ReAlign MVP – Technical Architecture Overview

Version: 1.8

Date: May 21, 2025

🧱 Overview

ReAlign is a mobile-first communication platform for short sale transactions. It focuses on fast, trackable document collection, flat threaded messaging, and clear visibility into bottlenecks. The architecture prioritizes simplicity, stability, and security with a clear separation of frontend, backend, and auth responsibilities. This document outlines the core architectural decisions, technology stack, and overarching strategies.

⚙️ Core Principles

* Simplicity: Choose straightforward solutions and avoid over-engineering, especially for the MVP.
* Stability: Leverage proven technologies and patterns to ensure a reliable platform.
* Security by Design: Integrate security considerations into the architecture foundation, with detailed policies outlined in the "ReAlign MVP – Security & Privacy Plan."
* Scalability (MVP Focus): While building for the MVP, make architectural choices that do not hinder future scalability.
* Developer Experience: Maintain clear documentation, consistent coding standards, and a streamlined development process.

🚀 Technology Stack

* Frontend:
  + Framework: React 18.x + Vite 5.x
  + UI Library: TailwindCSS 3.x with customized theme for ReAlign brand tokens
  + Component Toolkit: shadcn/ui for consistent, accessible components
  + Icons: lucide-react
  + Animation: framer-motion for light animation and transitions
  + Routing: Vite Router with lazy-loaded screens
  + State Management: React Context for session and user role; useState/useReducer for local state. No global state library in MVP.
* Backend:
  + Platform: Node.js 20.x + Express.js 4.x
  + ORM: Drizzle ORM (latest) with PostgreSQL 15.x
  + Validation: Zod for input validation and DTO typing
* Platform & Services:
  + Authentication: Supabase Auth
  + Database: Supabase PostgreSQL
  + Storage: Supabase Storage for file handling
  + Push Notifications: A third-party service (e.g., Firebase Cloud Messaging - FCM, or equivalent) will be integrated for sending device push notifications for key alerts.
* Development & Operations:
  + Code Style: Prettier
  + Linting: ESLint

🖥️ Frontend Architecture

* Framework & Build: React + Vite ensures a fast development experience and optimized builds.
* UI:
  + TailwindCSS is used for utility-first styling, customized with ReAlign brand tokens.
  + shadcn/ui provides a set of accessible and reusable components.
  + lucide-react for consistent iconography.
  + framer-motion for subtle animations and transitions enhancing UX.
* Routing:
  + Vite Router is used for client-side routing with support for lazy-loaded screens to improve initial load time.
  + Layouts are mobile-first, utilizing CSS grid and flex.
* State Management:
  + React Context is employed for managing global aspects like session data and user roles.
  + Local component state is managed using useState and useReducer.
* Authentication Integration:
  + Supabase Auth handles user authentication.
  + Email/password for negotiators; Magic links for all other roles.
  + Supabase handles JWT-based session creation.
  + Token auto-refresh is disabled in MVP. Session expiry redirects negotiators to login.
* Push Notification Integration:
  + The frontend will be responsible for requesting user permission for push notifications via browser/OS prompts.
  + Upon successful permission grant, the frontend will obtain the device registration token and send it to the backend to be associated with the user.
* Access Control (UI):
  + Frontend: useRoleAccess() hook for conditional UI logic.

🌐 Backend Architecture

* Platform: Node.js with Express.js provides a robust framework for the REST API.
* API Design Principles:
  + RESTful API under /api/v1.
  + Modular route structure (auth, transactions, uploads, messages, doc-requests).
  + Stateless request handling.
  + Standard HTTP response codes and a consistent error response format.
* Database & ORM:
  + Drizzle ORM with PostgreSQL, using a code-first schema and typed migrations.
* Input Validation:
  + Zod is used for all incoming API data validation and for typing DTOs.
* File Handling:
  + Supabase Storage with signed URL upload.
  + Metadata includes: userId, docType, visibility.
  + Client-side compression before upload; backend enforces max size limit (10MB post-compression).
* Messaging Structure:
  + Messages stored flat (no recursion).
  + Each message stores: senderId, text, optional replyTo, timestamp.
* Document Requests Logic:
  + Only negotiators can create requests from a hardcoded preset list.
  + Request table stores docType, assigned userId, transactionId, dueDate, status, and timestamps. (Note: revision\_note confirmed in DB schema document ).
* Push Notification Handling:
  + The backend will store and manage device registration tokens received from the frontend, associating them with specific users.
  + The "Unified Notification Service" (detailed in Notification Rules) will be responsible for constructing and triggering push notifications via an integrated third-party service (e.g., FCM) for events like document requests, reminders, and revision needs.
* Access Control (API):
  + Express middleware layer enforces route-level role access and user scoping.

🛠️ Development & Operations

* Environment Management:
  + .env files will be used for managing environment-specific variables (API keys, database connection strings, frontend URLs).
  + A .env.example file will be maintained in the repository.
  + Platform-specific environment variable management will be used for deployed environments (e.g., Supabase project settings, Vercel environment settings).
* Code Quality:
  + ESLint and Prettier will be configured for consistent code formatting and to catch potential errors early. Shared configurations will be part of the repository.
* Build & Deployment Strategy (High-Level):
  + Frontend: Built using Vite (npm run build). Intended for deployment to a static hosting provider (e.g., Vercel, Netlify).
  + Backend: Express application. Deployment options include Supabase Functions for serverless endpoints or containerizing (e.g., Docker) for cloud platform deployment.
  + CI/CD pipelines (e.g., GitHub Actions) are recommended for automated testing and deployment.
* Error Handling & Logging Strategy (Architectural):
  + Frontend: Design for graceful error display to users. Client-side errors may be reported to a logging service.
  + Backend: A centralized error handling middleware in Express.js is used. Comprehensive logging (e.g., using Winston or Pino) directed to Supabase Logs or a dedicated third-party logging service.

Future Considerations (Post-MVP)

* Global state management library (if application complexity grows significantly)
* Real-time updates beyond targeted push notifications (e.g., WebSockets for live messaging or full in-app feed real-time updates)
* Advanced analytics and reporting
* Full E2E testing automation
* Internationalization (i18n)