Snap Vision

Architectural Requirements Document

Demo 4 – Updated Version

Table of Contents

1. Architectural Design Strategy

Snap Vision follows a modular, layered architecture to balance scalability, maintainability, and the limited team size. This strategy ensures that mobile, backend and indoor/outdoor navigation services remain cleanly separated but still interoperable.

Key factors:

- Separation of concerns between the frontend and backend.
- Parallel development support.
- Future scalability to support new locations.

We make use of the following layers:

- **Presentation Layer (Mobile App):** Handles UI/UX, map overlays, AR navigation, admin tools, developed with React Native.

- **Business Logic Layer (Backend APIs):** Manages navigation logic, user sessions, and administrative actions.

- **Data Layer (Cloud Database):** Stores map data, building info, navigation points, user information, etc.

This approach aligns well with our use of mobile-first delivery and real-time data streaming, while keeping future migration paths open with an increase in scalability.

2. Architectural Design and Patterns

Model-View-ViewModel (MVVM)

MVVM is a frontend architectural pattern that separates the user interface (View), business logic/state (ViewModel), and the data (Model). The ViewModel acts as the middle layer, reacting to data and exposing only the necessary values to the UI.

- The View is our screens (e.g, MapScreen, ManageUsersScreen).
- The ViewModel is implemented via custom hooks and state managers (e.g, useUserManagement, useNavigation).
- The Model refers to data fetched from a cloud database (e.g, buildings, routes).

Justification:
This pattern enforces separation of concerns within the Presentation Layer by clearly distinguishing between the user interface, business logic/state management, and underlying data.
It enhances testability since the ViewModel can be validated independently of the UI, ensuring more reliable and maintainable code. Furthermore, it supports modular and scalable development.

Component-Based Architecture

Component-based architecture involves dividing the system into reusable, independent modules, each responsible for a specific feature or concern. These can be developed, tested, and deployed in isolation.

- The mobile app is split into atoms/molecules/organisms (atomic design).
- AR overlays, admin panels, pathfinding, and feedback tools are all self-contained components.

Justification:
Component-based architecture enhances parallel development (multiple team members can work independently), simplifies unit testing, and supports our layered architecture. It also supports future scalability, as each component can evolve without breaking others.

Monolithic Architecture

A monolith means that the server-side code is shipped as one deployable unit that serves all features, backed by one database.

- The mobile app includes all user-facing features (indoor navigation, QR scanning, admin tools, AR overlays) in a single build.

- The minimal backend (snap-vision-backend) is implemented as a single service that handles route generation and connects directly to the shared database used by the app.

- The database, file storage, and authentication system act as a single unified data source for all modules.

Justification:
Monolithic architecture is viable for Snap Vision’s current scale and our team size because it simplifies deployment and maintenance.
Having one deployable unit means faster development cycles, reduced operational overhead, and easier debugging.
All features share the same environment and data model, ensuring consistency across the app.
While the architecture remains monolithic, we still enforce clean boundaries inside the code through layered design, MVVM, and component-based structures, making it easier to scale or refactor into separate services in the future if needed.

3. Architectural Quality Requirements

Quality Requirement	Metric / Testable Criteria	Testing	Justification	How It Influenced Our Design Decisions
Latency	**Location** is fetched within 5 seconds of request. **Route** is fetched within 2 seconds of request.	Firebase Performance testing was used to monitor these metrics and track changes.	● A user can see the effects of their actions in the app immediately. ● This is important for **real-time updates**, such as tracking a user's location as they move. ● Immediate feedback ensures the user receives the most **up-to-date information** and **accurate routing**.	● Components like location tracking were decoupled to prevent delays across layers. ● **RAM-first logic** via local state helps to minimize delays.
Performance	**Map** is visible within 10 seconds. **Buildings** are loaded in under 3 seconds. **Indoor maps** are available within 3 seconds.	Firebase Performance testing was used to monitor these metrics and track changes.	● Performance is a critical quality attribute, as users expect to complete tasks with minimal delay. ● Map visibility is a central function of the system, so it is essential that outdoor and indoor maps are readily available to maintain a seamless and **responsive user experience**.	● **Rendering logic** is handled in React Native, independent of route generation. ● **Component-based architecture** was used to ensure that the map renders independently from other services. ● **Async updates** support this by loading POIs, routes and overlays asynchronously.

Security	Ensure that a **user's live location** is not accessible to anyone besides the user and who they've shared it with. Strict Firestore rules are enforced for **RBAC**.	Only real-time, non-persistent location data is stored. RBAC verified through Firestore Security Rules testing and role simulations.	● As the system supports real-time navigation and scalable location tracking across multiple users and maps, security is critical to protect sensitive user data. ● **No historical location data is stored**, minimizing the risk of unauthorized access and aligning with privacy best practices. ● Users cannot access unauthorized content due to Firestore rules.	● **Data layer** is designed with stateless, session-based location tracking. ● **Role-based access control** to ensure maximum security. ● **Token-based auth** is also used for access validation.
Scalability	Support 50+ **concurrent users** navigating a building	Simulated concurrent sessions via multiple emulator instances and staggered navigation starts.	● The system must support multiple maps and users across diverse locations. ● The application must remain **reliable and responsive**, even when many users are navigating to the same destination simultaneously.	● Firebase Firestore's **horizontal scaling** supports concurrent reads/writes with minimal latency. ● While the backend is a modular monolith, routing is a separate endpoint that can be **scaled independently** if needed.

Usability	All **POIs** are shown on the map. **Text to speech** is available to all users. Colours are acceptable according to **WCAG standards**.	Firebase Test Lab was used to simulate a user and report back on usability standards.	● An **intuitive design** is necessary for first time users in order to enhance the overall user experience. ● Text-to-speech enhances usability by making the app accessible to users who are visually impaired or need hands-free interaction. ● Accessible indoor routes are necessary for those who are unable to take the stairs.	● The **MVVM architecture** allows UI logic to be modular and updated independently of state logic. ● Our use of **atomic design** made it easy to reuse components and apply consistent styling. ● **Mobile-first UI** decisions prioritise small screen clarity and interaction.

4. Architectural Strategies

Strategy	Explanation	How It's Used
Separation of concerns	Break different parts of the system into distinct, isolated layers.	UI (React Native), logic, data (Firestore), and AR modules are clearly split.
Layered modular monolith	One deployable backend unit with most of the logic being in the client.	Everything shares one Firebase project to simplify data storage and deployment.
Event-driven design	System reacts to events, not constant polling.	Live location updates, QR scans, Crowd Report forms, and navigation are all event-triggered.
Component-based modularity	Features are built as reusable components within the app.	Atomic design is used to maximize reusability and efficiency.
Stateless compute	No session data is stored in the backend itself.	Routing endpoint validates token and computes route per request. State lives on-device.

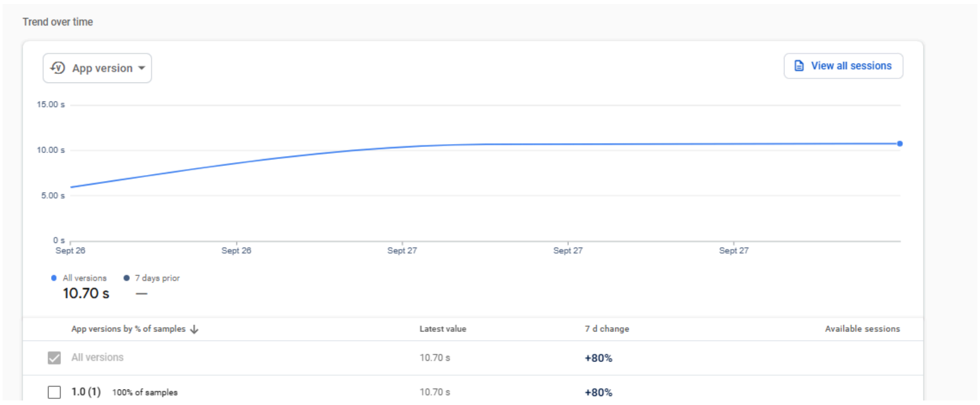
5. Architectural Constraints

Constraint	Description	Impact on Architecture
Mobile-first design	The system must work well on mobile phones first, not desktops	Forces a layered architecture where UI and backend are decoupled. Optimized layouts, gestures, and navigation for touch interfaces.
Security Rules complexity	Rules must reflect RBAC and location scoping without gaps.	The component-based architecture allows for the UI to be separate from the database calls.
Offline fallback required	The app should still function if the internet is unavailable.	Architecture needs caching and client-side storage to allow limited operation without server access.
Indoor environments lack GPS	Can't rely on satellite positioning inside buildings.	The architecture must support modular fallback positioning subsystems (e.g., QR code).
Privacy law compliance	No storage of personally identifiable location history (POPIA).	Data architecture avoids long-term tracking of users - uses session-based data models instead.

6. Quality Requirements Testing

Latency Tests

Quantification: Location is fetched within 5 seconds of request.
Tool: Firebase Performance Testing.

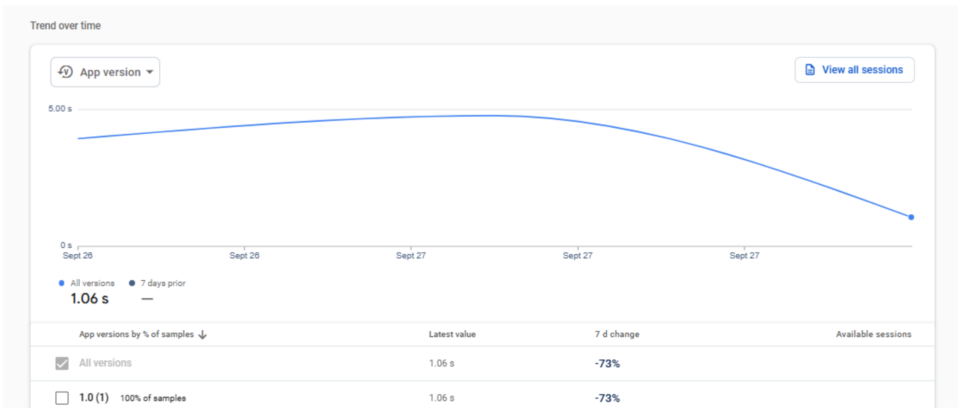


10.70 s —

App versions by % of samples	Latest value	7 d change	Available sessions
All versions	10.70 s	+80%	
1.0 (1) 100% of samples	10.70 s	+80%	

Conclusion: The Firebase Performance Testing logs prove that a user's location is fetched within 5-10 seconds, which is slightly slower than ideal. However, this is expected due to environmental factors and internet strength.

Quantification: Route is fetched within 2 seconds of request.
Tool: Firebase Performance Testing.



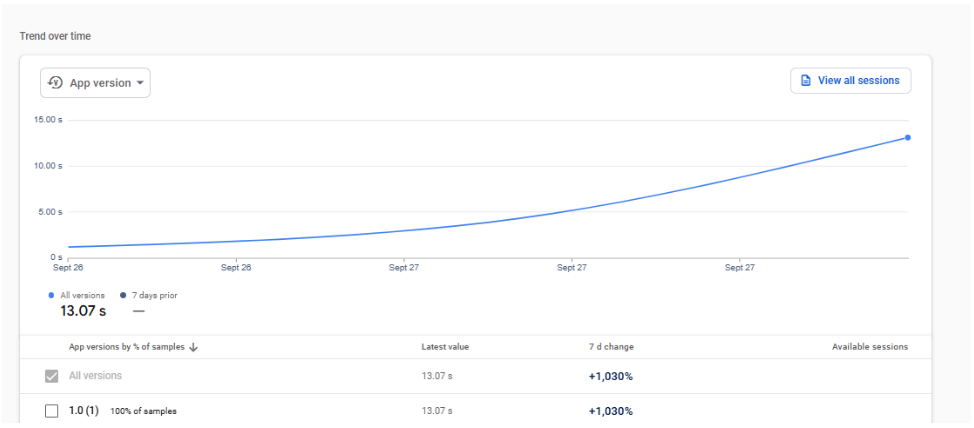
1.06 s —

App versions by % of samples	Latest value	7 d change	Available sessions
All versions	1.06 s	-73%	
1.0 (1) 100% of samples	1.06 s	-73%	

Conclusion: The above logs show that the route is fetched within 1 second, which surpasses our initial expectations.

Performance Tests

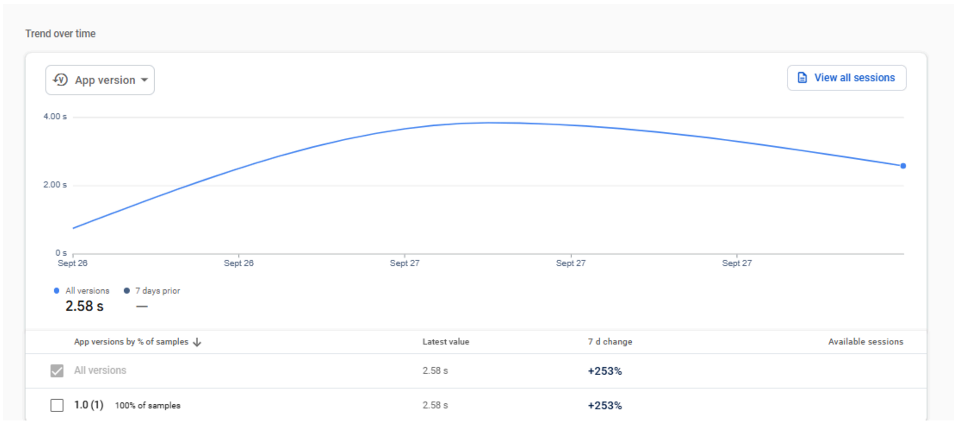
Quantification: Map is visible within 10 seconds.
Tool: Firebase Performance Testing.



App versions by % of samples ↓	Latest value	7 d change	Available sessions
☑ All versions	13.07 s	+1,030%	
☐ 1.0 (1) 100% of samples	13.07 s	+1,030%	

Conclusion: The map was visible within the range of 5-13 seconds. This measurement is dependent on external factors such as internet strength.

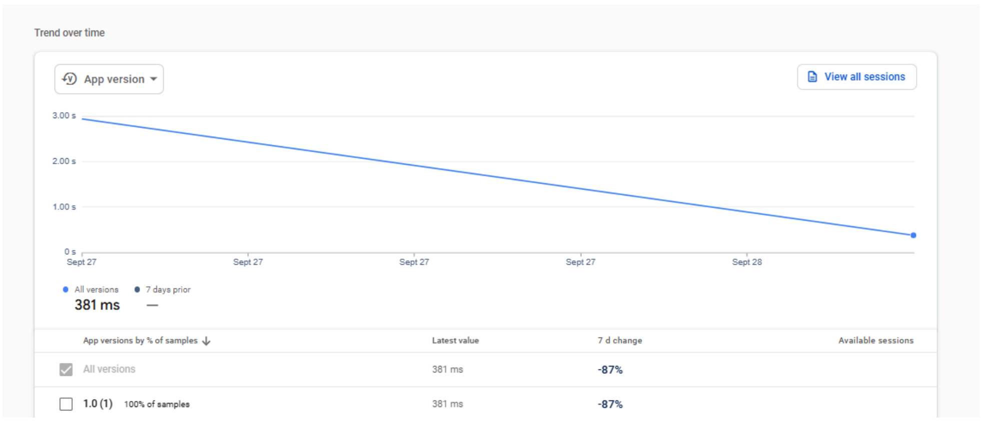
Quantification: Buildings are loaded in under 3 seconds.
Tool: Firebase Performance Testing.



App versions by % of samples ↓	Latest value	7 d change	Available sessions
☑ All versions	2.58 s	+253%	
☐ 1.0 (1) 100% of samples	2.58 s	+253%	

Conclusion: The above metrics show that the list of available buildings for uploading a floorplan consistently loads under 3 seconds. This is in line with our expectations, as caching is used to improve load speed.

Quantification: Indoor maps are available within 3 seconds.
Tool: Firebase Performance Testing.



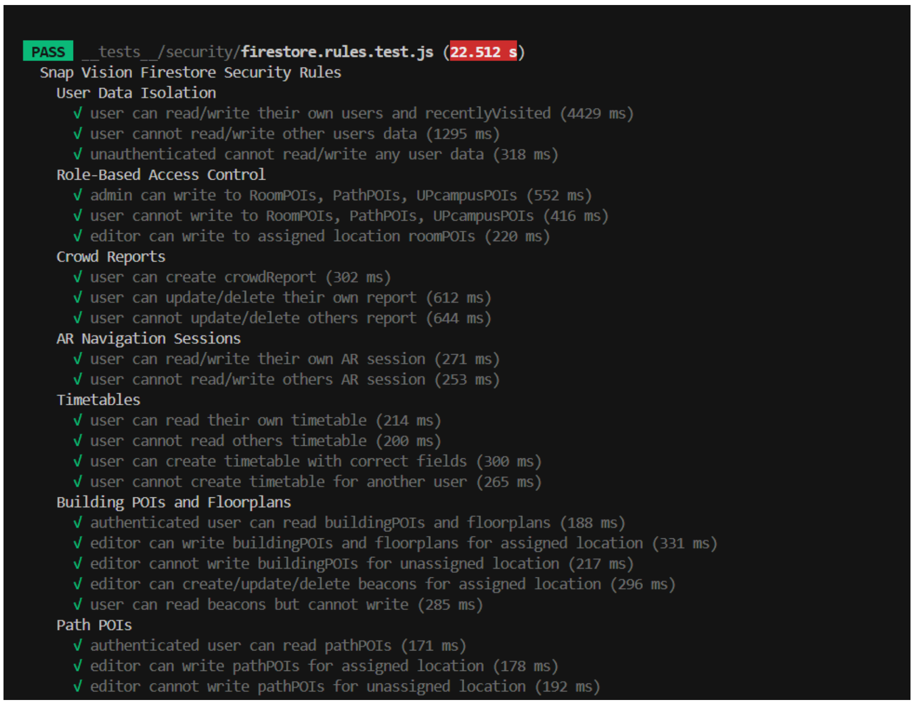
381 ms —

App versions by % of samples	Latest value	7 d change	Available sessions
✓ All versions	381 ms	-87%	
☐ 1.0 (1) 100% of samples	381 ms	-87%	

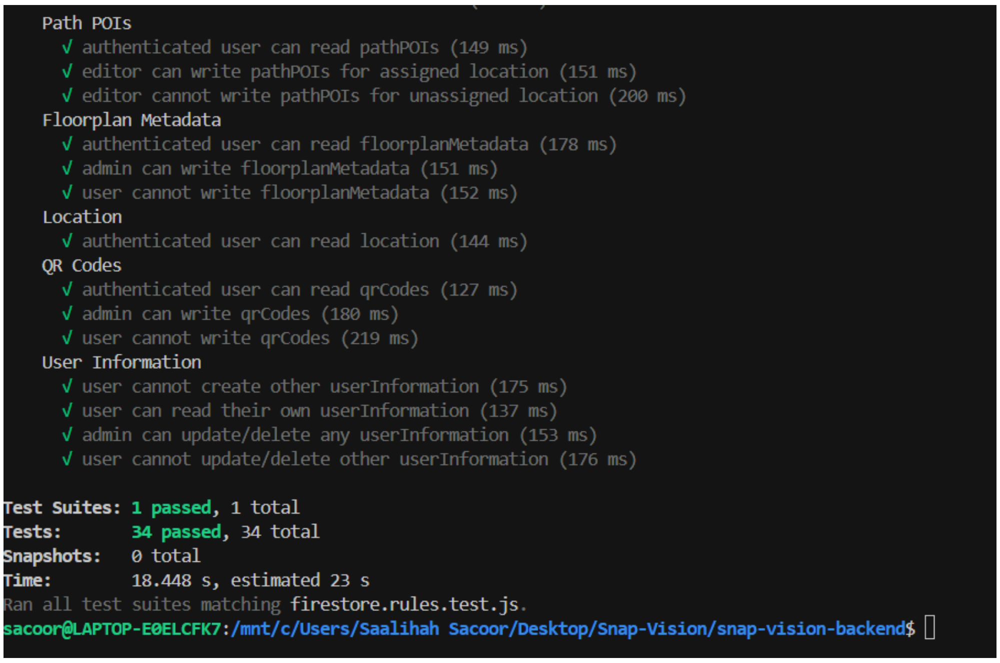
Conclusion: Indoor floorplans consistently load within 3ms. This aligns with our initial expectations.

Security Tests

Quantification: RBAC is enforced via Firebase Rules.
Tool: Firebase Security Unit Testing.



```
 PASS  __tests__/security/firestore.rules.test.js (22.512 s)
  Snap Vision Firestore Security Rules
    User Data Isolation
      √ user can read/write their own users and recentlyVisited (4429 ms)
      √ user cannot read/write other users data (1295 ms)
      √ unauthenticated cannot read/write any user data (318 ms)
    Role-Based Access Control
      √ admin can write to RoomPOIs, PathPOIs, UPcampusPOIs (552 ms)
      √ user cannot write to RoomPOIs, PathPOIs, UPcampusPOIs (416 ms)
      √ editor can write to assigned location roomPOIs (220 ms)
    Crowd Reports
      √ user can create crowdReport (302 ms)
      √ user can update/delete their own report (612 ms)
      √ user cannot update/delete others report (644 ms)
    AR Navigation Sessions
      √ user can read/write their own AR session (271 ms)
      √ user cannot read/write others AR session (253 ms)
    Timetables
      √ user can read their own timetable (214 ms)
      √ user cannot read others timetable (200 ms)
      √ user can create timetable with correct fields (300 ms)
      √ user cannot create timetable for another user (265 ms)
    Building POIs and Floorplans
      √ authenticated user can read buildingPOIs and floorplans (188 ms)
      √ editor can write buildingPOIs and floorplans for assigned location (331 ms)
      √ editor cannot write buildingPOIs for unassigned location (217 ms)
      √ editor can create/update/delete beacons for assigned location (296 ms)
      √ user can read beacons but cannot write (285 ms)
    Path POIs
      √ authenticated user can read pathPOIs (171 ms)
      √ editor can write pathPOIs for assigned location (178 ms)
      √ editor cannot write pathPOIs for unassigned location (192 ms)
```



```
    Path POIs
      √ authenticated user can read pathPOIs (149 ms)
      √ editor can write pathPOIs for assigned location (151 ms)
      √ editor cannot write pathPOIs for unassigned location (200 ms)
    Floorplan Metadata
      √ authenticated user can read floorplanMetadata (178 ms)
      √ admin can write floorplanMetadata (151 ms)
      √ user cannot write floorplanMetadata (152 ms)
    Location
      √ authenticated user can read location (144 ms)
    QR Codes
      √ authenticated user can read qrCodes (127 ms)
      √ admin can write qrCodes (180 ms)
      √ user cannot write qrCodes (219 ms)
    User Information
      √ user cannot create other userInformation (175 ms)
      √ user can read their own userInformation (137 ms)
      √ admin can update/delete any userInformation (153 ms)
      √ user cannot update/delete other userInformation (176 ms)

Test Suites: 1 passed, 1 total
Tests:       34 passed, 34 total
Snapshots:   0 total
Time:        18.448 s, estimated 23 s
Ran all test suites matching firestore.rules.test.js.
sacoor@LAPTOP-E0ELCFK7:/mnt/c/Users/Saalihah Sacoor/Desktop/Snap-Vision/snap-vision-backend$ 
```

Conclusion: The above tests thoroughly go through the enabled Firestore Rules to ensure that there is no unauthorized access. All of the tests pass, which aligns with our quantification of strict RBAC.

7. Technology Choices

Component: Frontend Framework

Option	Pros	Cons
React Native	Fast development, large ecosystem, cross-platform, reusable UI logic	Some native modules (e.g., ARKit, BLE) are trickier
Flutter	Excellent UI capabilities, performs close to native	Smaller ecosystem, Dart learning curve
Native (Kotlin/Swift)	Full control over device features	Time-consuming; separate codebases for Android and iOS

Final Choice: React Native
We selected React Native as it aligns with our mobile-first architecture and supports the Component-Based strategy through modular, reusable UI components. It also integrates cleanly into our MVVM frontend pattern, supporting separation of concerns.
This decision enables rapid development within our team's capabilities. Although native AR integration is slightly more complex, our architecture accommodates this through isolated AR modules, ensuring our design remains modular and maintainable.

Component: Backend Logic

Option	Pros	Cons
Express.js (Node)	Lightweight, widely supported, fully customizable, REST-friendly	Requires hosting and some server management
Firebase Functions	Serverless, mobile-first, scales automatically, easy auth integration	Limited long-running control and infrastructure flexibility
AWS Lambda	Language flexibility, scalable, serverless	Complex to set up and manage IAM and API Gateway

Final Choice: Express.js (with Node.js)
We chose Express.js because it provides a lightweight, modular backend framework that fits naturally with our layered architecture and componentized design. Express gives us full control over backend routes like /generateRoute which is central to our system.

While it does introduce some infrastructure responsibility, we mitigate this with simple deployment pipelines (CI/CD via GitHub Actions) and host our backend on cloud infrastructure that still respects our cost constraint (e.g., keeping usage within free-tier limits). Express also integrates well with Firestore, allowing us to coordinate logic and data layers without platform lock-in.

This approach supports our stateless design, makes testing easier, and aligns with our goal of flexibility across future deployments.

Component: Data Storage

Option	Pros	Cons
Supabase	SQL support, RLS (role-level security), file storage, real-time APIs	Still maturing; some features are in beta
Firebase Firestore	Real-time syncing, mobile-friendly, serverless, scalable	Complex querying is limited (e.g., joins)
MongoDB Atlas	Flexible document model, powerful querying	Requires server setup or third-party backend

Final Choice: Firestore (for sessions, routes, POIs)

Firestore handles real-time, loosely structured data like navigation sessions and feedback, supporting our event-driven design and low-latency QR updates

This optimizes for performance, scalability, and modularity. It also allows us to stay within free-tier limits, ensuring the system is cost-effective and responsive.

Component: AR/Positioning System

Option	Pros	Cons
ARCore/ ARKit	Native AR SDKs for Android/iOS, low latency, high realism	Requires tuning, platform-dependent setup
React Native Vision Camera + Custom AR	Cross-platform, integrates with RN ecosystem, customizable overlays	Limited AR capabilities
8thWall	High-quality web-based AR, supports image anchors	Paid license required, not mobile-native
Vuforia	Strong image tracking, cross-platform	Not designed for real-time indoor navigation

Final Choice: React Native Vision Camera + Custom AR implementation
We selected React Native Vision Camera as our AR foundation to maintain cross-platform compatibility within our React Native app. This approach allows us to build custom AR overlays using React Native components while accessing device camera capabilities. It also allowed for using react native libraries to capture device heading data. Whilst not having advanced AR features, it is able to provide essential navigation functionality through directional overlays and compass-based guidance that meets our core requirements.

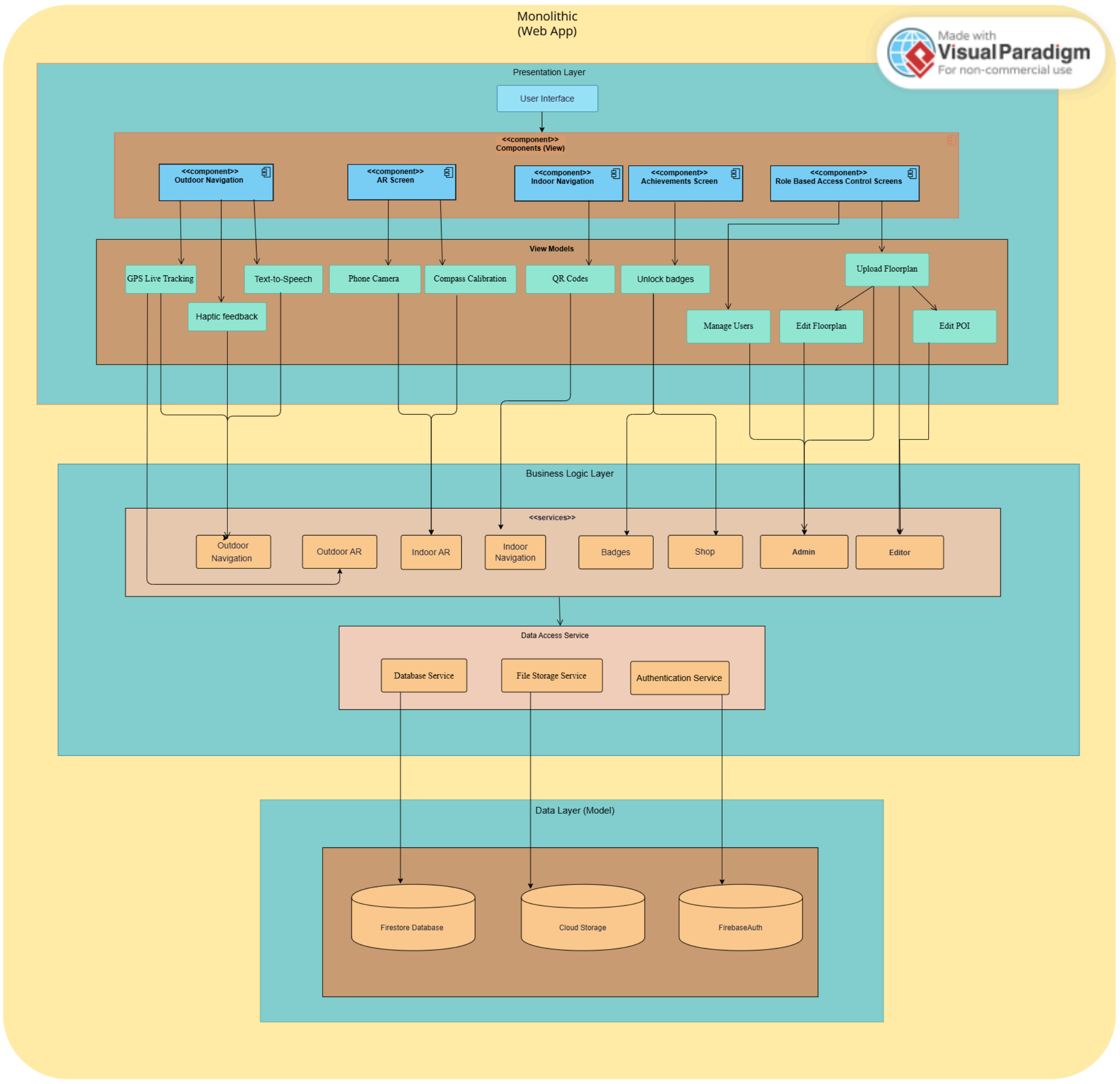
The AR navigation is handled by ARNavigationOverlay which provides:

- Camera-based AR view with directional overlays
- Compass-based bearing calculations
- Fallback non-camera guidance when camera access is unavailable

However, due to the no-GPS-indoor constraint, we also built a QR code fallback system to anchor indoor positioning. This modular fallback strategy supports our separation of concerns design, allowing AR, QR, and 2D views to operate independently.

This solution meets usability, latency, and availability quality requirements while respecting platform limits.

6. Architecture Diagram


Monolithic
(Web App)
Made with Visual Paradigm For non-commercial use
Presentation Layer
User Interface
<<component>>
Components (View)
<<component>>
Outdoor Navigation
<<component>>
AR Screen
<<component>>
Indoor Navigation
<<component>>
Achievements Screen
<<component>>
Role Based Access Control Screens
View Models
GPS Live Tracking
Text-to-Speech
Phone Camera
Compass Calibration
QR Codes
Unlock badges
Upload Floorplan
Haptic feedback
Manage Users
Edit Floorplan
Edit POI
Business Logic Layer
<<services>>
Outdoor Navigation
Outdoor AR
Indoor AR
Indoor Navigation
Badges
Shop
Admin
Editor
Data Access Service
Database Service
File Storage Service
Authentication Service
Data Layer (Model)
Firestore Database
Cloud Storage
FirebaseAuth