# ARISE — System Architecture

## Architecture Diagram (Layers)

```
┌─────────────────────────────────────────────────────────────┐
│                     USER (Browser)                          │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  ┌──────────────────────────────────────────────────────┐   │
│  │         UI LAYER (React + TypeScript)                │   │
│  ├──────────────────────────────────────────────────────┤   │
│  │ Auth Pages │ Dashboard │ Module Cards │ Forms │      │   │
│  │ Timeline   │ Vault     │ Log Progress │       │      │   │
│  └──────────────────────────────────────────────────────┘   │
│                        ▼                                    │
│  ┌──────────────────────────────────────────────────────┐   │
│  │  STATE LAYER (Zustand + Context API)                 │   │
│  │  - Global: user, auth session                        │   │
│  │  - Local: modals, filters, loading states            │   │
│  └──────────────────────────────────────────────────────┘   │
│                        ▼                                    │
│  ┌──────────────────────────────────────────────────────┐   │
│  │  SUPABASE CLIENT (HTTPS)                             │   │
│  │  - Auth API                                          │   │
│  │  - PostgREST API (CRUD)                              │   │
│  │  - Storage API (file upload/download)                │   │
│  └──────────────────────────────────────────────────────┘   │
└─────────────────────────────────────────────────────────────┘
                         ▼
┌─────────────────────────────────────────────────────────────┐
│          BACKEND SERVICES (Supabase)                        │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  ┌──────────────────────────────────────────────────────┐   │
│  │  Auth Service (email/password)                       │   │
│  │  - JWT tokens, session mgmt, RLS enforcement         │   │
│  └──────────────────────────────────────────────────────┘   │
│                                                             │
│  ┌──────────────────────────────────────────────────────┐   │
│  │  PostgreSQL Database                                 │   │
│  │  - items | timeline_entries | vault_files |          │   │
│  │  - profiles | modules                                │   │
│  │  - Row-Level Security policies (auth.uid check)      │   │
│  └──────────────────────────────────────────────────────┘   │
│                                                             │
│  ┌──────────────────────────────────────────────────────┐   │
│  │  Storage Service (S3-compatible)                     │   │
│  │  - vault_uploads bucket                              │   │
│  │  - Private access (auth required)                    │   │
│  └──────────────────────────────────────────────────────┘   │
│                                                             │
└─────────────────────────────────────────────────────────────┘
                         ▼
```

```
┌──────────────────────────────────────────────────────────────┐
│          HOSTING                                             │
├──────────────────────────────────────────────────────────────┤
│  Vercel (SPA distribution) │ Supabase (API + Storage)       │
└──────────────────────────────────────────────────────────────┘
```

## Core Components & Responsibilities

### Frontend Components

Component	Responsibility
`<AuthLayout>`	Sign up / sign in / password reset forms
`<Dashboard>`	Main view; renders 5 module cards; handles module toggle/reorder
`<ModuleCard>`	Displays module summary + quick-add button; collapse/expand
`<ItemEditor>`	Modal form for adding/editing items; adapts fields per `module_type`
`<ItemList>`	Displays items in a module; supports sorting/filtering
`<LogProgressModal>`	Hero flow: date (auto), short note, optional section link → saves to timeline + updates item
`<TimelineView>`	Global timeline list; filter by module_type and item_id; pagination support
`<VaultUpload>`	Drag-and-drop or file picker; submits to Supabase Storage + metadata DB
`<VaultList>`	Lists vault files; download/delete actions
`<ProtectedRoute>`	Guards routes; redirect to login if not authenticated

### Data Access Layer

Service	Responsibility
`supabaseClient`	Singleton Supabase client; Auth + PostgREST + Storage
`itemsAPI`	CRUD operations on `items` table (module-agnostic)
`timelineAPI`	CRUD on `timeline_entries`; supports filters
`vaultAPI`	Storage upload/download + `vault_files` metadata management
`authAPI`	Sign up, sign in, sign out, session refresh

### State Management

Scope	Tool	State
Global	Zustand store	user, session, currentUserTheme (future)
Local	React Context	modal visibility, form state, loading flags

---

## Data Flow (Key Scenarios)

### Flow 1: User Login
```
1. User enters email + password → Auth form
2. Form submits → authAPI.login() → Supabase Auth
3. Supabase returns session + JWT token
4. JWT stored in browser (via Supabase client auto-management)
5. Zustand store updated with user data
6. Dashboard renders; fetch user's items/timeline/vault
7. User redirected to Dashboard
```

### Flow 2: Create New Project Item
```
1. User clicks "+" on Projects module
2. ItemEditor modal opens (fields: title, status, next_action, notes)
3. User fills form + clicks Save
4. itemsAPI.create({ module_type: 'projects', title, status, ... }) → PostgREST
5. Supabase RLS policy checks auth.uid() = user_id → ✅ allowed
6. Row inserted into `items` table
7. UI refreshes; new item appears in Projects list
8. Toast confirmation shown
```

### Flow 3: Log Progress (Create Timeline Entry + Update Item)
```
1. User clicks primary "Log Progress" button
2. LogProgressModal opens (date auto-filled, short note field, optional section link)
3. User types note + optionally selects linked item (e.g., "On Project: Beta Launch")
4. User clicks "Log"
5. System creates:
   a. timelineAPI.create({ occurred_at: now, entry_text: note, item_id: selected, module_type: 'arise_log' })
   b. itemsAPI.update(item_id, { last_activity_at: now })
6. RLS policies check user_id matches → ✅ allowed
7. Both rows inserted/updated
8. Dashboard refreshes; timeline shows new entry; item's "last worked" updates
9. Toast confirmation
```

### Flow 4: Upload File to Vault
```
1. User clicks "Vault" module → sees upload area
2. User drags PDF/JPG/PNG/DOCX file
3. File picker validates type
4. Form collects: category tag, optional link to item
5. User clicks "Upload"
6. vaultAPI.upload(file, metadata) → Supabase Storage + metadata insert
7. Storage path: /vault_uploads/{user_id}/{file_id}_{original_name}
```

8. `vault_files` row inserted with metadata
9. RLS policies enforce user_id check → ✅ private to user
10. File appears in Vault list with download button
```

### Flow 5: View Global Timeline with Filters
```
1. User scrolls to Timeline section or clicks dedicated Timeline view
2. UI fetches timelineAPI.list({ user_id, limit: 50 }) (ordered by occurred_at DESC)
3. Supabase RLS policy restricts to current user's rows only
4. Timeline renders list of entries (newest first)
5. User selects filter (e.g., "Projects only")
6. Client-side filter applies → timelineAPI.list({ user_id, module_type: 'projects', limit: 50 })
7. Timeline re-renders filtered entries
8. User clicks entry → optionally jump to related item in module card (if item_id present)
~~~

Authentication & Authorization

Authentication (Email/Password via Supabase Auth)

Sign Up:
- User submits email + password.
- Supabase Auth validates: email uniqueness, password strength (min 6 chars default).
- User receives confirmation email (or auto-confirmed if email verification disabled).
- On confirmation, account activated; JWT issued.

Sign In:
- User submits email + password.
- Supabase Auth validates credentials.
- Returns JWT token + session cookie (httpOnly, Secure, SameSite).
- Token stored in browser; automatically refreshed by Supabase client when expired.

Session Management:
- Supabase client library handles token refresh transparently.
- On logout, tokens cleared; `auth.uid()` becomes null.

Authorization (Row-Level Security)

Every table enforces RLS policies:

```sql
-- Example for items table
CREATE POLICY user_items_access ON items
  FOR ALL
  USING (auth.uid() = user_id)
~~~

```sql
  WITH CHECK (auth.uid() = user_id);

-- Example for timeline_entries
CREATE POLICY user_timeline_access ON timeline_entries
  FOR ALL
  USING (auth.uid() = user_id);

-- Example for vault_files
CREATE POLICY user_vault_access ON vault_files
  FOR ALL
  USING (auth.uid() = user_id);
```

Result: User cannot query/update/delete any row where `user_id ≠ auth.uid()`. Even if they craft a malicious API request, the database rejects it.

## Database Schema (Detailed)

### Table: `profiles`
```sql
CREATE TABLE profiles (
  id UUID PRIMARY KEY DEFAULT auth.uid(),
  email TEXT NOT NULL (from auth.users),
  created_at TIMESTAMPTZ DEFAULT now(),
  updated_at TIMESTAMPTZ DEFAULT now()
);
```

### Table: `modules` (future; for now, hardcoded module list in frontend)
```sql
CREATE TABLE modules (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID NOT NULL REFERENCES auth.users(id) ON DELETE CASCADE,
  module_type TEXT NOT NULL (enum: 'projects' | 'money_moves' | 'skills'
| 'vault' | 'arise_log'),
  title TEXT NOT NULL,
  order_index INT DEFAULT 0,
  is_collapsed BOOLEAN DEFAULT FALSE,
  created_at TIMESTAMPTZ DEFAULT now(),
  updated_at TIMESTAMPTZ DEFAULT now(),
  UNIQUE(user_id, module_type)
);
```

### Table: `items` (Shared table with module_type + flexible fields)
```sql
CREATE TABLE items (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID NOT NULL REFERENCES auth.users(id) ON DELETE CASCADE,
  module_type TEXT NOT NULL (enum: 'projects' | 'money_moves' |
'skills'),
```

~~~~
  -- Universal fields
  title TEXT NOT NULL,
  notes TEXT,

  -- Projects-specific
  status TEXT (enum: 'active' | 'paused' | 'done', nullable),
  next_action TEXT,

  -- Money Moves-specific
  type TEXT (enum: 'internship' | 'hackathon', nullable),
  deadline_at TIMESTAMPTZ,
  prep_status TEXT,
  outcome TEXT,

  -- Skills-specific
  current_level TEXT (1-10 scale or custom, nullable),
  last_practiced_at TIMESTAMPTZ,

  -- Cross-module tracking
  last_activity_at TIMESTAMPTZ DEFAULT now(),

  created_at TIMESTAMPTZ DEFAULT now(),
  updated_at TIMESTAMPTZ DEFAULT now()
);
```

### Table: `timeline_entries`
```sql
CREATE TABLE timeline_entries (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID NOT NULL REFERENCES auth.users(id) ON DELETE CASCADE,

  occurred_at TIMESTAMPTZ NOT NULL,
  entry_text TEXT NOT NULL,
  module_type TEXT NOT NULL (enum: 'projects' | 'money_moves' | 'skills'
| 'vault' | 'arise_log'),

  -- Link to parent item (if applicable)
  item_id UUID REFERENCES items(id) ON DELETE SET NULL,

  -- Milestone marker
  is_milestone BOOLEAN DEFAULT FALSE,

  created_at TIMESTAMPTZ DEFAULT now()
);
```

### Table: `vault_files`
```sql
CREATE TABLE vault_files (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID NOT NULL REFERENCES auth.users(id) ON DELETE CASCADE,
~~~~

```
  -- Storage reference
  storage_bucket TEXT NOT NULL DEFAULT 'vault_uploads',
  storage_path TEXT NOT NULL,

  -- Metadata
  file_name TEXT NOT NULL,
  file_type TEXT NOT NULL (e.g., 'application/pdf', 'image/png'),
  file_size BIGINT NOT NULL,

  -- Categorization
  category TEXT (e.g., 'certification', 'offer_letter', 'notes'),
  related_module_type TEXT REFERENCES modules(module_type),
  related_item_id UUID REFERENCES items(id) ON DELETE SET NULL,

  uploaded_at TIMESTAMPTZ DEFAULT now()
);
```

## API Endpoints (Detailed)

### Items Endpoints (via Supabase PostgREST)

```
GET    /rest/v1/items?user_id=eq.{user_id}&module_type=eq.projects
       → Fetch all projects for current user

POST   /rest/v1/items
       Body: { user_id, module_type, title, status, next_action, notes,
... }
       → Create new item

PATCH  /rest/v1/items?id=eq.{id}
       Body: { status, next_action, last_activity_at, ... }
       → Update item (only user_id owner can)

DELETE /rest/v1/items?id=eq.{id}
       → Delete item (only user_id owner can)
```

### Timeline Endpoints

```
GET
/rest/v1/timeline_entries?user_id=eq.{user_id}&order=occurred_at.desc
       → Fetch timeline entries (newest first)

GET
/rest/v1/timeline_entries?user_id=eq.{user_id}&module_type=eq.projects
       → Fetch filtered timeline (e.g., Projects only)

POST   /rest/v1/timeline_entries
```

```
       Body: { user_id, occurred_at, entry_text, item_id, module_type,
is_milestone }
       → Create timeline entry

DELETE /rest/v1/timeline_entries?id=eq.{id}
       → Delete entry (owner only)
```

### Vault/Storage Endpoints

```
POST   /storage/v1/object/vault_uploads/upload
       Multipart form: file + metadata
       → Upload file; returns storage_path

GET    /storage/v1/object/vault_uploads/{storage_path}
       → Download file (public read if policy allows; private by default)

DELETE /storage/v1/object/vault_uploads/{storage_path}
       → Delete file from storage + remove metadata row

GET    /rest/v1/vault_files?user_id=eq.{user_id}
       → Fetch all vault metadata for user
```

### Auth Endpoints

```
POST   /auth/v1/signup
       Body: { email, password }
       → Create new user account

POST   /auth/v1/token?grant_type=password
       Body: { email, password }
       → Login; return JWT + session

POST   /auth/v1/logout
       → Clear session/token

GET    /auth/v1/user
       → Get current user info (requires valid JWT)

POST   /auth/v1/token?grant_type=refresh_token
       → Refresh expired JWT
```

## System Non-Negotiables (CRITICAL)

These are core principles that must NEVER be violated, even during bug fixes or refactoring:

### 1. **Row-Level Security Everywhere**

- Every table (`items`, `timeline_entries`, `vault_files`, `profiles`) MUST have an RLS policy enforcing `auth.uid() = user_id`.
- No exceptions; no bypass.
- If a bug occurs, fix it within the RLS model, not by disabling RLS.

### 2. **Low-Friction Core Flows**
- "Log Progress" must be ≤ 2 clicks from dashboard.
- Item add/edit must require ≤ 5 form fields visible (optional fields can be collapsed).
- No deep navigation (max 3 levels).
- If a feature feels "heavy," question if it belongs in V1.

### 3. **Not a To-Do App**
- No task scheduling, no priority systems, no sub-tasks, no recurring reminders.
- No "mark done" workflow pressure; just tracking progress.
- Timeline is read-only (history, not planning).

### 4. **No UI Clutter**
- Max 5 module cards visible at once (collapse/scroll if more added later).
- No dashboard widgets, no stats/metrics, no gamification (badges, streaks).
- Every element on screen must serve clarity or low-friction interaction.

### 5. **Single `items` Table Design**
- All projects, money moves, and skills go into one `items` table with `module_type` discriminator.
- This enables future cross-module filtering and analytics without schema refactors.
- Do not create separate tables (e.g., `projects`, `skills`) unless forced by business logic.

### 6. **Privacy by Default**
- All data is private to the user; no sharing, no public profiles in V1.
- Vault files are private; download only.
- If a feature needs to access another user's data, that's a new scope decision (not V1).

## Error Handling Strategy

### Frontend Error Handling
1. **Form validation:** Show inline errors (required fields, email format, file type).
2. **API errors:** Catch 4xx/5xx responses; show user-friendly toast ("Something went wrong. Please try again.").
3. **Auth errors:** If JWT expires during a request, refresh silently (Supabase client handles); if refresh fails, redirect to login.
4. **File upload errors:** Validate file size (< 100MB), type (PDF/JPG/PNG/DOCX only), before upload.

### Server-Side Error Handling (Supabase)

1. **RLS violations:** Request denied at database level; API returns 403 Forbidden.
2. **Invalid data:** Postgres constraints trigger; API returns 400 Bad Request.
3. **Auth token invalid:** API returns 401 Unauthorized.
4. **Storage quota exceeded:** API returns 413 Payload Too Large or 429 Too Many Requests.

### Error Logging (V1)
- Log errors to browser console during dev.
- Optionally add a `client_logs` table later to track production errors.
- Monitor Supabase dashboard for policy violations, quota alerts.

## Caching Strategy (V1)

### Client-side Caching
- **Approach:** Lightweight React state + optional React Query for fetch caching.
- **Items list:** Cache in state; refetch on "save" or manual refresh.
- **Timeline:** Paginate or virtualize if > 100 entries (don't fetch all at once).
- **Vault metadata:** Cache list; refetch after upload/delete.

### No server-side caching for V1.
- Supabase PostgREST handles query optimization.
- Add Redis caching only if analytics/summaries become a bottleneck (future).

## External Integrations (V1)

- **Email service:** Supabase Auth handles sign-up confirmations and password resets (via SendGrid or Supabase's default).
- **File storage:** Supabase Storage (S3-compatible); can migrate to AWS S3 later.
- **Analytics:** Optional; start with Vercel analytics.
- **Monitoring:** Vercel logs, Supabase dashboard.

## Performance Optimizations (V1)

1. **Code splitting:** React Router lazy-load dashboard, timeline, vault routes.
2. **Image/file handling:** Compress Vault uploads client-side before sending (sharp.js or similar).
3. **Pagination:** Timeline paginate or virtualize if > 100 entries.
4. **Build optimization:** Tailwind CSS tree-shaking removes unused classes; Vite minifies.

## Monitoring & Logging (V1)

### Frontend Monitoring
- Vercel Analytics: track page load, interaction latency.
- Console errors: log to browser dev tools during dev.

### Backend Monitoring
- Supabase dashboard: view request counts, errors, quota usage.
- Database logs: monitor slow queries, RLS violations.

### Logging Strategy (upgrade as needed)
- V1: Console logs + Supabase logs (free).
- V1.5: Add structured logging table (`client_logs`) if needed.
- V2+: Integrate error tracking service (Sentry, LogRocket).

## Key Architectural Decisions

Decision	Rationale
Shared `items` table with `module_type`	Enables future cross-module features without schema refactor
Supabase RLS (not custom auth backend)	Simpler, less code, scales automatically
Client-side state only (Zustand + Context)	No need for complex server state in V1
Vercel hosting	Optimal for React SPAs; auto-deployments from GitHub
No serverless functions (yet)	Supabase PostgREST + Storage sufficient for CRUD
Email/password only (no OAuth/social)	Simpler for personal use; add later if needed