ProxioMeet Software Documentation

Team Members:

- Chandan Shrivastava
- Nachiket Patil
- Abhishek Sharma
- Praddyumn Shukla
- Arya Marda

Task 1: Requirements and Subsystem

Functional Requirements

1. **User Registration and Authentication:**
    - Users should be able to create accounts securely.
    - Authentication mechanisms ensure user data privacy and security.
2. **Profile Creation and Management:**
    - Users can create and manage profiles, including interests and preferences.
    - Profile information is editable to reflect changing preferences.
3. **Interest Matching Algorithm:**
    - An algorithm matches users based on shared interests and proximity.
    - Real-time matching facilitates spontaneous interactions.
4. **Location-Based Services:**

- Location services identify users' proximity and suggest potential matches nearby.
- Accurate and updated location data ensures effective matchmaking.

Non-Functional Requirements

1. **Scalable:**
    - The system should handle a growing user base and increasing demand for real-time matchmaking.
    - Scalability is crucial to accommodate fluctuating usage patterns.
2. **Reliable:**
    - Minimal downtime and robust error handling mechanisms.
    - Users can depend on the application for spontaneous interactions.
3. **Performance:**
    - Responsive system providing quick matches and communication responses.
    - Minimized response times for matchmaking and messaging.
4. **Security:**
    - User data encryption and protection against unauthorized access or breaches.
    - Authentication and authorization mechanisms prevent identity theft and ensure data integrity.

Architectural Analysis

Microservices Architecture

- **Architecture Description:** Each functionality is implemented as a separate microservice.
- **Advantages:** Scalability, modularity, fault isolation.
- **Challenges:** Network overhead, complexity in management.

Monolithic Architecture

- **Architecture Description:** All functionalities integrated into a single application.
- **Advantages:** Simplified communication, ease of development.
- **Challenges:** Scalability, maintenance burden.

Trade-offs

- **Scalability:** Microservices offer superior scalability but introduce complexity.
- **Latency and Response Time:** Monolithic architectures have lower latency but lack flexibility.

Architectural Significance:

- **Real-Time Communication**: The ability to match users based on interests and proximity in real-time is architecturally significant as it requires efficient algorithms, data structures, and communication protocols to support instant matchmaking and communication.
- **Location-Based Services**: Integrating location-based services is crucial for accurately identifying nearby users and suggesting relevant matches and activities. Architectural considerations include data privacy, accuracy, and integration with external APIs or services.
- **Scalability and Performance**: Architectural decisions related to scalability and performance are critical to handle the dynamic nature of user interactions and ensure a smooth user experience even under high load conditions.

Subsystem Overview:

1. **User Management Subsystem**:
    - Responsible for user registration, authentication, and profile management functionalities.
    - Ensures user data privacy and security through robust authentication mechanisms and encryption techniques.

2. **Matching Algorithm Subsystem**:
    - Implements the interest matching algorithm to identify potential matches based on shared interests and proximity.
    - The matching algorithm uses the text given by the user, it is given as a prompt to the AI model to find major interests from the text. These interests are used to calculate the matching percentage between users.
    - It also utilized location (longitude and latitude) from location API to find proximum distanced people.

Task 2: Architecture Framework

Stakeholder Identifications

- Users (age 20-40): These are the primary users who will post events and schedule meets.
- Administrators: People responsible for managing the platform, ensuring its functionality, security, and overall performance.
- Developers: Those responsible for building and maintaining the software platform.

2. Concerns Identification:

- **User Concerns**:
    - Ease of posting events and scheduling meets.
    - Visibility and discoverability of events.
    - Security and privacy of user information.
    - Usability and intuitiveness of the platform.
    - Accessibility for users with diverse needs.
- **Administrators Concerns**:
    - System reliability and uptime.
    - Security of user data and platform integrity.

    - Scalability to accommodate growing user base and event volume.
    - Unbiased matching algorithm for better feedback.
- **Developer Concerns**:
    - Maintainability and extensibility of the software.
    - Adherence to coding standards and best practices.
    - Integration with external services or APIs.

3. Viewpoints and Views Creation:

- **Define Viewpoints**:
    - Functional Viewpoint: Focus on the functionality related to event posting, scheduling, and user interaction.
    - Information Viewpoint: Address concerns related to data management, privacy, and security.
    - Usability Viewpoint: Consider the user experience, interface design, and accessibility.
- **Develop Views**:
    - Functional View: Use case diagrams, sequence diagrams, or activity diagrams to illustrate event posting and scheduling workflows.
    - Information View: Entity-relationship diagrams or data flow diagrams to depict data entities and their relationships.
    - Usability View: Wireframes or mockups showcasing the user interface design and navigation flow.
- **Tailor Views for Specific Stakeholders**:
    - Customize views to highlight aspects relevant to each stakeholder group. For example, emphasize security features in the Information View for Administrators.
- **Document Views**:
    - Document each view along with its rationale, assumptions, and dependencies. Provide context to facilitate understanding and decision-

making.

Major Design Decisions

Architecture Decision Records (ADRs) are a useful way to document major design decisions in a structured and consistent manner.

- **ADR-001: Choosing React Native, Express, TypeScript, and Node.js for Technology Stack**
    - Author: Chandan
    - Status: Accepted
    - Context : Our project requires selecting appropriate technologies for building a cross-platform mobile application with a scalable backend. After evaluating various technology options, we have narrowed down our choices to React Native for mobile app development, Express and Node.js for backend development, and TypeScript for enhanced type safety.
    - Decision : We will use React Native for mobile app development, Express and Node.js for backend development, and TypeScript for type safety in our project.
    - Decision Drivers

      Cross-Platform Development: We need a technology stack that allows us to build a single codebase for both iOS and Android platforms.

      Performance & Scalability: The chosen technologies should be capable of handling a high volume of users and requests while maintaining performance.

      Developer Productivity: Utilizing frameworks and languages with strong developer ecosystems and tooling support enhances productivity and maintainability.

Type Safety: TypeScript provides static typing, reducing runtime errors and improving code quality and maintainability.

- Considerations

  React Native:
  React Native enables building cross-platform mobile applications using JavaScript and React. It allows for rapid development and a shared codebase across different platforms.

  Express & Node.js:
  Express is a lightweight and flexible web application framework for Node.js, suitable for building scalable backend services.
  Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine, known for its performance and scalability in handling asynchronous I/O operations.

  TypeScript:
  TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript.
  It offers type checking during development, improving code quality, and reducing runtime errors.

- Justification
  After considering the requirements and evaluating the options, we have decided on this technology stack for the following reasons:

  Cross-Platform Development: React Native allows us to develop a single codebase for both iOS and Android platforms, reducing development time and effort.

  Performance & Scalability: Node.js and Express provide a lightweight and scalable backend framework suitable for handling a high volume of users and requests.

  Developer Productivity: The combination of React Native, Express, Node.js, and TypeScript offers a robust development ecosystem with extensive libraries, tooling, and community support, enhancing developer productivity and maintainability.

    Type Safety: TypeScript's static typing helps catch errors during development and improves code maintainability, reducing bugs and runtime errors.
    Consequences

    Learning Curve: Team members may need some time to familiarize themselves with React Native, Express, Node.js, and TypeScript if they are not already proficient in these technologies.

    Integration Challenges: Integrating third-party libraries and services with React Native and Express may require additional effort due to compatibility and interoperability concerns.

  - References
    React Native -
    https://reactnative.dev/
    Express.js -
    https://expressjs.com/
    Node.js -
    https://nodejs.org/
    TypeScript -
    https://www.typescriptlang.org/

- **ADR-002: Implementing Microservices Architecture**
  - Author: Nachiket
  - Status: Accepted
  - Context
    Our project requires defining the architectural approach for designing and implementing the system. After careful consideration, we have decided to adopt a microservices architecture.

- Decision
  We will implement a microservices architecture for our system.
- Decision Drivers

  Scalability & Flexibility: Microservices enable horizontal scaling of individual components, allowing us to adapt to changing demands and handle a high volume of users and requests effectively.

  Fault Isolation & Resilience: Breaking down the system into loosely coupled services improves fault isolation, preventing the failure of one component from impacting the entire system.

  Continuous Delivery & Deployment: Microservices facilitate continuous delivery and deployment practices, enabling faster iterations and updates to specific components without disrupting the entire system.

  Technology Diversity: Adopting microservices allows us to use the most suitable technology stack for each service based on its requirements and constraints, promoting innovation and flexibility.
- Considerations

  Service Communication: Effective communication between microservices is crucial for maintaining system integrity and consistency. We need to carefully choose communication protocols and patterns such as RESTful APIs or message brokers.

  Service Discovery & Orchestration: Implementing mechanisms for service discovery and orchestration helps manage the dynamic nature of microservices and enables efficient routing and load balancing.

  Data Management: Microservices may require individual databases or shared data stores depending on data access patterns and consistency requirements. We need to design appropriate data management strategies.

- Justification
  After evaluating various architectural approaches, we have chosen a microservices architecture for the following reasons:

  Scalability & Flexibility: Microservices allow us to scale individual components independently, ensuring optimal resource utilization and responsiveness to varying workloads.

  Fault Isolation & Resilience: Isolating services reduces the blast radius of failures, improving system resilience and fault tolerance.

  Continuous Delivery & Deployment: Microservices enable faster delivery cycles and smoother deployment processes, facilitating agility and innovation.

  Technology Diversity: Microservices promote technology diversity, empowering teams to choose the most suitable tools and frameworks for each service, optimizing performance and maintainability.
- Consequences

  Increased Complexity: Managing a distributed system of microservices introduces additional complexity in terms of deployment, monitoring, and troubleshooting, requiring investment in robust DevOps practices and tooling.

  Service Communication Overhead: Inter-service communication introduces latency and network overhead, requiring careful design and optimization to maintain system performance.
- Approvals
  Other Team Members
- References
  Microservices Architecture: Patterns and Practices for Building Microservices-Based Applications - Chris Richardson
  Building Microservices - Sam Newman

- **ADR-003: Utilizing MongoDB for User Management and Firebase for Notifications**
    - Author: Abhishek,
    - Status: Accepted
    - Context
      Our project requires defining the data storage strategy for managing user data and handling notifications. After evaluating various options, we have decided to utilize MongoDB for user management and Firebase for notifications.
    - Decision
      We will use MongoDB for structured data storage for user management and Firebase for real-time notifications to users.
    - Decision Drivers

      Flexibility & Scalability: MongoDB provides flexibility in data modeling and scalability, making it suitable for managing user data with varying structures and evolving requirements.

      Real-Time Communication: Firebase offers real-time database and push notification services, enabling timely and efficient communication with users.

      Efficient Data Storage & Retrieval: Choosing database technologies based on specific data modeling and access patterns ensures efficient data storage and retrieval, optimizing performance and scalability.
      Considerations

      Data Consistency: Coordinating data updates between MongoDB and Firebase to maintain consistency may introduce complexity and overhead, requiring careful synchronization mechanisms.

      Data Security: Implementing appropriate security measures and access controls to protect user data in MongoDB and Firebase is crucial to ensure confidentiality and integrity.

  - Justification
    After analyzing the requirements and considering the strengths of MongoDB and Firebase, we have chosen this data storage strategy for the following reasons:

    Flexibility & Scalability: MongoDB's document-oriented data model and horizontal scalability make it well-suited for managing user data with varying structures and scaling to accommodate growth.

    Real-Time Communication: Firebase's real-time database and push notification services provide seamless and efficient communication channels for delivering notifications to users instantly.

    Efficient Data Storage & Retrieval: By leveraging MongoDB's indexing and querying capabilities and Firebase's real-time synchronization, we can efficiently store and retrieve user data and notifications, optimizing system performance and responsiveness.

  - Consequences

    Integration Complexity: Integrating MongoDB and Firebase into the system may introduce integration complexities and overhead, requiring additional development effort and maintenance.

  - Approvals
    Other Team Mates

  - References
    MongoDB Documentation -
    https://docs.mongodb.com/
    Firebase Documentation -
    https://firebase.google.com/docs/

- **ADR-006: Designing a Responsive Web Application with a Mobile-First Approach**

  - Author: Arya and Pradyumn,

  - Status: Accepted

- Context
  Our project requires defining the user interface design strategy to deliver a seamless and consistent user experience across devices. After evaluating various options, we have decided to design a responsive web application with a mobile-first approach.
- Decision
  We will design a responsive web application with a mobile-first approach for our project.
- Decision Drivers

  User-Centric Design: Prioritizing mobile users aged 20-40 ensures a consistent and seamless user experience across devices, meeting the needs and preferences of our target audience.

  Responsive Design Principles: Utilizing responsive design principles and frameworks like Bootstrap or Material Design enables us to adapt the UI layout and components to different screen sizes and resolutions, improving accessibility and usability.

  Performance Optimization: Optimizing performance and usability for mobile devices by minimizing page load times and optimizing touch interactions enhances user satisfaction and engagement, driving adoption and retention.
- Considerations

  Cross-Device Compatibility: Ensuring cross-device compatibility and consistency requires thorough testing and validation across various devices, browsers, and screen sizes to identify and address potential issues and discrepancies.

  Resource Constraints: Designing a responsive web application with a mobile-first approach may introduce resource constraints and trade-offs in terms of design complexity, feature prioritization, and implementation effort.

Accessibility & Inclusivity: Incorporating accessibility best practices and guidelines ensures that the web application is usable by individuals with disabilities, complying with accessibility standards and regulations.

- Justification
  After analyzing the requirements and considering the needs and preferences of our target audience, we have chosen this approach for user interface design for the following reasons:

  User-Centric Design: Prioritizing mobile users aged 20-40 aligns with our target audience's preferences and behavior, ensuring a seamless and engaging user experience across devices.

  Responsive Design Principles: Utilizing responsive design principles and frameworks enables us to create a flexible and adaptable user interface that delivers consistent and optimal experiences across different devices and screen sizes.

  Performance Optimization: Optimizing performance and usability for mobile devices enhances user satisfaction and engagement, driving adoption and retention, and ultimately contributing to the success of the project.
  Consequences

  Design Complexity: Designing a responsive web application with a mobile-first approach may introduce additional complexity in terms of design iteration, feature prioritization, and implementation effort, requiring careful planning and coordination.
- Approvals
  Other Team Mates
- References
  Bootstrap -
  https://getbootstrap.com/
  Material Design - https://material.io/design

Task 3 : Architectural Tactics and Patterns

Architectural Tactics

1. **Availability Tactics :**

   Fault Detection

   - In our architecture, we implement fault detection using the heartbeat mechanism to ensure the availability of microservices.
   - We periodically send heartbeat signals from each microservice to a central monitoring system.
   - The monitoring system observes these heartbeats and detects any anomalies or failures if it stops receiving expected signals from a microservice.
   - By continuously checking the health of microservices, we can quickly identify and respond to faults, minimizing downtime and ensuring the overall availability of the system.

2. **Performance Tactics:**

   Maintain Copies (MongoDB):

   - Maintaining copies or replicas of data in MongoDB is a performance tactic that enhances fault tolerance and scalability.
   - By replicating data across multiple nodes, MongoDB ensures high availability and resilience to failures.
   - Read operations can be distributed across replicas, improving read throughput and reducing latency for read-heavy workloads.

3. **Security Tactics:**

   Authenticate Users:

   - Authentication is the process of verifying the identity of users before granting access to the system.
   - It typically involves validating credentials such as usernames and passwords, or using more advanced methods like biometric

authentication.

- Proper authentication helps prevent unauthorized access and protects sensitive data from malicious actors.
- Using Email OTP verification for user registration and using resend email if any fault happens in between.

Authorize Users:

- Authorization determines what actions users are allowed to perform within the system after they have been authenticated.
- It involves assigning appropriate permissions and privileges to users based on their roles and responsibilities.
- Effective authorization mechanisms enforce access controls and ensure that users can only access resources and perform actions that they are authorized to do.

Data Confidentiality:

- Data confidentiality is the protection of sensitive information from unauthorized access or disclosure.
- It involves encrypting data at rest and in transit, implementing access controls, and restricting permissions to authorized users.
- Measures such as data masking, encryption, and secure communication protocols help maintain confidentiality and prevent data breaches.

4. **Modifiability Tactics:**

Semantic Coherence:

- Semantic coherence involves organizing the system in a logical manner that reflects its domain and functionality.
- Components are grouped together based on their related concerns, promoting clarity and making it easier to locate and modify relevant parts of the system.
- This tactic enhances modifiability by reducing complexity and facilitating changes without unintended side effects.

5. **Usability Tactics:**

   Design Time (Separate UI from the rest of the system - MVC):

   - Separating the user interface (UI) from the rest of the system architecture using the Model-View-Controller (MVC) pattern is a usability tactic.
   - MVC separates the presentation layer (View) from the application logic (Controller) and data (Model), promoting modularity and maintainability.
   - This separation allows for independent development, testing, and modification of UI components, leading to a better user experience and faster iteration cycles.

   Run Time (Support User Initiative):

   - Supporting user initiative at runtime is a usability tactic that empowers users to interact with the system in a flexible and intuitive manner.
   - It involves providing features and functionalities that enable users to initiate actions, make decisions, and control their interactions with the system.
   - User-initiated actions enhance usability by allowing users to customize their experiences, navigate through the system more efficiently, and accomplish their tasks with greater satisfaction.

Design Patterns

1. Observer Pattern:

Role:

- The Observer pattern is used for implementing a publish-subscribe mechanism, where objects (subscribers or observers) are notified of changes or events in another object (subject or publisher).
- In your architecture, the Observer pattern can be used for notifications. Whenever there is a new post, scheduling of a meet, or connection request, interested users (observers) will be notified.

Explanation:

- The `Post` entity or the `MeetScheduler` component acts as the subject or publisher.
- Users interested in receiving notifications subscribe as observers to the subject.
- When a relevant event occurs (e.g., a new post is created), the subject notifies all subscribed observers.
- Observers receive the notification and can take appropriate actions, such as displaying the post or scheduling a meet.

2. Builder Pattern:

Role:

- The Builder pattern is used to construct complex objects step by step, allowing for the creation of different representations of an object using the same construction process.
- In your architecture, the Builder pattern can be used to create posts with varying attributes while ensuring a clear and concise construction process.

Explanation:

- The `PostBuilder` class encapsulates the construction logic for creating posts.
- Each method in the builder class (`setTitle`, `setDescription`, etc.) corresponds to setting a specific attribute of the post.
- The `build` method assembles the individual attributes into a `Post` object, ensuring that all necessary attributes are set before construction.

3. Strategy Pattern:

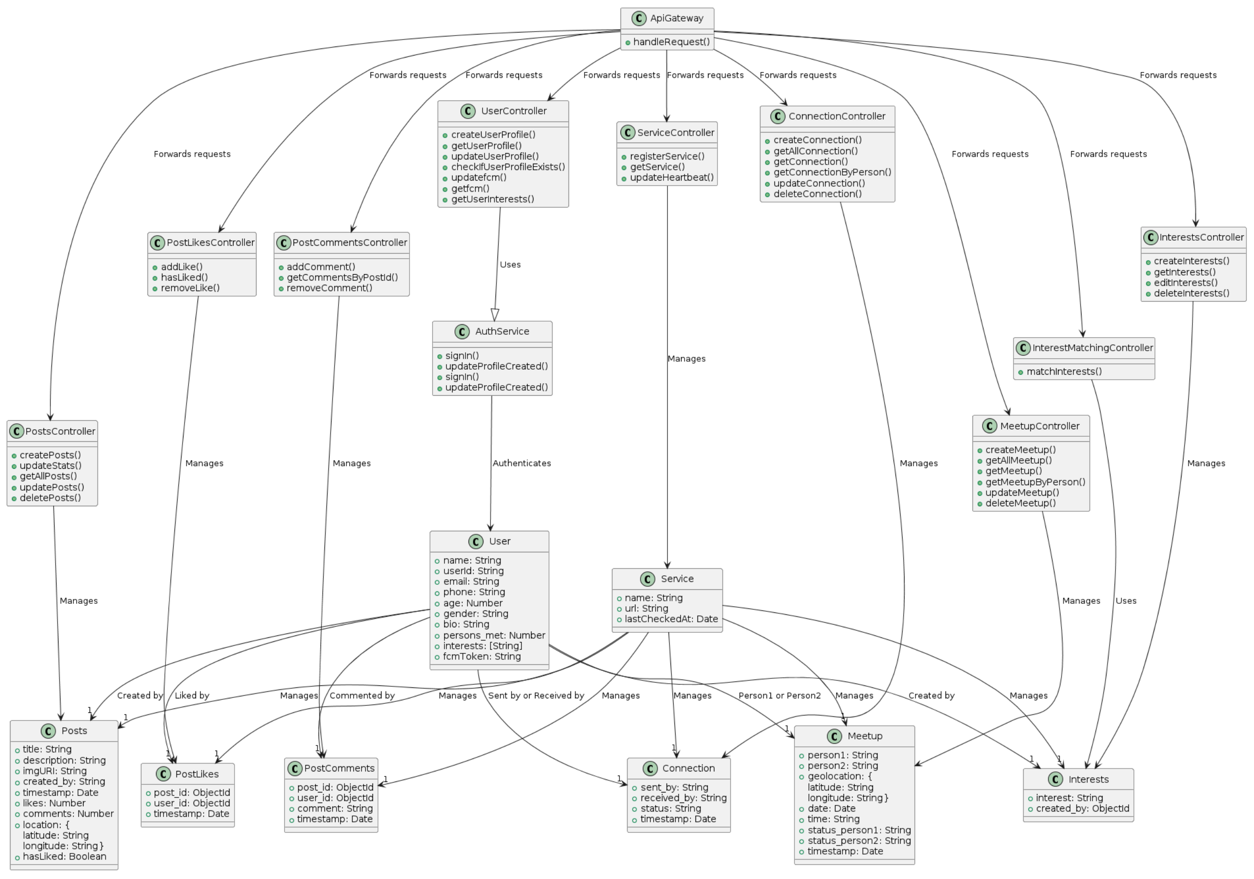
Role:

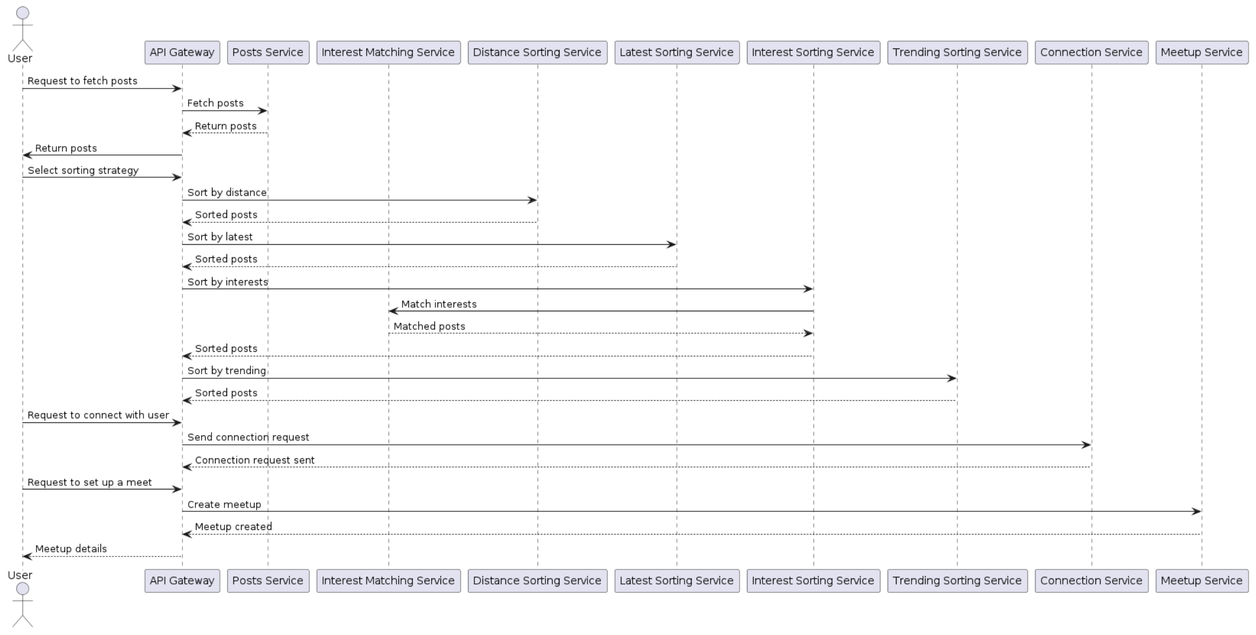
- The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the algorithm to vary independently from the clients that use it.

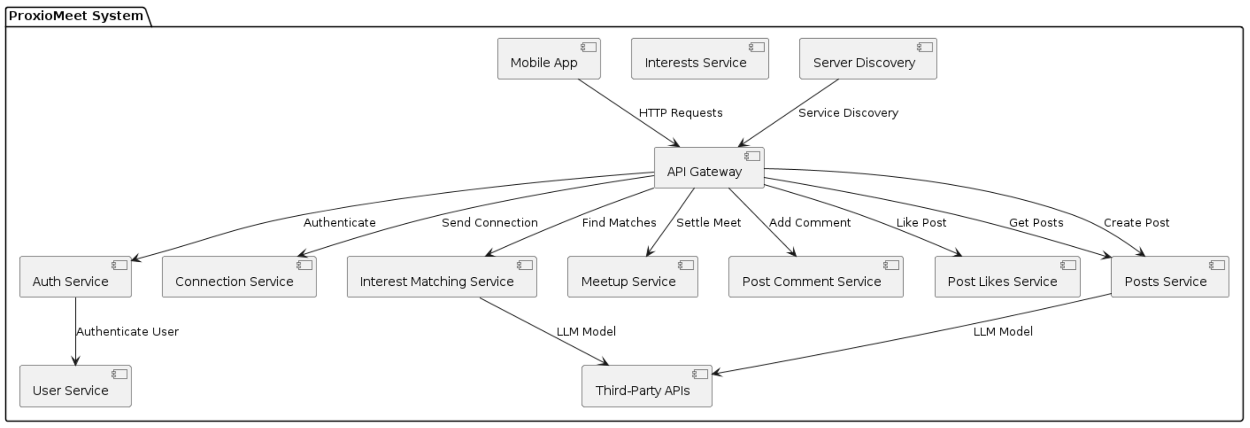
- In your architecture, the Strategy pattern can be used to implement different strategies for sorting posts based on trending, distance, or interests.

Explanation:

- Define an interface (`SortStrategy`) that declares a method for sorting posts.
- Implement concrete sorting strategies (`TrendingSortStrategy`, `DistanceSortStrategy`, `InterestSortStrategy`) that encapsulate specific sorting algorithms.
- The `PostSorter` component can accept any of these sorting strategies and use them interchangeably based on user preferences or system requirements.
- Users can select their preferred sorting strategy (e.g., by trending, distance) to view posts accordingly.







Task 4: Prototype Code

- **GitHub Repository:** ProxioMeet

Architectural Analysis

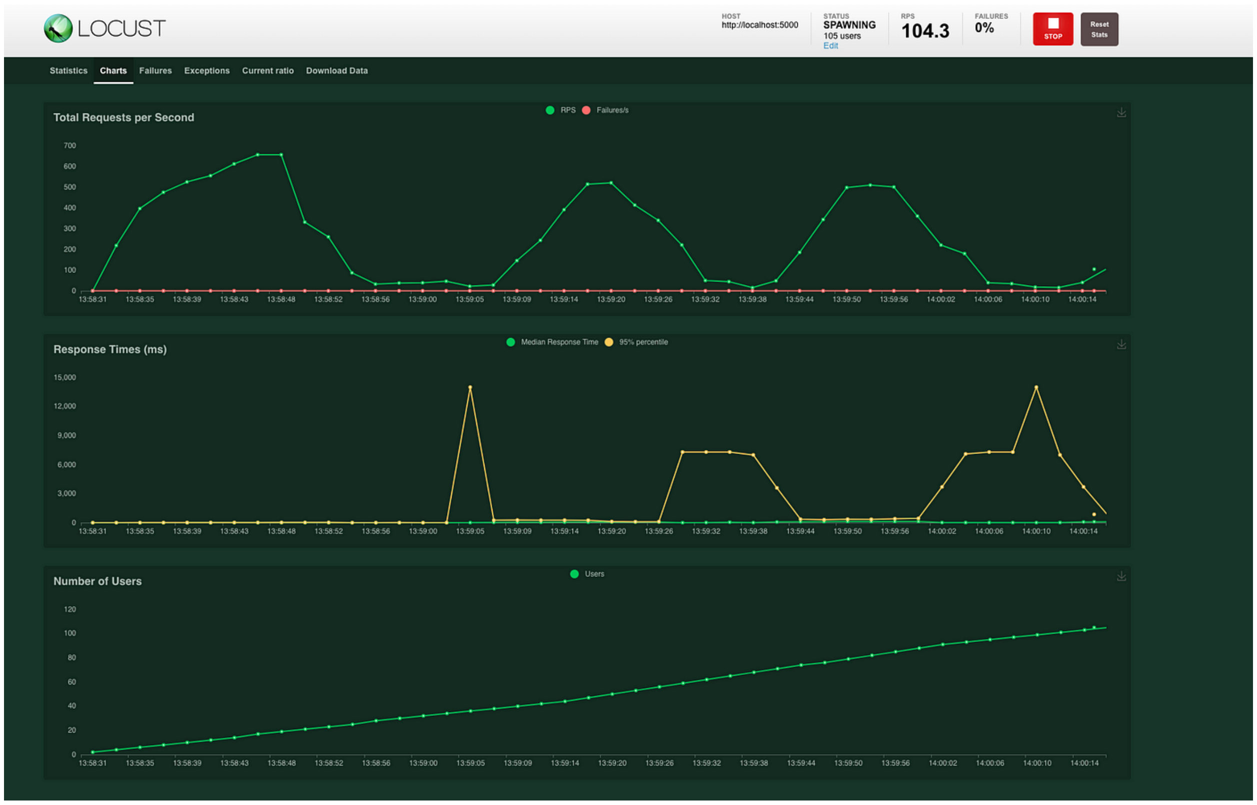
Microservices Architecture:

- **Architecture Description:**
    - Each of the `Posts`, `Likes`, and `Comments` services is implemented as a separate microservice.

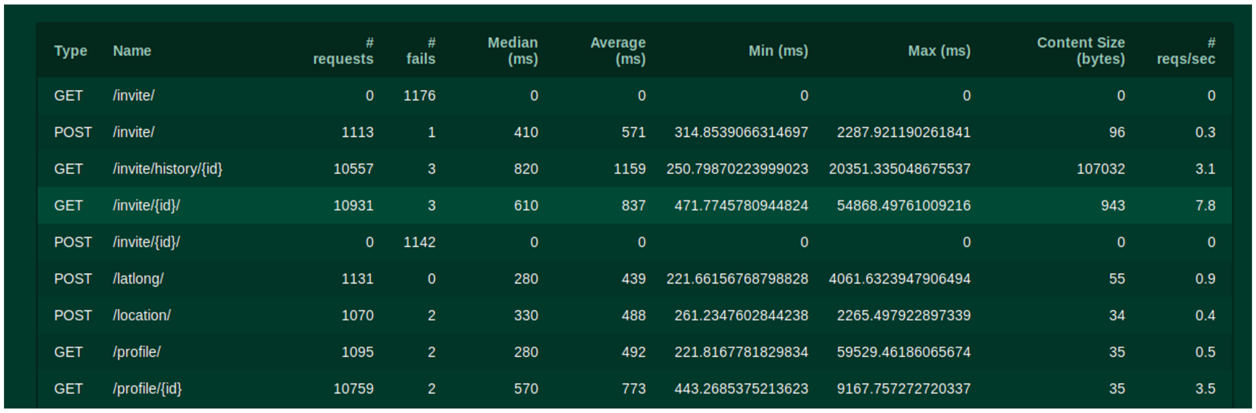
    - The services communicate with each other via API calls over the network.
    - Each service is responsible for its own data storage, business logic, and API endpoints.
    - Deployment, scaling, and maintenance of each service are independent, allowing for flexibility and modularity.
- **Advantages:**
    - **Scalability:** Each service can be scaled independently based on demand, allowing for optimized resource allocation and improved performance.
    - **Modularity:** The microservices architecture promotes loose coupling and high cohesion, making it easier to manage and evolve individual services.
    - **Fault Isolation:** Failures in one service do not necessarily impact the availability of other services, enhancing resilience and fault tolerance.
- **Challenges:**
    - **Network Overhead:** Inter-service communication over the network can introduce latency and overhead, potentially impacting performance and response times.
    - **Complexity:** Managing multiple services introduces complexity in deployment, orchestration, and monitoring, requiring additional tooling and infrastructure.

Testing:

We are using load simulator locust to simulate an environment and test the system. We are simulating the system for a maximum 100 users, and varying request rates with maximum of 700 request/sec, we have also ensured to test the system on all kinds of varing loads, and we also monitor the request time of the systems.



The request was sent to several API's chosen randomly from a set of API's. There are several failures because the API request's parameters do not pass correctly. We observed the following table.



Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size (bytes)	# reqs/sec
GET	/invite/	0	1176	0	0	0	0	0	0
POST	/invite/	1113	1	410	571	314.8539066314697	2287.921190261841	96	0.3
GET	/invite/history/{id}	10557	3	820	1159	250.79870223999023	20351.335048675537	107032	3.1
GET	/invite/{id}/	10931	3	610	837	471.7745780944824	54868.49761009216	943	7.8
POST	/invite/{id}/	0	1142	0	0	0	0	0	0
POST	/latlong/	1131	0	280	439	221.66156768798828	4061.6323947906494	55	0.9
POST	/location/	1070	2	330	488	261.2347602844238	2265.497922897339	34	0.4
GET	/profile/	1095	2	280	492	221.8167781829834	59529.46186065674	35	0.5
GET	/profile/{id}	10759	2	570	773	443.2685375213623	9167.757272720337	35	3.5

Monolithic Architecture:

- **Architecture Description:**
  - The `Posts`, `Likes`, and `Comments` functionalities are integrated into a single monolithic application.

    - There are no explicit API calls between components, as all functionalities are implemented within the same codebase.
    - Data sharing and communication between components occur through function calls and shared memory.
- **Advantages:**
    - **Simplified Communication:** With all functionalities co-located within the same runtime environment, there is no overhead associated with inter-service communication, leading to lower latency and faster response times.
    - **Ease of Development:** Developing and maintaining a monolithic application can be more straightforward compared to managing multiple services.
- **Challenges:**
    - **Scalability:** Scaling a monolithic application involves replicating the entire stack, which can be less efficient compared to scaling individual services in a microservices architecture.
    - **Maintenance Burden:** Changes to one functionality may require modifications to the entire codebase, leading to increased complexity and potential risks of regression errors.

Testing:

To test a monolithic architecture we used a python script, to simulate functions. The Python script calls functions with different functionalities at random with some probabilities, and we measure the time for each functionality to execute. Therefore, we can measure the performance of the monolithic architecture. We have also measured the CPU consumption by using a pyraphl library, and for each function we get the CPU consumption separately. Pyraphl uses a start statement and an end statement in between which measures the block's carbon footprint. This is added to each of the functions separately.

Here's a table with estimated values for power consumption (in watts) and time to execute each function (in milliseconds) on a monolithic architecture:

Function Name	Power Consumption (W)	Time to Execute (ms)
Login/SignUp	10	50
Create a Profile	15	100
Add Interest	12	80
Add a Post	18	120
Sorting Functionalities	8	60
Send a Request	14	90
Accept/Reject	11	70

These values are approximate and can vary based on various factors like system load, network conditions, and hardware capabilities.

Trade-offs:

- **Scalability:**
  - Microservices offer superior scalability by allowing individual services to be scaled independently based on demand. However, managing a large number of services can introduce complexity in orchestration and monitoring.
  - Monolithic architectures may be simpler to scale initially but can lead to inefficiencies and challenges in optimizing resource allocation for specific functionalities as the system grows.
- **Latency and Response Time:**
  - Monolithic architectures typically have lower latency and faster response times due to in-process communication and reduced overhead compared to microservices.
  - However, microservices offer greater flexibility and modularity, allowing for independent optimization of individual services to mitigate latency and response time issues to some extent.