Architectural Requirements

A project for EPI-USE by Syntex Squad

Demo 2

27 June 2025

Architectural Requirements

Summary

Below is a summary of our architectural requirements:

Quality Requirement	Architectural Tactic	Architectural Patterns
Scalability	Vertical scaling	N-layered pattern
Performance	Increase resource efficiency, maintain copies of data frequently accessed, parallel data fetching	Caching Modular Monolith
Availability	Replication	PostgreSQL streaming replication
Usability	Support system initiative, Real-time UI responsiveness	MVVM
Security	Security detection and resistance	OAuth2 and Spring security features

Architectural Design Strategy

Our design strategy focuses on two key approaches: Decomposition Strategy and Quality-Driven Requirements Strategy.

Decomposition Strategy

Concept: This approach involves breaking down the system into smaller, independent components or subsystems, much like constructing a building by dividing it into tasks like laying the foundation, building the walls, and installing the roof. Each component addresses specific

aspects of the system, ensuring a modular and manageable architecture.

Benefits:

- Modularity: Each component operates independently, enhancing system understanding and management. This modularity simplifies the complexity, allowing developers to focus on individual parts without getting overwhelmed by the entire system.
- Maintainability: Changes or fixes can be applied to individual components without affecting the entire system. This isolation of components means that updating or debugging one part doesn't require extensive changes to others, thereby reducing the risk of introducing new bugs.
- Extensibility: Adding new features becomes easier as new components can be integrated without disrupting existing ones. This ensures that the system can evolve over time, incorporating new functionalities seamlessly.

Quality-Driven Requirements Strategy

Concept: The design process is guided by the key quality requirements of the system. These requirements define the essential characteristics that ensure the system's success, such as reliability, efficiency, security, and usability.

Benefits:

- Focus on Quality: By prioritizing quality from the beginning, the final system is more likely to meet or exceed user expectations. This approach ensures that the design addresses crucial aspects like performance, reliability, and security, resulting in a robust system.
- Improved User Experience: The system will be reliable, perform well, be secure, and easy to use. By emphasizing these quality attributes, the system not only fulfills its functional requirements but also provides a pleasant and efficient user experience.

Architectural Quality Requirements

1. Scalability
   - 1.1. The system design shall allow for adding new features with minimal impact on existing functionality
   - 1.2. Quantification
     - 1.2.1. The architecture shall support vertical scaling to manage increased user load.

2. Performance
   - 2.1. Performance is considered a crucial underlying factor for our system, highlighting the importance of processing power and all-round data analysis capabilities. On the frontend side, effective and rapid rendering is crucial for real time posts ,bot feeds and feeds are provided to the end user, while being backed by a solid backend system with a RESTful API. We use a form of client-side orchestration by controlling flow in fetching data from multiple modules In parallel, the flow is deciding what to call, in what order and how to handle failures. This reduces round-trip latency and avoids sequential bottlenecks.
   - 2.2. Quantification
     - 2.2.1. Feed generation and bot post creation should be within 500ms for 95% of requests(for atleast 1000 concurrent users)
     - 2.2.2. App loading time should be less than 10 seconds.
     - 2.2.3. API response time should be less than 5 seconds.
     - 2.2.4. The average response time for user actions should not exceed 3 seconds under normal load.
3. Availability
   - 3.1. Our system is designed to have the ability to mask or repair faults such that the cumulative service outage period does not exceed an arbitrary value over a specified time interval.
   - 3.2. Quantification
     - 3.2.1. The system shall have an uptime of 98% per month.
     - 3.2.2. Backup procedures must be in place to restore service within 1

hour in case of failure. A replication of some sort must be considered

4. Usability
   - 4.1. A system designed to help users express themselves on a social media platform in the form of posts, AI generated posts, following other users as well as Feed interaction.
     Usability refers to how easy and enjoyable it is for users to use the system.
   - 4.2. To ensure usability is met, real-time UI responsiveness is achieved with the aid of MVVM.
   - 4.3. Quantification
     - 4.3.1. The system should help users achieve their goals (e.g.,expressing themselves via posts) by
       - 4.3.1.1. Providing a clean, intuitive interface with user-friendly navigation.
       - 4.3.1.2. Ensure consistency in the design by using established design systems and component libraries.
     - 4.3.2. The system should be easy and quick to use, allowing users to complete tasks without unnecessary effort or time by providing an easy-to-access help menu within the app for users to get help when needed."
     - 4.3.3. The system should be user-friendly and enjoyable to interact with, leaving the users feeling satisfied with their experience by
       - 4.3.3.1. Simplify data entry and interaction processes to minimize user effort.

4.3.4. Provide clear, concise instructions and tooltips to guide users through complex tasks.

5. Security
   5.1. A measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized.
   5.2. Quantification
      5.2.1. All user data shall be transmitted over encrypted channels (HTTPS)
      5.2.2. The system shall use secure password storage such as password hashing
   5.3.
6. Testability
   6.1. Our system's testability measures our test coverage and the ability to place a clear distinction between failing and passing units in our code. This will be required to provide a robust system that works under various circumstances.
   6.2. Quantification
      6.2.1. Presentation Layer: Make use of Usability Tests with our user base to ensure frontend achieves maximum user experience
      6.2.2. Logic Layer: Make use of SpringBoot testing framework to write unit tests for our backend.
      6.2.3. Data Layer: Make use of Springboot testing framework to test the database operations and to ensure consistency in the database to serve accurate and representative data.
      6.2.4. Between layers: Make use of Postman to test our interactions between layers.

6.2.5. System-wide: Make use of Springboot testing framework to write end-to-end tests simulating system flow.

6.2.6. Atleast 60% coverage overall.

Architectural Strategies

This project aims to develop a user-friendly and reliable social media platform where users can share posts, content and engage with posts generated by Bots as well as posts that are trending in general.

Our architectural strategy focuses on Scalability, Performance, Availability , Usability and Security by implementing performance tracking, automated testing, and service monitoring. The system prioritizes accessibility for diverse users by offering compatibility across devices(Desktop and Mobile) and a user-centric design, ensuring its meets the needs of all our targeted users for the application. We argue that for an application like Future Feed, the most important Quality Requirements are Scalability and Performance. Our project needs to scale as it grows and perform efficiently when handling a large number of active users

7. *Scalability*

7.1. Vertical Scaling: optimizing hardware and resources in backend

7.2. Data sharding: Design the database in such a way that large datasets are split up into smaller chunks or shards.(each shard stored on a separate database server). This also improves performance and fault tolerance.

8. *Performance*

8.1. For Control of Resource Demand

- Increase resource efficiency**:** Improving the algorithms used in critical areas will decrease latency.

8.2. For Managing Resources

- Maintain multiple copies of data(caching): keeping copies of data frequently accessed on storage with different access speeds

9. *Availability*

9.1. PostgreSQL streaming replication/Replication**:** Components are exact clones of each other, multiple copies of identical components in protecting against unexpected failures of hardware

10. *Usability*

10.1. User-Centric Design: Focus on creating an intuitive and user-friendly interface to enhance the user experience for our social media targeted user base.

10.2. Accessibility: Ensure the application is accessible to a wide range of users, including those with varying levels of technical proficiency.

10.3. Responsive Design: Implement a responsive design to ensure optimal usability across different screen sizes and orientations. This includes but is not limited to

Desktop and Mobile interfaces to support System Initiative

10.3.1. Maintain System Model: Here the system maintains an explicit model of itself. This is used to determine expected system behavior so that appropriate feedback can be given to the user. Interface metaphors and components such as progress bars to indicate and predict the time needed to complete a current activity.

11. Security

Two main categories of security: Detection and Resistance

11.1. For Security Detection we have:

- Verify message integrity**:** Using checksums and hash values to verify the integrity of messages, resource files, permitted actions and so on. **Checksum** maintains redundant information from messages and uses this to verify the message when it is used. Using **Hash values** to hash and check against protected user passwords in Future Feed.

11.2. For Security Resistance we have:

- Identify actors: Using user IDs to identify source of external input to system.
- Authenticate actors**:** Ensuring an actor is who they claim to be(user authentication). Passwords, hashed passwords and google gmail accounts to provide a means for authentication
- Authorize actors**:** Authorization means ensuring that an authenticated actor has the rights to access and modify either data or services. Here, our system differentiates between Bots and actual users each providing different access and permissions to data and or services.

11.3. Data encryption**:** Data is protected from unauthorized users by applying encryption to data such as passwords. This provides extra protection to persistently maintained data beyond that available from authorization

12. Testability

12.1. Automated Testing Framework: Establish a comprehensive automated testing framework to ensure rigorous testing across different layers of the architecture.

12.2. Unit Tests: Implement unit tests to validate the functionality of individual components.

12.3. Integration Tests: Develop integration tests to verify the interactions between different units.

12.4. End-to-End Tests: Conduct end-to-end testing to ensure smooth system flow and user experience.

12.5. Coverage Reports: Generate and review coverage reports to ensure all critical paths and components are adequately tested.

Architectural Patterns

For a social media platform like Future Feed, we feel that a hybrid architecture combining several patterns would be the best approach, rather than relying on a single one.

1. MVVM

1.2. it is important to keep modifications to the user interface software separate from the rest of the system

1.3. The model-view-controller (MVC) pattern separates application functionality into three kinds of components:

1.3.1. A model, which contains the application's data and business logic

1.3.2. A view, which displays the underlying data and handles user interactions.

1.3.3. A viewmodel, which mediates between the model and view, providing data binding and state

management

2. Modular Monolith

4.1. Modular monolith structures the platform into distinct, domain-specific modules within a single deployable unit. Each module incapsulates its own logic, data access, and API calls. This also achieves separation of concerns

4.2. Benefits:

4.2.1. Teams can optimize resource usage and development focus per module, enabling parallel development and clearer ownership without the operational overhead of distributed services.

Constraints

Budget Constraints

- Limited financial resources for initial development, deployment, and maintenance.
- Preference for open-source tools and free-tier services to minimise costs.

Hardware Constraints

- Compatibility with common farming hardware, including sensors and IoT devices.
- Limited processing power and memory on devices used by farmers (e.g., older smartphones and tablets).

Software Constraints

- AWS free-tier limits performance to 1000 users
- Bot content quality tied to LLM, requires internet access
- LLM cost

Network Constraints

- Ensuring functionality in areas with limited or intermittent internet connectivity.
- Efficient data transfer protocols to minimise bandwidth usage.

Technology Choices

1. Frontend
   1.1. Vite
      1.1.1. Vite is a progressive JavaScript framework used for building user interfaces. It is designed to be incrementally adoptable and focuses on the view layer.
      It serves as our build tool and deployment server

Pros:

1.1.2.1. Ease of Use: Vite's learning curve is gentle, making it accessible for developers with varying levels of experience. It also provides seamless support for React with typescript and JSX

1.1.2.2. Performance: Vite is known for its high performance and efficient rendering.

1.1.2.3. Efficient: Vite is built to work hand in hand with modern libraries like React and it supports Typescript.

1.2. React

1.2.1. React has good, tested UI Library for complex and interactive UI's.

Pros:

1.2.2.1. Ecosystem & Community: React offers a massive ecosystem of tools, libraries, and third-party integrations—crucial when you need flexibility in building social features, chat systems, or news feeds.

1.2.2.2. Component-Driven Architecture: Perfect for building scalable, reusable UI pieces like posts, comments, stories, or notifications.

1.2.2.3. TypeScript Integration: Adds strong typing for better maintainability and fewer runtime errors—vital for a complex app with a long lifespan.

1.2.3. Cons:

1.2.3.1. Learning Curve: React introduces additional concepts and configurations that require some learning.

1.2.3.2. Opinionated Structure: React enforces a certain project structure, which might not be suitable for every use case.

1.3. TailwindCSS

1.3.1. TailwindCSS is a utility-first CSS framework that provides low-level utility classes to build custom designs directly in the markup.

1.3.2. Pros:

1.3.2.1. Consistency: Ensures a consistent design system across the application by using predefined classes.

1.3.2.2. Utility-First Approach: Provides a wide range of utility classes that eliminate the need for custom CSS, speeding up the styling process.

1.3.3. Cons:

1.3.3.1. Verbose HTML: Using utility classes directly in the markup can lead to more verbose HTML.

1.4. Typescript

1.4.1 typescript ensures type safety in our application and prevents unwanted code

1.4.2. Pros

1.4.2.1 Compile-time validation: Caches errors before runtime in complex social interactions.

1.4.2.2. Self-documenting Code: Type definitioins serve as living documentation

2. Testing
   2.1. SpringBoot built-in testing framework
   2.1.2. Benefits:
   2.1.2.1. Rapid Setup: Spring Boot auto-configures the environment, so you can go from zero to RESTful endpoints in minutes.
   2.1.2.2. Built-in Testing Tools: It offers out-of-the-box support for JUnit, Mockito, and integration testing via @SpringBootTest.
   2.1.2.3. Easy Data Setup: Supports data injection for test cases using tools like Flyway or Liquibase.
3. Backend
   3.1. PostgreSQL
      3.1.1. PostgreSQL is a powerful, open-source relational database management system (RDBMS) known for its robustness, scalability, and standards compliance. It's widely used in many backend systems, including those involving complex data models and high transaction volumes.
      3.1.2. Pros:
         3.1.2.1. Advanced Features: PostgreSQL supports complex queries, foreign keys, triggers, views, and transactional integrity.
         3.1.2.2. Extensibility: It allows users to define their own data types, operators, and index types.
         3.1.2.3. Performance Optimization: Offers powerful indexing, partitioning, and parallelization features.
         3.1.2.4. Community and Support: Large, active community and plenty of documentation and third-party tools.
         3.1.2.5. Standards Compliance: Highly compliant with SQL standards, ensuring portability and compatibility.
      3.1.3. Cons:
         3.1.3.1. Complexity: Its vast array of features can be overwhelming and may require a steep learning curve.
         3.1.3.2. Resource Intensive: Requires more system resources compared to simpler databases.
         3.1.3.3. Setup and Maintenance: Requires careful setup and ongoing maintenance, especially in high-availability and high-transaction environments.
   3.2. Java
      3.2.1. Java is a versatile programming language widely used in backend and is very reliable
         3.2.1.1. Benefits:
         3.2.1.2. Robust Ecosystem: Libraries for messaging, security (OAuth2/JWT), database access (JPA/Hibernate), and more.
         3.2.1.3. Concurrency Handling: Java’s multithreading makes it well-suited to real-time features like notifications, messaging, and

live updates.