

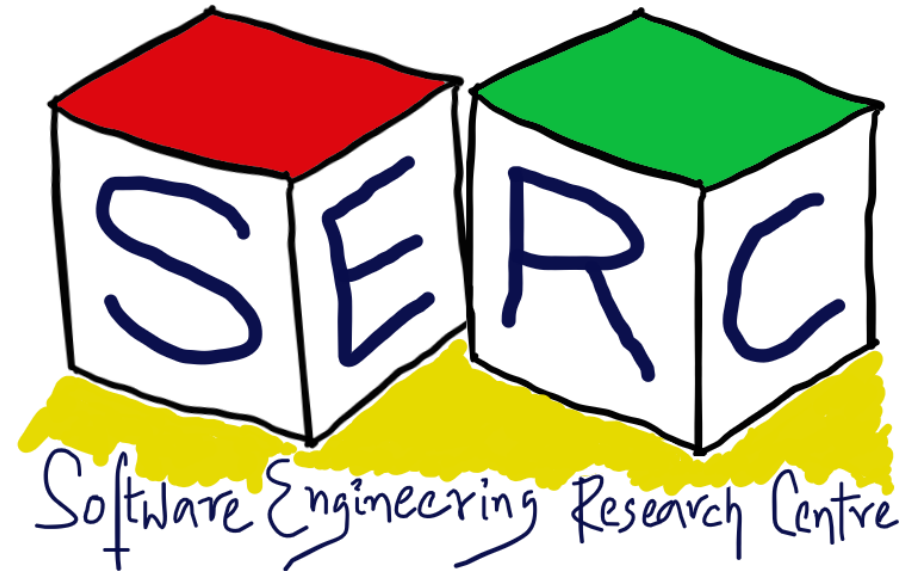
Architectural Styles & Patterns

CS6.401 Software Engineering

Dr. Karthik Vaidhyathan

karthik.vaidhyathan@iiit.ac.in

<https://karthikvaidhyathan.com>



Acknowledgements

The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

1. Software Architecture in Practice, Len Bass, 2nd, 3rd edition
2. Various sources from the web that has been duly credited in the respective slide

Software Architecture

The Software Architecture is the earliest model of the whole software system created along the software lifecycle

- A set of components and connectors communicating through interface
- A set of architecture design decisions
- Focus on set of views and viewpoints
- Developed according to **architectural styles**





Software Architectural Styles

Architectural Styles



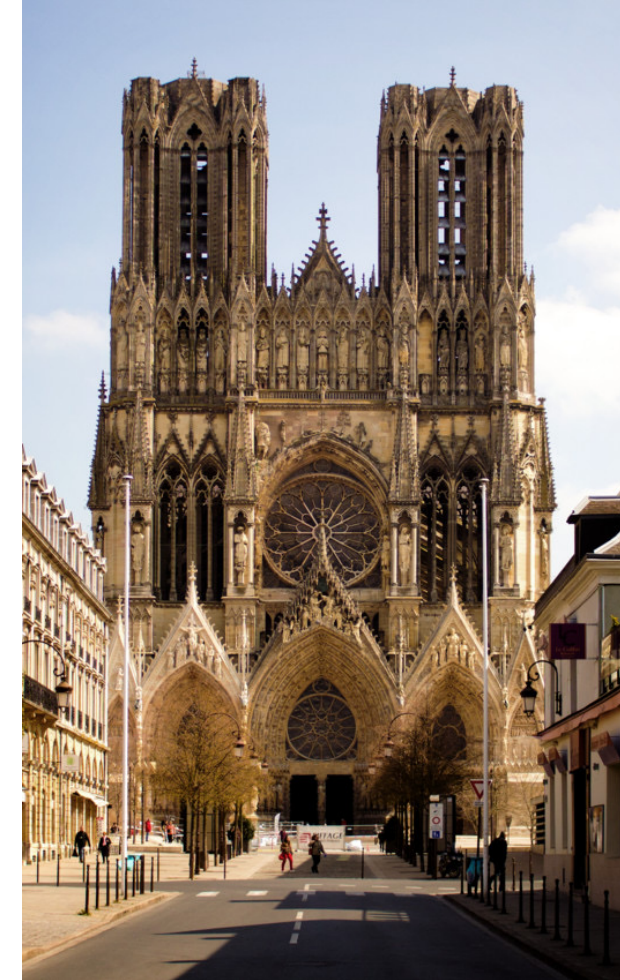
Classic architecture



Romanesque Architecture



Dravidian Architecture



Gothic architecture



Software Architectural Styles

Set of **design rules** that identify the **kinds of** components and connectors that may be used to compose a system or subsystem, together with **local or global constraints** on the way the composition is done

[Shaw and Clements, 1996]



Software Architectural Patterns

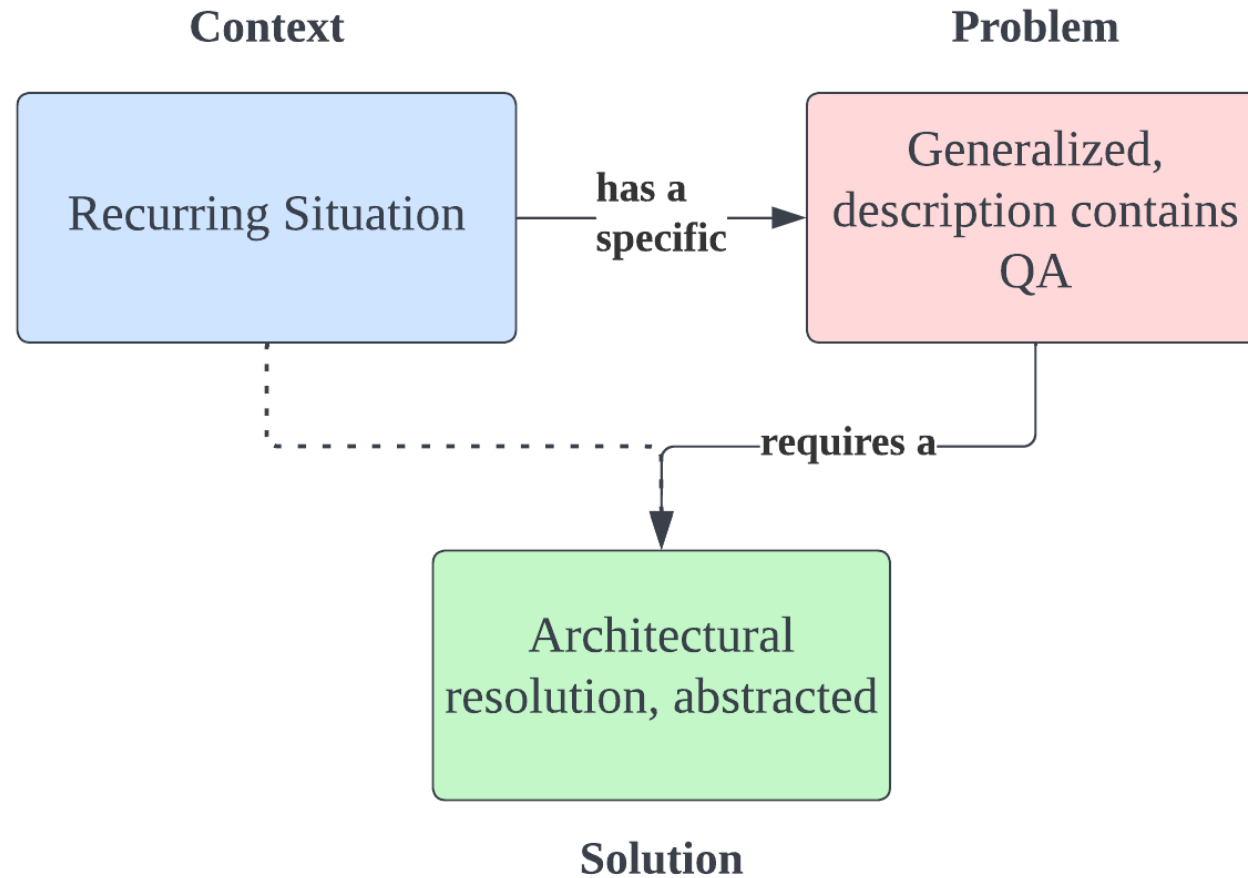
Architectural Patterns

1. Collection of design decisions found in practice
2. Has known properties that permit reuse
3. Describes a class of architecture

One does not invent patterns, one just discovers them – They are found in practice

There is never a complete list of patterns

Architectural Patterns

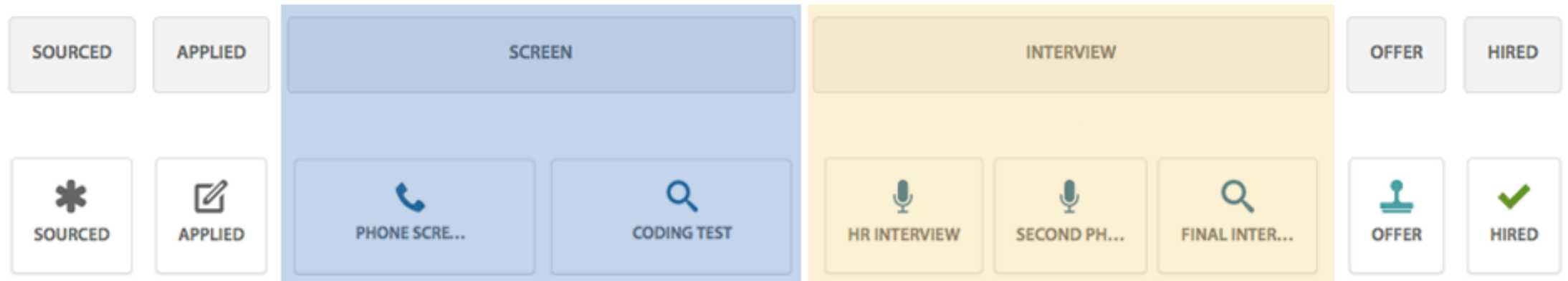


Pattern documentation template: $\{context, problem, solution\}$



Pipe and Filter

The Pipe and Filter Pattern - Intuition



The Pipe and Filter Pattern

Context

Many systems required to transform discrete stream of data. Occur repeatedly in practice

Problem

Reusable, loosely coupled components, simple and generic interactions

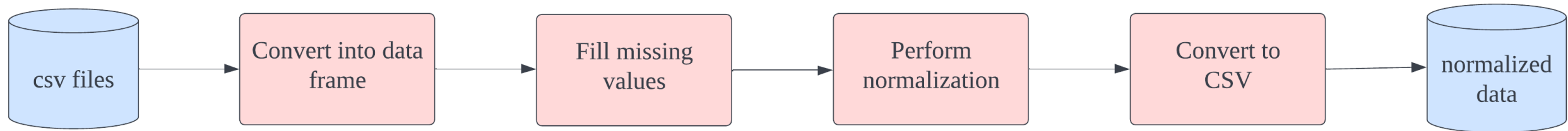
flexibility, resuability

Divide into pipes and filters, pipes transport, filters process/transform

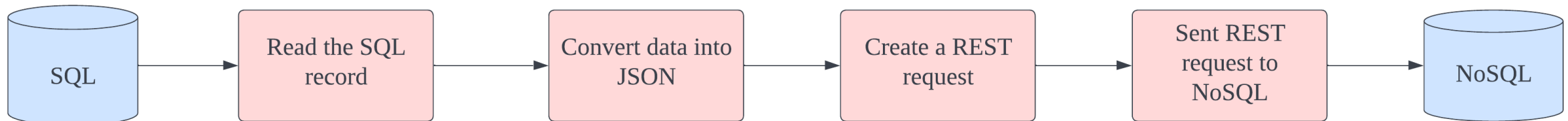
Solution

The Pipe and Filter Pattern – Some Use Cases

Data preparation for ML



Data Migration



The Pipe and Filter Pattern



Architectural Elements

1. Filter (Component)

Transforms data from input to output

Can execute concurrently, incrementally transform

2. Pipe (Connector)

Single source for input, single target for output

Does not alter data passing through pipe

The Pipe and Filter Pattern

Constraints

1. Pipes connect filter output to filter input
2. Filters must agree on type of data being passed from pipe
3. Specialization is more like a linear sequence of actions => Pipelines

Weakness

1. Not good for interactive system
2. Large number of filters can add substantial overhead
3. Not suitable for long running jobs



Blackboard

The Blackboard Pattern - Intuition



The Blackboard Pattern

Context

Open problem domain with various partial solutions

Problem

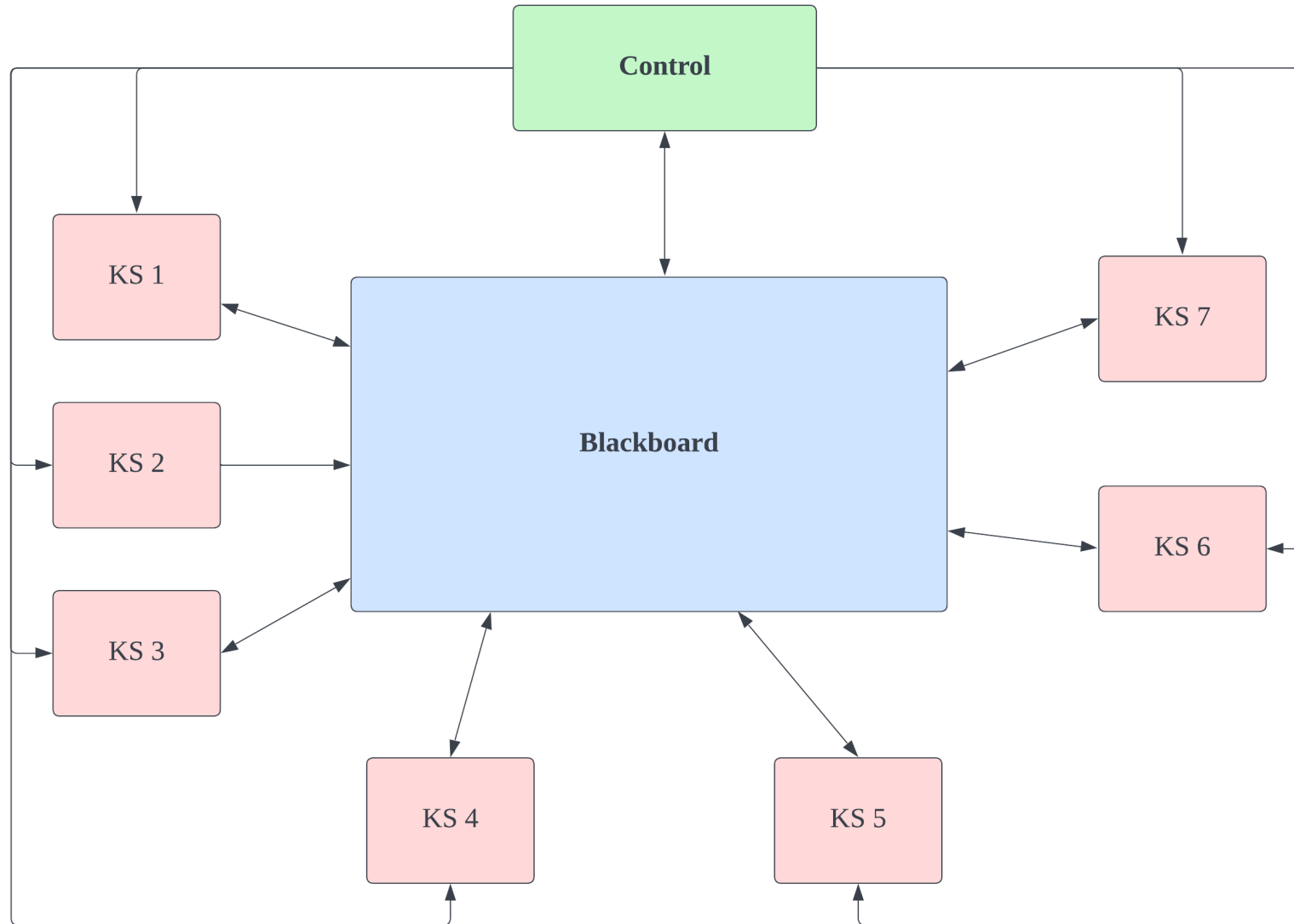
The partial solutions needs to be integrated

Flexibility, Maintainability,

Decompose the software into blackboard, knowledge source and control

Solution

The Blackboard Pattern



The Blackboard Pattern

Architectural Elements

Blackboard

1. Global repository containing input data and partial solutions

Knowledge Sources (KS)

1. Separate and independent components
2. Contains the knowledge required to solve the problem

Controller

1. Component managing course of problem solving (eg: manage KS)

Relation: Attachment relation (KS's attached to the blackboard)

The Blackboard Pattern

Constraints

1. No direct communication among the KS
2. Any interaction happens via the blackboard

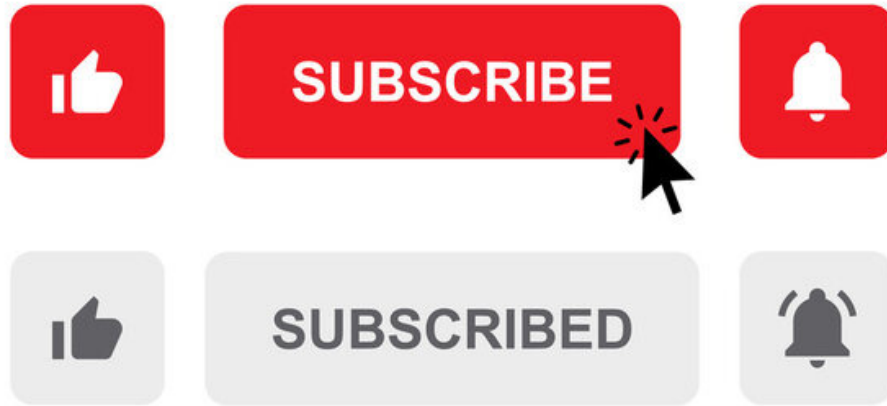
Weakness

1. Blackboard can become a bottleneck (too many KS)
2. Difficult to determine the partitioning of knowledge
3. Control can be very complex



Publish Subscribe

Publish Subscribe Pattern - Intuition



Youtube subscription



Newspaper subscription

The Publish Subscribe Pattern

Context

Number of independent producers and consumers that must interact. The number or nature of data is not fixed

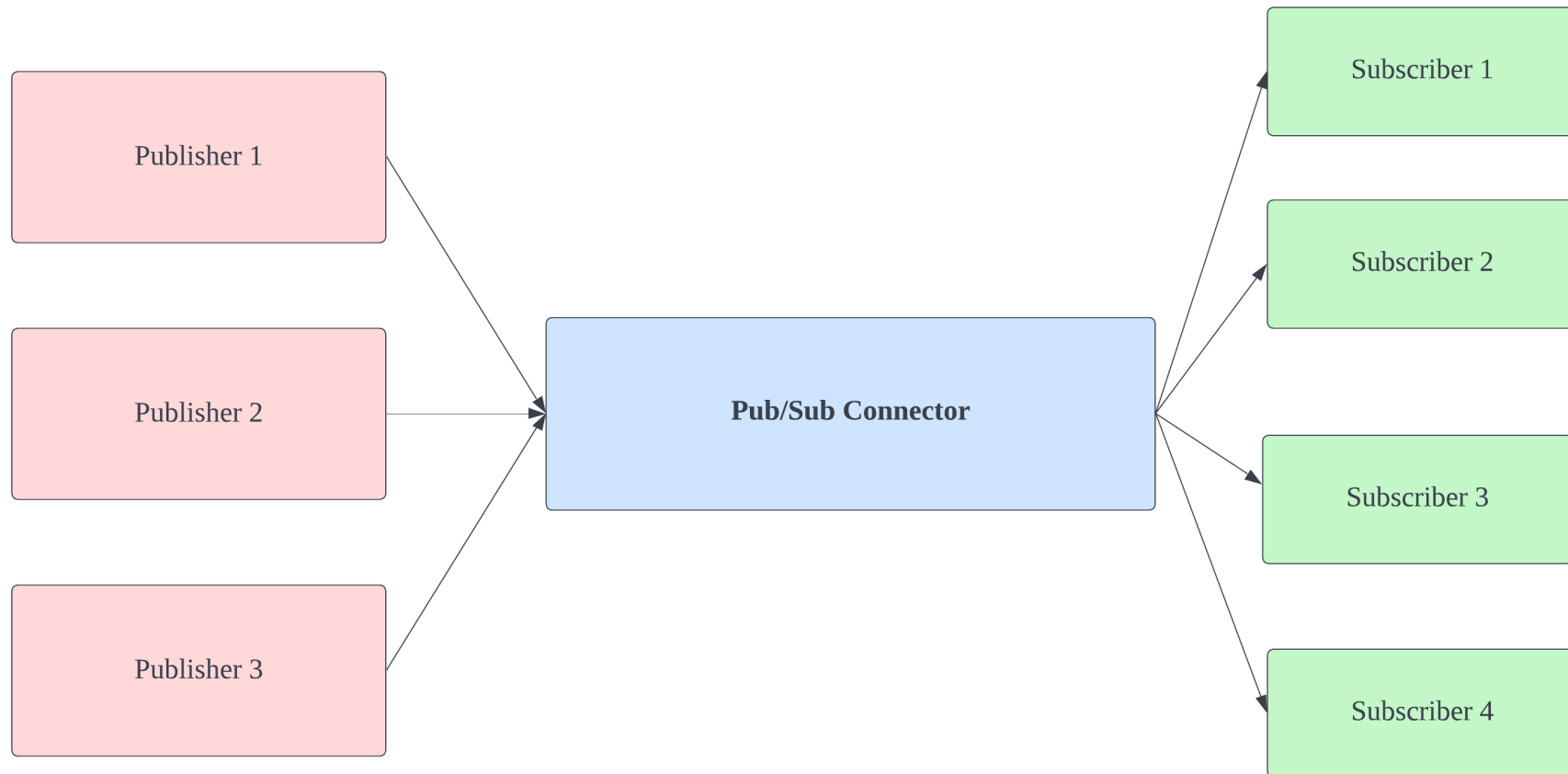
Problem

How to create integration mechanisms that support transmission without coupling
scalability, manageability

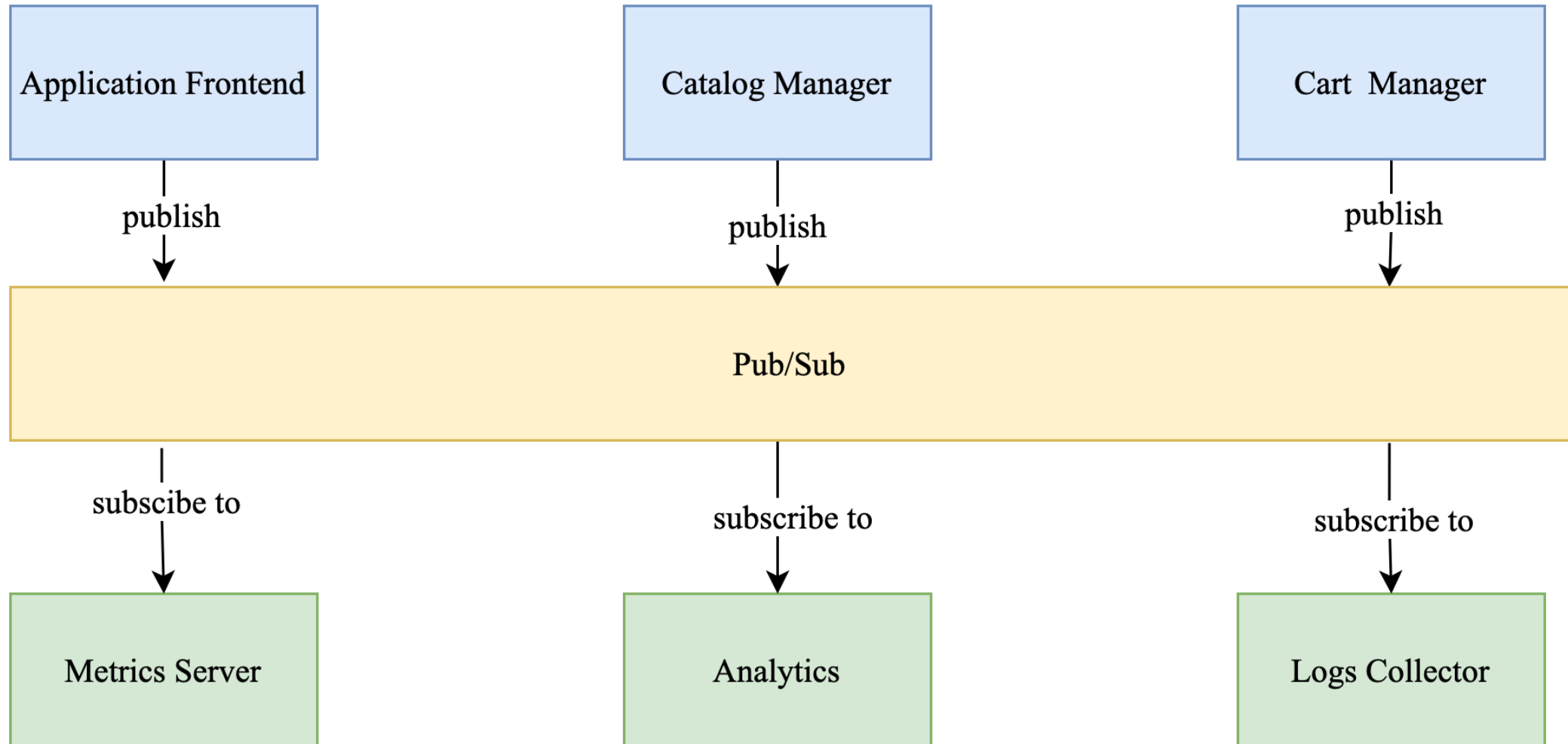
Publishers publish information which can be subscribed to by the subscribers. Have connectors to manage

Solution

The Publish Subscribe Pattern



Publish-Subscribe Pattern – An Example



Publish-Subscribe Pattern

Architectural Elements

Publisher

Components that produces messages/events

Subscriber

Components that consume the messages/events produced by publisher

Pub-Sub Connector

Component that has *announce* and *listen* roles for publishers and subscribers

Relation: Attachment relation associates pub/sub components with the connectors

Publish-Subscribe Pattern

Constraints

1. All components are connected to a connector (bus or a component)
2. Restrictions on which component can listen to what
3. A component may be both a publisher and a subscriber

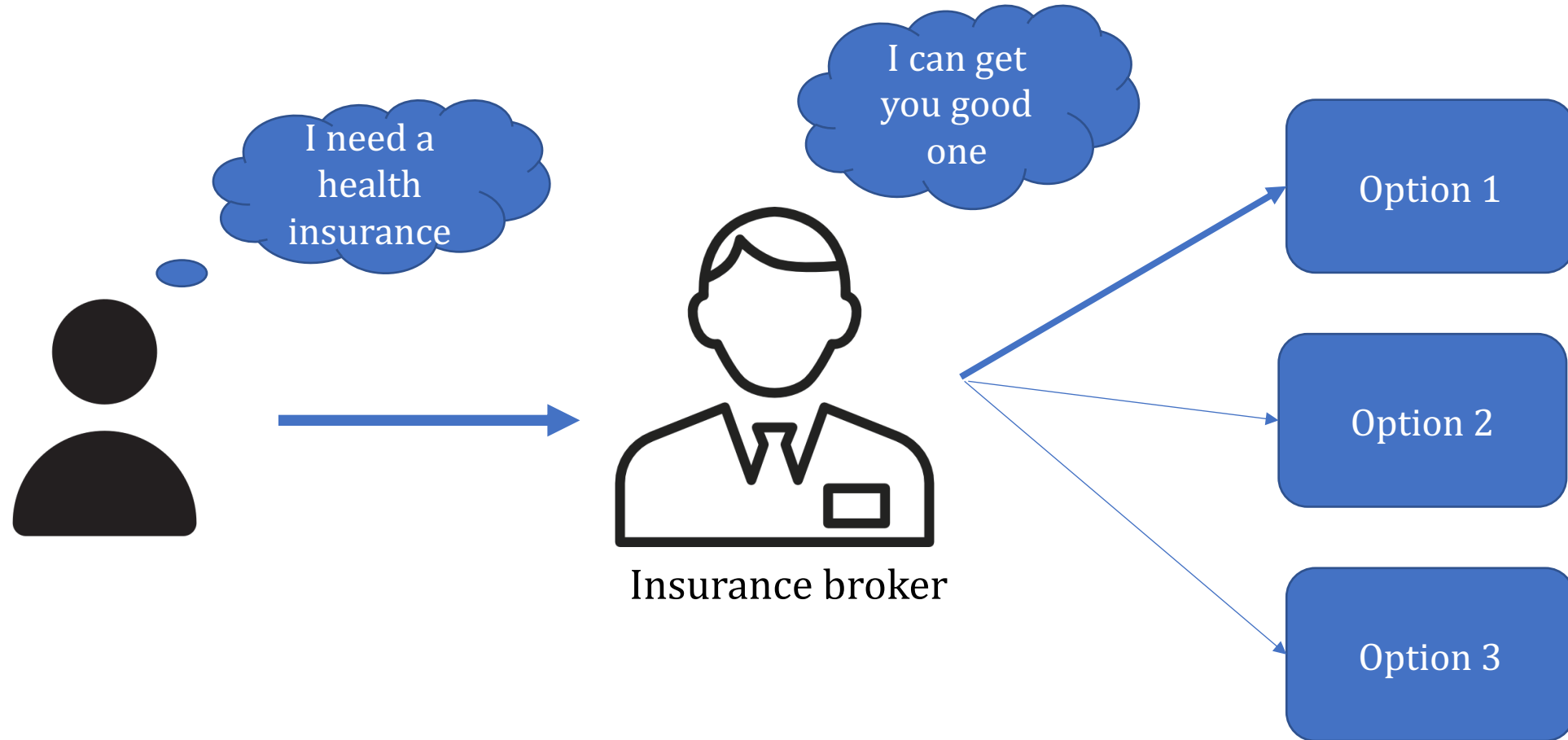
Weakness

1. May increase latency
2. Can have a negative impact on predictability of message delivery time
3. Less control on ordering of messages
4. Delivery of message is not guranteed



Broker

The Broker Pattern - Intuition



The Broker Pattern

Context

Many systems are collection of distributed programs. They need to exchange information and be available

Problem

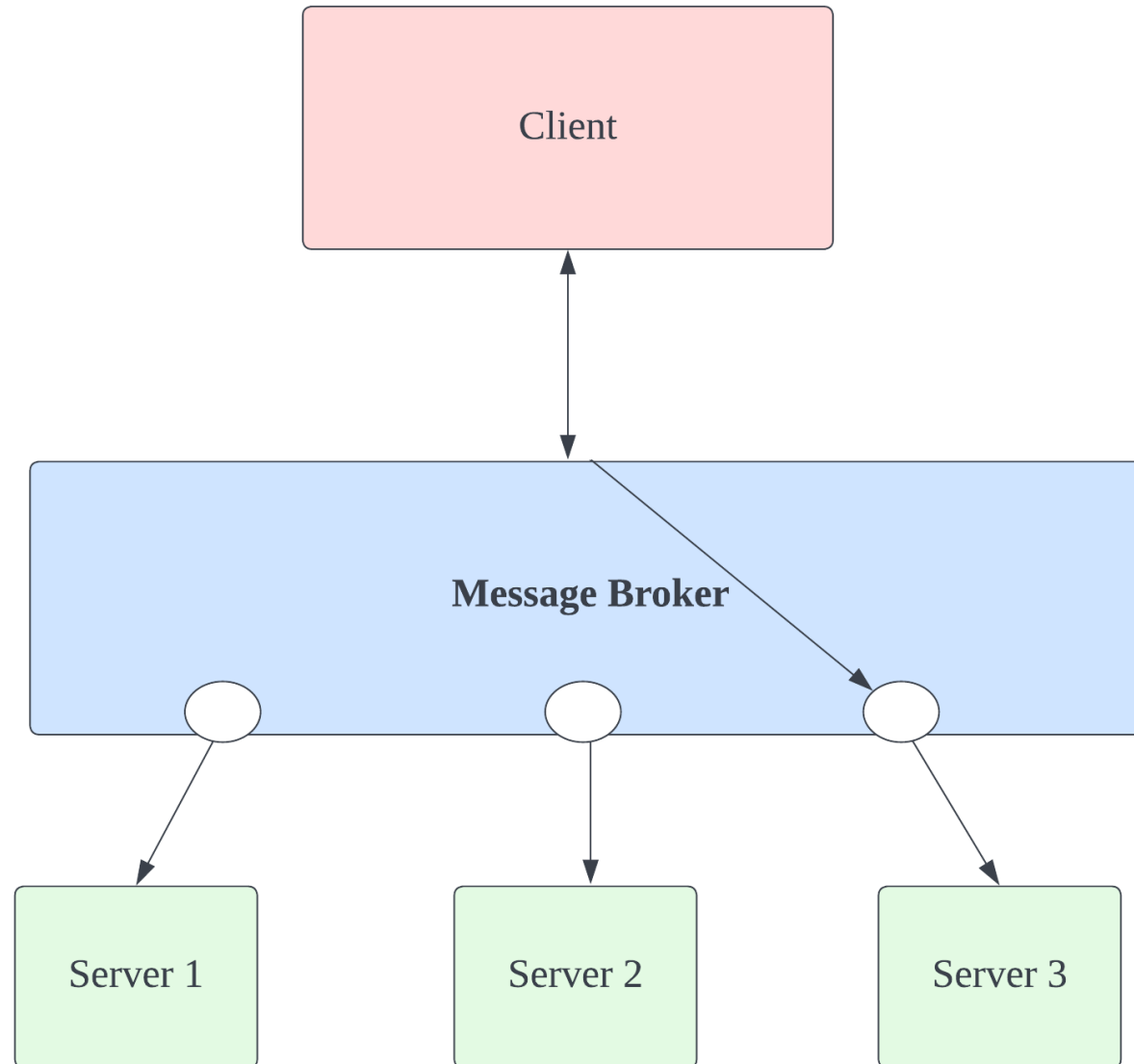
How to structure a distributed system such that service users need not worry about location of providers

availability, interoperability

Separate user of functionalities from provider of functionalities using an intermediary component called broker

Solution

The Broker Pattern



The Broker Pattern

Architectural Elements

Client

Component which is the requester of functionalities/services

Server

Component(s) which is the provider of functionalities/services

Broker

Component that locates appropriate server to fulfill client's request

Client-side and Server-side proxy

Manages actual communication with the broker

Relation: attachment associates clients and servers with brokers

The Broker Pattern

Constraints

1. Client can only attach to a broker (potentially via client-side proxy)
2. Server can only attach to a broker (potentially via server-side proxy)

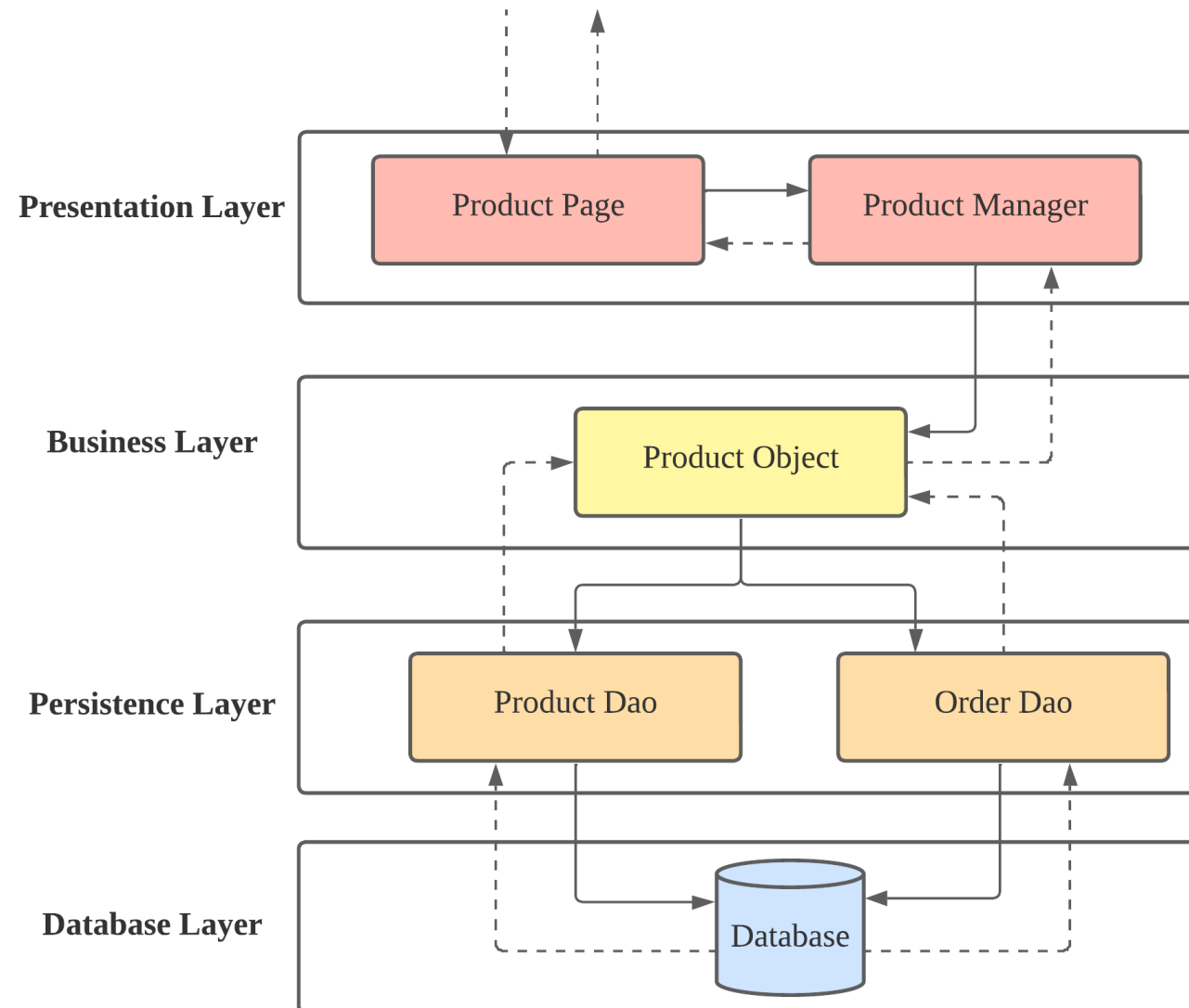
Weakness

1. Brokers can result in performance bottleneck (latency)
2. Broker can be a single point of failure
3. Can be subjected to security attack
4. Adds up-front complexity
5. It may be difficult to test



Layered

Layered Architectural Pattern - Example



Layered Architectural Pattern

Context

Develop and evolve portions of systems independently. Promote separation of concerns.

Problem

Modules can be developed and evolved separately with little interaction

**modifiability, portability,
reuse**

Divide the software into units called layers. Each layer is a grouping of modules

Solution

Layered Architectural Pattern

Architectural Elements

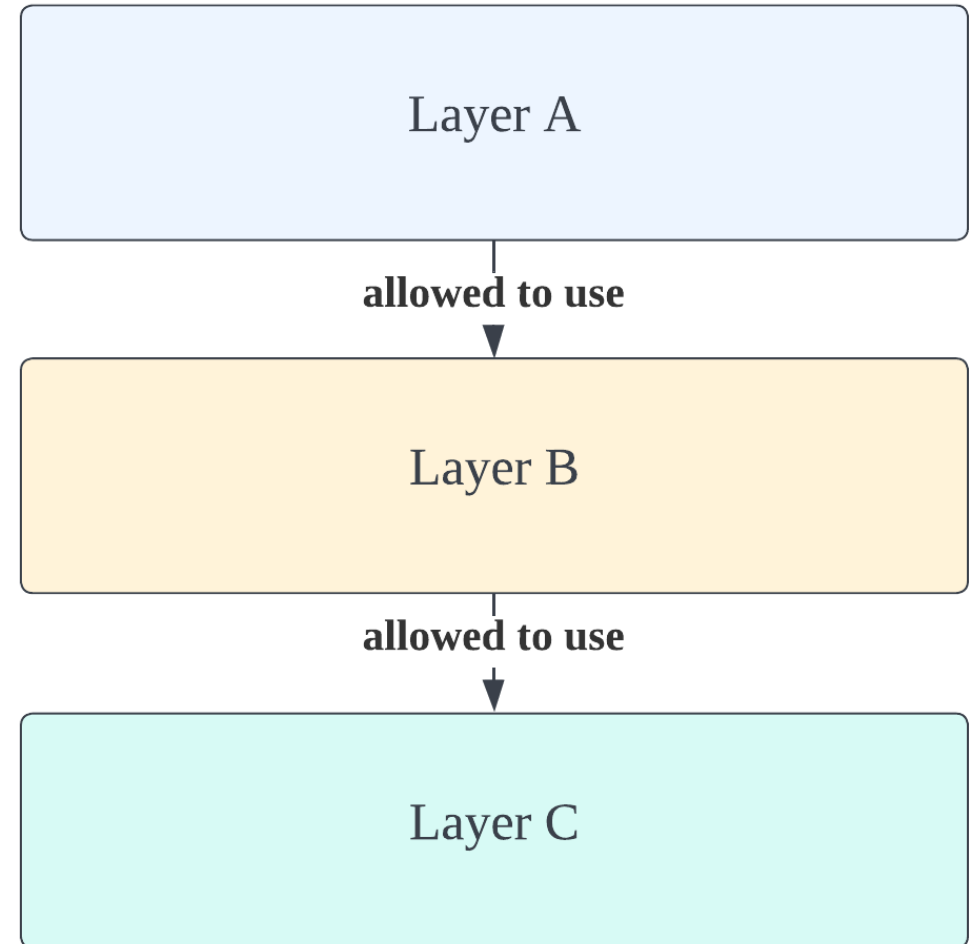
Layer

1. Kind of a module
2. Description should define the what modules it can contain

Relation

1. Allowed to use

The design should always define the usage rules



Layered Architectural Pattern

Constraints

1. Every piece of software is exactly allocated to one layer
2. There are at least two layers (often more!)
3. Allowed to use relations should be acyclic

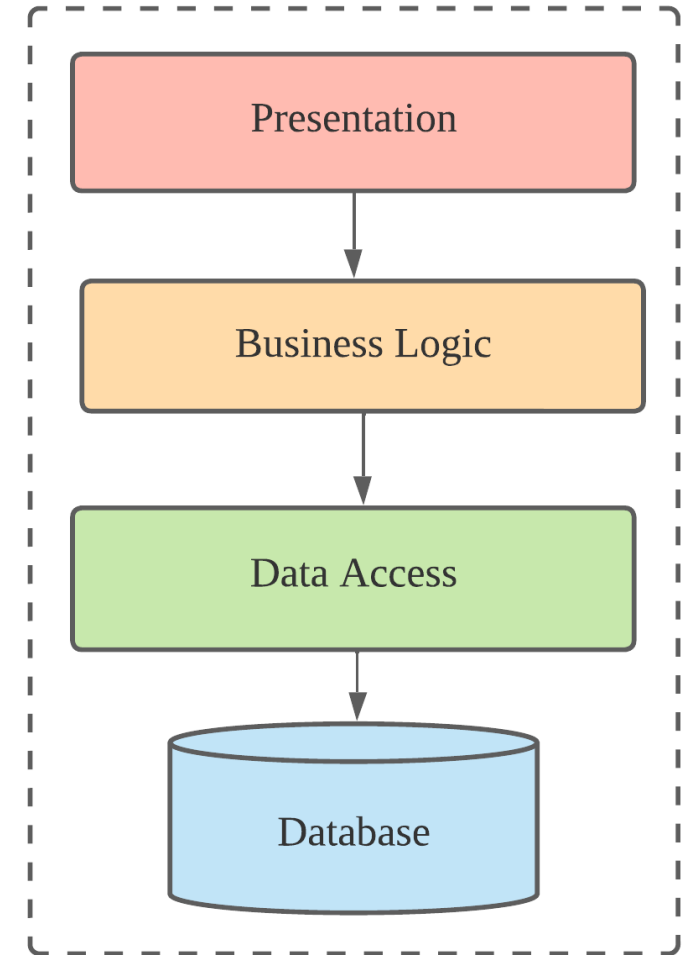
Weakness

1. The addition of layers adds up-front cost and complexity
2. Performance bottlenecks

Layered Architectural Pattern – Sommon Issues

One of the most commonly used patterns – still people get it wrong!

1. Define proper relations with key (which layer can use what)
2. Stack of boxes lined up does not belong to layered
3. A layer isn't allowed to use any layer above it.



Monolith?



Monolith of Utah, USA



Menhir (monolith), France



Ponce Monolith, Bolivia

Monolithic Approach to E-commerce

Organizational



UI Developers/
App developers



App backend team(s)



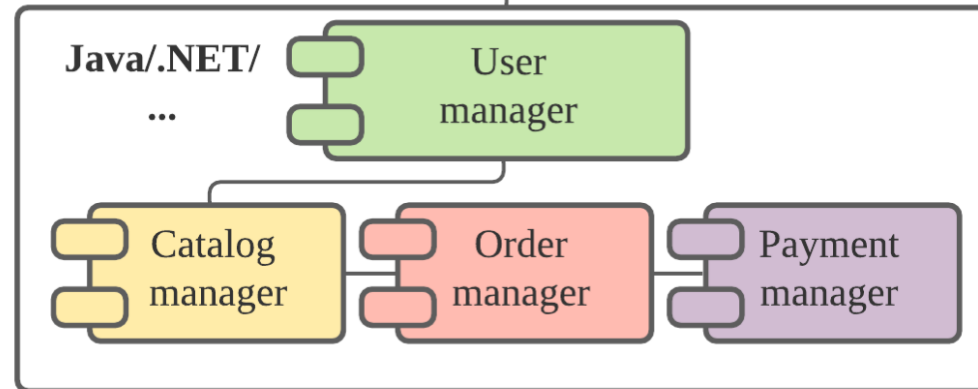
DB Administrators

Architecture

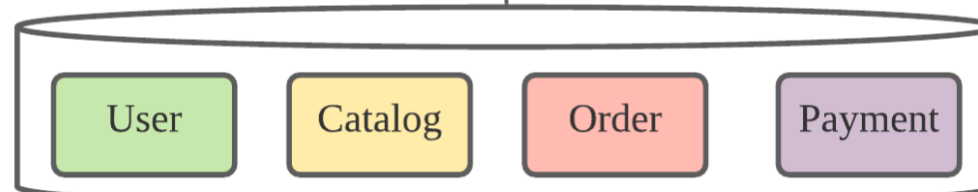
HTML/CSS
and JS



Android/iOS



Oracle/MySQL/..



Deployment



Application
webserver
(for web)



Backend
server




Database
server

Monolithic Approach – What are some pitfalls?

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

Monolith has its own advantages too!



Service-Oriented

The Service-Oriented Pattern

Context

A number of services offered by service providers and consumed by service consumers. Service consumer should be able to use services without knowing detailed implementation

Problem

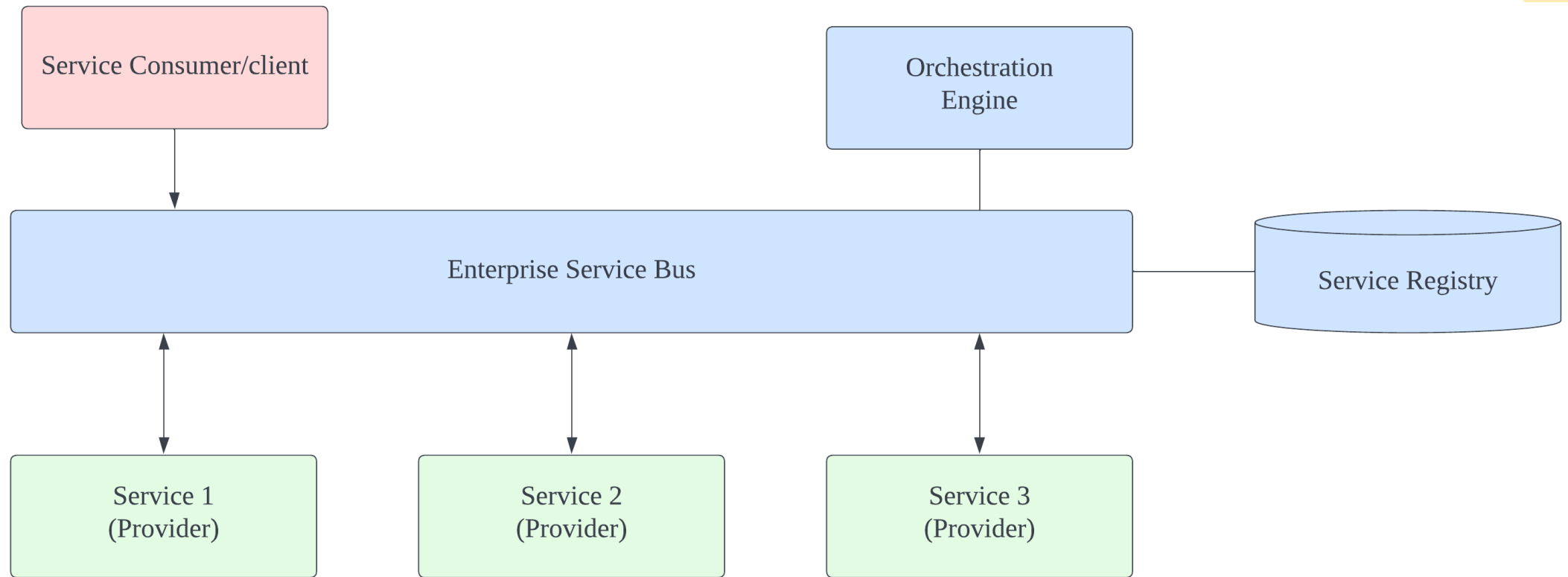
How to provide support for interoperability among different components running in different platforms implemented in different languages

availability, performance, security

Collection of loosely coupled services with clearly defined interfaces. Can be implemented in different languages. Supports communication between and to/from services

Solution

The Service Oriented Pattern



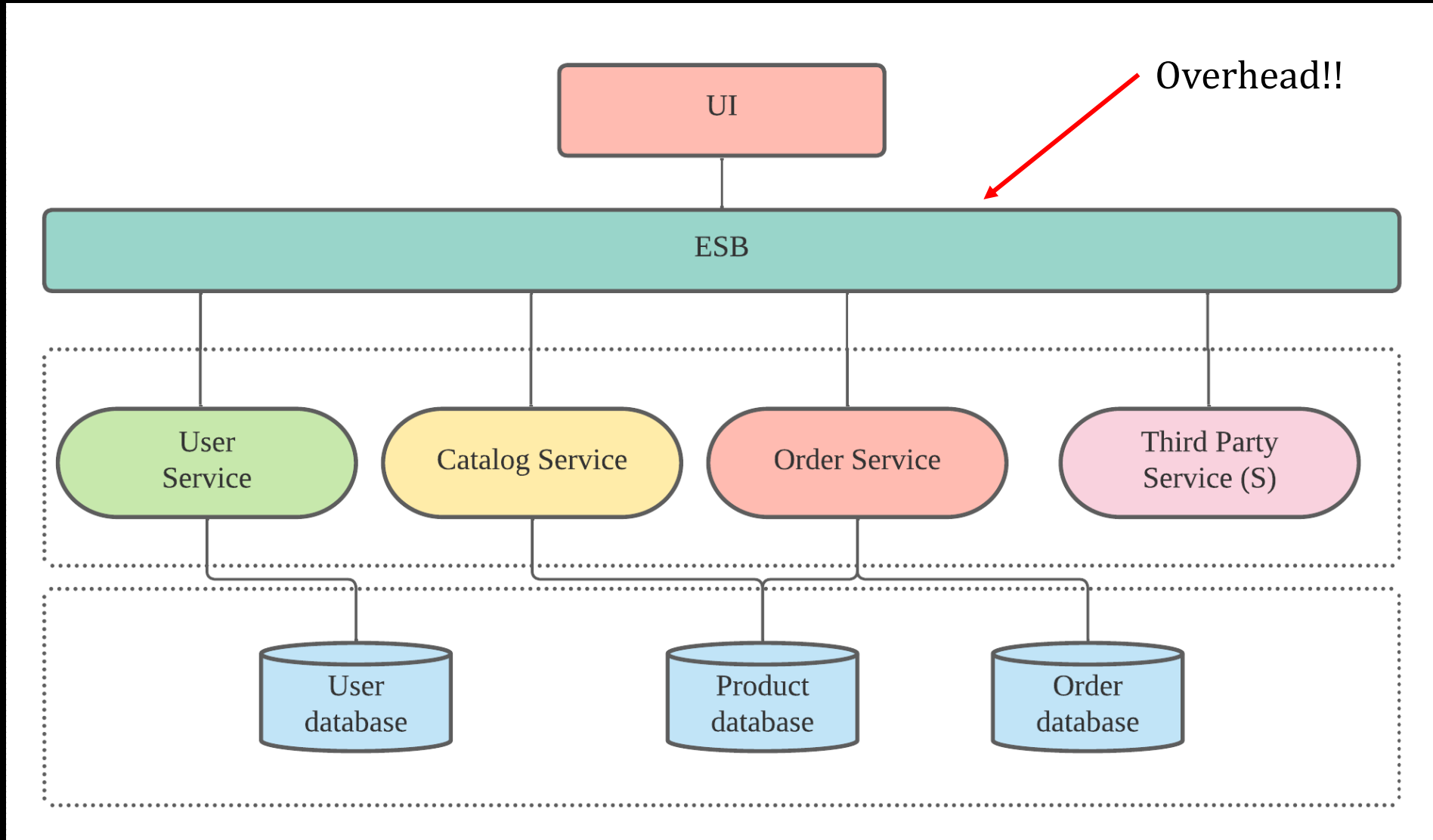
SOA Pattern - Architectural Elements (Components)

1. Service Providers: Components that provide 1 or more services through defined interfaces
2. Service Consumers: Invoke services directly or through intermediary
3. ESB: Intermediary component that can route and transform messages
4. Service Registry: Providers can register services, consumers can discover services
5. Orchestration Server: Coordinates interaction between consumers and providers based on languages

SOA Pattern - Architectural Elements (Connectors)

1. SOAP Connector: SOAP Protocol for synchronous communication over HTTP
2. REST Connector: Relies on request/response operations over HTTP
3. Asynchronous messaging connector: For point-to-point asynchronous message exchanges or pub-sub exchanges

SOA Applied to E-Commerce



SOA Pattern

Relations

Attachments of different components to available connectors

Constraints

1. Service consumers are connected to providers (ESBs or other intermediary component may be used)

Weakness

1. Complex to build
2. Performance bottlenecks due to middleware
3. Performance guarantees are usually not met



Time to Evolve: Microservices

SOA Pattern - Architectural Elements (Connectors)

Domain Driven Design

Large-Scale Systems

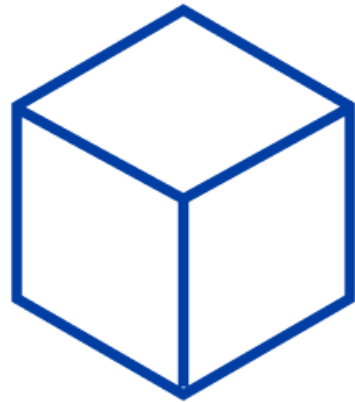
Continuous Delivery

Infrastructure Automation

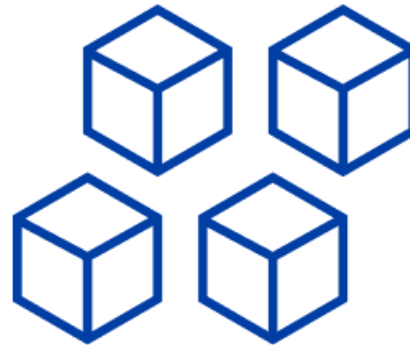
Small Autonomous Teams

Microservices

Moving Towards Microservices



MONOLITHIC
Single unit



SOA
Coarse-grained



MICROSERVICES
Fine-grained

1990

2000

2010

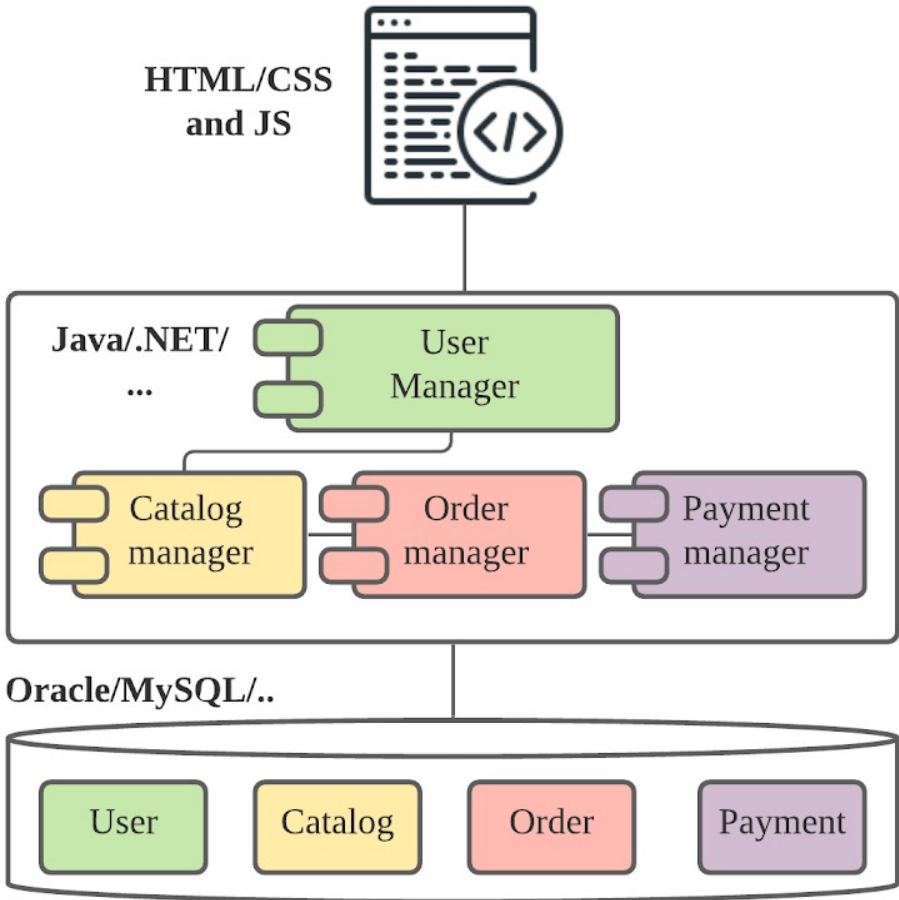
Microservices: What does it Mean?

“Small autonomous services that work together” -- Sam Newman

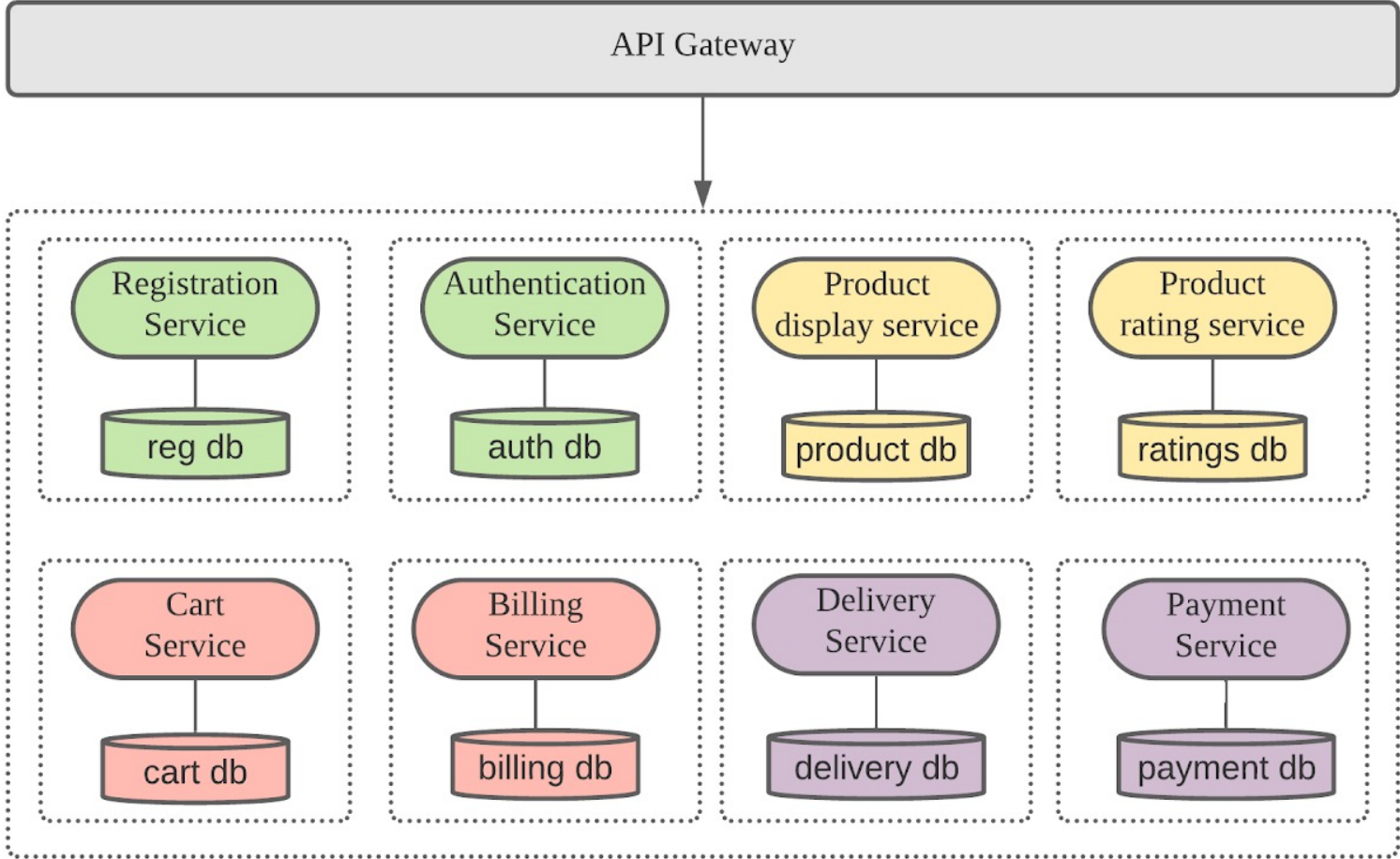
“It is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API” -- Martin Fowler

Microservices: What does it Mean?

Monolithic Version



Microservices Version

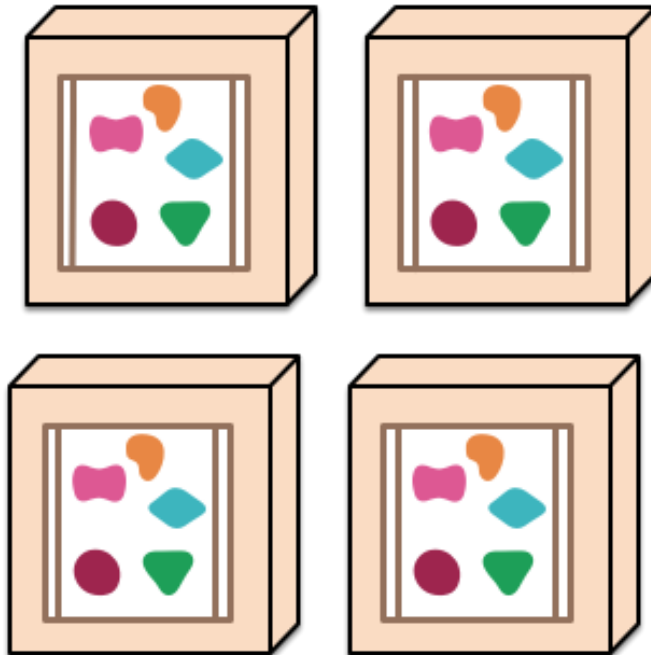


Microservices: What does it Mean?

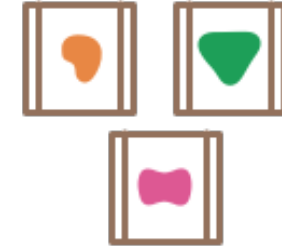
A monolithic application puts all its functionality into a single process...



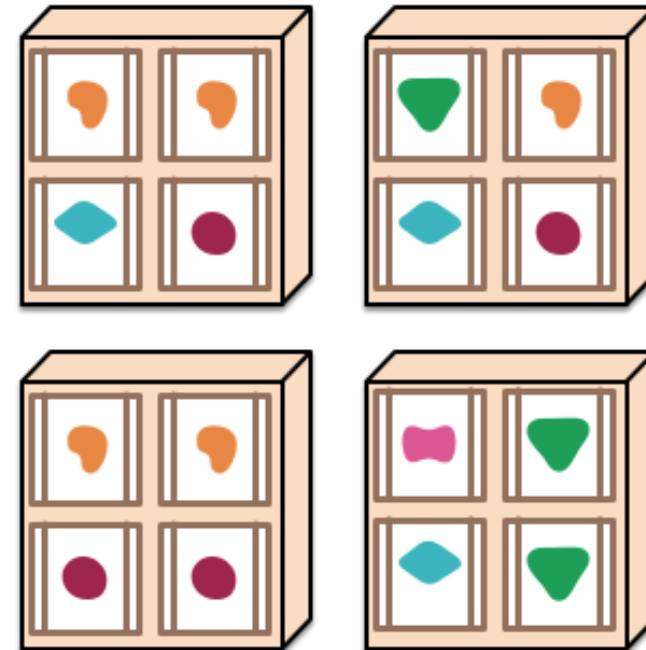
... and scales by replicating the monolith on multiple servers



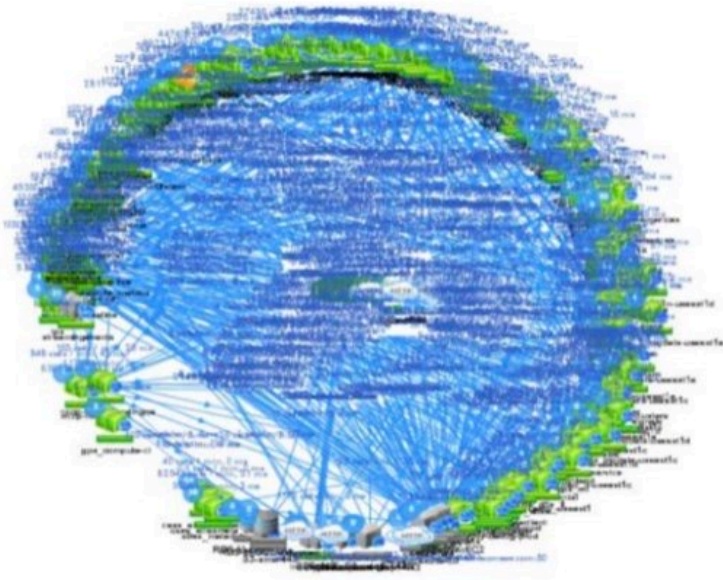
A microservices architecture puts each element of functionality into a separate service...



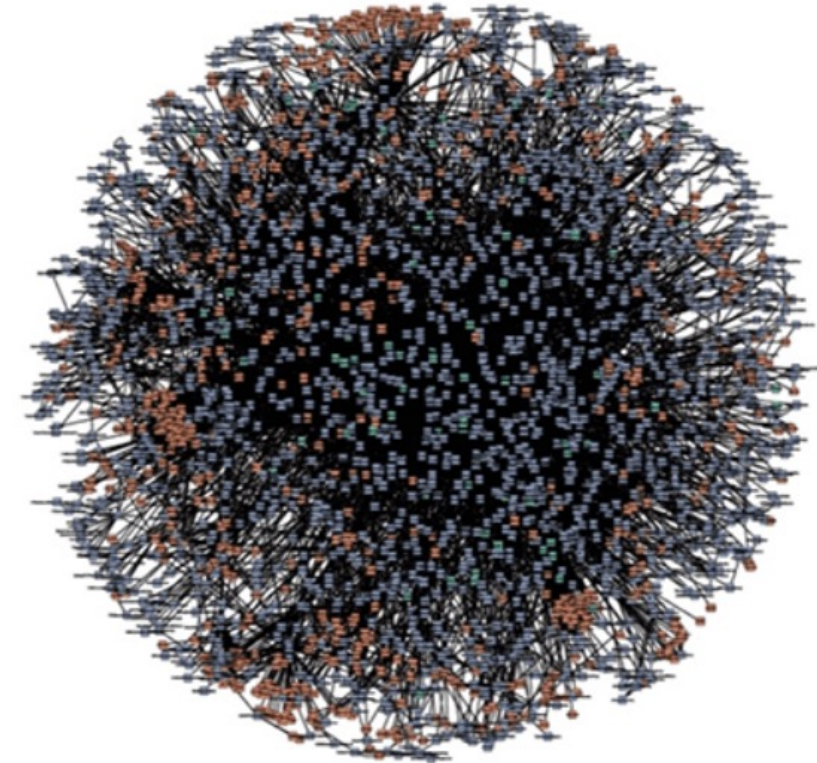
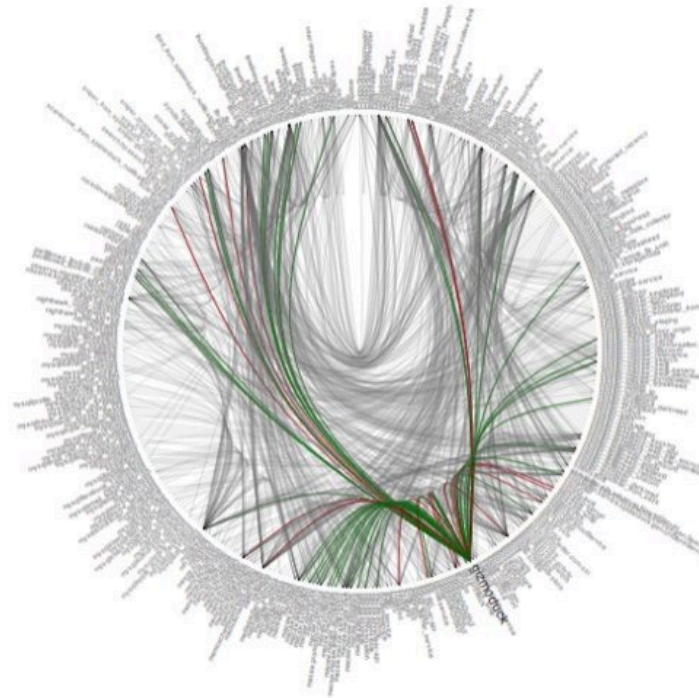
... and scales by distributing these services across servers, replicating as needed.



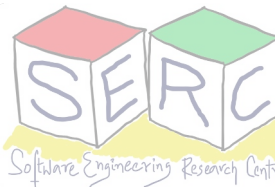
Microservices: Who Uses Them?



NETFLIX



amazon.com



Amazon's API Mandate



Jeff Bezos,
Founder and President, Amazon

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols – doesn't matter.
5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
6. Anyone who doesn't do this will be fired.
7. Thank you; have a nice day!

Microservices: Key Advantages

Scaling is Easy

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

Heterogeneity

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

Resilience

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

Microservices: Key Advantages

Organizational Alignment

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

Composability

- Easily compose microservices to get new functionality

Replaceability

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

Ease of Deployment

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



How to identify Microservices?

Main Takeaways

- Architectural Pattern serves as guidelines
- Always be aware of trade-offs
- A complex system can consists of multiple architectures
- Think about an IoT system, e-commerce system or any production system



Thank You



Course website: karthikv1392.github.io/cs6401_se

Email: karthik.vaidhyanathan@iiit.ac.in

Web: <https://karthikvaidhyanathan.com>

Twitter: @karthi_ishere

