Online - Meeting platform

Team 10 : Event Drive Engineers - Members:

1) Susheel Krishna - 2022101006
2) Srisai Krishna - 2022101115
3) Sreeja - 2022101081
4) Vishnu Sai - 2023121009
5) Ramandeep - 2021101050

Components Overview:

We have classified our system into four main components :

- User management :
    - It authenticates the user with user credentials and maintains a single sign on feature. Also helps in managing user content such as personal details. In future, we can store additional details related to the user such as friends list, followers, user dedicated profiles, user's usual settings etc.
- Meeting Management :
    - Manages meeting metadata, stores meeting participants and room allocated to them in real time. In the future, we can store additional meeting related data such as chats, participants' accesses, useful links, and transcriptions of the meeting. We can also integrate AI for generating summaries.
- Room Management :
    - Handles groups. Such that we can maintain rooms for each group. This will help in privacy and also helps in better communication and notifies each participant in the room if something happens. This was also helpful in managing a huge number of rooms which helps in conducting concurrent meetings.
- Media Management :
    - This manages media access for the meeting. Users can change the media settings such as video resolution, video frame rates etc. this helps in customization. In the future, we can add filterings (such as changing voice modulations or removing background noise etc. and also adding avatars in videos, changing video background etc.)

Components & Interactions

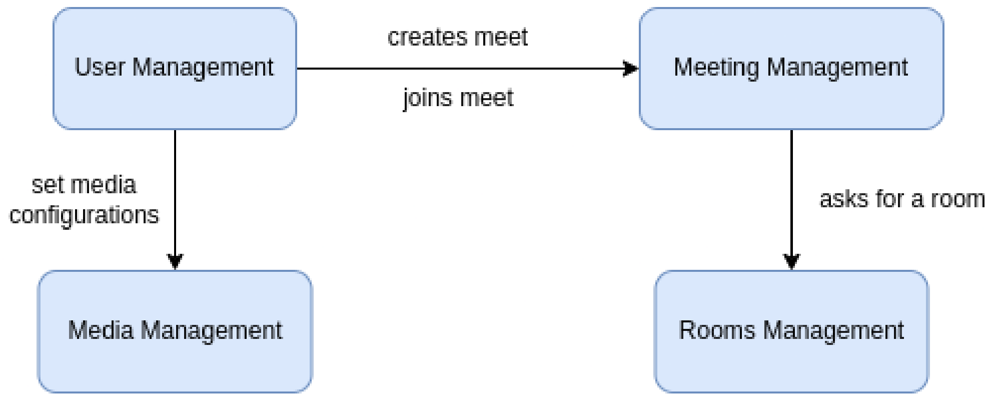


Fig-1 : Describes components interactions

Functional Requirements:

1. User Authentication : The system must allow users to register, log in, and manage their accounts globally. And provides access to each service in future.
2. Meeting Scheduling : Users should be able to create, join, and manage online meetings.
3. Real-time Communication : Enable real-time video and audio communication using WebRTC.
4. Room Management: manage meeting rooms based on the servers availability.
5. Participant Tracking: Track the number of participants in each meeting.

Non-functional Requirements:

1. Scalability: The system should handle a large number of concurrent meetings with peer to peer connection only. Like 10000 group meetings at a time.
2. Performance: Ensure low latency for real-time communication. At most the latency should be 200ms (only during media streaming in case of 2 members. And 300ms in case of 4 to 5 members).
3. Security: make sure that anonymous users won't join.
4. Usability: Provide an intuitive and user-friendly interface.
5. Availability : assuming hosting is done in google cloud run : >= 99.5% [src]

Here, in our project, Performance (Latency) and Scalability heavily depends on the architecture we are choosing. Because in our case, user will be always connected to the signalling server. So the signalling server may get overhead. Thus if we combine all the systems or keep its relation with the other systems, then latency still increases. That's why we separated all other servers from the signalling server. Also in the future, if one signalling server fails then we need to know and shift the users to another server. And also we should keep tracking the available signalling servers and rooms available in them. So we created rooms manager. And we are having a common database between the signalling server and the rooms manager. We give update access to signalling server to the interface and read only access to the meeting server through the interface. This interface is maintained by the rooms_manager server. By doing so, we can increase the performance and also we can scale the signalling servers in future.which would help in handling large number of users.

<u>Stakeholder Identification:</u>

1. End Users:

- **Description**: These are the primary users of the online meeting platform, including individuals and organizations who rely on the system for communication and collaboration.
- **Concerns**:
  - **Usability**: The platform must be intuitive and easy to use, with minimal learning curve. They should not be aware with room allocation and peer connection sharing [Hiding implementation Details]
  - **Performance**: Real-time communication should have minimal latency and high-quality audio/video.
  - **Availability**: We should ensure that users can use our application at any point of time.

2. System Administrators:

- **Description**: Responsible for deploying, maintaining, and monitoring the platform's infrastructure.
- **Concerns**:
  - **Scalability**: The system should handle increasing user loads without performance degradation.
  - **Ease of Monitoring**: Administrators need tools to monitor system health and performance. At Least expect better logging.
  - **Easy to Deploy**: Should not face errors of dependencies and versions.

3. Developers

- **Description**: Engineers responsible for building and maintaining the platform.
- **Concerns**:
    - **Code Maintainability**: The codebase should be modular and well-documented and readable.
    - Divide subsystems into services. So that the developer can work individually
    - Use a popular and a stable stack so that it is easy for developers to learn it and work with it.

4. Business Stakeholders

- **Description**: Includes investors, product managers, and other decision-makers driving the platform's development.
- **Concerns**:
    - Return on Investment (ROI): The platform should generate revenue and justify development costs.
    - **Market Competitiveness:** The platform must offer features that differentiate it from competitors.
    - **Feature Roadmap:** Stakeholders need a clear plan for future development.
    - Use popular stack so that it is easy to Hire 😅.

<u>**Views Addressing Stakeholder Concerns :**</u>

1. User Experience View:

Ensure the platform is intuitive and user-friendly for end users.

- The `src/App.jsx` and `src/Meeting.jsx` files implement a responsive and clean UI (We hope so 😅) for creating and joining meetings.
- Modular React components in `src/Meeting.jsx` allow for easy testing and iteration of user workflows.

2. Operational View:

Provide tools and architecture for system administrators to maintain and monitor the platform.

- Deployment Architecture: Backend services like `server.cjs` and `rooms_manager.cjs` are modular, making them easy to deploy and scale. We can use docker and kubernetes further for easy deployment.
- Monitoring Tools: deployment service providers such as Google Cloud provide request monitoring along with that we are storing meetings metadata.

3. Development View:
Provide developers with a maintainable and clear codebase.

- Software Architecture: The codebase uses a modular structure (`server.cjs`, `rooms_manager.cjs`, `meet_server.cjs`) for scalability and maintainability.
- Coding Standards: The `eslint.config.js` enforces consistent coding practices.
- Debugging Tools: Console logs in backend files (`server.cjs`, `meet_server.cjs`) and React DevTools for frontend debugging.

4. Business View:
Purpose: Align the platform's development with business goals and market demands.

- Key Focus Areas:
    - Feature Roadmap: Modular design in `rooms_manager.cjs` and `meet_server.cjs` allows for easy addition of new features like advanced room management.
    - Market Competitiveness: WebRTC integration in `src/Meeting.jsx` ensures high-quality real-time communication.
    - Revenue Generation: The architecture supports future monetization features like subscription plans.

Architectural Tactics

1. Load Balancing:

- **Implementation**: The backend services (server.cjs, rooms_manager.cjs) are modular and can be deployed behind an Nginx load balancer or a cloud-based load balancer like Google Cloud Load Balancing to distribute traffic effectively. Also in case of signalling servers, we are implicitly handling load by allocating room based on throughput on the server.
    - Improved Scalability: Distributes traffic across multiple servers, ensuring the system can handle increased user load without performance degradation.
    - Enhanced Availability: Prevents a single point of failure by routing traffic to healthy servers, ensuring continuous service availability.
    - Performance Optimization: Reduces response times by balancing the load across servers with lower throughput.

2. Caching:

- **Implementation**: The rooms_manager.cjs file uses in-memory data structures (e.g., Map) to temporarily store room details and participant data. This can be extended with Redis for more robust caching. We can use caching in case of the room checking near the Meeting server.

  - Reduced Latency: Speeds up room availability checks by storing frequently accessed data in memory or Redis, reducing database queries.
  - Improved Scalability: Reduces backend load by serving cached data, allowing the system to handle more concurrent users.
  - Cost Efficiency: Minimizes database usage, lowering operational costs for high-traffic scenarios.

3. Authentication and Authorization:

- **Implementation**: Firebase Authentication is integrated in src/fire_auth.jsx to handle secure user authentication. JWT can be used for session management to ensure only authorized users access the platform.
  - Security: Ensures only authenticated and authorized users can access the platform, protecting sensitive data and resources.
  - Compliance: Meets security standards and regulations (e.g., GDPR, HIPAA) by implementing secure authentication mechanisms.

4. Geographically Distributed Servers:

- **Implementation**: The modular backend services (server.cjs, meet_server.cjs) are designed to be deployed in multiple regions using Google Cloud's infrastructure, reducing latency for global users.
  - Reduced Latency: Deploying servers closer to users minimizes network delays, improving the user experience for global participants.
  - Fault Tolerance: Ensures service continuity by routing traffic to other regions in case of regional server failures.
  - Scalability: Supports a growing global user base by distributing the load across multiple regions.

5. Service Isolation (Microservices):

- **Implementation**: The system is divided into independent services:
- **Signaling**: meet_server.cjs handles WebRTC signaling.
- **Room Management**: rooms_manager.cjs manages room allocation and availability.
- **User Management**: src/authentication.jsx and src/fire_auth.jsx handle user authentication and profile management.

Architectural Decision Records:

ADR 1: Use of WebRTC for Peer-to-Peer Communication

Status: Accepted
Date: 2025-04-16

Context

We needed a way to enable real-time video and audio communication between users in a meeting. This required low latency, direct peer-to-peer communication, and the ability to handle NAT traversal.

Decision

We decided to use **WebRTC** for establishing peer-to-peer media streams between clients. WebRTC is an open standard supported by all major browsers and is optimized for real-time media exchange. The signaling process is handled by a custom **Signaling Server** via **WebSocket**.

Consequences

- Low-latency, high-quality audio/video transmission
- Native support in modern browsers (no plugins needed)
- Requires a signaling layer for peer discovery
- Additional work required to handle ICE candidates, NAT traversal, and fallback mechanisms

ADR 2: Separate Signaling Server for WebRTC

Status: Accepted
Date: 2025-04-17

Context

To establish WebRTC connections, clients need to exchange signaling data such as session descriptions and ICE candidates. This is not handled by WebRTC itself.

Decision

We implemented a dedicated **Signaling Server** using **Socket.IO** running on Node.js. This server is responsible for managing WebSocket connections and relaying signaling messages (e.g., offer, answer, candidate) between peers.

Consequences

- Clean separation of concerns (media handled by clients, signaling by server)
- Scalability via standalone WebSocket server
- Must ensure the signaling server remains synchronized and stateless
- Increases complexity due to an additional backend service

ADR 3: Microservices Architecture

Status: Accepted
Date: 2025-04-17

Context

The application has multiple responsibilities — frontend, user authentication, meeting scheduling, signaling, and room management — which can grow in complexity over time.

Decision

We adopted a **microservices architecture**, dividing the backend into:

- **Signaling Server**
- **Meeting Server**
- **Room Manager**

Each service runs independently, communicates via REST or WebSocket, and is focused on a single responsibility.

Consequences

- Improved modularity, scalability, and maintainability
- Easier to debug and test individual services
- Higher operational overhead (multiple services to run and monitor)
- Requires coordination between services and consistent interfaces

ADR 4: Use of React with Vite for Frontend Development

Status: Accepted
Date: 2025-04-14

Context

We required a modern frontend framework that supports reactive UI development, rapid prototyping, and tight integration with JavaScript. Build speed and developer experience were important criteria.

Decision

We selected **React** for building the user interface due to its component-based architecture and wide community support. To improve build speed and local development experience, we used **Vite** instead of traditional bundlers like Webpack.

Consequences

- Fast development server with hot module replacement (HMR)
- Easier state management and reusable components using React
- Optimized build output with Vite's esbuild integration
- Learning curve for state management and React lifecycle
- Must manage bundling compatibility when integrating third-party tools

ADR 5: Use of Dedicated Room Manager for Scalability

Status: Accepted
Date: 2025-04-18

Context

The application must support multiple concurrent meetings, track room availability, and handle dynamic session allocation. Without centralized control, this becomes difficult to scale and debug.

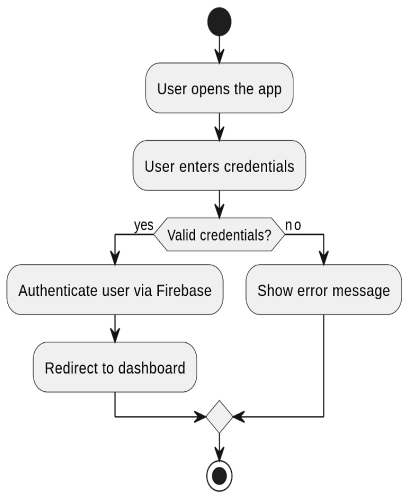
Decision

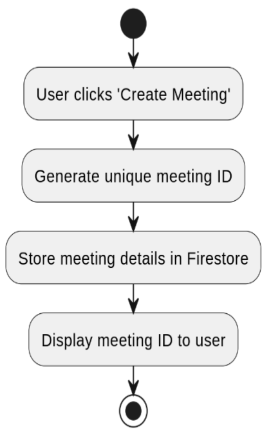
We created a **dedicated Room Manager** service that tracks available rooms, manages room assignments, and ensures meetings do not conflict. This service exposes REST endpoints to update and retrieve room availability.

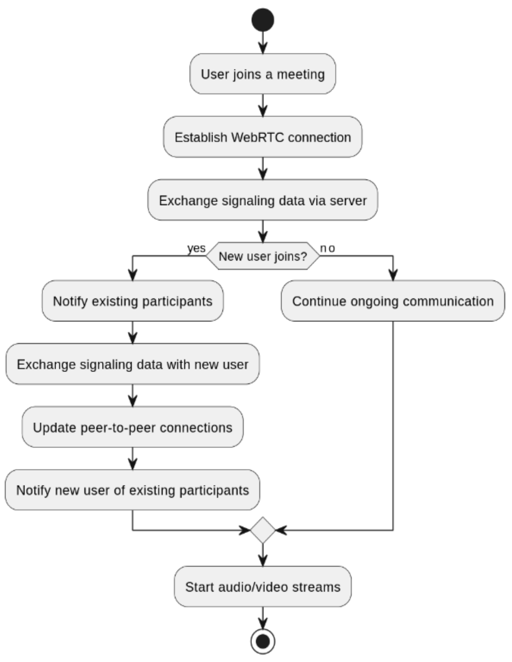
Consequences

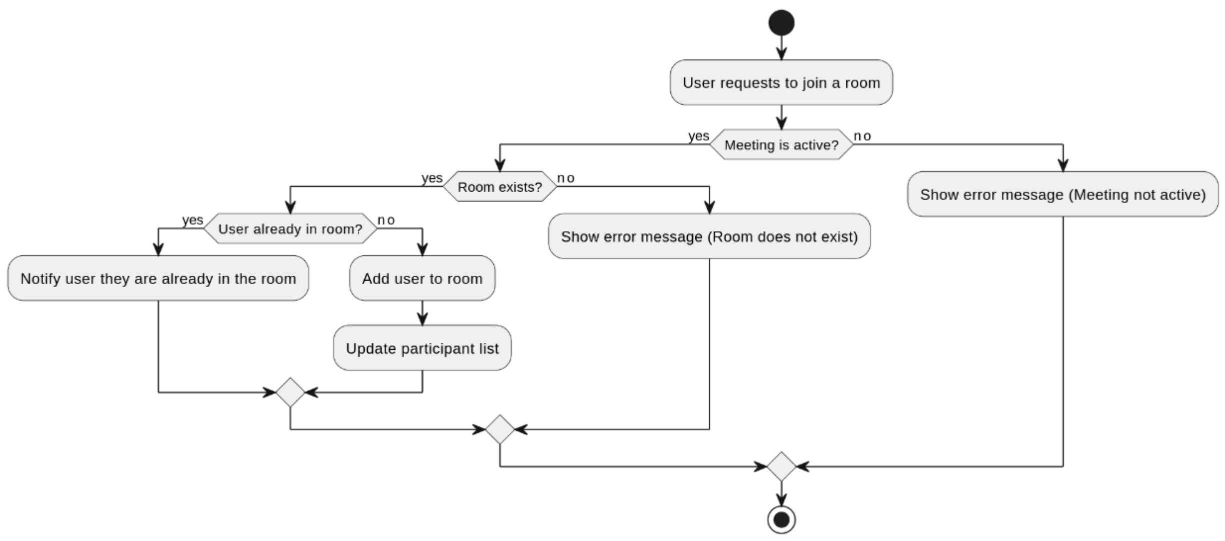
- Clear separation of room logic from meeting or signaling servers
- Easier to scale the room logic independently
- Simplifies room validation and access control
- Adds inter-service communication complexity
- Must handle synchronization to avoid race conditions in room allocation

System WorkFlows




User clicks 'Create Meeting'
Generate unique meeting ID
Store meeting details in Firestore
Display meeting ID to user

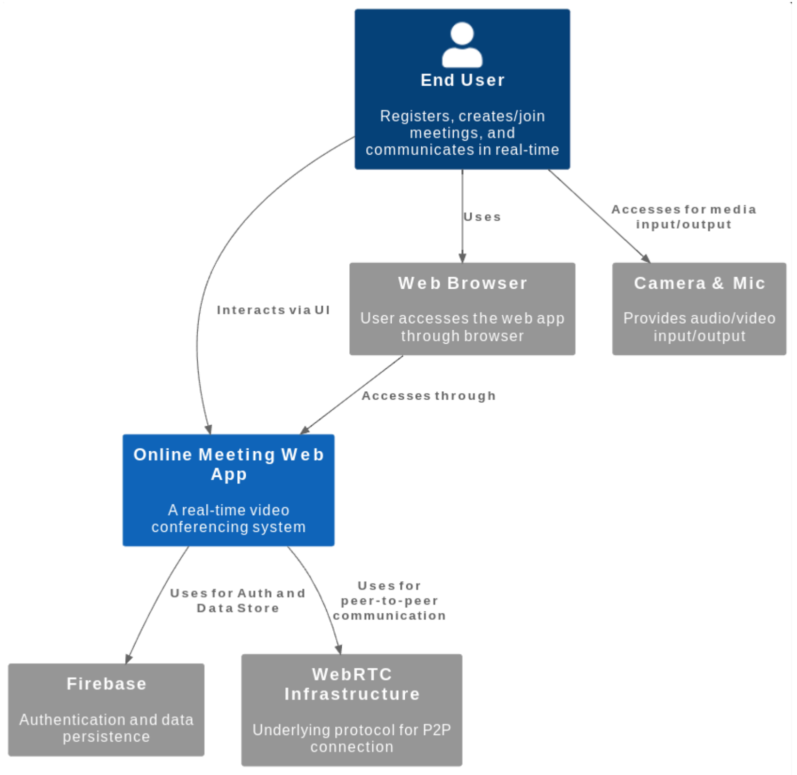

User joins a meeting
Establish WebRTC connection
Exchange signaling data via server
yes
New user joins?
no
Notify existing participants
Continue ongoing communication
Exchange signaling data with new user
Update peer-to-peer connections
Notify new user of existing participants
Start audio/video streams



User requests to join a room
yes
Meeting is active?
no
yes
Room exists?
no
Show error message (Meeting not active)
yes
User already in room?
no
Show error message (Room does not exist)
Notify user they are already in the room
Add user to room
Update participant list

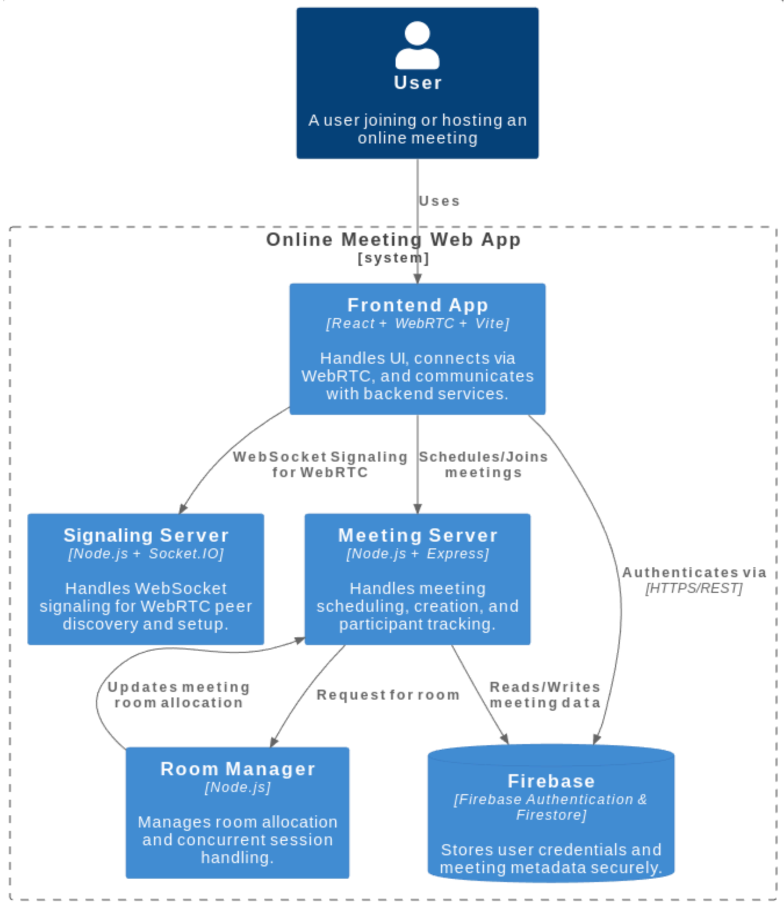
C4 model

1. Context level

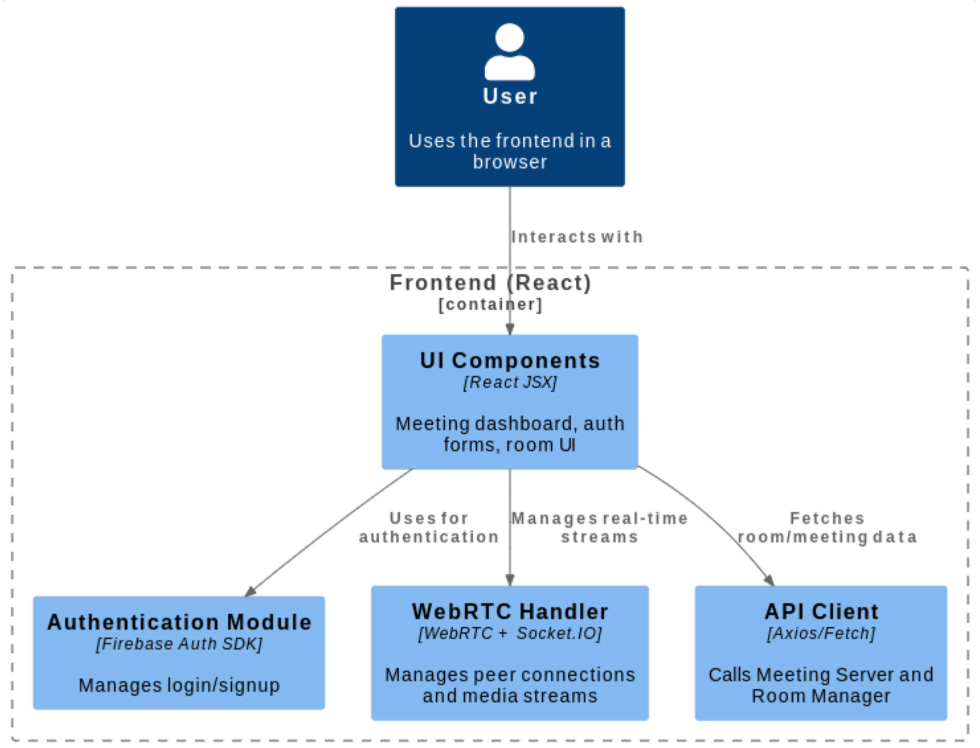


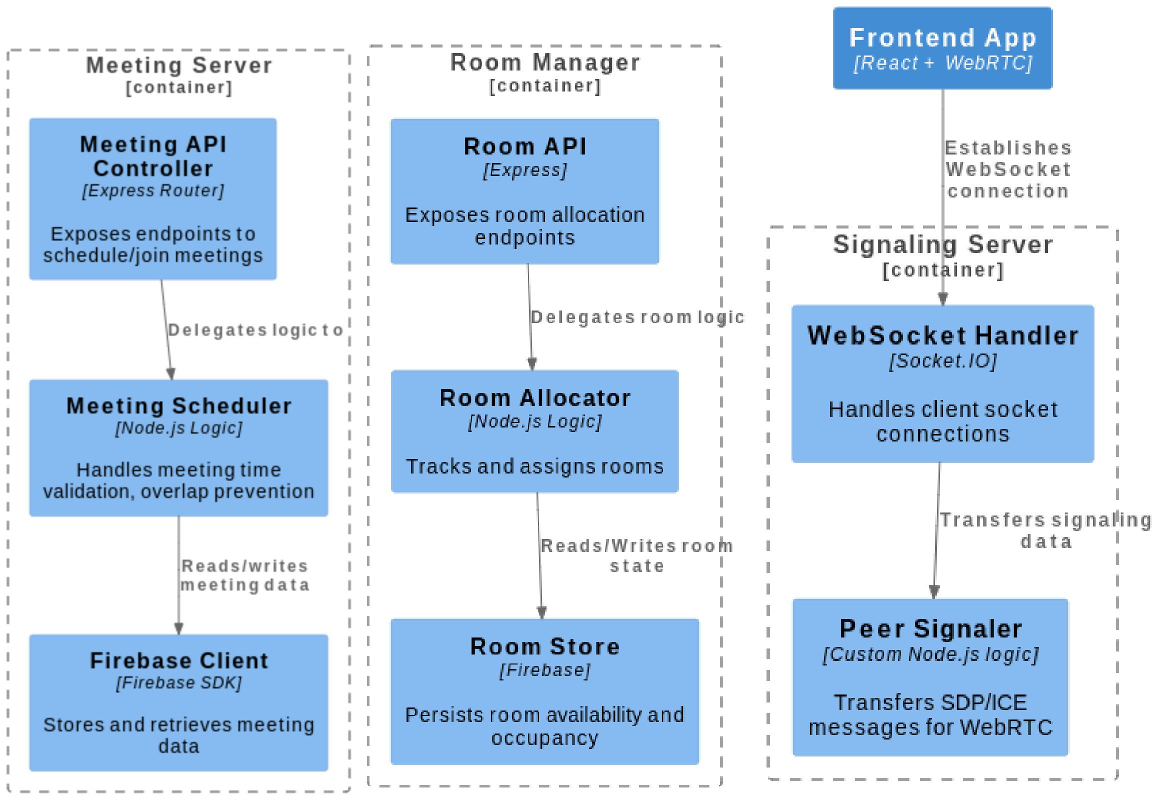
End User
Registers, creates/join meetings, and communicates in real-time
Uses
Accesses for media input/output
Interacts via UI
Web Browser
User accesses the web app through browser
Camera & Mic
Provides audio/video input/output
Accesses through
Online Meeting Web App
A real-time video conferencing system
Uses for Auth and Data Store
Uses for peer-to-peer communication
Firebase
Authentication and data persistence
WebRTC Infrastructure
Underlying protocol for P2P connection

2. Container level


User
A user joining or hosting an online meeting
Uses
Online Meeting Web App
[system]
Frontend App
[React + WebRTC + Vite]
Handles UI, connects via WebRTC, and communicates with backend services.
WebSocket Signaling for WebRTC
Schedules/Joins meetings
Authenticates via
[HTTPS/REST]
Signaling Server
[Node.js + Socket.IO]
Handles WebSocket signaling for WebRTC peer discovery and setup.
Meeting Server
[Node.js + Express]
Handles meeting scheduling, creation, and participant tracking.
Updates meeting room allocation
Request for room
Reads/Writes meeting data
Room Manager
[Node.js]
Manages room allocation and concurrent session handling.
Firebase
[Firebase Authentication & Firestore]
Stores user credentials and meeting metadata securely.

3. Component level





Design Patterns that can be implemented in the microservices :

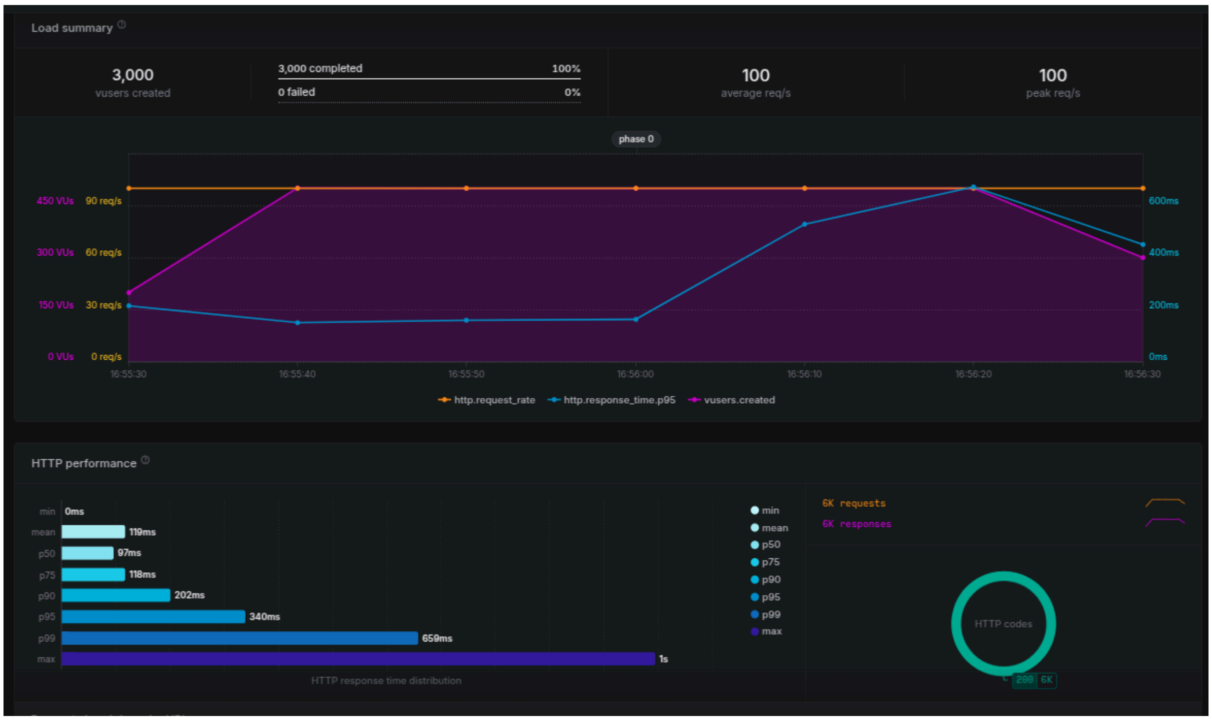
1. Gateway Pattern (API Gateway)
   - Role: Acts as a single entry point for all client requests, routing them to the appropriate microservices.
   - Where to Apply:
     - Introduce an API Gateway to handle requests for the Meeting Server, Room Manager, and Signaling Server.
     - This can centralize authentication, rate limiting, and request routing.
   - Benefits:
     - Simplifies client interactions.
     - Decouples clients from individual microservices.
     - Enables cross-cutting concerns like logging and monitoring.
2. Observer Pattern
   - Role: Enables real-time communication between components by notifying subscribers of events.
   - Where to Apply:
     - Already used in the Signaling Server (server.cjs) with WebSocket events like join-room, signal, and peer-disconnected.
     - Extend this pattern to notify the Room Manager when room statuses change.
   - Benefits:
     - Facilitates real-time updates.
     - Decouples event producers from consumers.

3. Repository Pattern
    - Role: Abstracts database operations, providing a clean interface for data access.
    - Where to Apply:
        - Already partially implemented in firestore_Database.cjs with functions like add_Data, get_Data, and update_Data.
        - Extend this pattern to encapsulate all database interactions for better testability and separation of concerns.
    - Benefits:
        - Simplifies database access logic.
        - Makes the codebase more testable and maintainable.

4. Singleton Pattern
    - Role: Ensures a single instance of a class or resource is created and shared.
    - Where to Apply:
        - Use for shared resources like the Firebase App initialization in firestore_Database.cjs and fire_auth.jsx.
        - Apply to the Redis Cache (if introduced) to ensure a single connection pool.
    - Benefits:
        - Reduces resource overhead.
        - Ensures consistent state across the application.

5. Rate Limiter Pattern
    - Role: Controls the rate of requests to a service.
    - Where to Apply:
        - Use in the Signaling Server to limit WebSocket connections or signaling messages.
        - Apply to the Meeting Server to prevent abuse of APIs like /scheduleMeeting.
    - Benefits:
        - Protects services from being overwhelmed.
        - Ensures fair usage of resources.

Architecture Analysis:



Comparison of Present Architecture with monolithic :

Scalability

- Microservices:
  - Advantage: Each service (e.g., rooms_manager.cjs, meet_server.cjs, server.cjs) can be scaled independently based on its load. For example, if the signaling server (server.cjs) experiences high traffic, you can scale it without affecting other services.
  - Consequence: Requires orchestration tools like Kubernetes or Docker Swarm to manage scaling, which adds complexity.
- Monolithic:
  - Advantage: Scaling is simpler because the entire application is deployed as a single unit.
  - Consequence: Scaling is less efficient. For example, if only the signaling logic is under heavy load, the entire application must be scaled, wasting resources.

Development and Deployment

- Microservices:
  - Advantage: Teams can work on different services independently. For example: The rooms_manager.cjs team can focus on room allocation. The meet_server.cjs team can handle meeting scheduling.
  - Consequence: Deployment is more complex. Each service must be deployed and managed separately, requiring CI/CD pipelines for each service.
- Monolithic:
  - Advantage: Easier to develop and deploy as a single unit. A single deployment pipeline suffices.
  - Consequence: Changes in one part of the application (e.g., room management) can affect unrelated parts, increasing the risk of bugs.

Fault Isolation

- Microservices:
    - Advantage: Faults in one service (e.g., rooms_manager.cjs) do not bring down the entire system. For example, if the room manager fails, the signaling server can still function.
    - Consequence: Requires robust inter-service communication and error handling (e.g., retries, circuit breakers).
- Monolithic:
    - Advantage: No inter-service communication issues since everything is in one process.
    - Consequence: A single bug (e.g., in room allocation) can crash the entire application.

Performance

- Microservices:
    - Advantage: Services can be optimized individually. For example, the signaling server can use WebSocket (server.cjs), while the meeting server uses REST APIs (meet_server.cjs).
    - Consequence: Inter-service communication (e.g., HTTP requests between services) adds latency. For example, meet_server.cjs calls rooms_manager.cjs to allocate rooms, which introduces network overhead.
- Monolithic:
    - Advantage: No inter-service communication overhead. All function calls are in-process, making them faster.
    - Consequence: As the application grows, performance may degrade due to a lack of modularity.

Maintainability

- Microservices:
    - Advantage: Easier to maintain and update individual services. For example, you can update firestore_Database.cjs without affecting the signaling server.
    - Consequence: Requires careful versioning and API contracts between services to avoid breaking changes.
- Monolithic:
    - Advantage: Simpler to maintain initially since all code is in one place.
    - Consequence: As the codebase grows, it becomes harder to maintain due to tight coupling between components.

Technology Stack

- Microservices:
    - Advantage: Each service can use the most suitable technology. For example: rooms_manager.cjs uses HTTP for room management. server.cjs uses WebSocket for signaling.
    - Consequence: Increases complexity in managing multiple technologies and dependencies.
- Monolithic:
    - Advantage: A single technology stack simplifies development and reduces the learning curve.
    - Consequence: Limits flexibility in choosing the best tools for specific tasks.

Testing

- Microservices:

        - Advantage: Easier to test individual services in isolation. For example, you can test rooms_manager.cjs independently using tools like Artillery (artillery-config-rooms_manager.yml).
        - Consequence: Integration testing becomes more complex due to inter-service dependencies.
    - Monolithic:
        - Advantage: Easier to perform end-to-end testing since all components are in one application.
        - Consequence: Unit testing individual components is harder due to tight coupling.

Security

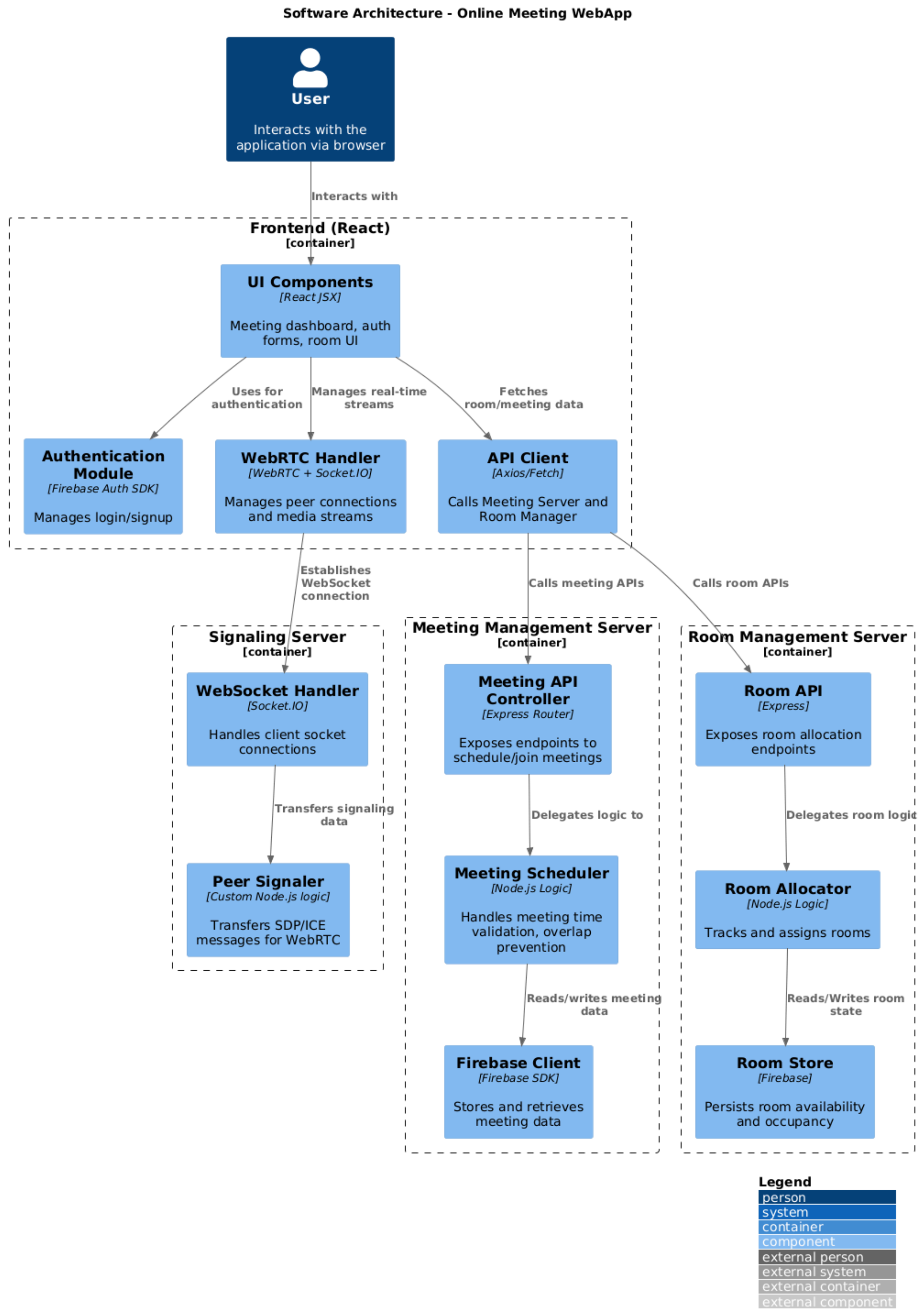
- Microservices:
    - Advantage: Each service can have its own security policies. For example: firestore_Database.cjs can restrict access to sensitive data. server.cjs can enforce WebSocket authentication.
    - Consequence: Requires securing inter-service communication (e.g., using HTTPS, tokens).
- Monolithic:
    - Advantage: Security is centralized, making it easier to manage.
    - Consequence: A single vulnerability can compromise the entire application.

Latency would increase by 3 times for each operation. Because in our case, we have separated servers. So while one server performs, the other server can parallelly process other tasks. Thus Latency with microservices is on average 90ms. But it would become 270ms.

Similarly, if we consider users handling, scalability might be a concern. Presently our system is handling 3000 users parallelly. But when you consider it monolithic, you need huge processing to handle the same number of users.

Source code - repository link: Check it out

Final Architecture Diagram for our project:



Thank You