Top 3 Prioritized NFRs and Justification

1. **Usability**: This non-functional requirement was prioritized in our development process due to its potential impact on user experience. Specifically, we focused on maintaining page persistence when a user reloads or navigates away from the page. Previously, if a user switched tabs or refreshed the page, they would be redirected to the dashboard, often causing frustration—especially if they were in the middle of completing a form. This disruption highlighted the need to enhance usability, prompting us to address this issue as a priority.
    a. *Trade-off 1: Performance Overhead:* Persisting web pages requires storing and retrieving the user's state, which can introduce performance overhead. This may lead to longer load times or increased use of browser resources, potentially slowing down the user experience in other areas. In fact, load times have become noticeably slower compared to previous versions of the application.
    b. *Trade-off 2: Security and Data Privacy Concerns:* Storing sensitive form data in the browser's local storage increases security risks. Because this data persists beyond the active browsing session, it may be accessible to unauthorized users—particularly on shared or public devices—raising concerns about data privacy and potential misuse.
2. **Scalability**: Over the last few sprints, we continuously added features such as chat and multi-user interactions. As the application expanded, the need to support concurrent users and handle increasing data volumes became more critical. During development, we encountered challenges related to this non-functional requirement. To address them, we implemented rate limiting to manage Supabase's request limits and integrated the map feature in a way that preserved overall application performance.
    a. *Trade-off 1: Rate Limiting vs. User Experience:* We implemented client-side rate limiting to manage Supabase's request limits. While this helped prevent API overload, it came at the cost of user experience. For instance, during periods of high traffic, users could experience delays, and more complex interactions might be throttled, leading to frustration or interrupted workflows.
    b. *Trade-off 2: Database Query Optimization vs Development Speed:* We implemented pagination and filtering in listing queries, allowing the application to handle large datasets efficiently while supporting complex search functionality. However, this introduced the trade-off of more complex query logic to maintain, along with the potential for performance degradation when applying intricate filters.
3. **Reliability**: We prioritized this non-functional requirement because, like the previous ones, it has a significant impact on user experience. If the application only functions correctly under certain conditions, users are likely to become frustrated. To address this, we implemented features such as retry mechanisms and graceful degradation to improve overall reliability and ensure consistent performance.
    a. *Trade-off 1: Caching Strategy vs. Data Freshness:* We implemented aggressive caching to enhance reliability and reduce server load. However, this approach

introduces the risk that users may occasionally see outdated information, especially in rapidly changing data scenarios.

b. Trade-off 2: Retry Mechanisms vs. User Experience:As previously mentioned, retry mechanisms were implemented to improve reliability. The trade-off, however, is that during network issues, these retries can make the application feel slower or less responsive to users. As a result, the user experience may be negatively affected.

Performance Testing Report

Test Environment:

- Browser: Chrome 120.0.6099.109
- Device: Windows 10, 16GB RAM, Intel i7
- Network: WiFi (50 Mbps)
- Date: Sunday, August 3rd. 2025
- Application Version: Roomzi V4

Key Findings:

Page Load Performance:

- Homepage: 6.0 seconds
    - Performance: 55/100
    - Accessibility: 100/100
    - Best Practices: 100/100
    - SEO: 100/100
- Tenant Dashboard: 6.1 seconds
    - Performance: 36/100
    - Accessibility: 92/100
    - Best Practices: 100/100
    - SEO: 100/100
- Matches Page: 6.2 seconds
    - Performance: 37/100
    - Accessibility: 89/100
    - Best Practices: 100/100
    - SEO: 100/100
- My Leased Properties Page: 6.4 seconds
    - Performance: 37/100
    - Accessibility: 92/100
    - Best Practices: 100/100
    - SEO: 100/100
- My House Page: 6.5 seconds
    - Performance: 54/100
    - Accessibility: 94/100
    - Best Practices: 100/100
    - SEO: 100/100

- Profile Page:
    - Performance: 54/100
    - Accessibility: 93/100
    - Best Practices: 100/100
    - SEO: 100/100

Performance Expectations Met:

Performance testing for this was done using the Google Developer Tool Lighthouse. Testing was done for five metrics which were page load times, performance (lighthouse measurement), Accessibility (Lighthouse Measurement), Best Practices (Lighthouse Measurement), and SEO (Lighthouse Measurement). The performance category measures the speed and responsiveness of a webpage. The accessibility category checks if the website is difficult for people with disabilities to use, while best practices is the metric to check if most best practices were used for the creation of this website. SEO is search engine optimization which checks for the likelihood of the visibility of the webpage in search results. For accessibility, Best practices, and SEO I would say the system surpassed expectations. It achieved high scores even though these were not targeted. Performance fell short off expectations, this is most likely because the tradeoffs made for user experience were larger than expected.

Security Measures and Testing

The application employs a comprehensive security architecture encompassing authentication, authorization, data protection, and API security. Supabase handles authentication using JWTs with PKCE and magic links, ensuring secure session management and passwordless access. Role-based access control (RBAC) supports tenant and landlord roles with frontend and backend route protection. Data security is reinforced through Prisma ORM, row-level security, file type/size validation, and private Supabase storage with signed URLs. APIs are safeguarded via strict CORS rules, input sanitization, and standardized error handling. Sessions and tokens are securely managed with automatic refresh and persistent storage, while additional safeguards ensure graceful error handling, validation, and role synchronization.

Based on preliminary and basic testing, the system appears to be reasonably secure. Unauthorized users are unable to access protected pages without authentication, and users can only access their own information. Basic SQL injection attempts on the login page were unsuccessful. However, one notable security concern is the file upload feature on the tenant lease page. There is currently no proper validation or restriction in place, which means a malicious file could potentially be uploaded and cause issues.

Scalability and Availability Considerations

The system is designed with scalability and availability in mind, using a modular architecture with a stateless Node.js/Express backend and a React/TypeScript frontend. Supabase's managed PostgreSQL database, paired with Prisma connection pooling, supports efficient handling of increased data and users—up to 1000 concurrent users as per the release plan. While current deployment uses a single server without load balancing or caching, the

architecture supports horizontal scaling. Future improvements include adding Redis for caching, load balancers for backend redundancy, and a CDN for faster asset delivery. Availability is supported by Supabase’s 99.9% SLA, health check endpoints, and graceful shutdown handling.