HLD for Professional Client Portal for Document Delivery

This document describes the high-level architecture of the Professional Client Portal for Document Delivery using Mermaid diagrams. It aligns with the PRD and Roadmap.

Revision: MVP · Last updated with current implementation (file browser, Drive picker, members, invitations).

Design Principles

Principle	Description
Direct-to-Drive	File bytes go browser → Google Drive (resumable upload). Portal servers never store or proxy file content; only metadata and upload URLs.
Multi-tenant	All data is scoped by Organization. Clients and Projects belong to an org; access is enforced via roles (ORG_OWNER, ORG_MEMBER, ORG_GUEST) and project personas.
Headless Drive	Portal is the UI; Google Drive is the storage backend. Folder structure is created at project creation; file list and uploads use Drive API and Picker.
Session-first auth	Supabase handles Google OAuth and session. API routes validate session and resolve org/client/project context; connector tokens are used for Drive API on behalf of the org.

Technology Stack

Layer	Technology
Frontend	Next.js (App Router), React, TypeScript
API	Next.js Route Handlers (REST), Server Actions where applicable
Database	PostgreSQL (Supavisor), Prisma ORM
Auth	Supabase Auth (Google OAuth)
Storage / Files	Google Drive API (metadata + resumable uploads), no Portal file storage
Hosting	Vercel (Next.js build + postbuild Prisma migrate deploy)
Observability	Sentry (client, server, edge), structured logging

---

Database Schema Organization

The database uses multiple PostgreSQL schemas to separate user-facing application data from administrative data:

Schema	Purpose	Tables
`portal`	User-facing application data	`organizations`, `clients`, `projects`, `organization_members`, `roles`, `role_permissions`, `connectors`, `documents`, `linked_files`, `project_members`, `project_personas`, `project_invitations`, `customer_requests`
`admin`	Administrative/internal data	`contact_submissions`
`public`	Default schema (minimal use)	Reserved for PostgreSQL system objects

Rationale:

- **Portal schema**: Contains all tables that users interact with directly. This includes support requests (`customer_requests`) so users can view their own tickets.
- **Admin schema**: Contains tables used only by internal administrators (e.g., contact form submissions from the marketing site).
- **Separation benefits**: Clear data boundaries, simplified access control, and easier compliance auditing.

All enums (`ConnectorType` , `ConnectorStatus` , `DocumentStatus` , `InvitationStatus` , `TicketType` , `TicketStatus`) are created in the `portal` schema since they are used by portal tables.

URL Structure

Main application routes:

Route	Purpose
`/onboarding`	New user workspace creation (no org yet)
`/dash`	Dashboard; redirects to last-used client workspace
`/o/[slug]`	Organization scope (e.g. org home, connectors, insights)
`/o/[slug]/c/[clientSlug]`	Client scope; project list
`/o/[slug]/c/[clientSlug]/p/[projectSlug]`	Project workspace (Files, Members, Shares, Insights, Sources tabs)
`/invite/[token]`	Invitation redemption (sign-in/sign-up → project)

Slugs are URL-friendly (org, client, project names). IDs are used in API and DB.

Key API Surface

Main API route groups used by the app (Next.js Route Handlers under `app/api/`). LLD should specify request/response schemas, auth, and error handling per endpoint.

Route group	Purpose
`POST /api/organization/create`	Create organization (onboarding).
`GET/POST /api/organization`, `GET /api/organizations`	Org CRUD and list.
`GET /api/connectors/google-drive?action=token`	Get Google access token for Picker/Drive API.
`POST /api/connectors/google-drive/upload`	Init resumable upload (returns Drive upload URL); no file body.
`POST /api/connectors/google-drive/import`	Process Import from Drive (copy/shortcut selected files).
`GET /api/connectors/google-drive/callback`	OAuth callback for Google Drive connector.
`GET /api/documents/download`	Proxy or signed URL for document download.
`GET /api/drive-summary`, `GET /api/drive-metrics`, `POST /api/drive-action`	Drive summary, metrics, and actions (e.g. list children).
`POST /api/provision`	Provision project + Drive folder (project creation).

All authenticated routes expect `Authorization: Bearer <session.access_token>`. Org/client/project context is derived from request or path.

Key Frontend Anchors

Main UI entry points and components that LLD can break down into subcomponents, state, and API calls.

Area	Entry / key components
Onboarding	`app/onboarding/page.tsx` — workspace name, slug, create org.
Dashboard	`app/dash/` — redirect to last client; sidebar layout.
Org / Client / Project	`app/o/[slug]/layout.tsx`, `c/[clientSlug]/page.tsx`, `p/[projectSlug]/page.tsx` — hierarchy; `ProjectWorkspace` with tabs.
Project Files	`ProjectFileList` — file table, breadcrumbs, Add menu, filters, sort, upload queue, Import from Drive, row actions.
Project Members	`ProjectMembersTab` — member list, invite modal, persona assignment.
Connectors	`app/o/[slug]/connectors/page.tsx` — Google Drive connect, link folders.
Invitation	`app/invite/[token]/page.tsx` — redeem invite, sign-in/sign-up, join project.

Security & Compliance

Authentication & Authorization

- **Authentication:** Supabase session (JWT). All API routes that need auth validate the session and resolve the user.
- **Authorization:** Org and project membership plus roles (ORG_OWNER, ORG_MEMBER, ORG_GUEST) and project personas. Invitation flow ensures invitee email matches authenticated user (no link forwarding).
- **Data scope:** Queries are scoped by `organizationId` (and client/project where applicable) in application code. Connector tokens are org-scoped; Drive folder IDs are stored per project. **Database:** Row-Level Security (RLS) — see Data & PII Protection.

Direct-to-Drive Upload Security

File content never transits or persists on Portal servers. Prescribed approach:

Measure	Prescribed approach
Transport	Enforce TLS 1.2+ and HSTS in production. Browser → Google Drive over HTTPS; Portal API only returns a resumable upload URL.
Server-side file handling	Do not store or proxy file bytes. Do not introduce server-side buffering of file content.
Upload URL lifecycle	Short-lived upload URLs (e.g. 1-hour expiry); no reuse. Revoke or scope URLs to a single session/request.
Token handling	Store tokens in server-side secrets or vault; never log or expose in responses. Rotate tokens on revoke. Use connector OAuth tokens server-side only to obtain upload URL; do not send tokens to the browser.
Validation	Validate project membership and folder ownership before issuing upload URL; rate-limit per user/org. Scope upload URL to the project's Drive folder.

Audit	Log upload events (user, org, project, folderId, timestamp) for compliance and forensics; do not log file content.

Data & PII Protection

Area	Prescribed approach
Row-Level Security (RLS)	Enable PostgreSQL RLS per table as below. Use `current_setting('app.current_org_id')` and, for project-scoped tables, project-membership checks. See **RLS multi-tenancy strategy** below.
Encryption in transit	TLS 1.2+ everywhere; DB connection over TLS (Supavisor supports this).
Encryption at rest (DB)	Confirm provider uses AES-256 or equivalent; use encrypted backups.
PII in database	Field-level or column-level encryption for sensitive PII (e.g. email, display name in invitations/members). Use a KMS or env-based key and encrypt before write, decrypt in app layer. Key rotation without re-encrypting all data (e.g. envelope encryption) as roadmap.
Secrets	Secrets in a vault (e.g. Vercel env, Doppler, AWS Secrets Manager); no secrets in code or logs.
Logging	Redact or omit PII from logs (email, names, tokens). Use placeholders (e.g. `user_id=xxx`) for debugging.
Retention	Define retention for PII and audit logs; automated purge or archive per policy and jurisdiction (e.g. GDPR).

Encryption: what to encrypt and how

Advice only (no code changes). Which data in the DB should be encrypted, and how to do it technically.

Category	What	Should encrypt?	How (technical)
PII	User-identifying data: email (invitees, members, connectors), display names if stored. **Exclude:** `contact_submissions` (for Portal admins; do not encrypt).	**Yes** for enterprise/compliance.	**Application-level (field/column) encryption** with a KMS (e.g. AWS KMS, HashiCorp Vault). Encrypt before write, decrypt in app layer after read. Use envelope encryption: data key encrypted by KMS key; rotate KMS key without re-encrypting all rows. Store ciphertext in existing columns or dedicated columns; avoid indexing encrypted values for search (or use deterministic encryption for equality-only search with higher leakage risk).
Business data of Portal clients	Client name, project name, document title; you noted you only store fileId for Drive files (metadata like name comes from Drive API).	**Optional but recommended** for high-sensitivity clients.	Same as PII: application-level encryption with KMS. Encrypt `clients.name`, `projects.name`, `documents.title` (and any other business labels) before write. Decrypt in app for display. Search by name would require application-side decrypt-then-filter or a separate search index with encrypted/hashed tokens.

Invitee / member emails	`project_invitations.email`, `organization_members` (no email column today; user resolved via `userId` from Supabase). Connector `email` in `connectors`.	**Yes, if you want Portal admins (or DB access) to not see them in plaintext.**	Encrypt `project_invitations.email` and `connectors.email` (and any other email columns) with application-level encryption + KMS. Then only the app (with access to the key) can decrypt; DB backups and direct DB access show ciphertext. For invite redemption you must decrypt by token (token is unique) or store a hash for lookup and keep ciphertext for display.

Summary

- **PII:** Encrypt email, display names, and other PII (application-level, KMS, envelope encryption). Do **not** encrypt `contact_submissions` (for Portal admins).
- **Business data:** Encrypt client/project/document names if you want them protected at rest; you already avoid storing file content (only fileId/metadata).
- **Invitee/member emails:** Encrypt so Portal admins (or anyone with DB access) cannot read them directly; use app-level encryption and KMS; design invite flow so redemption works (e.g. lookup by token, decrypt email only in app).

Technical approach (short): One encryption layer that: gets a data key from KMS, encrypts (e.g. AES-256-GCM), stores ciphertext + IV/key id in DB; on read, decrypt in app. No plaintext or keys in logs. Use deterministic encryption only where you need equality search (e.g. email hash), accepting that equal values give equal ciphertext.

What is KMS?

A **Key Management Service (KMS)** is a managed service that creates, stores, and protects **master keys** (Customer Master Keys, CMK). You never get the raw master key; you call the KMS via API to:

- **Generate a data key** — KMS returns a plaintext data key (used once in memory to encrypt your payload) and an encrypted copy of that data key (stored with your ciphertext; the "key id" identifies the CMK used to encrypt it).
- **Decrypt a data key** — You send the encrypted data key; KMS returns the plaintext data key so your app can decrypt the payload.

The master key never leaves the KMS. Encryption/decryption of your actual data (e.g. AES-256-GCM) happens **in your app** using the data key; the KMS only protects the data keys. That pattern is **envelope encryption**: one key (data key) encrypts the data, another key (master key in KMS) encrypts the data key.

KMS design with Vercel + Supabase

In a Vercel (app) + Supabase (DB) deployment:

- **Supabase** does not provide a KMS for application-level field encryption. It gives you encryption at rest (disk) and secure DB credentials; it does not hold "your" keys for encrypting columns. So the KMS is **outside** Supabase.
- **Vercel** runs your app (Next.js API routes, Server Actions). It does not provide a KMS either. Your app must call an **external KMS** (e.g. AWS KMS, Google Cloud KMS, HashiCorp Vault) over the network.

Typical design:

Component	Role
KMS (e.g. AWS KMS, GCP KMS, Vault)	Holds the master key(s). Your app calls it to generate or decrypt data keys. Credentials (e.g. IAM role, service account key, Vault token) are provided via env vars or Vercel integrations.
App (Vercel)	On write: asks KMS for a new data key (or decrypts an existing encrypted data key by key id); encrypts plaintext with the data key (e.g. AES-256-GCM); stores **ciphertext + IV + key id** in

	the DB. On read: loads ciphertext + IV + key id; asks KMS to decrypt the data key; decrypts in app; returns plaintext. Never logs plaintext or keys.
Supabase (Postgres)	Stores only **ciphertext**, **IV**, and **key id** (and optional hash for equality search). No plaintext PII and no keys.

Data flow (write): User input (plaintext) → Vercel → KMS (generate data key; get back plaintext data key + encrypted data key) → App encrypts with data key → App sends ciphertext + IV + key id to Supabase → Supabase stores. The plaintext data key is discarded after use.

Data flow (read): Supabase returns ciphertext + IV + key id → Vercel → KMS (decrypt data key by key id) → App decrypts with data key → Plaintext returned to caller. Again, no plaintext or keys in logs.

Practical options: Use **AWS KMS** (if you use AWS for anything else), **Google Cloud KMS** (if you use GCP), or **HashiCorp Vault** (self-hosted or HCP). Configure credentials in Vercel (e.g. `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_REGION`, and KMS key id) or Vault address + token, and ensure only server-side code (API routes / Server Actions) can access them.

Do you strictly need an external KMS? (Env-based key)

No. You can use a **single secret in Vercel env** (e.g. a base encryption key or a SALT used to derive a key) as the "key source" for your one encryption layer. The same technical approach applies: get the key from env (or derive from env + fixed context), encrypt with AES-256-GCM, store ciphertext + IV in DB; on read, decrypt in app. No plaintext or keys in logs; deterministic encryption only where you need equality search.

What you get with env-based key:

- One encryption layer, no external service. Simpler ops and no KMS cost.
- Strong crypto (AES-256-GCM) and no PII/keys in logs still satisfy many security and compliance expectations.
- For **ISO 27001**, use of cryptography (A.10) can be met by documenting that you use approved algorithms and protect the key (e.g. env var, access control, no logging). You do **not** strictly need an HSM or external KMS for certification, though auditors may prefer key separation and rotation.

What you give up vs external KMS:

- **Key rotation:** Rotating the env key means re-encrypting all existing ciphertext with a new key; with a KMS and envelope encryption, you can rotate the master key without re-encrypting every row.
- **Key separation:** The encryption key lives in the same place as other app config (Vercel env). A KMS keeps the master key in a dedicated, often HSM-backed, service.
- **Auditor preference:** Some auditors or customers expect "keys in a KMS/HSM" for higher-assurance or regulated workloads; an env-based key is easier to question in those contexts.

Recommendation: For MVP and many ISO 27001 / ISO 27701–oriented deployments, **a strong secret in Vercel env is acceptable** if you: (1) use a long, random key (e.g. 256-bit); (2) restrict env access (e.g. Vercel project settings, no client exposure); (3) document key handling and that plaintext/keys are never logged. Introduce an external KMS when you need key rotation without re-encrypting everything or when a customer/auditor requires it.

RLS multi-tenancy strategy

Multi-tenancy is at **organization level** (tenant = Organization). RLS is applied to tables in the `portal` schema. The `admin` schema tables (`contact_submissions`) do not use RLS as they are admin-only.

RLS can and should be applied at **different levels for different tables**:

Level	Tables (in `portal` schema)	Policy basis	Notes
Org-level	`organizations`, `clients`, `projects`, `connectors`,	Restrict by `organization_id = current_setting('app.current_org_id')::uuid`. For `organizations`, `id =`	User may only see rows belonging to the org they are

	`project_personas`, `organization_members`, `customer_requests`	`current_setting('app.current_org_id')::uuid`. For `customer_requests`, filter by `userId` or `organizationId` as appropriate.	acting in. App sets `app.current_org_id` per request (e.g. from path or session).
Project-level (project-membership)	`project_members`, `project_invitations`	Restrict to rows where the user is a member of that project, e.g. `project_id IN (SELECT project_id FROM project_members WHERE user_id = current_setting('app.current_user_id')::uuid)`.	Only project members may see that project's members and invitations. Avoids org-wide visibility of project-scoped data.
Org or project (by model)	`documents`, `linked_files`	If project-scoped: same as project-level. If org-scoped: by `organization_id`.	Align with how the app uses these tables (e.g. `documents` has `organizationId` and optional `projectId`).

Summary: Use **org-level RLS** for org-owned tables in the `portal` schema; use **project-level (membership-based) RLS** for project-scoped tables. Different tables may have RLS at different levels. The `admin` schema is excluded from RLS as it contains admin-only data.

Implementation: The app must set session variables at the start of each request (before any Prisma query): `SET LOCAL app.current_org_id = '<uuid>'; SET LOCAL app.current_user_id = '<uuid>';` (e.g. in middleware or an API wrapper that resolves org and user from the session). These variables apply to queries in the `portal` schema.

Enterprise Best Practices

Practice	Prescribed approach
Access reviews	Periodic review of org/project members and roles; deprovision on leave.
SSO / SAML	Optional SAML/SSO for orgs (e.g. Okta, Azure AD) for enterprise customers.
Audit logging	Immutable audit log for sensitive actions (invite, role change, connector link, project create/delete). Store in append-only store or dedicated audit table with tamper detection.
Backup & DR	Automated DB backups; tested restore; RPO/RTO defined. Document recovery runbook.
Incident response	Runbook for breach or exposure; notification process; post-incident review.
Compliance	Map controls to SOC 2, GDPR, or ISO as needed; document in a security/compliance doc.

Gap analysis (internal)

Gap analysis against ISO 27001 and ISO 27701 is documented separately in **`hld-nfr.md`** (internal; not for customer sharing).

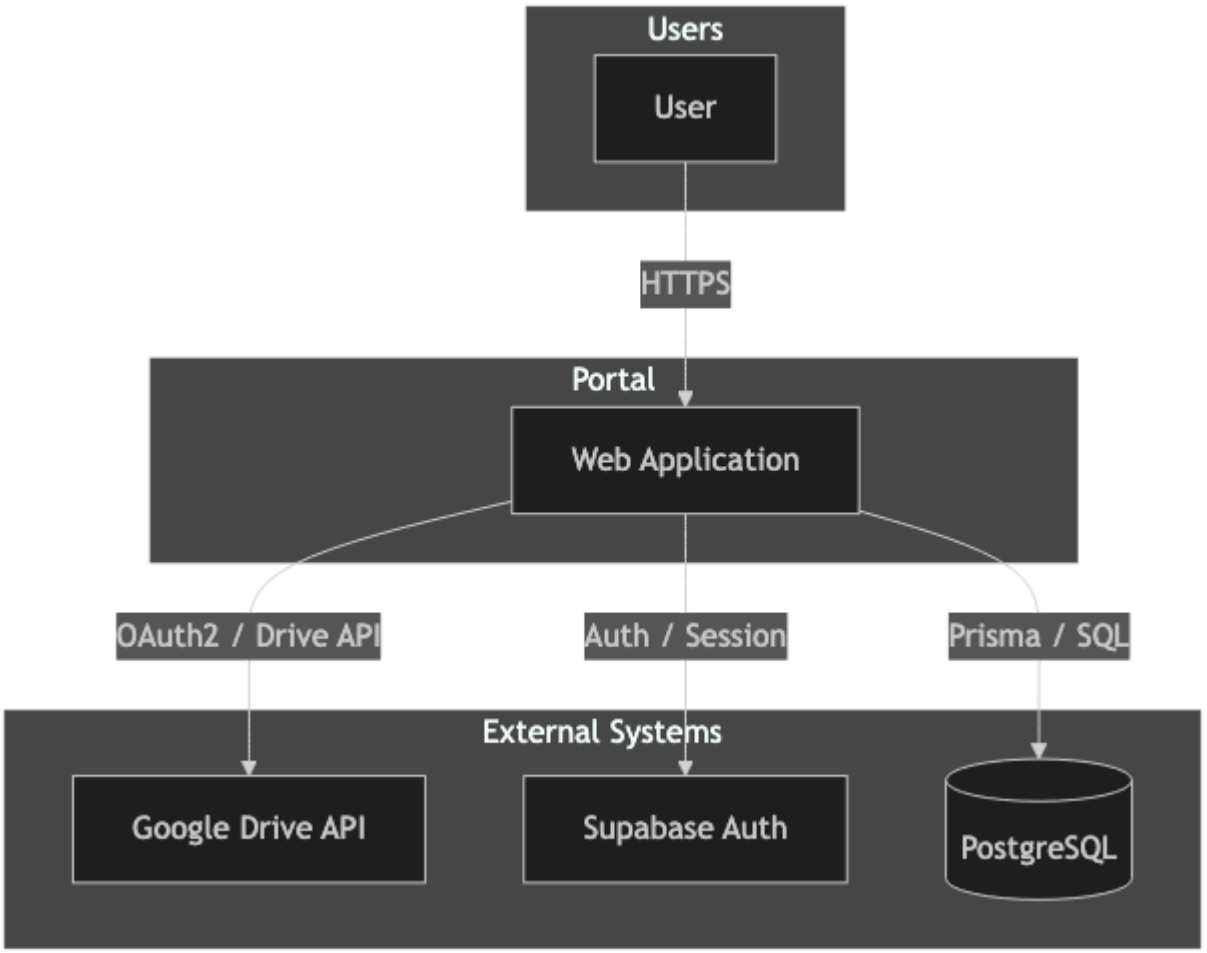
How to view the diagrams (Markdown Preview Mermaid Support)

1. Open this file (`hld.md`) and use **Markdown: Open Preview** (**Cmd+Shift+V** / **Ctrl+Shift+V**) — *not* another extension's preview (e.g. "Markdown Preview Enhanced").
2. **If diagrams disappear after closing and reopening the preview:** Use **Markdown: Open Preview to the Side** (**Cmd+K V** / **Ctrl+K V**) and keep that preview pane open. The side preview tends to re-render Mermaid more reliably than reopening a closed tab. If it still doesn't show diagrams, run **Developer: Reload Window** (Cmd+Shift+P → "Reload Window"), then open preview again.
3. Alternative: copy a `mermaid` code block into mermaid.live or push and view on GitHub.

---

1. System Context (C4 Level 1)

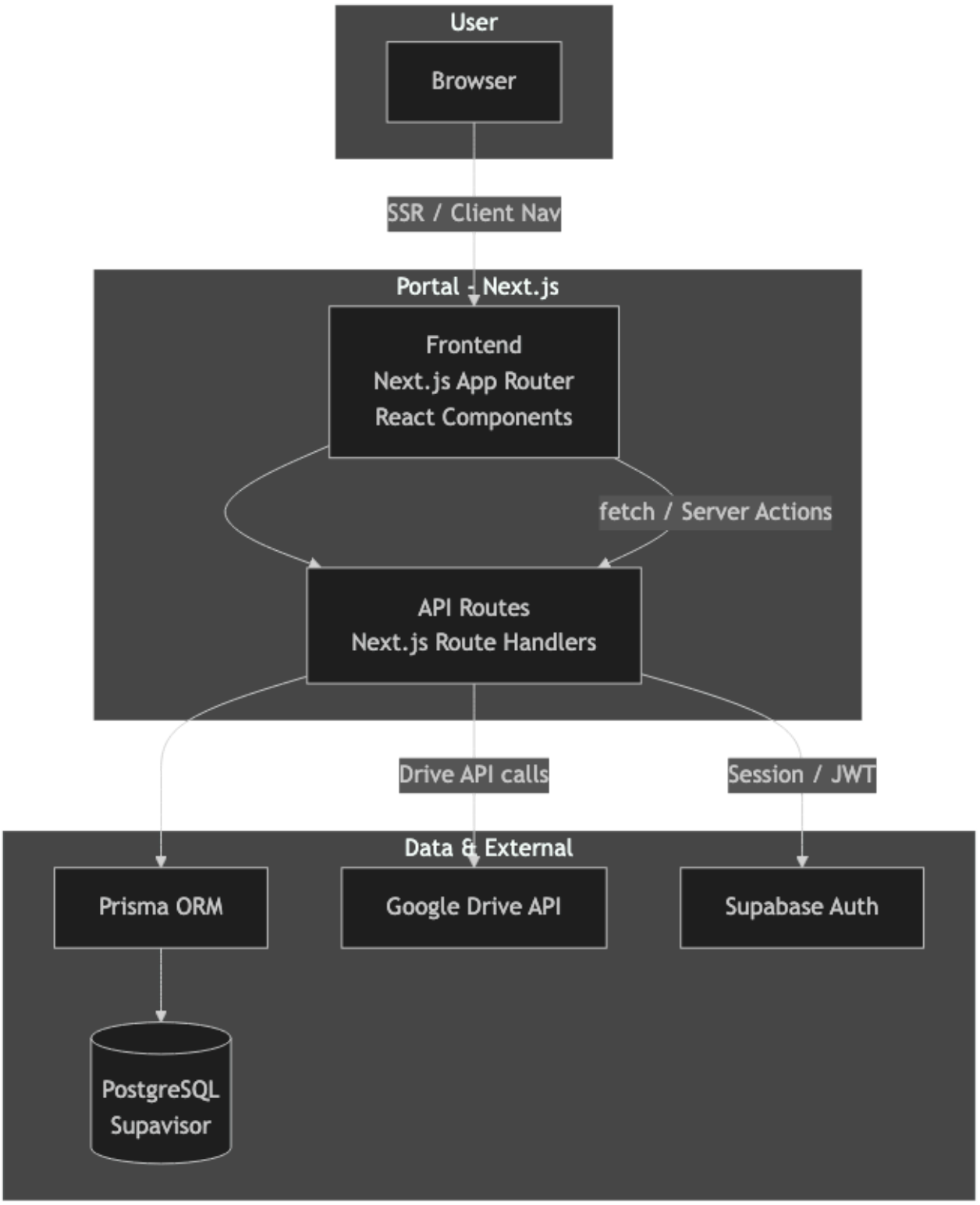
Users interact with the Portal web application. Portal uses Google Drive for file storage, Supabase for authentication, and PostgreSQL (Supavisor) for application data.



---

2. Container Diagram (C4 Level 2)

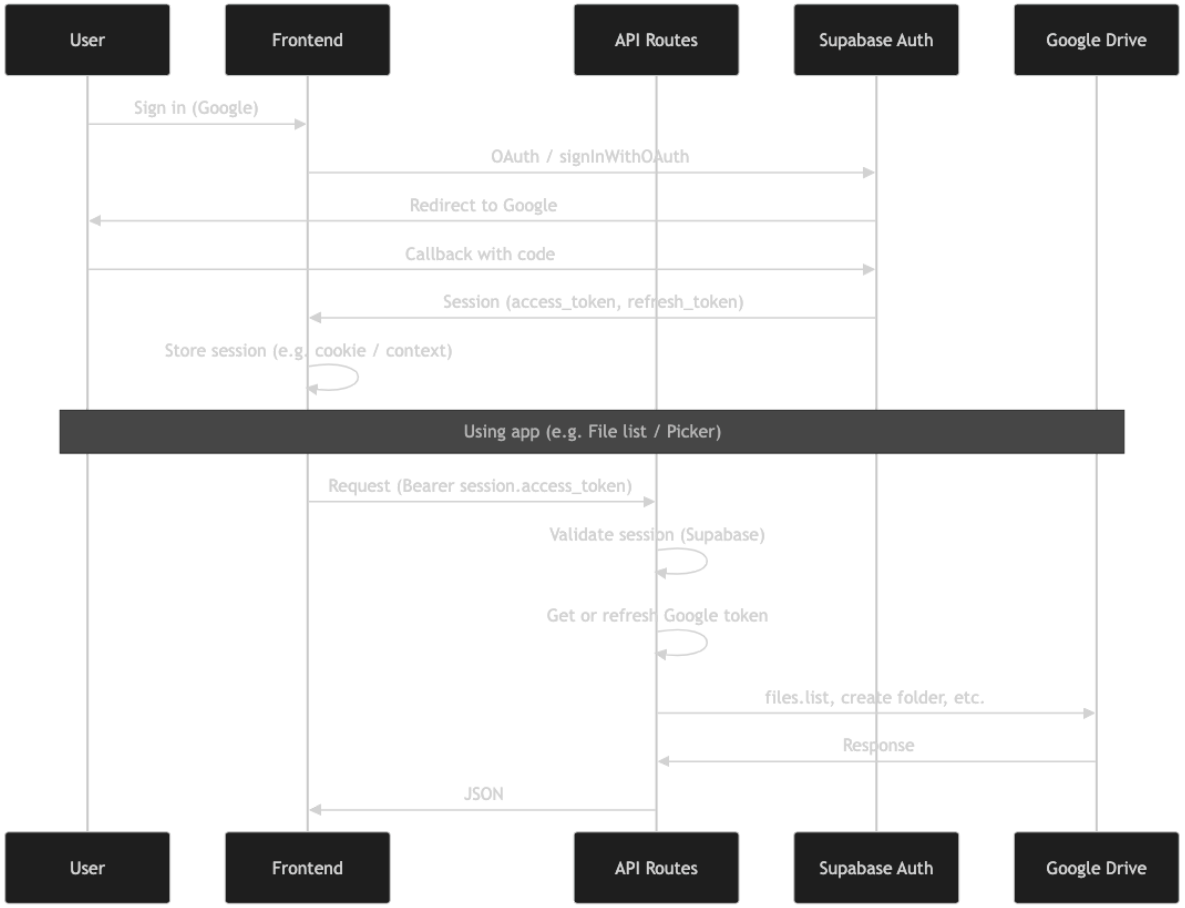
The web application is a Next.js app comprising the browser UI and API routes. Application data is stored in PostgreSQL; file content lives in Google Drive.



---

3. Authentication Flow

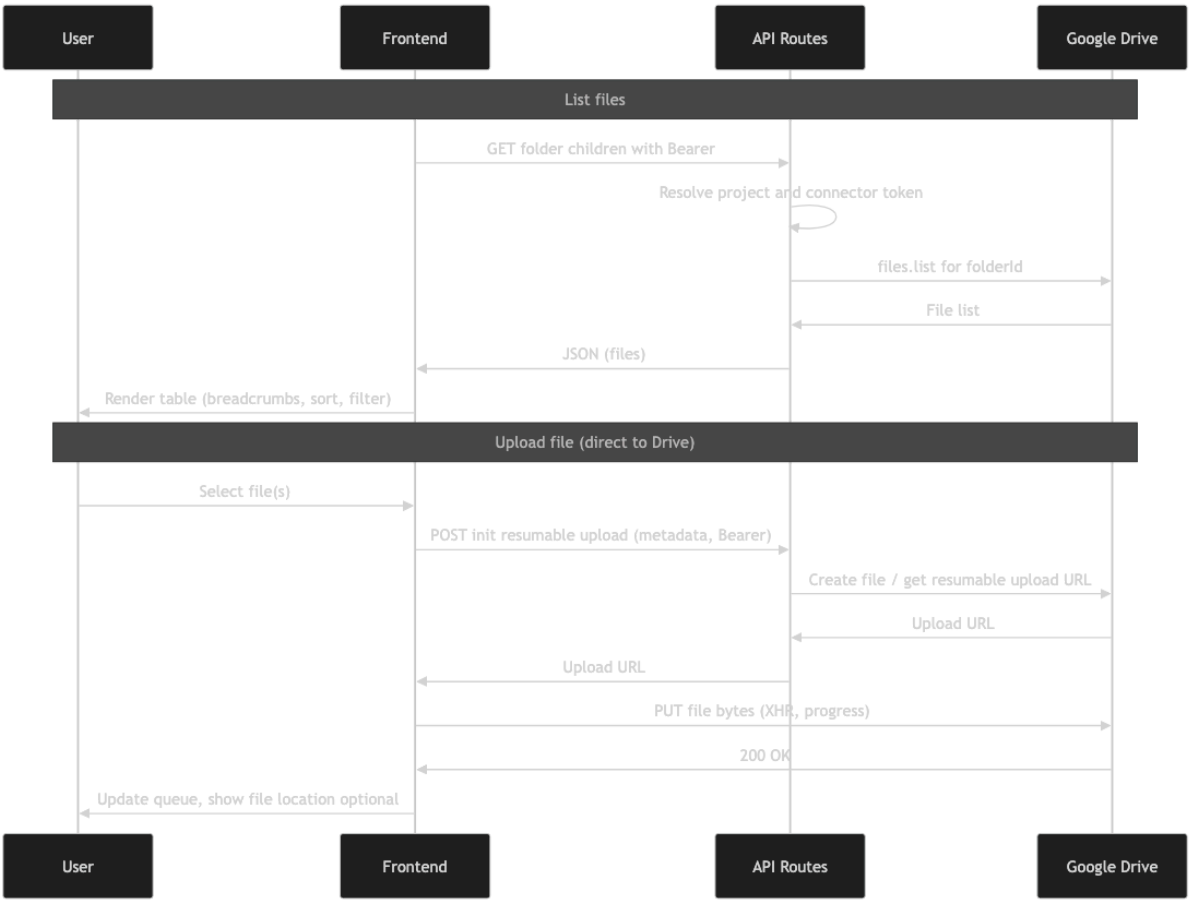
Users sign in with Google (Supabase). Session is used for API authorization and for obtaining a Google access token (e.g. for Drive Picker and Drive API).



---

4. Project File List & Upload Flow

File browser lists contents of the project's Drive folder. Uploads go directly from the browser to Google Drive (resumable upload); the API only issues the upload URL and metadata.



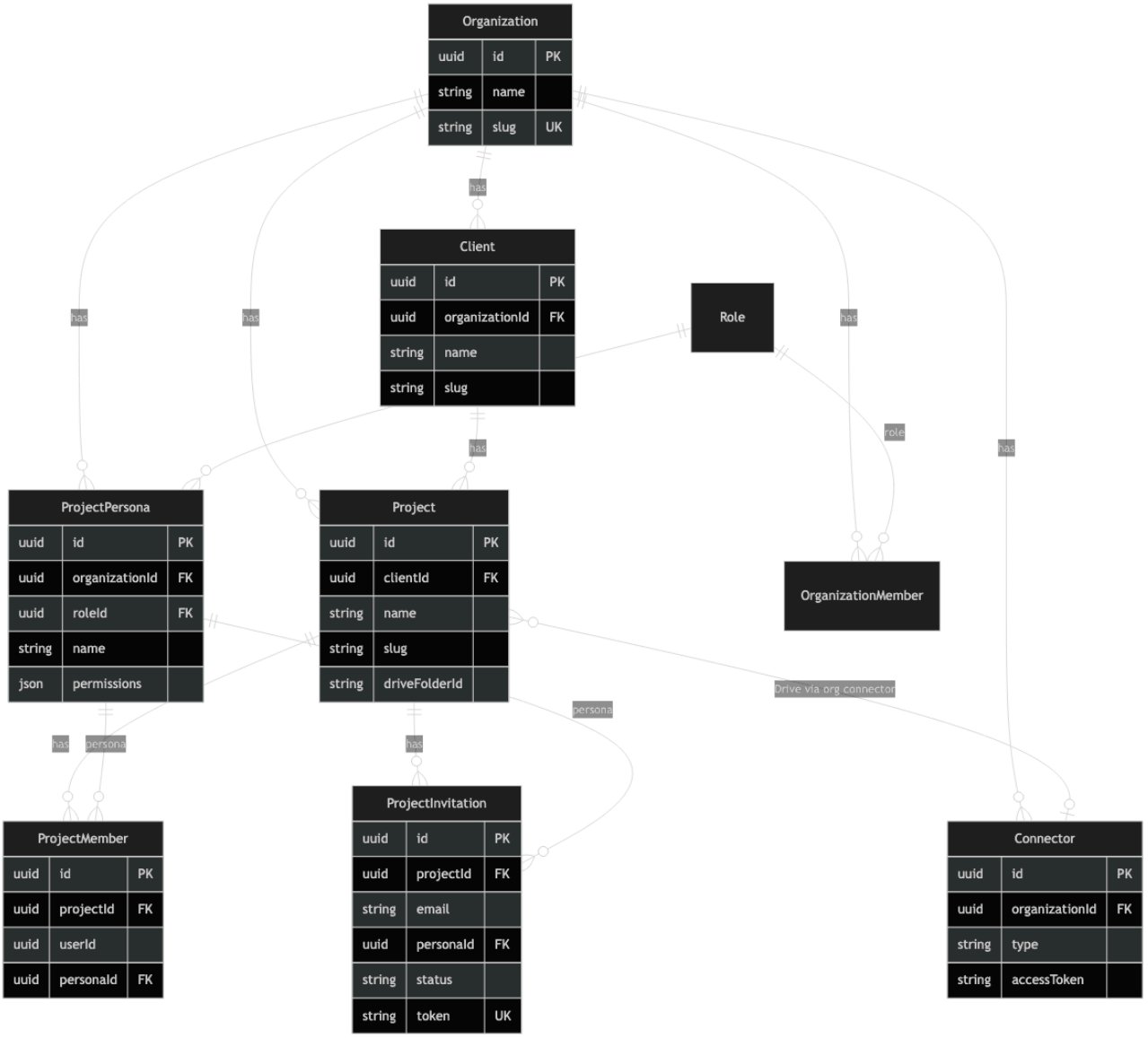
---

5. Core Data Model (Simplified)

Organizations contain Clients and Connectors. Projects belong to a Client and reference a Drive folder. Members and Invitations are scoped to Organization and Project; Personas define project-level roles.

Schema organization: All tables shown below (except `contact_submissions`) are in the **`portal`** schema. The `admin` schema contains `contact_submissions` (not shown in diagram).

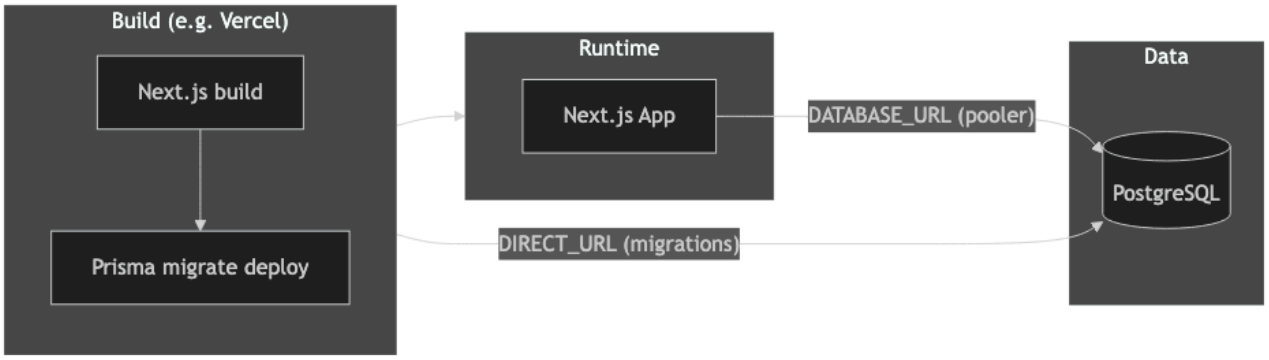
Note: File Assignment feature (planned) will require a new `FileAssignment` table to track which files are assigned to which external members. This table will link `fileId` (Google Drive file ID), `projectId`, `memberId` (ProjectMember ID), `assignedAt`, and `assignedBy` (userId).



---

6. Deployment Context

Next.js is built and deployed (e.g. Vercel). PostgreSQL is hosted (e.g. Supabase/Supavisor). Environment distinguishes runtime URL (transaction pooler) vs migration URL (session pooler).



Glossary

Term	Definition
Organization	Top-level tenant; owns clients, projects, connectors, and personas. Users belong to one or more orgs via roles. Stored in `portal` schema.
Client	Customer or entity (e.g. "Acme Corp"); belongs to an org; contains projects. Stored in `portal` schema.
Project	Engagement or case; belongs to a client; linked to one Google Drive folder; has tabs (Files, Members, Shares, Insights, Sources). Stored in `portal` schema.
Connector	Org-level link to an external service (e.g. Google Drive); stores OAuth tokens; used for Drive folder sync and Import from Drive. Stored in `portal` schema.
Persona	Project-level role template (e.g. Project Lead, Team Member, External Collaborator, Client Contact); defines permissions (`can_view`, `can_edit`, `can_manage`, `can_comment`); assigned to members and invitations. Stored in `portal` schema. Default personas are seeded per organization when first accessed. Persona names can be renamed by Project Leads per project (low priority feature).
Portal schema	PostgreSQL schema containing all user-facing application data (organizations, clients, projects, documents, customer requests, etc.). RLS is applied to tables in this schema.
Admin schema	PostgreSQL schema containing administrative/internal data (contact form submissions). RLS is not applied to this schema as it is admin-only.

From HLD to Low-Level Design (LLD)

This HLD is structured so that a Low-Level Design can be derived from it without ambiguity. The sections below provide **steps to create an LLD** and a **mapping from each HLD section to LLD deliverables**, demonstrating that the HLD is detailed enough to prepare an LLD.

Sufficiency for LLD

The HLD provides:

- **System and container boundaries** (C4 L1/L2) → LLD can zoom into components and classes per container.
- **Flows** (auth, file list, upload) → LLD can add API-level and DB-level sequence diagrams, request/response schemas.
- **Data model** (ER diagram + tables) → LLD can define physical schema, indexes, migrations, and RLS policies.
- **API surface** (route groups and purpose) → LLD can specify each endpoint's method, path, body, headers, errors, and auth.
- **Frontend anchors** (pages and key components) → LLD can specify component tree, props, state, and API calls per screen.
- **Security and compliance** (direct-to-Drive, PII, RLS, enterprise practices) → LLD can specify concrete controls (e.g. RLS policy SQL, encryption fields, audit log schema).

Steps to Create an LLD from This HLD

1. **Choose a scope** — Pick one or more areas (e.g. "Project Files", "Invitations", "Connectors") so the LLD stays bounded.
2. **Derive component view (C4 L3)** — For each container (Frontend, API), list components/modules that implement the flows in §3–§4. For example: `ProjectFileList`, `useDrivePicker`, `GET /api/connectors/google-drive/upload`, `listDriveChildren` in `google-drive-connector`.
3. **Specify API contracts** — For each route in **Key API Surface**, document: HTTP method and path, request body/query/headers, response shape, error codes, and how auth/org/project context is passed. Optionally add OpenAPI or TypeScript types.

4. **Specify data access** — For each flow that touches the DB, list Prisma models and queries (or raw SQL). From **Core Data Model** and **Security (RLS)**, write migration-ready schema changes and RLS policy definitions (e.g. `CREATE POLICY ... ON clients USING (organization_id = current_setting('app.current_org_id')::uuid)`).
5. **Specify UI behavior** — For each **Key Frontend Anchors** component, document: props, local state, server state (e.g. SWR/query keys), API calls, and error/loading UI. Add wireframes or acceptance criteria if needed.
6. **Specify security controls** — From **Security & Compliance**, turn recommendations into concrete LLD items: which tables get RLS, which PII columns get encryption, audit log table schema, and where to set `app.current_org_id` (e.g. middleware or API wrapper).
7. **Cross-check** — Ensure every flow in §3–§4 is covered by at least one API contract, one data-access spec, and one UI spec; and that every security requirement maps to an implementable control.

HLD Section → LLD Deliverable Mapping

HLD section	LLD deliverable
Design principles	Constraints and non-goals in LLD intro; used to reject out-of-scope designs.
Technology stack	Stack is fixed; LLD specifies libraries, versions, and patterns within that stack.
URL structure	Route table; LLD adds page components, layout, and data-fetching per route.
Key API surface	Per-endpoint spec: method, path, request/response, errors, auth. Optionally OpenAPI.
Key frontend anchors	Per-component spec: props, state, API calls, subcomponents. Optionally wireframes.
Security & compliance	RLS policy definitions; PII encryption fields and KMS usage; audit log schema; secrets handling.
1 System context	No LLD expansion; used to validate that LLD does not introduce new system-level actors or boundaries.
2 Container diagram	C4 L3 component diagram per container (Frontend modules, API route groups, shared libs).
3 Authentication flow	Sequence diagram at API/DB level; session validation and token refresh logic; middleware spec.
4 File list & upload flow	Sequence diagram at API/DB level; upload init and Drive API call specs; frontend upload state machine.
5 Core data model	Physical schema (Prisma or SQL); indexes; migration files; RLS policies per table.
6 Deployment context	Build and deploy steps; env vars; DATABASE_URL vs DIRECT_URL usage.
Glossary	Terms used consistently in LLD; extend with domain terms introduced in LLD.

Using this mapping, an implementer can produce LLD documents (e.g. one per epic or module) that trace back to this HLD and prove that the HLD is detailed enough to prepare an LLD.

References

- PRD – Product requirements and feature list
- Roadmap – Milestones and schedule
- AGENTS.md – Database migrations, Vercel, Git workflow