2024

Research Document

KREMER,YORDI Y.C.T.J.

What architecture is best suited for my project Requirements?

With this sub question, I hope to find out what a good architecture to use is for my project and my learning outcomes this semester I already have some in my like Microservices which is used a lot for Enterprise software but I want to compare it to other architectures and explain why I will choose a certain architecture so I have a better understanding why I am making this project in a certain way.

Field Research	Library Research	Workshop
Explore user requirements	Literature study	IT architecture sketching

Explore user requirements

This application will be about streaming a video to multiple users and for the users to be able to interact with the video like playing and pausing it, this should all work in sync with the users in the same room. This is a very important things to keep in mind when choosing my architecture for this project.

Non-functional requirements

- Scalability
- Security by Design
- Cloud native development
- Distributed Data

These are the non functional requirements that I should implement in this project to fulfill my learning outcomes so I have to think about these requirements when choosing my architecture.

Literature Study

I will be comparing system architectures with each other, for my research I chose the most Prevalent ones.

For each architecture I will explain gather an explanation of the architecture good and bad things about this architecture according to my requirements and I will be finding designs of this architecture to get a visualization of how the architecture works.

Monolithic Architecture

In this architecture, the entire application is built as a single, self-contained unit. All components and functionalities are tightly coupled and deployed together. Monolithic architecture is relatively simple and suitable for smaller applications, but it can become challenging to maintain and scale as the system grows.

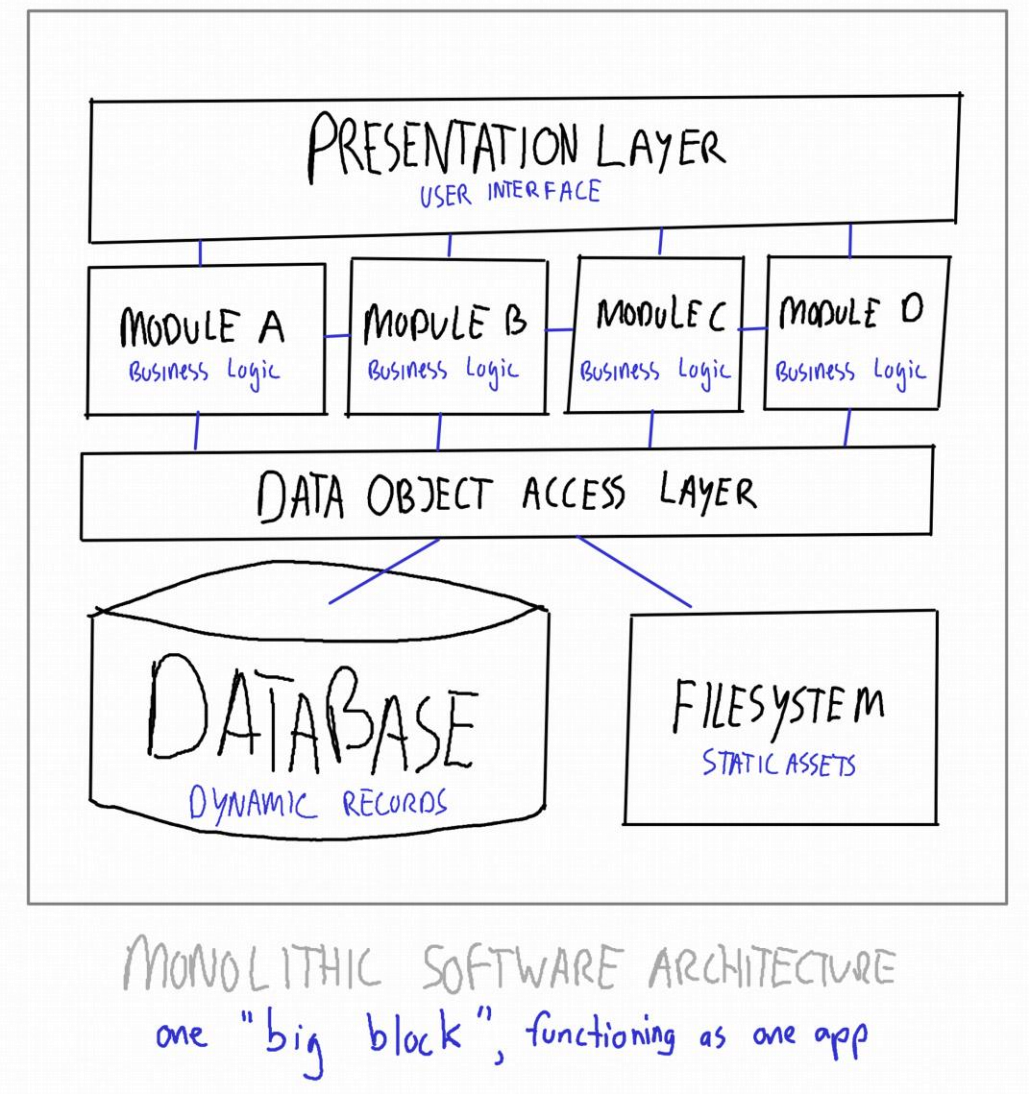
Important Pros

- Simplicity: Easier to develop, deploy, and maintain due to its single-codebase nature.
- Rapid Development: Faster development cycles as all components are tightly integrated.
- Ease of Testing: Simplified testing processes with all components running in the same environment.

Important Cons

- Scalability Limitations: Difficulty in scaling individual components independently, limiting scalability.
- Lack of Flexibility: Challenges in adopting new technologies or making changes due to tight coupling.
- Maintenance Challenges: Complexities in maintaining and updating a large, monolithic codebase over time.

Diagram



Client-Server Architecture

This architecture involves dividing the system into two main components: clients and servers. Clients are end-user devices that request services from servers, which provide resources or perform specific tasks. Client-server architecture allows for distributed processing, scalability, and separation of concerns between client-side and server-side logic.

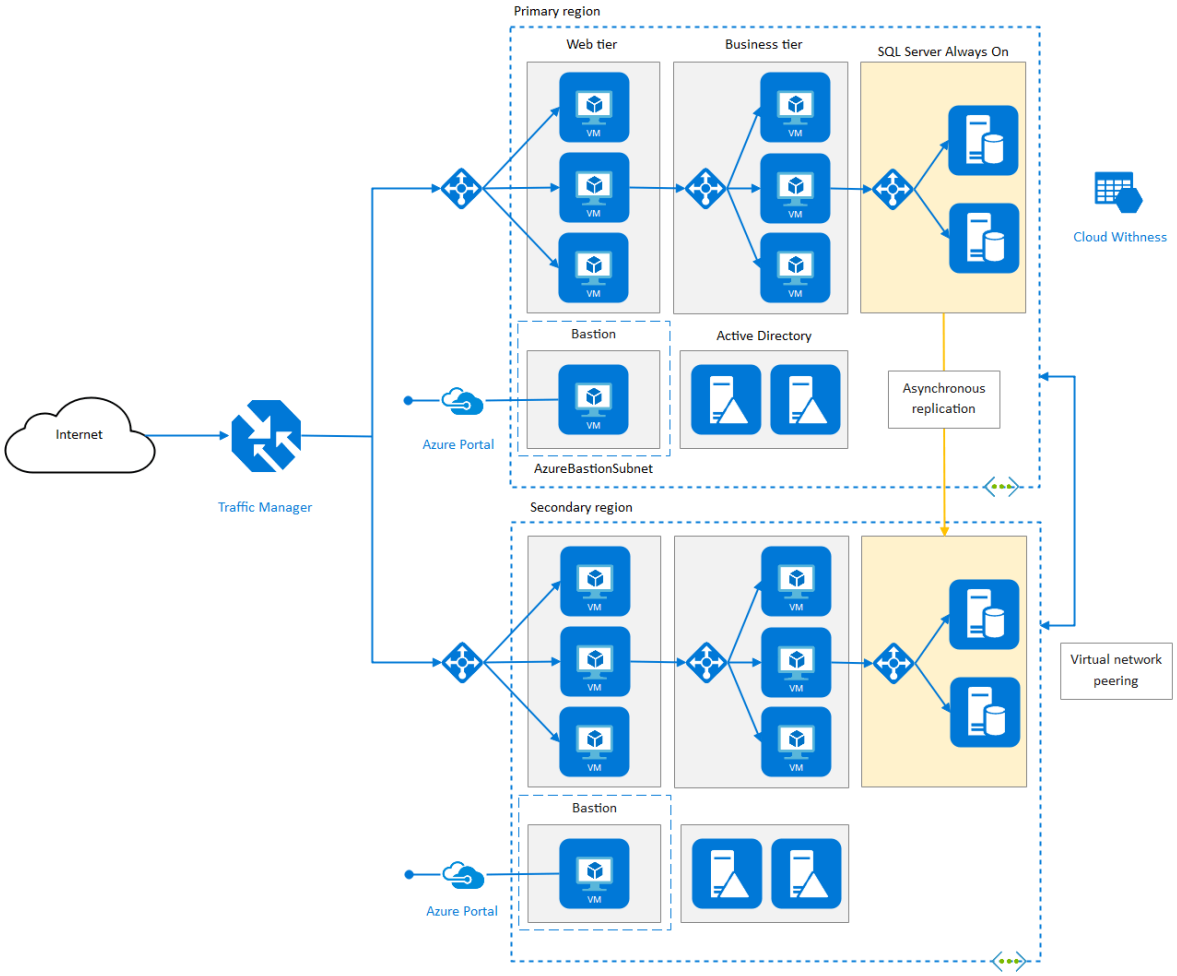
Important Pros

- Scalability: Allows for scaling server resources independently of client devices.
- Centralized Data Management: Easier to manage and secure data on centralized servers.
- Security Control: Enhanced security measures can be implemented centrally, reducing vulnerabilities.

Important Cons

- Single Point of Failure: Dependency on a central server can lead to system-wide failures if the server goes down.
- Network Dependency: Requires stable and reliable network connectivity for communication between clients and servers.
- Potential Performance Bottlenecks: Heavy reliance on server resources can lead to performance issues under high load.

Diagram



Service-Oriented Architecture

SOA is an architectural style that focuses on designing systems as a collection of loosely coupled, interoperable services. Each service performs a specific business functionality and can be independently developed, deployed, and consumed. SOA promotes reusability, flexibility, and interoperability between different components and systems.

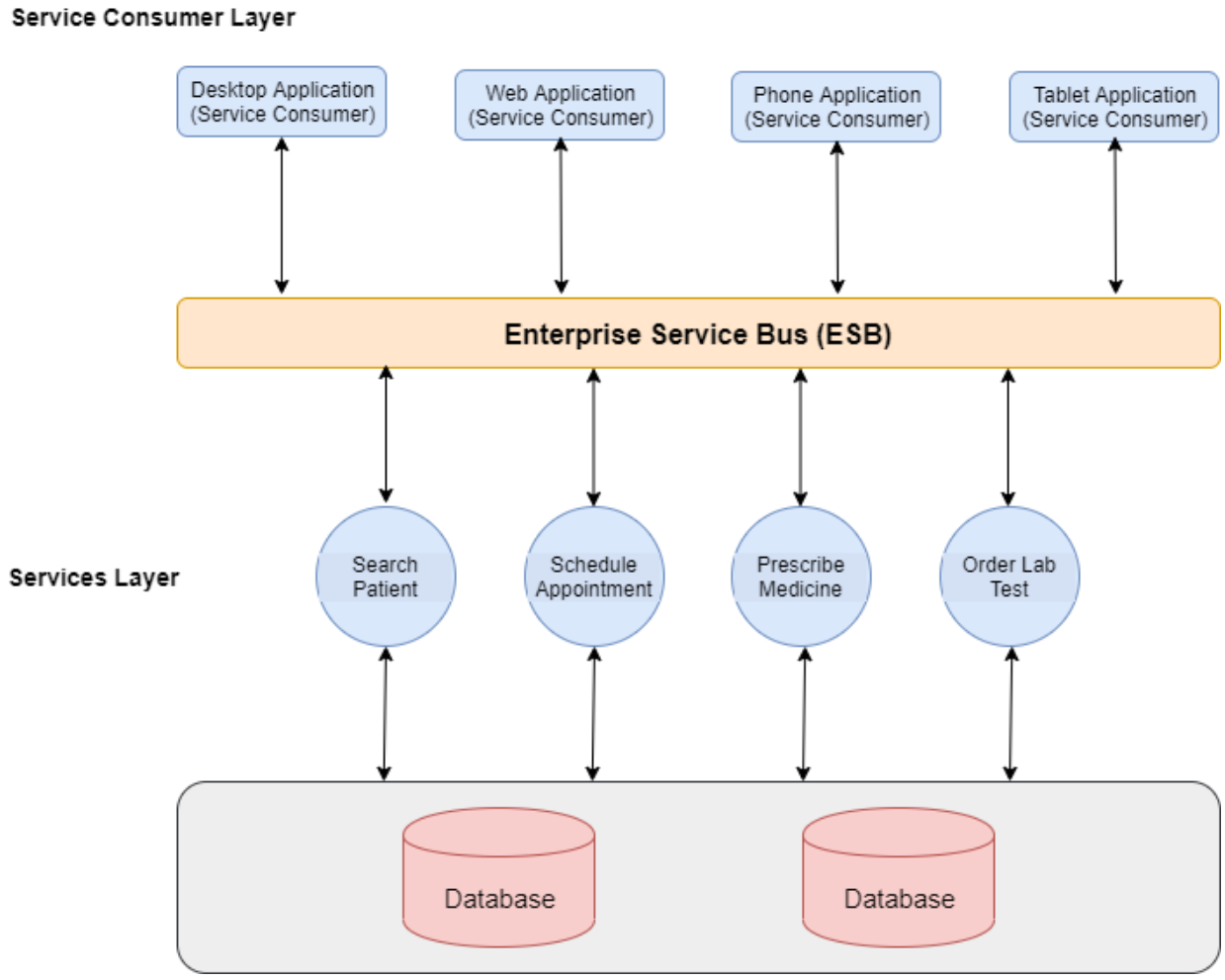
Most important Pros

- Modularity: Allows for independent development, deployment, and scaling of services.
- Flexibility: Facilitates easy integration and adaptation to changing business requirements.
- Reusability: Promotes reuse of services across multiple applications or components.

Most important Cons

- Increased Complexity: SOA introduces complexity in development, deployment, and maintenance due to managing multiple services and their interactions.
- Latency and Overhead: Communication between services over a network can lead to increased latency and overhead, impacting performance.
- Dependency Management: Services may have dependencies on each other, leading to cascading failures if one service goes down or undergoes changes.
- Testing Complexity: Testing distributed systems in an SOA can be challenging due to the need to simulate various service interactions and states.
- Versioning and Compatibility: Ensuring backward compatibility and managing different versions of services can be complex and require careful coordination.

Diagram



Microservices Architecture

Microservices architecture is an extension of the SOA concept, where the system is divided into a set of small, independent services. Each service represents a specific business capability and communicates with other services using lightweight protocols such as HTTP/REST. Microservices allow for independent scalability, ease of deployment, and fault isolation.

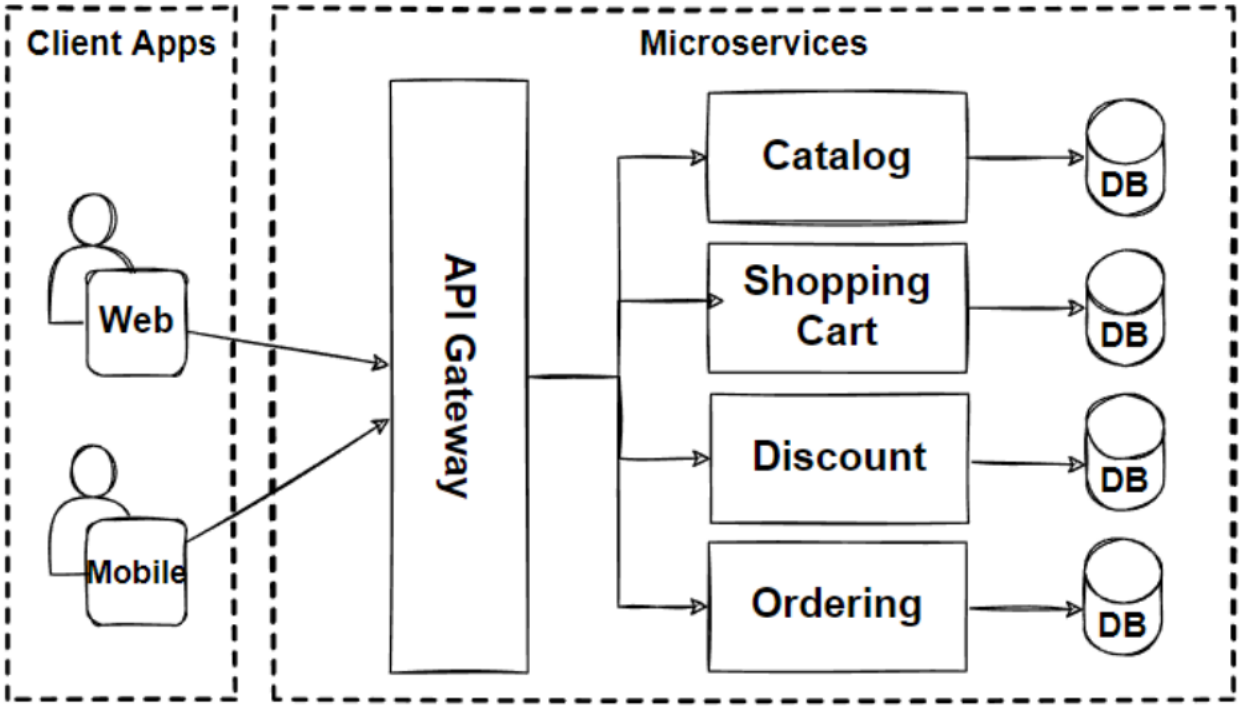
Most important Pros

- Scalability: Enables scaling individual components independently, allowing for greater flexibility in managing resources.
- Modularity and Agility: Promotes modular development, deployment, and scaling, facilitating rapid iteration and adaptation to changing requirements.
- Fault Isolation: Isolates failures to specific services, minimizing the impact of failures and improving system resilience.
- Technology Diversity: Allows for using different technologies and programming languages for different services, optimizing for specific use cases and requirements.
- Team Autonomy: Facilitates team autonomy and ownership, with each team responsible for developing, deploying, and maintaining their own services.

Most important Cons

- Increased Complexity: Microservices introduce complexity in development, deployment, and maintenance due to managing a larger number of services and their interactions.
- Operational Overhead: Requires additional infrastructure and operational overhead to manage and monitor multiple services, increasing costs and complexity.
- Communication Overhead: Communication between microservices over a network can introduce latency and overhead, impacting performance.
- Data Management Complexity: Data consistency and transaction management can be challenging in a distributed microservices environment, requiring careful design and implementation.
- Testing Challenges: Testing distributed systems in a microservices architecture can be complex and require sophisticated testing strategies to ensure the overall system behaves as expected.

Diagram



Event-Driven Architecture

EDA is an architectural pattern that emphasizes the production, detection, and consumption of events within a system. Events are used to trigger actions or communicate changes between components. This architecture promotes loose coupling and scalability by decoupling components through event-based communication.

Most important Pros

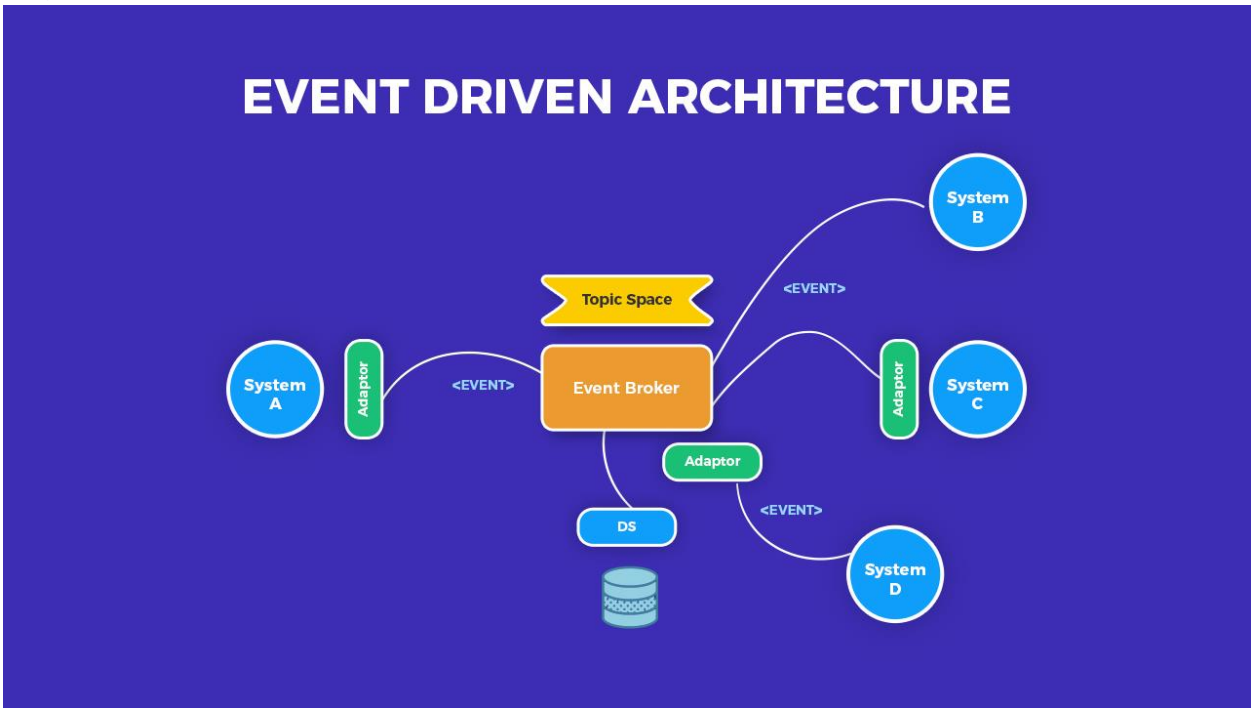
- Scalability: Allows for asynchronous and distributed processing, enabling scaling of individual components independently.
- Flexibility and Responsiveness: Enables decoupling of components, facilitating rapid development and adaptation to changing requirements.

- Real-time Processing: Supports real-time processing of events, enabling timely responses to business events and reducing latency.
- Loose Coupling: Promotes loose coupling between components, making the system more resilient to changes and failures.
- Integration with Legacy Systems: Facilitates integration with legacy systems and third-party services through event-driven communication patterns.

Most important Cons

- Complexity: Implementing and managing event-driven systems can introduce complexity in development, deployment, and maintenance due to asynchronous communication and event handling.
- Overhead: Event-driven systems may introduce additional overhead in terms of event processing, message queues, and infrastructure, potentially impacting performance and resource utilization.
- Synchronization Challenges: Coordinating and synchronizing events across distributed components can be challenging, leading to potential race conditions or data inconsistencies
- Debugging and Testing Difficulty: Debugging and testing event-driven systems can be more complex compared to traditional synchronous systems, requiring specialized tools and techniques.
- Learning Curve: Event-driven architecture introduces new concepts and paradigms, requiring developers to learn and adapt to event-driven patterns and practices.

Diagram



Layered Architecture

Layered architecture involves organizing the system into multiple layers, where each layer represents a different level of abstraction or functionality. Typically, layers include presentation, business logic, and data access. This architecture promotes separation of concerns, modularity, and maintainability.

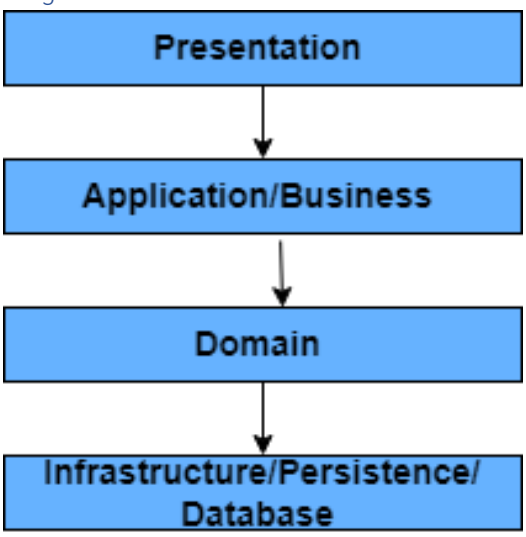
Most important Pros

- Modularity: Promotes a modular design, enhancing maintainability and scalability.
- Separation of Concerns: Facilitates clear separation of different components, improving code organization and readability.
- Flexibility: Enables independent development and modification of individual layers, supporting agile development practices.

Most important Cons

- Tight Coupling: Layered architectures can lead to tight coupling between layers, making it difficult to change one layer without affecting others.
- Performance Overhead: Passing data through multiple layers can introduce performance overhead, especially if layers are not optimized.
- Complexity: Managing interactions and dependencies between layers can become complex, especially as the system grows in size and complexity.

Diagram



Peer-to-Peer Architecture

P2P architecture allows for decentralized communication between nodes or peers in a network. Peers can act as both clients and servers, sharing resources and collaborating directly with each other. P2P architectures are often used for file sharing, distributed computing, or decentralized systems.

Most important Pros

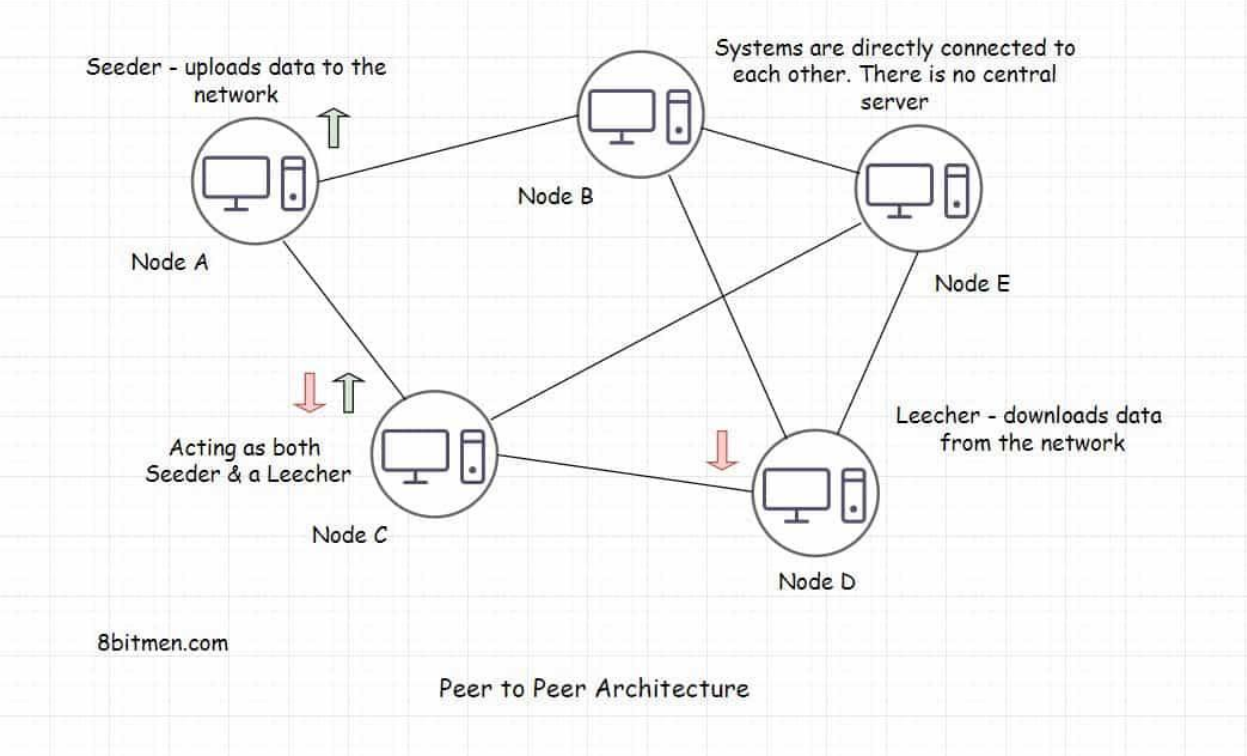
- Decentralization: P2P architectures distribute data and processing across multiple nodes, reducing reliance on centralized servers and improving fault tolerance and scalability.
- Redundancy: With data distributed across multiple nodes, P2P architectures inherently provide redundancy, ensuring data availability even if some nodes fail or leave the network.

- Scalability: P2P architectures can scale effectively by adding more nodes to the network, distributing the load and accommodating growth without the need for centralized infrastructure upgrades.
- Resilience: P2P networks are resilient to single points of failure, as each node can act independently and contribute to the network's functionality.
- Privacy and Security: P2P architectures can offer improved privacy and security by eliminating the need for centralized data storage and reducing the risk of data breaches or unauthorized access.

Most important Cons

- Security Concerns: P2P networks can be vulnerable to security threats such as malicious attacks, unauthorized access, and data manipulation due to the decentralized nature of communication.
- Reliability and Consistency: Ensuring data consistency and reliability in a P2P network can be challenging, especially when dealing with unreliable or malicious nodes.
- Performance Limitations: P2P architectures may suffer from performance limitations, particularly in large-scale networks or when handling large volumes of data, due to the decentralized nature of communication and potential network congestion.
- Complexity: Implementing and managing a P2P network can be complex, requiring sophisticated protocols and algorithms to handle node discovery, routing, data replication, and synchronization.
- Regulatory Compliance: P2P architectures may face regulatory challenges, particularly in industries with strict data privacy and compliance requirements, as ensuring regulatory compliance can be more complex in decentralized networks.

Diagram



Messaging Architecture

This architecture utilizes message queues or event streams to enable asynchronous communication and decoupling between components. Messages or events are sent between different components to trigger actions or exchange information. It enables scalability, fault tolerance, and loose coupling.

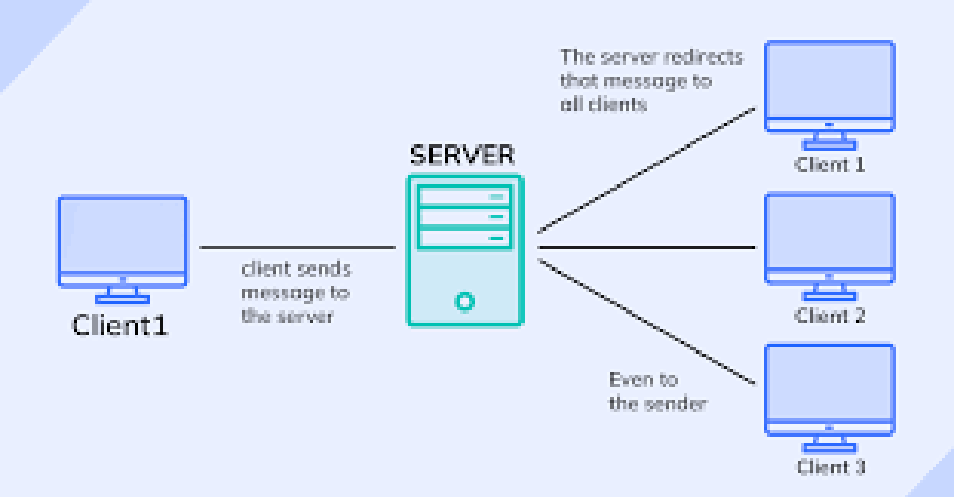
Most important Pros

- Scalability: Messaging architectures allow for horizontal scalability, enabling your application to handle increasing loads by distributing work across multiple message queues and consumers.
- Decoupling: Messaging decouples the components of your application, reducing dependencies and allowing them to evolve independently. This makes your system more resilient to changes and promotes modularity.
- Asynchronous Communication: Messaging enables asynchronous communication between components, improving responsiveness and reducing latency. This is particularly beneficial for long-running or resource-intensive tasks.
- Reliability: Messaging systems often provide features such as message persistence, delivery guarantees, and error handling mechanisms, ensuring reliable message delivery even in the presence of failures.
- Flexibility: Messaging architectures are flexible and can support various communication patterns, including publish-subscribe, point-to-point, and request-reply, allowing you to choose the best pattern for your specific use case.

Most important Cons

- Complexity: Implementing and managing a messaging architecture can introduce complexity, particularly in terms of message routing, handling, and ensuring message delivery guarantees.
- Overhead: Messaging systems may introduce additional overhead, both in terms of resource consumption and operational complexity, which can impact performance and scalability.
- Latency: Asynchronous messaging can introduce latency, especially when dealing with high message volumes or when relying on external message brokers or queues.
- Single Point of Failure: If your messaging infrastructure, such as the message broker or queue, becomes a single point of failure, it can adversely affect the reliability and availability of your entire system.
- Learning Curve: Adopting a messaging architecture may require your team to learn new concepts and technologies, which can increase ramp-up time and training costs.

Diagram


The server redirects
that message to
all clients
SERVER
Client 1
Client 2
Client 3
Client1
client sends
message to
the server
Even to
the sender

Conclusion

For my project I will be using a Microservice architecture combined with a Layered architecture below you can find my conclusion on each architecture for why this is or is not a good choice for my project.

- Monolithic architecture: A monolithic architecture is not a smart architectural choice for enterprise software which should be scalable and this is one of the biggest downsides of a monolithic architecture.

- Client-Server architecture: Client server architecture seems like a very good choice for my enterprise software project, the main problem is if one server fails the whole system goes down which is not a very secure design.

- Service oriented architecture: I will not be using Service Oriented Architecture because this is used to reuse the services that you create in other applications and the scale of these services are on the larger side.

- Microservice architecture: Microservice architecture seems like an excellent choice for my application I can divide my application in small services for single functionalities and have the security benefits of the system not failing when one service goes down. This architecture also aligns very well with my learning outcomes.

- Event Driven architecture: Event driven architecture is not a good choice for my application as I am creating a streaming platform which needs to function as much in sync as possible so having a architecture that works asynchronous will slow this process down a lot and cause data inconsistency.

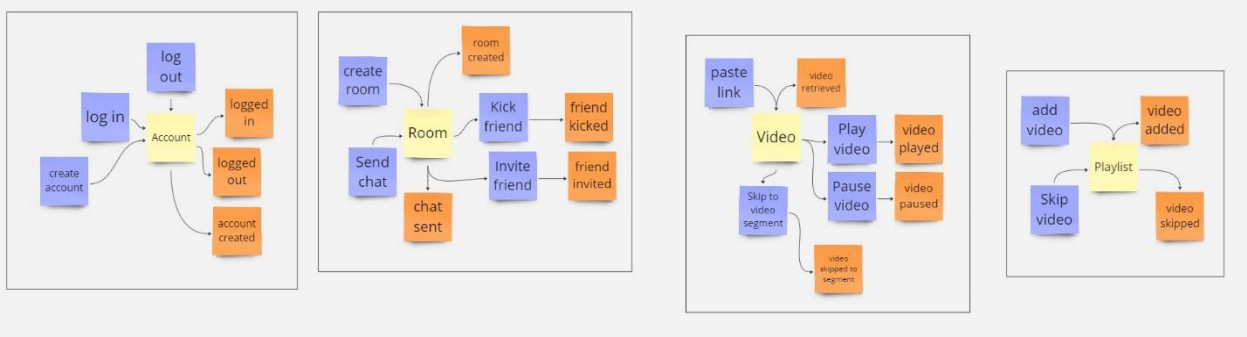
- Layered architecture: I will probably implement layered architecture inside of my individual services so that is well organized so that each layer offers a different function.

- Peer to Peer architecture: A peer to peer architecture is not a good fit for my enterprise software as It will have very slow performance when is use if for my streaming service.

- Message Architecture: Message Queuing architecture is not a good choice for my application as it is a asynchronous communication protocol which will give my application a lot of latency and I do not want that for a streaming platform.

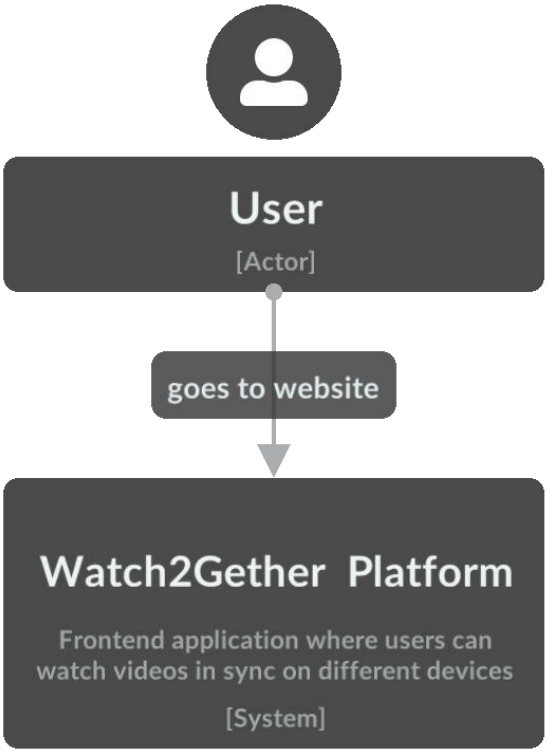
IT-architecture Sketching

I did an event storming session for this project to get a separation for my functionalities. Out of these groups I can make a service each for my microservice architecture, this way the code is divided well and every service is responsible for its own part.



C1 Diagram

In this level diagram you can see what external actors have access to my Application, The only actors to have access to my application are regular users who can create an account and watch videos with their friends. I will not have an administrative side on the project.



C2 Diagram

In this diagram you can see how my application functions, it contains a frontend that has a WebSocket connection to an API gateway which has a WebSocket connection to each microservice. Each microservice in my application has its own database. the choice for this design was made for scalability purposes.