ChatGPT

Doliver Investment Lifecycle App – Product Requirements Document (PRD)

Introduction and Overview

The Doliver Investment Lifecycle App is a comprehensive deal flow and investment pipeline management platform for Doliver Advisors. It streamlines the entire lifecycle of an investment deal – from initial intake and due diligence through funding, monitoring, and exit – within a single collaborative system. This PRD consolidates the current **DealFlowLifecycle v1** architecture and features, requirements from PRD 2.0, and new enhancements into a refined blueprint for version 2.0 of the app. The document is structured for use by a Replit AI development agent as direct implementation guidance, ensuring all features and technical details are explicitly defined.

Goals and Objectives: The primary goal is to provide Doliver's investment team with a **centralized, efficient, and secure platform** to manage deals, track fund allocations, generate AI-assisted investment memos, forecast capital calls, and analyze portfolio metrics. Key objectives include: - Eliminating scattered spreadsheets and email threads by capturing all deal data, notes, and documents in one system.
- Enforcing a **structured deal review workflow** that can be customized to Doliver's process (e.g. stage gating, checklists) [1].
- Enhancing decision-making with analytics dashboards that turn data into actionable insights (e.g. pipeline conversion rates, sector trends) [2].
- Integrating an **AI Analyst** to automatically draft investment memos and surface insights, reducing manual effort in documentation.
- Ensuring compliance with regulatory requirements through audit logs, permission controls, and a built-in compliance checklist.
- Providing a modern, modular technical architecture (separated front-end and back-end, robust API) to support easy maintenance, testing, and future growth.

Scope: This PRD covers all functional modules (deal intake, stage workflow, fund management, AI memos, analytics, etc.), the full system architecture (client, server, database, external integrations), an expanded data model, API design (REST and GraphQL), implementation details (file structure, deployment), and security/compliance considerations. It is assumed that basic CRM features (e.g. contact management) are included only insofar as they support the deal flow process, and that general accounting or portfolio performance tracking beyond deal close is out of scope for this app (those might be handled by separate portfolio management tools).

System Architecture Overview

Doliver's app follows a **client-server architecture** with a decoupled front-end and back-end. This separation of concerns allows the UI to be highly interactive and the server to expose a robust API for both the web client and potential external integrations. The design is modular, with distinct services for core domains (deals, funds, users, etc.) and a unified data store. Key principles include scalability (the

architecture can handle growing data and users), maintainability (clear separation of modules and layers), and security at every layer.

High-Level Architecture

- **Client:** A web front-end (Single Page Application) implementing the user interface and client-side logic. It communicates with the server via REST and GraphQL API calls over HTTPS. The client handles presentation, user input, and local state (e.g. form data, caching for performance). It is built with modern web technologies (e.g. **React** framework with component-based architecture), allowing dynamic and responsive user experience. The front-end is deployed as static assets (HTML/CSS/JS) served by the back-end or a CDN. Users access the client via browser (the app is optimized for Chrome and modern browsers, and responsive for tablet use; mobile-specific optimization is a future consideration).

- **Server:** A back-end application (e.g. **Node.js** with Express and Apollo Server, or Python Flask/FastAPI with Graphene for GraphQL) that provides business logic and data access. The server exposes a well-defined **REST API** for standard CRUD operations and an **GraphQL API** for flexible queries. It implements all core functionalities: handling incoming deal data, enforcing workflow rules, generating AI memos (by calling external AI services), running analytics computations, etc. The server architecture is layered into controllers/resolvers (handling API requests), services (business logic, e.g. deal service, fund service), and data access layers (ORM models and queries). This design allows each module's logic to be maintained and tested in isolation.

- **Database:** A **relational database** (e.g. PostgreSQL) holds the persistent data for all deals, funds, users, and related entities. A SQL database is chosen for its robustness with relational data and support for complex queries (analytics, reporting). An ORM (Object-Relational Mapper) such as Prisma (for Node) or SQLAlchemy (for Python) is used to map database tables to classes and to ensure database-agnostic development. The data model (detailed in a later section) is carefully designed with **referential integrity constraints** (primary/foreign keys) to enforce consistency (for example, a fund allocation record cannot exist without valid references to a deal and a fund). The database may be hosted via Replit's built-in database support for development and an external managed PostgreSQL instance for production (ensuring data durability and backups).

- **Storage:** For file storage (e.g. documents attached to deals or exported reports), the system uses a cloud storage service or the database for small files. Documents like term sheets, pitch decks, and NDAs can be uploaded via the app's UI. In development on Replit, files can be stored in the `/mnt/data` directory (persisted storage), while in production a service like AWS S3 or Google Cloud Storage may be configured. The app stores file metadata (filename, link, associated deal) in the database, and the files themselves in secure storage.

- **External Integrations:** The app integrates with several external services to extend functionality:

- **AI NLP Service:** Integration with OpenAI (or a similar LLM provider) for the AI Analyst module. The server sends deal context (e.g. description, financials, etc.) to the AI API and receives generated content (memos or answers). All API keys (for OpenAI) are stored securely in environment variables, and usage complies with data security (sensitive info is either sanitized or an on-premise model could be considered if needed for compliance).

- **Calendar API:** Integration with Google Calendar (via Google Calendar API and OAuth2) or Outlook 365 Calendar to sync scheduled events (e.g. due diligence calls, investor meetings, or capital call dates). The app will use OAuth credentials to create or update events on users' calendars with appropriate reminders. Alternatively, for a simpler approach, the app can generate industry-standard `.ics` calendar files for download or emailing to participants.
- **Email Service:** (Planned) Integration with an email API (like SendGrid or SMTP server) to send notifications such as: new deal intake confirmations, task reminders, or a generated investment memo to stakeholders. Email notifications ensure users not logged into the app are still alerted of important events (this is complementary to in-app notifications).
- **Authentication Provider:** (Optional, if using SSO) Integration with an identity provider (like Okta or Google OAuth) for single sign-on. Otherwise, the app uses its own user management (with JWTs).

- **Market Data APIs:** (Future consideration for AI Analyst) APIs that provide financial or market data (for example, stock data, industry benchmarks) could be integrated to enrich the AI analysis. This is an extended feature not in scope for this release but the architecture allows adding a service module for such data when needed.

- **Events and Messaging:** The back-end will implement an internal event system to decouple modules. For example, when a deal's stage is advanced, an event "DealStageChanged" can be emitted. Listeners for this event could trigger actions like: send a notification email, update an analytics counter, or create a follow-up task. This could be done through a simple publish/subscribe pattern in the application (for instance, using Node.js EventEmitter or a message queue like BullMQ for background jobs if needed). Similarly, when a new deal is created, an event might trigger the AI module to pre-generate a draft memo in the background. This event-driven approach ensures **loose coupling** – new features (like notifications or auto AI analysis) can subscribe to events without modifying the core transaction flow.

- **Client-Server Communication:** All client-server interactions occur over secure HTTP (HTTPS). The app will use **JSON** for REST API payloads and standard GraphQL query/mutation syntax for GraphQL API. Real-time updates (if needed for notifications or live collaboration) could be handled with WebSockets or GraphQL subscriptions; however, in this PRD we assume near-real-time needs are low and can be handled by periodic polling or manual refresh for simplicity. WebSocket support can be added later for things like instant notifications of stage changes if required.

- **Deployment Model:** The app is expected to run on Replit during development (which provides an environment to host both front-end and back-end). For production, it could continue to run on Replit's Always-On deployment or be containerized (Docker) and hosted on cloud platforms. The architecture supports horizontal scaling on the back-end (stateless API servers behind a load balancer, as the database is central), though initial deployments will likely be small scale (single instance for simplicity). Static assets of the front-end can be served by a CDN for scalability if needed. We also separate environments for Dev/Staging and Production to allow safe testing of new features.

Diagram – System Context: *A high-level architecture diagram would show:* Users accessing the **Web Client** (browser UI), which communicates with the **Application Server** (REST/GraphQL API). The server reads/writes from the **Database** (PostgreSQL). The server also communicates with external services: the **OpenAI API** for memos, the **Google Calendar API** for event syncing, and an **Email SMTP** for notifications. All

interactions are secured via encryption. *(This diagram can be visualized as a typical 3-tier web app: UI → API → DB, with side integrations branching off the API.)*

Client-Side Architecture (Front-End)

The front-end is a **single-page application** (SPA) built with **React** (using JSX and modern ESNext JavaScript). It provides an intuitive user interface for all the functional modules (deals, funds, analytics, etc.). The client-side code is organized into a modular structure: - **Pages/Views:** Each main feature corresponds to a page or view in the app (e.g., Deals Pipeline page, Deal Details page, Funds page, Analytics Dashboard page, etc.). A routing library (like React Router) manages navigation between pages without full page reloads (for a smooth SPA experience). - **Components:** Reusable UI components encapsulate sections of the UI (e.g., **DealCard**, **DealForm**, **StageKanbanBoard**, **FundAllocationTable**, **MemoEditor**, **AnalyticsChart**). Components are organized by feature domain. For example, a `deals` directory might contain components for listing deals, deal detail view, and forms related to deals. - **State Management:** The app will use a combination of React's built-in state and context for local component state and a global state solution for cross-cutting data. For moderate complexity, React Context or lightweight state libraries can suffice (e.g., Context + useReducer for things like the current user or global loading states). If the app grows in complexity, adopting Redux or Zustand can be considered, but initially the focus is on simplicity. Data that is fetched from the server (like list of deals or user info) will be cached in memory so that components can use it without refetch on every navigation. The GraphQL client (if using GraphQL) such as Apollo Client can manage caching of query results. - **API Layer:** The front-end includes an API utility layer for communicating with the back-end. This could be a set of functions or a service class abstracting calls. For GraphQL, an Apollo Client (or URQL) is configured with the GraphQL endpoint, handling queries and mutations from React (with hooks or higher-order components). For any REST calls, a lightweight wrapper using `fetch` or axios is included. All API calls include the user's auth token (e.g., a JWT stored in memory or localStorage) in headers for authentication. - **UI/Styling:** The app uses a modern UI toolkit or custom component library for consistency. It could use a component library like **Material-UI** (MUI) or Ant Design for pre-styled components (tables, buttons, forms) and then apply custom Doliver branding on top. CSS is managed either through CSS-in-JS (if using styled-components or emotion with React) or through SCSS/CSS modules depending on developer preference. The goal is a clean, professional interface that is also responsive. Key UI elements include: - A navigation sidebar or header menu for accessing main sections (Deals, Funds, Analytics, etc.). - List and detail views: e.g., a scrollable list or table of deals on one side and details on the other (for larger screens), or separate pages on smaller screens. - Forms for data input (new deal form, edit deal, add fund allocation form, etc.) with validation and helpful defaults. - Interactive dashboard widgets for analytics (charts and graphs). - Dialogs and modals for tasks like adding notes, confirming delete, generating AI memos, etc. - **Frontend Modules Mapping:** Each functional module described later has a front-end component aspect: - *Deal Intake:* A **New Deal Form** page to input deal information and a **Deal Import** option if needed (for example, parsing an email or a CSV list of prospects). - *Stage Workflow:* A **Pipeline Board** view showing columns for each stage (Kanban style) where deal cards can be dragged and dropped between stages. Also, a **Deal Detail** view where the stage can be advanced via a dropdown or "Next Stage" button (with confirmation). - *Fund Allocation:* Within the Deal Detail view, a **Fund Allocation sub-section** (table listing which funds invest how much in this deal, with an "Add Allocation" button). Also, a **Funds List** page showing all funds, and each Fund's detail page listing its investments. - *AI Memo:* In the Deal Detail view, an **AI Memo panel** shows the latest AI-generated investment memo. If none exists or if user wants a refresh, a "Generate AI Memo" button triggers the AI service. The content can be displayed in a rich text viewer/editor (allowing the user to make manual edits or add comments). - *Analytics:* An **Analytics Dashboard** page with various charts (e.g., bar chart of deals by stage, pie chart of deals by sector, line chart

of cumulative investments over time, etc.). The sector-level analytics might allow filtering the charts by industry sector, or a separate section highlighting sector breakdowns. - *Call Forecasting:* Likely integrated in the Funds detail or a dedicated **Capital Calls** page. This could present a timeline or table of upcoming capital calls for each fund. For example, on a Fund's page, show "Projected Capital Calls" with dates and amounts, possibly visualized in a calendar view or timeline chart. Users might adjust assumptions and see scenarios (if implemented). - *Calendar Integration:* A settings page or modal for linking a calendar account (OAuth flow). Once linked, events (like due diligence calls or IC meetings) can be created from within the app. For instance, a **Schedule Meeting** button on a Deal or Task could open a form to enter meeting details and then push to the external calendar. Upcoming events might also be listed on a dashboard or the deal's page. - *Export/Compliance:* Buttons or links to export data appear in relevant places (e.g., "Export Deals to CSV" on the Deals page, "Download Memo PDF" on a memo, "Export Audit Log" in admin). Compliance checklists are shown on the Deal Detail view (perhaps under a **Compliance** tab or section), and an admin-only **Compliance Dashboard** could list all outstanding compliance tasks across deals. - **Frontend Routing and Navigation:** All navigation should be intuitive with clear URLs for each major view (e.g., `/deals`, `/deals/123` for a specific deal by ID, `/funds`, `/analytics`, etc.). Where appropriate, deep linking is enabled (users can share a URL of a deal detail). The app will handle unknown routes gracefully (show a 404 page or redirect to a safe page like the deals list). - **Form Validation and UX:** Use client-side validation for required fields and proper formats (e.g., email format, date pickers for dates, numeric inputs for money with appropriate formatting). Provide inline error messages and prevent submission until errors are fixed. Also use **optimistic UI** updates where it makes sense (e.g., moving a deal card to a new stage could immediately reflect on UI while the request is sent to server, then confirm or rollback if server fails). - **Performance Considerations:** Use code-splitting (dynamic import of modules) to keep initial load fast (e.g., load the deals list code on startup, but only load the analytics components when the user navigates there). Use pagination or infinite scroll for long lists of deals to avoid rendering too many DOM elements at once. Leverage caching from GraphQL or localStorage if needed for large data that doesn't change often (like list of sectors or static reference data).

Server-Side Architecture (Back-End)

The back-end is designed as a **RESTful API service with GraphQL**, implemented in a modular way. We will outline it assuming a Node.js environment for concreteness, but the design is generally applicable. Major back-end components and structure: - **Web Server & Framework:** The server uses **Express.js** (a minimalist web framework for Node) to define REST API endpoints and handle middleware (auth, logging, etc.). In parallel, it runs an **Apollo Server** (or Express middleware) for GraphQL at a `/graphql` endpoint. The two interfaces (REST and GraphQL) operate on the same business logic underneath. Cross-origin resource sharing (CORS) is configured to allow the front-end origin to call the APIs if hosted separately. - **Authentication & Middleware:** The server enforces authentication via a middleware that checks for a valid JWT in the `Authorization` header for protected routes. Public routes (like login, health check) skip this. On each request, the JWT is decoded (using a secret or public key if using JWT signing) to identify the user and attach user info (ID, roles) to the request context for use in downstream logic. Additional middleware include request logging (to console or file), body parsing (JSON parser for REST, which Apollo does for GraphQL automatically), and rate limiting (to prevent abuse of certain endpoints, e.g., too many login attempts). - **Service Modules (Business Logic):** The core logic is organized into service modules corresponding to each domain: - **DealService:** Functions for creating a new deal, updating deal info, advancing a deal's stage (with checks), adding notes or attachments to deals, etc. This service encapsulates all rules about deals (e.g., required fields, allowed stage transitions, notifications to trigger on stage change). For example, `DealService.advanceStage(dealId, newStage)` will: verify the user has

permission, ensure the new stage is valid (maybe you cannot jump backwards or skip stages without approval), update the deal's stage in the database, log the change in history, and emit a `DealStageChanged` event. - **FundService:** Functions to create/edit funds and manage allocations. E.g., `FundService.allocateToDeal(fundId, dealId, amount)` will check that amount is available, create or update a fund allocation entry, and perhaps recalc some summary fields (like how much of the fund is now invested). It might also trigger recalculation of capital call forecasts. - **AIService (Analyst):** Functions to generate AI content. For instance, `AIService.generateInvestmentMemo(dealId)` will gather necessary data (deal details, company info, perhaps recent news if we integrate that in future), craft a prompt, call the OpenAI API, and get a response. It then saves the AI memo to the database (in a Memo table or as part of the deal record) and returns it. If the call is lengthy, this could be done asynchronously: the request could immediately return a status and the client polls, but for MVP it can be synchronous behind a loading indicator. The AIService also could handle other AI-driven features, like answering ad-hoc questions about a deal (if an interactive Q&A feature is added). - **AnalyticsService:** Functions to compute various metrics for the dashboard or reports. This might aggregate data across deals. For example, `AnalyticsService.getPipelineStats()` might query the database to count deals per stage, per sector, conversion rates (deals that moved from stage X to closed/invested), etc. More complex analytics (like running scenario simulations for call forecasting) could also live here. These functions are used by the API endpoints when the client requests analytics data. - **ComplianceService:** Functions related to compliance and audit. For instance, it might validate that all required compliance tasks for a deal are completed, or produce an audit log report. It might also encapsulate permission logic (though much of that can be simple checks in each service function). - **UserService & Auth:** Managing users (creating accounts, password hashing, login tokens) and roles. For login, `AuthService.login(email, password)` verifies credentials (using hashed password comparison) and returns a signed JWT containing user ID and role claims. There may be functions for password resets, etc. UserService might also handle user profile updates or listing users (for admin). - **NotificationService:** (Optional module) If the system sends emails or in-app notifications, this service would contain functions to compose and send those (triggered by events like stage changes or upcoming deadlines). - **Controllers/Resolvers:** On the REST side, **controller functions** correspond to each API route and simply call the appropriate service functions, then format the response. For example, a POST `/api/deals` route uses `DealController.createDeal(req, res)` which extracts data from the request body, calls `DealService.createDeal(data, currentUser)`, and returns the created deal or an error. On the GraphQL side, **resolvers** are defined for Query and Mutation types mapping to service calls. For example, `Mutation.createDeal(_, {input}, context)` would similarly call the service. Using service layer ensures consistency between REST and GraphQL – they both leverage the same underlying logic. - **Data Access Layer (Models/ORM):** The app will define **database models** that map to the database tables (or use an ORM's definitions). For example, a `Deal` model class (or Prisma schema for Deal) defines fields like id, name, description, stage, etc., and relationships to other models (e.g., hasMany notes, hasMany allocations). This layer is responsible for constructing queries. Using an ORM, simple operations are method calls (e.g., `DealModel.findById(id)` or `DealModel.query().filter({stage: ...})`). For complex reports, custom SQL or ORM query builder can be used (ensuring to optimize with proper indexing on the database side). The data layer also enforces validations beyond what the database does (though critical constraints are also mirrored in the DB schema). - **GraphQL Schema:** The GraphQL API is described by a schema (SDL) defining types like `Deal`, `Fund`, `User`, and input types for mutations. The schema and resolvers are maintained alongside the models and services. For example, a GraphQL type `Deal` will have fields that may map directly to model fields (like `name`, `stage`) and relational fields that resolvers populate (like `company` or `allocations`). If using Apollo, the schema might be defined with type definitions and resolver maps, or using code-first (like Nexus or type-graphql) to generate from code. This PRD will include a detailed API contract section later. - **Caching**

& Performance: On the server side, certain heavy operations might benefit from caching. For example, an analytics query that is expensive could cache results for a short period (e.g., update the stats every hour or on certain events rather than every request). The system could use an in-memory cache (like a simple Node cache or Redis if we introduce an external cache layer) for such use cases. Another performance aspect is using database indices on frequently filtered fields (e.g., stage, deal name, sector) to speed up queries. - **Error Handling:** The server uniformly handles errors. If a request fails validation or a business rule (e.g., trying to allocate more funds than available), the service throws an error which is caught by the controller or GraphQL resolver and translated to an appropriate HTTP error code or GraphQL error response. The error messages are designed to be user-friendly where they may surface to the UI (e.g., "Allocation exceeds remaining fund capacity"), and technical details (stack traces) are logged but not exposed. - **Logging & Monitoring:** The backend will log key events and errors. At minimum, access logs (method, URL, response time, status code) are kept for requests (could use morgan library in Express for this). Important errors and warnings are logged to console or a file. In production, integration with a monitoring service (like Sentry or DataDog) could be set up to catch exceptions and performance metrics. This ensures issues can be diagnosed. Also, security-relevant events (failed logins, permission denials) could be logged for audit. - **Scalability & Concurrency:** The stateless nature of the API allows running multiple instances if needed. Replit might not support multiple instances, but the design is cloud-agnostic. If heavy background processing is needed (e.g., a very large AI analysis or PDF generation), a background job queue (like Bull in Node) could be used so that such tasks don't block the main request thread. The job runner can be another process or thread. For now, the scale of operations (moderate number of deals and periodic AI calls) can be handled by a single instance.

In summary, the back-end provides a robust, modular foundation. The clear separation into controllers, services, and data models makes it easier for Replit's AI (or any developer) to implement each piece in a structured way. Each functional module (detailed next) will specify the particular back-end logic needed, which will correspond to service functions and possibly new data models or external calls as described above.

Database Schema and Data Model

(A full data model is described in a dedicated section below; this overview highlights the approach.) The application uses a relational schema capturing the core entities of Doliver's business: - **Deals:** Represent investment opportunities. Linked to a Company (the target company/startup) and have attributes like current stage, status (active/passed), source, etc. - **Companies:** Represent the companies or assets being invested in (with data like industry sector, key contacts, etc.). Separated from Deals to allow multiple rounds or opportunities for the same company to be tracked. - **Funds:** Investment funds or pools of capital that Doliver manages. Each fund has a total committed capital and tracks how much is already invested or called. - **Allocations:** Join entity linking Funds to Deals, indicating how much each fund will invest (or has invested) in a particular deal. - **Users:** System users (Doliver team members) with roles determining permissions (e.g., Admin, Analyst, Partner, Compliance). - **Notes/Memos:** Textual notes or memos attached to deals (including AI-generated memos as a subtype). - **Tasks/Checklist Items:** To-do items or due diligence checklist entries for a deal, often relating to compliance or next steps (e.g., "Complete KYC"). - **Events:** Scheduled events like calls or meetings, possibly synced to external calendars. - **Reference Data:** Tables for lookup values such as deal stages definitions, industry sectors, and deal sources.

The database schema enforces relationships (foreign keys from deals to companies, from allocations to deals and funds, etc.) and uses indexes on key fields (like deal stage, company sector, etc.) to optimize

queries. All string fields that represent enumerations (like stage) are either using a lookup table or an enum type in the DB to maintain data integrity. The model also accounts for **history/audit** where needed (for example, a separate table logging stage changes over time).

External Integrations in Depth

- **AI Analyst Integration:** The integration with the AI service (OpenAI) is encapsulated in the AIService. The PRD calls for an *Enhanced AI Analyst* feature, meaning we will extend beyond the basic "generate memo" of v1. The enhancement could include interactive Q&A with the AI or deeper analysis. Technically, the integration involves sending a prompt to OpenAI's API. For instance, to generate an investment memo, the prompt might include a summary of the company (business model, market, financials, etc. either entered by the user or from documents) and instruct the AI to produce a structured memo covering specific aspects (market, team, financials, risks, etc.), similar to how Flybridge's tool processes a pitch deck and uses a structured prompt [3]. The returned text is then saved. If an interactive chatbot analyst is needed, the system could maintain a conversation context per deal. **Note on cost and performance:** Calls to AI APIs should be mindful (perhaps requiring a user action like clicking "Generate Memo" to avoid excessive calls), and the system should handle errors (e.g., API down or prompt too long) gracefully with retry or user messaging.
- **Calendar Integration:** The app will have credentials for a Google API (client ID/secret) to perform OAuth for each user who wants calendar sync. Upon linking, the server obtains a refresh token to act on behalf of that user's calendar. When the user schedules an event (like "IC Meeting for Deal X"), the server uses Google Calendar API to create an event on the user's calendar (and possibly invite other participants via the event's attendee field). The integration module will handle token refresh and API calls. Additionally, it could subscribe to calendar update events if needed, but likely we only need one-way sync (from app to calendar). Security: tokens are stored encrypted in the DB and only used by the calendar module.
- **Email/Notification Integration:** Using a service like SendGrid or a simple SMTP, the server can send emails for certain triggers. For example, when a deal is moved to "IC Review" stage, an email can automatically go to the Investment Committee members with the meeting info or the AI memo attached. Or daily summary emails can be sent for upcoming tasks/events. This integration simply requires SMTP credentials or API keys, and using a library to send out messages.

Deployment and Replit Considerations

- **Replit Workspace Setup:** The project repository will be structured as described in the "File/Folder Structure" section below, which is optimized for a Replit environment. On Replit, a single environment can run both the front-end and back-end. We can use a **Procfile** or Replit's `.replit` configuration to run a start script that concurrently launches the front-end build (if needed) and the back-end server. However, the simpler approach is to build the front-end into static files and serve them via the back-end (Express can serve the React build output). During development, we might run the React dev server on one port and the API on another; Replit might only expose one port publicly, so in-editor development might use one or the other at a time, or integrate the dev server as a background process.
- **Environment Variables:** The Replit Secrets management will be used to store sensitive config such as the database URL, AI API key, and Google API credentials. The code will read these from `process.env` (for Node) at runtime. The PRD will later enumerate required environment variables.
- **Continuous Deployment:** Each commit could trigger Replit's deployment if connected to a GitHub repo or using Replit's own version control. We will ensure tests can be run on Replit (with a test

database or test mode) before deployment. Replit's always-on feature will keep the app running. For scaling beyond Replit, Dockerfiles and cloud deployment scripts can be added but are not detailed here as immediate focus is Replit.

- **Testing in Replit:** We plan to have a set of unit tests that can be executed in the Replit environment (perhaps via a `npm test` command for Node). Replit's environment can support running these tests, and this PRD includes guidance on what tests to implement for each module.

Having established the overall architecture, we now detail each functional module and feature set, explaining what it does and how it will be implemented on both the front-end and back-end.

Functional Modules and Features

This section enumerates all major functional modules of the Doliver Investment Lifecycle App, including both existing features (from v1 and PRD 2.0) and new features introduced in this version. Each module description covers the user story (what the feature does for the user), UI components involved, back-end logic (including any special rules or algorithms), data model entities, and any integration or implementation notes. The modules are presented in a logical flow from deal inception to analysis and compliance.

1. Deal Intake & Lead Management

Description: The Deal Intake module handles the entry of new investment opportunities into the system. This can include startups seeking funding or any deal lead Doliver wants to track. The goal is to capture all relevant information at the point of intake and ensure the deal is properly initialized in the pipeline.

User Stories: - As an investment analyst, I can **add a new deal** by entering key information (company name, brief description, amount seeking, source of the deal, etc.) so that it appears in our pipeline for evaluation. - As a user, I want to see a list of recently added deals and basic info at a glance, so I know what new opportunities have come in. - (If applicable) As an external party (entrepreneur or referrer), I might fill an **online application form** that feeds into deal intake (this could be a future enhancement, but the system should accommodate importing deals from such sources).

Frontend Implementation: - A **"New Deal" form page** or modal dialog is provided. This form includes fields such as: - Company Name (text) - Deal Name or Short Title (if different from company, otherwise can use company name) - Description/Overview (long text) - Industry Sector (dropdown selection – e.g., FinTech, Health, etc., based on predefined sectors) - Deal Source (dropdown or text – e.g., "Referral", "Inbound Email", "Conference", etc.) [4] - Source Details (e.g., name of person who referred, if applicable) - Stage (initial stage, defaults to "New" or "Screening" as first stage) - Amount Seeking or Deal Size (number, possibly currency) - Attachments (option to upload a teaser or pitch deck PDF at intake) - Assigned To (which internal team member is responsible, if applicable) - The form performs validation (required fields like Company Name, and proper data types for amount). It will use auto-complete or suggestions for certain fields if available (e.g., previously used sources or sectors). - On submission, the form calls `POST /api/deals` or a GraphQL `createDeal` mutation with the input data. If attachments are included, these are uploaded (either first to get a file ID or as part of form data if using REST multipart). - After creation, the user is redirected to the **Deal Detail view** for the newly created deal, or back to the deals list with a success message. - The UI also features a **Deals List/Directory** page where all deals are listed (optionally filterable by stage, sector, etc.). Newly added deals show up here. Each entry shows basic info (name, company, stage, and maybe a tag for source or date added). - If many deals are present, a search bar allows searching by

company or deal name. Filters can narrow by stage (e.g., view only “Screening” stage deals). - Possibly implement pagination or infinite scroll for the deals list if the count is large, but initially a paginated list with 20 per page is fine.

Backend Implementation: - **API Endpoint:** A route `POST /api/deals` accepts new deal data. Alternatively, GraphQL `createDeal(input: DealInput): Deal`. Both will invoke the DealService. - **DealService.createDeal(data, user):** - Validates that required fields are present (and user has permission to create deals – assume all internal users can, or perhaps only certain roles; this can be refined in the Security section, but likely any investment team member can add a deal). - If a Company with the same name already exists in the database, it may either reuse it or warn the user of a duplicate (we might prompt on UI if we detect duplicates by name). For now, assume company is unique per deal intake unless future linking is needed (we will create a Company record behind the scenes for each deal, and possibly merge if duplicates found manually). - Creates a new Company record (with name, sector, etc.) if provided and not already present. - Creates a new Deal record with all the input fields. Initial stage is set (likely “New” or whatever default stage is called in the Stage table). If using an enumeration, ensure it’s valid. - Sets the “created_by” field to the current user and timestamps. - If an attachment was included, store the file (in storage) and create a Document record linked to the deal. - If there is an “assigned_to” or owner field, store that (and potentially send a notification to that user). - Saves the Deal and returns the data (including generated ID). - Optionally, emit an event `DealCreated` for further processing (e.g., an event listener could auto-generate an AI summary or send a welcome email to the deal’s contact if we had that info). - **Data Model Considerations:** Creating a deal will involve the `deals` table and possibly `companies` and `documents`. The deals table has a foreign key `company_id`. The system might allow multiple deals per company (for different rounds) so linking ensures we don’t duplicate company info. Each deal also has a `stage` (could be stored as a string or a foreign key to a `stages` table; we will detail this in Data Model). - **Initial Stage/Status:** By default, new deals might be given a status of “Active” (meaning in pipeline) and stage = “Intake” or “Screening” (whatever the first stage is named). In the expanded data model, we will have a Stage table with an order, and one field likely marks the first stage. - **Deal Source Tracking:** The `source` field (and details) from input are saved. This allows reporting later (like “Who is our best referral source?”) [5]. - **Permissions/Visibility:** Typically, all team members can see all deals (especially at Doliver if it’s one team). If in the future deals are assigned and should be private, we could implement an ownership model. For now, assume full visibility within the org. - **Integration:** If we integrate an external intake form or email parsing, that would feed into this endpoint. For example, a web form could call an open endpoint (with an API key for security) to create a deal. This is outside current scope, but the architecture is open to it. - **Testing Considerations:** Unit tests for `createDeal` should cover scenarios: normal creation, duplicate company handling, missing fields (should throw validation error), and ensuring database entries are correctly linked. Integration test: calling the API with sample payload and verifying the DB state. - **Edge Cases:** Handling if a required external ID from another system is needed (not in this case), or if attachments fail to save (should rollback deal creation perhaps), but likely we can save deal first, then handle file. If file upload fails, we can still have the deal and let user attach later with a separate function. - **Example:** A new deal “ABC Tech Series A” is added, source “Referral by John Doe”, sector “Fintech”. After creation, it appears under Stage “Screening” in the pipeline.

Implementation Notes: This module is relatively straightforward but sets the stage for all others. Make sure the UI is user-friendly to encourage complete data entry. Also, consider a future feature: an **Import** function (upload CSV or integrate with email) – while not required now, design the `DealService.createDeal` to be reusable for bulk operations if needed (e.g., could call it in a loop for each CSV row).

2. Pipeline Stage Workflow Management

Description: Once deals are in the system, the Pipeline Stage Workflow module manages the progression of each deal through defined stages of the investment process. It ensures every deal is categorized by its current stage, provides a visual representation of the pipeline, and restricts or triggers actions at stage transitions. Stages might include, for example: *New Lead*, *Screening*, *Due Diligence*, *IC (Investment Committee) Review*, *Approved/Term Sheet*, *Funded/Closed*, or *Rejected/Passed*. This module allows customization of these stages and enforces the order of progression [1].

User Stories: - As an investment team member, I can view all active deals organized by stage, so I have a clear picture of our pipeline and what phase each deal is in. - I can **change the stage** of a deal (e.g., from Screening to Due Diligence) as it progresses, with appropriate controls (e.g., maybe only certain users can move to Approved stage). - I want to capture a reason or note when a deal is marked as *Passed/Rejected* so that we have context for future reference. - I want the system to automatically log when a stage change happened and by whom, for audit and process tracking. - If possible, I want certain stage changes to prompt automated actions (like generating a due diligence checklist or scheduling a meeting).

Frontend Implementation: - The primary UI is a **Pipeline Board** (Kanban-style interface) on a Pipeline page. Each stage is a column with cards for each deal in that stage. Cards display summary info: company name, deal name, maybe amount, and a highlight if any pending tasks or notes. - Users can drag and drop deal cards between stage columns to update the stage. Alternatively, on the deal detail page, there's a stage dropdown or stepper control to move to a new stage. - When a user attempts to move a deal to a stage that represents a closed state (like "Passed" or "Funded"), a modal may ask for confirmation and additional info (e.g., "Are you sure you want to mark this deal as Passed? Please provide a reason: [text box]"). The reason can be stored in the deal's record or as a note. - The board should horizontally scroll if there are many stages. Stages are displayed in defined order. If the organization wants to customize stage names or order, the UI should reflect that (for MVP, stage definitions are static or database-driven but set by an admin or developer). - Each stage column shows a count of deals in it and possibly color codes (e.g., final stages in green, rejected in red). - The Deal Detail view shows the current stage prominently, and an action (button or dropdown) to change stage. If a stage transition is not allowed (for example, moving backwards from Due Diligence back to Screening might be disallowed or require a reason), the UI can disable or warn accordingly. - If stage change triggers events like tasks generation (e.g., moving to Due Diligence might auto-populate a checklist of diligence tasks), the UI after stage change will reflect those new tasks in the deal's compliance section. - Possibly a timeline of stage changes can be shown in the deal detail (history of status) so users can review how long it took and past decisions.

Backend Implementation: - **Stage Model:** A `stages` reference table (or enumerated type) defines all possible stages. Fields might include: id, name, order_index, category (perhaps to identify which are terminal like "Closed Won" vs "Closed Lost"). Alternatively, two booleans: `is_won_stage`, `is_lost_stage` to mark outcomes, and maybe `is_active = false` if it's an end state. - **Deal Record Fields:** Each deal record has a `stage_id` (FK to stages) or `stage` string, and possibly a `status` that can be "Active/Open" vs "Closed" (with outcome). But we can infer status from stage (like any stage with is_active=false means closed). - **Stage Transitions:** The logic for moving stages is handled in `DealService.advanceStage(deal, newStageId, user, optionalReason)`. This function will: - Check that the new stage is different and is a valid next step. We could maintain a simple rule that any forward progression is allowed, or a more complex rule via a transition matrix. For initial version, assume stages have an implicit ordering and you should generally not go backwards (except maybe reopening a

passed deal, which an admin could allow). - If the new stage is a closed stage (like “Passed” or “Funded”), handle accordingly: - If “Funded/Closed-Won”, the system might prompt to input or confirm the final investment details (like confirm which fund invested and amount) if not already done. This could be part of Fund Allocation module rather than here, but the stage change might enforce that fund allocation exists. - If “Passed/Closed-Lost”, save the reason provided (e.g., in a field `pass_reason` on the deal or as a note). - Mark the deal as closed and maybe remove from active pipeline queries (still in DB for record). - If moving into a stage like “Due Diligence”, the service could auto-create a set of `ChecklistItem` entries for that deal, based on a template. (We might have a predefined list of tasks e.g., “KYC form”, “Background check”, etc.) - If moving to “IC Review”, perhaps trigger a notification or calendar event to schedule an IC meeting (if such automation is desired). - Always update the deal’s `stage_id` and save. Create an entry in `deal_stage_history` table logging old stage, new stage, timestamp, and user. - Emit a `DealStageChanged` event with details, which NotificationService or others can listen to. - **API Endpoints:** - For REST: we could allow stage change via a generic update `PUT /api/deals/{id}` if stage is just one of the fields (with server logic intercepting if needed), or have a special endpoint `POST /api/deals/{id}/stage` with { newStage, reason } in body to explicitly handle transition. The latter might be cleaner to encapsulate logic and validations. - For GraphQL: A mutation `advanceDealStage(dealId, newStageId, reason): Deal` which performs the action and returns the updated deal. - **Data Model:** - `stages` table (stage_id PK, name, order, is_closed, is_won) – populated with something like: 1: Screening, 2: Due Diligence, 3: IC Review, 4: Approved, 5: Funded (won), 6: Passed (lost). (Exact stages may differ per Doliver’s process – this should be refined with their input and made configurable if needed.) - `deal_stage_history` table: id, deal_id (FK), from_stage_id, to_stage_id, changed_by (FK to users), changed_at timestamp, reason (nullable text). - **Access Control:** Possibly restrict certain transitions: - E.g., Only users with role “Partner” or “Admin” can move to “Approved” or “Funded” stage as that implies an investment decision. - Only Admin can reopen a passed deal. - The system will check `user.role` in `advanceStage` and throw an authorization error if not permitted. The front-end should hide or disable such moves if the user lacks rights. - **Integration with other Modules:** - Stage changes to Funded should tie into Fund Allocation: likely, to mark a deal Funded, at least one FundAllocation entry should exist, so `advanceStage` might validate that condition. - Stage changes to any stage could be used to trigger an AI memo refresh. For instance, after Due Diligence, the user might want an updated memo; we could auto-run AIService on reaching IC Review stage to prepare a final memo draft for IC. - If calendar integration is in use, reaching IC Review stage could auto-create a calendar event draft for the IC meeting (if date known or to be set). - **Frontend Feedback:** On a successful stage change, the UI should reflect it immediately (moving card to new column, updating detail view). If error (e.g., not allowed or missing data), show a message. - **Reporting:** The pipeline view is itself a live representation, but we might also provide counts. The analytics module (later) will use stage info to compute metrics like conversion rates (how many deals from Screening eventually got Funded) [6] . - **Testing:** Ensure that stage transitions enforce rules: e.g., cannot move to Funded without allocation (if that rule is set), cannot move to IC if compliance tasks not done (if we choose to enforce that), etc. Also test that history is logged correctly. Also test permission scenarios.

Implementation Notes: Customizing stages – if Doliver wants to rename or add stages – would require either a UI to manage the `stages` table or at least making it easy in config/seed data. For now, we assume we know the stages and seed them. The PRD should list the assumed stage names according to Doliver’s process (to be refined by stakeholders). Example: - *Stage 1:* Deal Intake/Screening (initial review)
- *Stage 2:* Due Diligence (deep dive)
- *Stage 3:* IC Review (prepare for investment committee)
- *Stage 4:* Approved (IC approved, term sheet issued)

- *Stage 5:* Funded (deal closed successfully) – closed-won
- *Stage 6:* Passed (deal declined) – closed-lost
These can be adjusted to their nomenclature.

By providing a robust pipeline management interface, the app ensures nothing falls through the cracks and everyone knows the status of each opportunity [7] [8]. A centralized pipeline "with **customization and collaboration**" improves the investment team's efficiency [9].

3. Fund Management and Allocation Tracking

Description: Doliver handles multiple investment funds or pools of capital. This module tracks the funds and how each deal is allocated to one or more funds. It ensures that for every investment made, the source of capital is recorded and that fund utilization is monitored. Additionally, it sets the stage for capital call forecasting by showing how much each fund has committed and what remains uncalled.

User Stories: - As a fund manager or CFO, I can create and edit **Fund** records with details like fund name, total committed capital, and available capital, so the system knows how much we have to invest. - As an investment team member, I can **allocate a deal to one or more funds** when we decide to invest, specifying how much each fund will invest, so that our records reflect which fund finances the deal. - I want to see on each deal which fund(s) and what amounts are allocated, and on each fund, see all deals (investments) that fund has made. - If I try to allocate more money than a fund has available, the system should warn or prevent it. - I want to track the status of an allocation (e.g., proposed vs. committed vs. funded), possibly to differentiate between an allocation plan and actual money transferred.

Frontend Implementation: - **Funds Management UI:** There will be a **Funds List page** showing all funds as cards or rows. Each entry includes fund name and key stats (committed capital, % invested or remaining, number of deals). - A **Fund Detail page** shows details: fund name, total committed, capital called to date, remaining to invest, and a list of deals allocated to that fund. Possibly a chart (like a pie) of allocation by sector or stage to see diversification. - An **Add/Edit Fund form** (accessible to authorized users, likely only Admin or CFO roles) to create a new fund or update details. Fields: Name, Description, Total Committed Capital, Launch Date, perhaps Fund Type (e.g., Fund I, II or separate accounts). - **Deal Detail Integration:** On each Deal's detail page, under a section "Fund Allocation" or "Investment Plan," list current allocations: e.g., "Fund Alpha: $500k (Committed)" – including status if relevant. There's an **"Allocate Fund"** button or link, which opens a form to add a new allocation or edit existing: - The allocation form allows selection of a fund (dropdown of available funds), entering an amount, possibly selecting a status (like "Proposed" vs "Committed" vs "Executed"), and a date (e.g., date of commitment or expected funding date). - Validation: amount should be >0 and <= fund's remaining available. - If the deal is already closed/funded, mark allocations as committed or executed. - For deals in early stages, team might do "what-if" allocations (proposed). The UI should perhaps mark those differently (italic or a tag "proposed"). - Possibly, if multiple funds can co-invest, allow adding multiple entries. If only one fund can invest per deal in Doliver's context, then it's simpler (but typically multiple funds could co-invest especially if Doliver has different funds or SPVs). - The fund selection dropdown should also show how much is available in that fund for quick reference. - On the fund detail page, the list of deals could show each deal's allocated amount and stage or date invested.

Backend Implementation: - **Fund Model:** `funds` table with fields: id, name, total_committed (the size of the fund), capital_called (how much of that total has been called from investors – initially 0 until calls

happen), capital_invested (how much actually invested, or this can be derived by sum of allocations marked executed), maybe start_date, end_date (fund life). - **Allocation Model:** `fund_allocations` (or `investments`) table. Fields: id, deal_id (FK), fund_id (FK), amount, status, date_allocated (or expected date). Status might be an enum: "Proposed, Committed, Executed (Funded)" etc. Alternatively, a boolean like `is_committed` and separate field `date_funded`. But an enum is clearer for states. - **Relationships:** Each allocation links one deal to one fund with an amount. If a deal has multiple funds, it will have multiple allocation rows. If a fund invests in multiple deals, multiple rows with that fund_id. - **API Endpoints:** - `GET /api/funds` (list funds), `POST /api/funds` (create new fund), `PUT /api/funds/{id}` (update fund info), maybe `DELETE /api/funds/{id}` (if needed, though probably funds won't be deleted if they had allocations). - `GET /api/funds/{id}/allocations` or include in fund detail response the list of deals (with fields of deal and amount). - `POST /api/deals/{id}/allocations` to create a new allocation for a deal (with body including fund_id, amount, date, status). Or a more RESTful could be `POST /api/allocations` with deal_id in body. - `PUT /api/allocations/{id}` to edit an allocation (e.g., change amount or mark executed). - GraphQL: types `Fund`, `Allocation`, and updated `Deal` type that contains `[Allocation] allocations`. Mutations: `createFund, updateFund`, `allocateToDeal(dealId, fundId, amount, status)`, etc. - **FundService & Allocation Logic:** - A `FundService.createFund(data)` for admin to add funds. - A `AllocationService.allocate(dealId, fundId, amount, status)` (or this can be in FundService or DealService as well since it ties both). This will: - Validate the deal and fund exist and are active. - If status is "Committed/Executed", ensure that the deal's stage is appropriate (maybe you can only mark executed once the deal is closing) or allow anyway but the presence of an executed allocation likely implies the deal will be marked Funded. - Check fund availability: For example, a rule: `amount <= fund.total_committed - fund.capital_invested`. If not, throw error. - Create or update the allocation record. If new, insert; if editing existing (like changing from proposed to committed or adjusting amount), update accordingly. - Possibly update cached totals on fund: e.g., increase a field `capital_allocated` on fund by that amount (especially if committed or executed). We might maintain `capital_allocated` (sum of all committed allocations for the fund) and `capital_remaining = total_committed - capital_allocated`. Alternatively, calculate on the fly or via a view. For simplicity, updating a cached field can be done but must be consistent (update on every allocation change). - If an allocation is marked executed, we could also update `fund.capital_invested` (which might be same as allocated if we only mark once actual invested). - For multiple proposed allocations, we could allow going over temporarily, but perhaps not – best to always enforce fund limit even for proposed, to be realistic. - Return the created allocation or updated info. - Possibly trigger events: If a deal now has any committed allocations, maybe that triggers the stage change to Funded (but better to let user explicitly change stage when ready). - If a fund gets fully allocated (capital_remaining = 0), maybe notify or mark it closed for new investments. - `FundService.getFundDetails(fundId)` to return fund info plus list of allocations (with deal info joined). - **Call Forecasting Tie-in:** - The data from allocations feeds the Call Forecasting module. Each allocation likely has or can have an expected funding date. For forecasting, we might assume deals at IC or Approved stage will close by a certain date (perhaps store an expected close date on the deal or allocation). The forecasting will sum amounts by date per fund. This is covered in the Call Forecasting section, but note that the input for forecasting is here. - **Edge Cases:** - If an allocation is removed or reduced after being committed (maybe a deal fell through after planning), the system should adjust fund availability back. So allow deletion or editing and carefully update totals. - If one deal is funded by multiple funds, ensure the sum might exceed deal's required amount? Actually, define if the deal has a target amount and enforce sum of allocations <= target or not needed. Possibly track `deal.target_amount` and ensure not allocating beyond it. Or the target can be just informational. - If Doliver co-invests with external investors (not Doliver's funds), those external portions might be out of scope to track. We might only track our funds.

If needed, we can have an “external share” field in deal but not needed for core function. - **Permissions:** Creating or editing funds maybe admin only. Adding allocations likely any deal team user can propose, but committing perhaps restricted to senior roles. For now, assume all authenticated can do it, refine later if needed. - **UI Notifications:** - If allocation is successfully added, update the UI list on deal detail immediately. Possibly also update the fund detail if that is open. - If insufficient fund balance, the API returns error which UI shows: “Allocation exceeds available capital in the selected fund.” - **Testing:** - Test allocation under normal conditions and exceeding conditions. - Test multiple funds allocated to one deal. - Test editing/deleting allocations updates totals correctly. - Test that retrieving fund detail yields correct aggregated info.

Implementation Notes: This module is crucial for tying the pipeline to actual investment decisions. Even before deals close, tracking which fund *would* invest can guide strategy (ensuring we don’t overcommit a fund). Additionally, because we plan to implement call forecasting, capturing not just that a fund will invest, but *when* the cash is needed (via expected close date) is important. One approach is adding `expected_close_date` on deal or on allocation. We will incorporate that in the data model, as it drives the timeline of capital calls.

4. AI Analyst & Investment Memo Generation

Description: The AI Analyst module (also referred to as AI Investment Memo) leverages artificial intelligence (likely GPT-4 or similar LLM) to generate analytical content and summaries for deals. In v1, there may have been a basic AI memo feature; in this refined version, we **enhance the AI Analyst** to provide more robust support – including generating full investment memos, answering questions about deals, and possibly analyzing provided documents or data. The aim is to save the team time in drafting memos and to surface insights that might not be immediately obvious.

User Stories: - As an analyst, I can click a button to **generate an investment memo** for a deal. The AI will produce a well-structured draft covering key aspects (company overview, market analysis, team, financials, risks, etc.), which I can then refine. - As a user, I can provide additional inputs to the AI (like a pitch deck file or specific questions) to get more tailored outputs. *(For example, I upload a pitch deck PDF and the AI summarizes it, or I ask “What are the biggest risks for this deal?” and get an answer.)* - I can see the AI-generated memo on the deal’s page, edit it if needed, and save final notes. - If the AI generation fails or seems inaccurate, I can regenerate or adjust inputs. - (Future/Stretch) I can have a Q&A chat with an AI agent about the deal, which remembers the deal context during that chat session.

Frontend Implementation: - On the Deal Detail page, have an **“AI Analyst” section** or tab. This might have a default view showing the latest AI memo (if generated). - If no memo is present yet, show a prompt like “No AI memo generated yet. Click below to generate a comprehensive investment memo using AI.” and a **Generate Memo button**. - Possibly allow the user to choose what data to base the memo on: e.g., checkboxes or options like “Include company description, include team bios, include financials”. Or simpler, always include everything we have. - When generating, show a loading indicator (and possibly some witty “AI is thinking…” message). - Once generated, display the memo in a scrollable text area or rich text viewer. Allow user to click an **“Edit”** button if they want to make changes (which then perhaps allows them to save those changes as a user-edited memo separate from AI original – or just override it, depending on design). - Provide a **Regenerate** option if they want a new draft (maybe warn it will overwrite current AI draft unless we version it). - If interactive Q&A is supported: a sub-component like a chat interface where the user can type a question (“What is the valuation of the company?”) and the AI responds. The context given to the AI

would be the deal's info. This requires storing a conversation state. If implemented, it could be another tab "Ask AI" with a chat log. - Support **file upload** in this section (for example, "Upload Pitch Deck for AI analysis"). If user uploads a PDF, the front-end sends it to the server (which might use OCR or extract text to feed into AI). This is a complex feature; if not fully doing, we mention it as future, but the UI could allow it if we integrate an API that can read PDFs (OpenAI can take text if we extract, or other services). - The AI section might also allow selecting a template or style for the memo (maybe one day multiple output formats). For now, one standard template is fine. - Provide disclaimers in UI: e.g., "AI-generated content – please verify for accuracy."

Backend Implementation: - **AI Service Integration:** Using OpenAI's API (for example, GPT-4 model) via their REST API. We need an API key (stored in env). The `AIService.generateInvestmentMemo(dealId)` will: 1. Gather relevant data about the deal from the database: company description, sector, what problem they solve, any financial info (if we have fields for revenue, etc., though maybe not), the stage, maybe team info if recorded (not sure if we track team bios, likely not in our data model unless stored in notes). 2. Construct a prompt. Possibly a template like: - "You are an AI investment analyst. Write a detailed investment memo for the following opportunity. Company: {name}. Description: {desc}. Sector: {sector}. Amount seeking: {amount}. The market: {maybe some text if available}. Team: {if we have any data}. Provide sections: Company Overview, Market Opportunity, Team, Financials, Risks, Recommendation." - We might have some static text plus plug in the fields. - If the user provided specific input (like a question or an uploaded text from a document), append that as well. 3. Call the OpenAI completion API with that prompt, requesting a reasonably large max tokens to get a few paragraphs for each section. Use appropriate temperature (maybe low-ish for factual consistency). 4. Receive the response (which is the draft memo text). 5. Save this in a `memos` (or `ai_memo`) table linked to the deal (content, created_at, perhaps store which model was used and prompt for auditing). 6. Return the content. - **Document Analysis (if implemented):** If user uploads a document, we need to extract text. We could use an external library or API for PDF text extraction (perhaps a Python library or an online service). If integrated, `AIService.extractAndSummarize(dealId, file)` would extract text and maybe summarize it or feed chunks to GPT to summarize the deck. This can get complex (maybe out-of-scope for initial PRD, but mention as future). - **Q&A Feature:** If implementing, the service might maintain a conversation state by storing the last few Q&A in memory or context. Possibly simpler: treat each question independently but always prepend deal info. Or use OpenAI's chat endpoint with a messages array that includes system message (with deal context) + previous QAs. Given complexity, we might limit to independent queries with context rather than a long chat memory. - **Performance:** Generating memos could be slow (several seconds). We might mark the operation as asynchronous. In a synchronous API call, we might hit request timeout if it takes too long. For the initial version, a sync approach with a reasonably high timeout might be okay (OpenAI responses might come in ~5-15 seconds for a long memo). If needed, an asynchronous approach: user triggers generation, we immediately respond with a job ID and status=processing; then the client polls an endpoint or uses WebSocket to get the result when done. However, that adds complexity. Possibly simpler: just have the frontend wait (with loading spinner) until response, which is acceptable as long as it's not extremely long. - **AI Result Storage:** Save memos to avoid re-calling API unnecessarily. We can store one memo per deal or multiple versions (maybe keep a history or just the latest). If storing multiple, have a boolean for `is_latest` or timestamp. - **Editing and Saving:** If user edits the memo text in UI, we should save that to perhaps the same table but mark it as "human edited". We might want to distinguish AI version vs final version. Perhaps fields like `memo.content` and `memo.is_draft` or `memo.is_ai_generated`. Or simpler: after generation, user edits in UI and saves, we just update that record. The risk is if they regenerate, it might overwrite. We can decide to either not allow regenerate after editing (force new record), or warn that it will override. Possibly easiest: keep one

record per deal, always overwritten on regenerate, because the user can always manually save their edited copy offline if needed. But losing work is not ideal. Maybe we do: always create a new memo record on each generation, but mark only one as active. The user's edits can be saved on that record. Overwriting only happens if regenerate is pressed *before* user edits or if they confirm replacing their edits. - **API Endpoints:** - `POST /api/deals/{id}/generate-memo` triggers generation. (Alternatively a GraphQL `generateMemo(dealId): Memo` mutation.) - `GET /api/deals/{id}/memos` to retrieve memos (or include in deal detail query the latest memo). - `PUT /api/memos/{id}` to update content if user edits. - Possibly `POST /api/deals/{id}/ask-ai` for Q&A, which expects a question in body and returns answer (without necessarily storing in DB except maybe logging). - **Error Handling:** If the AI API fails (network issue or prompt too long etc.), catch and return a friendly error to UI ("AI service is currently unavailable, please try again later"). If content is inappropriate or incomplete, user can just regenerate. - **OpenAI Usage Costs:** The design should allow configuration of model and be mindful of token usage. We will likely use GPT-4 or GPT-3.5 depending on needed quality. The PRD doesn't need cost details, but just to note this should be an environment setting (model name). - **AI Compliance:** Ensure that no confidential or regulated data (like personal data of individuals beyond basic info) is being sent to OpenAI in violation of any policies. We might need to allow opting out of AI for highly sensitive deals. Possibly tag deals as "do not use AI" and skip. - **Testing:** - Since we can't easily test AI output deterministically, tests might mock the AI API response to verify our logic. For example, simulate a short dummy response from AI and ensure our service saves it to DB and returns correctly. - Test that prompt generation works with various data (some fields missing). - Test that user editing is saved and not overwritten incorrectly. - **UX Considerations:** - The AI content might contain errors or hallucinations. The UI should encourage users to verify. Perhaps a watermark or note "Draft – not verified". - Possibly highlight any fields that were unknown (some AI might just make up financials if none given; we could try to avoid that by explicitly stating "if financials not provided, say 'No data' rather than guess").

Enhanced Capabilities (New in this version): The PRD specifically asks for an **Enhanced AI Analyst**. Beyond the basic memo generation, possible enhancements: - **Multi-source analysis:** ability to incorporate multiple data sources (e.g., feed the AI both the internal deal data and an external source like the company's website content or pitch deck). - **Sector Intelligence:** the AI might incorporate knowledge of the sector (maybe via fine-tuned data or by us including a sector overview in the prompt) so that the memo includes context about the industry. - **Interactive Q&A:** as discussed, a chat interface for follow-up questions, which can help the user dig deeper into analysis without manually searching. - **Automated Insights:** The AI could not only summarize but also highlight potential red flags or positive signals by comparing the deal data to known benchmarks (this is complex but an idea: e.g., "The revenue growth is below average for startups at this stage in FinTech" – this would require data which we may not have). - **Continuous learning:** In future, the AI could learn from user edits (if many users always fix certain parts of the memo, adapt prompts). For now, we implement the core generation and possibly a basic Q&A.

Integration with Other Modules: - The generation might be triggered automatically at certain points (maybe when a deal reaches IC Review stage, call `generateMemo` so that by the time the user clicks on the deal, a draft memo is ready). - The AI content can be exported (via Export module, e.g., "Download Memo as PDF"). - If we have a compliance concern, maybe the compliance officer must approve AI memos if they are to be shared externally. - We might also use AI for other small features, like generating a one-sentence summary for the deals list or extracting keywords – those are minor, but could be future.

By leveraging AI, Doliver can drastically **speed up the memo writing process** and gain insights. This is in line with industry trends where VC firms use AI to draft memos in minutes [10]. For example, Flybridge's

open-sourced memo generator ingests a pitch deck and outputs a structured memo automatically [3] – our system similarly aims to provide a first draft that Doliver's team can refine, maintaining consistency and saving effort.

5. Analytics and Dashboard Reporting

Description: The Analytics module provides data-driven insights into Doliver's deal flow and investment activities. It aggregates data from the pipeline and presents metrics, trends, and visualizations that help users understand performance and make strategic decisions. This includes both high-level summaries (pipeline overview) and detailed breakdowns (e.g., by sector, by source, by stage conversion). A new emphasis in this version is on **Sector-level analytics**, highlighting the distribution and success of deals across industry sectors.

User Stories: - As a user, I can view a **dashboard** with key metrics such as: number of deals in pipeline, how many reached each stage, how many investments made year-to-date, and conversion rates (percentage of deals from intake that got funded). - I can see a breakdown of deals by **sector** to identify which industries we are most invested in, and how those deals are faring (e.g., success rate per sector). - I can analyze deals by **source** to see which sources yield the most investments (e.g., referrals vs inbound). - I can filter or select a timeframe (e.g., Q1 2025, year 2024) to see the metrics for that period. - I can view charts such as: bar chart of deals per stage, pie chart of active deals by sector, line chart of cumulative invested capital over time. - I want an **exportable report** (PDF or PPT) of some analytics for meetings (overlaps with Export module). - Possibly, I can customize which metrics are shown on my dashboard.

Frontend Implementation: - **Analytics Dashboard Page:** A top-level page (e.g., at `/analytics` route) containing multiple sections/cards: - **Pipeline Summary:** Show total active deals and maybe totals by stage. For example: "Active Deals: 25 (5 in Screening, 10 in Due Diligence, 3 in IC Review, 2 Funded, 5 Passed this year)". Could be textual or small bar. - **Conversion Funnel Chart:** A funnel visualization or bar series showing: 100% at intake, X% made it to due diligence, Y% to IC, Z% to funded. This visually shows pipeline efficiency [6] [11]. - **Deals by Sector:** A bar or pie chart. If the number of sectors is large, a bar chart sorted by count of deals is good. It might allow toggling between count of deals and amount invested (if we consider invested deals and sum their allocation in each sector). - **Deals by Source:** Perhaps a pie chart or table listing sources and how many deals (and how many funded) came from each [4] [5]. - **Capital Deployed Over Time:** A line chart showing cumulative capital invested (sum of allocation amounts) on the y-axis vs time on x-axis (by quarter or month). This requires deals marked as funded with dates. - **Upcoming Capital Needs:** If call forecasting is integrated, a chart or table of projected capital calls by quarter (some might put this in a separate "Capital Forecast" section, but it can also appear on analytics). - Possibly **Top N metrics:** e.g., "Top 5 largest deals closed" or "Average time to move from Screening to Funded" as a metric. - **Sector Filter:** The user might filter all charts by a particular sector or fund. However, often these dashboards are general. We might provide a filter dropdown to focus on deals within a certain fund, or certain stage (though then some charts lose meaning). - **Interactivity:** Charts can be interactive (hover tooltips with exact values, legend to toggle series). Using a library like Chart.js or Recharts. Alternatively, simple summary numbers can be shown in text if charts not needed for every metric. - **Sector Drill-down:** If user clicks on a sector in the chart, optionally navigate to a page or modal listing all deals in that sector with statuses. - **Timeframe Filter:** Possibly a dropdown for "All Time, Last 12 months, Year 2024, Year 2025, Custom Range". Filtering by date would probably filter deals by their intake date or funded date depending on context (for pipeline counts maybe intake date in range, for funded maybe funded date). - The design should not be cluttered. We can start with a fixed set of key visuals and refine with user feedback.

Backend Implementation: - **AnalyticsService (continued):** Functions to gather these metrics: - `getPipelineSummary()` – returns counts of deals grouped by stage (for active deals) and maybe count of closed deals (funded vs passed) in a given period. - `getConversionRates()` – calculates percentage of deals that progressed from one stage to the next. This might derive from stage history: e.g., out of deals created in last year, how many reached due diligence, etc. (This can get complicated; maybe a simpler metric is ratio of number funded to number intake for period). - `getDealsBySector()` – returns data of sectors with count of deals (and optionally count funded). - `getDealsBySource()` – returns data by source. - `getCapitalDeployedOverTime(interval)` – returns points for each time interval (month/quarter) with sum of allocation of deals that closed in that period. - `getSectorAnalytics(sector)` – if sector-level drill down separate, could give details for one sector. - Many of these can be done via SQL queries: - e.g., count*() from deals group by sector. - pipeline counts: group by stage for deals where stage is not a closed stage; count of funded deals (stage=Funded) and passed deals (stage=Passed) for a period. - conversion: maybe use stage_history: count deals that have a history entry reaching certain stage / total deals. - If performance is an issue, these queries might be expensive on the fly. But since data size is not extremely large for an investment firm (maybe hundreds or thousands of deals at most), it's fine. We can generate these on-demand when user opens dashboard. Possibly cache for short durations if needed. - API Endpoints: - Could have a dedicated endpoint:* `GET /api/analytics/summary` *that returns a JSON with all needed metrics. Alternatively, multiple endpoints:* `/api/analytics/deals-by-sector`*, etc. Combining into one reduces number of calls from front-end and can be efficient. - GraphQL: could either have separate queries or one query that returns a custom type with all fields for dashboard. - Perhaps define a GraphQL type like:*

```
type AnalyticsSummary {
  totalActiveDeals: Int
  dealsByStage: [StageCount!]
  dealsBySector: [SectorCount!]
  dealsBySource: [SourceCount!]
  fundedDealsYTD: Int
  passedDealsYTD: Int
  capitalDeployedTimeline: [TimeSeriesPoint!]
}
```

and a query `analyticsSummary(year: Int): AnalyticsSummary`*. - Sector-level Analytics (New Feature Emphasis): - Additional detail: The request mentions "sector-level analytics" explicitly. Possibly beyond just counting deals, it could involve analyzing performance by sector. For instance: - How many deals in sector X did we invest in vs passed? - What is the average time to close for sector X deals? - If we had portfolio outcomes (like ROI), we could compare by sector, but we likely do not track ROI here (that's post-investment). - We could incorporate external data, but likely not for now. - At least, provide a page per sector. Perhaps on clicking a sector on the dashboard, navigate to a Sector Analytics page: * Show that sector's deal list, and summary: total deals considered in that sector, # funded, # passed, conversion rate, total invested amount in that sector. * Possibly show sub-sector if applicable or comparisons. * But if Doliver's focus is broad, maybe sectors are limited and such a page is manageable. - Data for sector analysis is directly from deals and their statuses. We ensure the model has a sector field (likely via company.sector). - Compliance Note: If some analytics involve sensitive personal data (like founder demographics as mentioned in Seraf article* [12] *), our system likely doesn't track those (not requested explicitly), so we skip that. If Doliver did want to track things like founder gender or similar, it could be added, but not in our scope unless mentioned. - Integration with Export: Possibly an "Export Report" button on dashboard that calls export service to generate a PDF of the dashboard charts or data tables (discussed later in Export Tools).*

- Testing: - The analytics endpoints should be tested for correctness on known dataset (maybe create some dummy deals in different categories and verify the counts). - Also test that filters (time range or by fund if implemented) are applied correctly. - Ensure performance of queries is okay by analyzing query plans if needed (but again, small data likely fine). - Real-time Updates: The dashboard need not live-update in real-time, but perhaps if the user has it open while others are editing, the data could go stale. Manual refresh or a refresh button is acceptable for now. In future, we could use WebSocket to push a refresh event if, say, a deal stage changed that significantly affects stats, but not necessary.

Sector-level Analytical Insights Example: Suppose Doliver has done 20 FinTech deals and 5 got funded. The sector analytics can highlight that 25% of FinTech deals convert to investment. Maybe in Healthcare, 10 deals and 1 funded (10%). This can inform strategy (maybe our filter or criteria is stricter in Health, or maybe fewer good opportunities). Similarly, it can show allocation: e.g., $10M invested in FinTech vs $2M in Health, which might or might not match Doliver's intended sector allocation strategy.

By providing these analytics, the app ensures Doliver can **"turn data into insights" with simple metrics and dashboards** [2] [11], helping them refine sourcing and investment focus.

6. Capital Call Forecasting (New Feature)

Description: This new module provides forecasts for capital calls needed for each fund based on the pipeline of deals and their fund allocations. In a private fund context, a **capital call** is when the fund asks its investors (Limited Partners) to contribute a portion of their committed capital to finance an investment. This feature will project upcoming capital calls (amounts and timing) so that Doliver's finance team can plan and so that LPs can be notified ahead of time.

User Stories: - As a fund manager, I can see a **schedule of projected capital calls** for each fund, so I know when we will likely need to call money from investors and how much. - The system uses deals that are likely to close (e.g., at IC approved stage) and their allocated amounts to determine these projections. - I can adjust assumptions such as expected close dates or probability of closing for deals to see how it changes the forecast. - I want reminders or calendar events for upcoming capital call dates so nothing is missed. - I can mark a projection as executed once we actually call the capital.

Frontend Implementation: - Possibly integrate this into the **Analytics page** or have a dedicated **Capital Calls** page under Funds. - **Fund Detail Page Augmentation:** On each Fund's page, include a section "Capital Call Forecast": - Show the fund's basic data (committed vs called vs remaining). - A timeline view or list of upcoming calls. For example: * Q3 2025: Projected call \$1.2M for Deal X and Deal Y. * Q4 2025: Projected call \$0.8M for Deal Z. * Total remaining uncalled after these: \$ (some number). - The timeline could be quarterly or monthly. The resolution depends on how precisely deals have expected dates. Possibly just by quarter is enough for planning, but we can allow exact months if we have close dates. - If a deal's expected close date is known, use that. If not, perhaps assume next upcoming quarter or half-year. - If deals are probabilistic, one could weigh them, but initial likely treat all active deals beyond a certain stage as going to happen. - **Global Capital Calls Page:** Alternatively or additionally, a page that shows all funds' projected calls aggregated, or a calendar view combining them. Maybe a calendar interface highlighting months with large calls. - **Interactivity:** Let user adjust: - On each deal in the pipeline, possibly set an "Expected close date" field. We should include that field in deals or allocations. The UI may allow editing it on the deal detail (for forecasting). - Possibly set a "Probability" (like 80% chance to close). But to keep scope manageable, we might not do probabilities now; treat it binary (either we include it or not, e.g., maybe only

include deals that are at least at IC Review stage, meaning high likelihood). - If including probability, we could show both weighted expected calls vs full value. That might be overkill for now. - Marking done: After a deal closes and we actually call capital, the forecast entry should be marked as done. Possibly the UI for each fund could allow marking a projected call as executed (which essentially means the deal is funded and capital was called). - **Notifications:** The UI could have an option "Add to Calendar" or automatically do so via Calendar integration (see Calendar Integration module). E.g., a button next to a projected call "Schedule LP call notice" which triggers adding an event to Outlook/Google on the expected date.

Backend Implementation: - **Data Requirements:** For forecasting, each relevant allocation needs an expected timing. Options: - Add `expected_call_date` or `expected_close_date` to the `fund_allocations` table (or use deal's expected close date if all allocations likely called at deal close). - Alternatively, track at deal level if all funds are called at once: e.g., `deal.expected_close_date`. If a deal has multiple fund allocations, presumably all happen at that close event (assuming simultaneous closing). - We'll plan to add `expected_close_date` on Deal, which can be set when the deal is near closing (like at Approved stage). If not set, we may use a default like end of current quarter or something. - **Forecast Algorithm (simple):** - Consider all deals in pipeline that are not yet funded but likely will be (maybe those in stage >= "Approved" or maybe "IC Review" or beyond). Alternatively, allow user to flag which deals to include. - Sum up their fund allocations per fund grouped by an expected date. - For each fund, produce a list of (date, amount) pairs. - If multiple deals in same quarter for a fund, sum them to one call or maybe separate if user wants detail. Possibly one call per quarter containing multiple investments is how they'd typically do it. - Ensure that the cumulative called doesn't exceed total committed for the fund (which it shouldn't if data is consistent). - Also consider funds might have already called some capital for past deals; those we treat as done (they contribute to capital_called). - We might also incorporate a minimum call amount (some funds prefer not to call very small amounts individually, but likely out-of-scope detail). - **Forecast Data Output:** - `CapitalCallService.getForecast(fundId)` returns an array of forecast entries with fields: fund_id, expected_date (could be standardized to quarter start or specific date), total_amount, details of which deals. - `CapitalCallService.getAllForecasts()` to get data for all funds (for a combined view). - **API Endpoints:** - `GET /api/funds/{id}/forecast` returns upcoming calls for that fund. - Perhaps `GET /api/forecast` returns for all funds or across funds. - If editing expected date or probability: - `PUT /api/deals/{id}` with expected_close_date or `PUT /api/allocations/{id}` with expected_date if per allocation. - We could also have an endpoint to mark a forecast executed, but executing is basically the deal closing which we already handle via stage and allocations set as executed. So maybe not needed separately. - **Integration with Fund Data:** - When an allocation is marked executed (deal funded), we should increment `fund.capital_called` by that amount (since an actual call happened). - The forecast calculations then should consider capital_called to know how much of total is left. If a fund is fully called, no further calls obviously. - **Integrations:** - **Calendar:** Possibly link forecast entries to Calendar as events. This could be done by an option on UI or automatically create events for any upcoming calls within next X days. - **Notifications/Reminders:** The system could email reminders a few weeks before a planned call date. - **Testing:** - Create scenario: Fund A total 10M, deals: Deal1 expected Q3 2025 $2M, Deal2 expected Q4 2025 $3M -> ensure forecast lists those and the sum. Mark Deal1 funded -> ensure forecast updates (Deal1 removed from future, capital_called updated). - Test edge: no deals -> no forecast. Test extremely large deals oversubscribing fund -> probably prevented at allocation stage, but if present, forecast should show fund as over-allocated (maybe highlight in output). - **Accuracy Considerations:** Forecasting is only as good as the assumptions. We assume straightforward that all deals at certain stage will close. This might overstate calls if some deals drop. If we allowed probabilities, we could generate multiple scenarios. However, initial requirement likely just to have a heads-up of potential calls.

Example: If Doliver Fund I has \$50M total, \$30M already called (invested in past deals), remaining \$20M. There are two deals in pipeline nearing close needing \$5M and \$7M from Fund I, expected in Q1 and Q2 next year. The system would list those, totaling \$12M, meaning after them the fund would have \$8M uncalled. If one deal slips, user can adjust date, etc. The feature essentially acts like a calendar of upcoming cash outflows.

In practice, tools like Copia's capital call tracker highlight the importance of a **centralized hub to forecast future calls and review uncalled capital** [13] . Our app mirrors this, giving Doliver full visibility into pending commitments and the timing of cash needs. By forecasting and setting reminders, Doliver can ensure it never misses a funding opportunity due to timing issues [14] .

7. Calendar Integration (New Feature)

Description: This module integrates the app with calendar services (like Google Calendar or Outlook) to sync important dates and events. The goal is to ensure meetings, deadlines, and tasks related to deals are on the users' calendars, improving workflow integration and reducing the chance of missing critical meetings (like due diligence calls or IC meetings).

User Stories: - As an investment team member, I can **schedule a meeting or call** (such as a due diligence call with a company, or an internal IC meeting) from within the app, and have it automatically appear on my Google Calendar with details and invites. - I can link my Google/Outlook calendar to the app via a one-time authentication. - Key dates (like expected close dates for deals, or target dates for tasks) can be added to my calendar as events or reminders. - Calendar events are updated if changed or can be canceled from the app. - Optionally, I can see a consolidated calendar view in the app itself of upcoming events related to my deals.

Frontend Implementation: - **Calendar Sync Settings:** Under user profile or a settings page, an option to "Connect Calendar". When clicked: - If not connected, initiate OAuth flow (pop up or redirect to Google auth). - After connecting, show status like "Connected to Google Calendar as [user@example.com]" and perhaps an option to disconnect. - **Scheduling UI:** In relevant places: - On a Deal Detail page, perhaps under a "Actions" menu or next to a task, include "Schedule Meeting" or "Add to Calendar". - If a due diligence meeting date is a field in the system, an icon to add to calendar. - For IC meetings: maybe when a deal reaches IC stage, show a suggestion "Schedule IC meeting". - For capital call forecasting: a button "Add capital call event" which creates an event on the fund manager's calendar. - This likely opens a modal form where user enters event details: Title, Date & Time, Duration, Participants (emails to invite), Location/Video Link (text), Description. - Some of these can be pre-filled: e.g., Title could default to "Due Diligence: [Company Name]". Description might include a link to the deal in the app, and any context. - Participants: maybe default internal team or contact persons if we have them (we might store a company's contact email if entered). - On submit, the app calls backend to create event via API. - **Viewing Events:** We may not implement full calendar UI, but perhaps a simple list of upcoming events on a dashboard or sidebar. Or after connecting calendar, we could fetch events tagged a certain way. A simpler approach: once events are on Google Calendar, user can see them there; we just ensure creation. We might not need to read them back. - **Visual cues:** On deals with an upcoming scheduled event, the deal list or detail could show a calendar icon with date, indicating e.g., "Next meeting on Sep 10". - **Edge case UI:** If user is not connected to a calendar and tries to schedule, prompt them to connect first.

Backend Implementation: - **OAuth Setup:** - Use Google's OAuth2 endpoints. We register the app and get a client ID/secret for Google API. For Replit dev, redirect URI might be something like a Replit domain. We'll

handle the OAuth code exchange on the backend (e.g., an endpoint `/api/auth/google-calendar/callback`). - On success, store the refresh token (and access token initially) in the database, associated with the user (like fields user.google_refresh_token, user.google_cal_email). - Use scopes: `https://www.googleapis.com/auth/calendar.events` etc. - We likely only need to store refresh token and use it to get new access tokens when needed. Use a library (like googleapis for Node or MSAL for Microsoft if we did Outlook). - **Event Creation:** - Provide an endpoint `POST /api/calendar/events` (or `/api/deals/{id}/calendar-event`) that accepts event details (title, datetime, etc., plus maybe dealId or context). - Server will: - Verify user is linked to calendar (if not, return error prompting link). - Use Google Calendar API (via their Node client or direct HTTP) to create an event on the user's primary calendar. - Construct the event with given details. Include in `description` field a reference like "Deal: [Deal Name] (link: https://...)" if helpful. - Add attendees emails if provided. The user themselves is implicit as organizer or added too. - Optionally set reminders (like email 1 day before, etc., could be default). - The API returns an event ID if created. Save that in the database linked to whatever context: * We could create a `events` table in our DB to log events we created, with columns: id, deal_id (if related), user_id (owner), external_event_id (Google event ID), datetime, title, etc. This allows us to track or display them. - Return success. - If needed, similarly support `PUT /api/calendar/events/{eventId}` to update or `DELETE /api/calendar/events/{eventId}` to cancel, but we might not fully implement unless needed. - Possibly simpler: For capital call events, where maybe the event is firm-wide (like notify multiple team members), we might create one event and invite others. But each user might prefer to create their own events. - **Event Fetching (optional):** If we want to show events in-app, we could call Google Calendar API to list events (pull events for certain calendars/time range). But this requires storing access and doing periodic fetch or on page load. To limit scope, we might skip real-time fetching; the app front-end could rely on the user's actual calendar for that. If a user wants to see within app, we can fetch events that have our app's tag (if we tag them). - **Microsoft Outlook integration:** Could be added similarly using Microsoft Graph API and OAuth2, but focusing on Google first (most common). - **Calendar for tasks vs meetings:** Not all tasks should be calendar events. Only things that have a date/time. E.g., a "prepare memo by Oct 1" might not be a calendar event (that's a to-do with a deadline rather than a fixed meeting). For such deadlines, we might instead add a reminder or leave it to user to decide if they want an event. We can allow scheduling any type of reminder if user chooses. - **Security:** Store refresh tokens encrypted. Ensure that if a user disconnects or leaves, we can delete their tokens. Also handle token refresh expiration gracefully. - **Testing:** - Integration tests might simulate the Google API with a dummy stub if possible. Otherwise, manual testing needed. - Ensure that after connecting, event creation returns success and actual event appears in Google Calendar. - Test error paths: invalid token (maybe refresh and retry). - **Rate limiting and quotas:** Google Calendar API has limits; our usage should be low though (just on user actions). We should also ensure not to spam events or duplicates.

Integration Points: - Combined with Pipeline: as noted, maybe auto-suggest scheduling at IC stage. - Combined with Compliance: maybe schedule a compliance review meeting if needed. - Combined with Notifications: after event creation, we could also send invites (which Google does by emailing attendees by default). - The existence of the events table in our DB can also serve as part of the compliance logs (proving that meeting was scheduled).

With calendar integration, the Doliver app **bridges the gap between pipeline management and day-to-day scheduling**, embedding itself in the users' regular workflows. Instead of manually creating calendar events for each meeting, users save time and reduce error by one-click scheduling from the context of a deal. This fosters better follow-through on tasks and ensures key events are not overlooked.

8. Export and Report Generation Tools (New Feature)

Description: This module provides the ability to export data and generate documents for sharing or compliance. Users can export lists (like deals or investors) to CSV/Excel for further analysis, and generate formatted PDF reports of specific items (like an investment memo or a pipeline report) to share with stakeholders or meet record-keeping requirements.

User Stories: - As a user, I can **export the deal pipeline to Excel/CSV**, so I can perform custom analysis or share a snapshot with someone who doesn't use the app. - I can export a **single deal's data** (including its details, notes, and allocations) as a PDF report for filing or sharing with the investment committee. - I can export the **analytics dashboard** or specific charts as an image or PDF for presentations. - I can download the **AI-generated memo** as a Word document or PDF, preserving formatting. - As a compliance officer, I want to export the **audit log or compliance checklist** for a deal (or all deals) to provide to regulators or include in our compliance files.

Frontend Implementation: - **Export Options in UI:** - On the Deals list page, an "Export" button that when clicked might either immediately download a CSV of all current filter results or open a dialog for options (e.g., "Export all deals vs only current filtered deals"). - On a Deal Detail page, a "Export Deal Report" button that triggers generation of a PDF containing that deal's info. - On the AI Memo view, a "Download Memo" link that yields a nicely formatted DOCX or PDF of the memo text. - On the Analytics dashboard, an "Export PDF Report" that captures the key charts and numbers in a PDF (maybe by server-side generation or even client-side using a library). - Possibly multiple formats: CSV for data tables (like deals, funds, allocations), and PDF for human-readable reports. - The UI should indicate when an export is being generated if it's not instant (like generating a PDF might take a couple seconds). Possibly show a spinner and then a download link. - For CSV, it's likely immediate (just constructing a string and downloading is quick, even client-side could handle it by converting JSON to CSV and creating a blob). - For PDF, often done server-side (for consistent formatting and including images like charts). But charts are on client side; to include them server might need to regenerate or screenshot. Simpler: we could implement a client-side "print to PDF" using the browser's print if allowed. But to automate, a server approach is more controlled. - Potential approach: Use a library like jsPDF or html2canvas in front-end to capture the dashboard as an image/PDF. But those can be finicky. Alternatively, send data to server and use a PDF generation library (like ReportLab for Python or Puppeteer to print a hidden browser page to PDF for Node). - For MVP, focus on data export (CSV) and memo export (PDF). - Also, compliance export: perhaps on a Compliance page or for each deal's compliance tab, an "Export Compliance Report" which includes the checklist items and their status, plus stage history.

Backend Implementation: - **CSV Export:** - Could be done client-side (less burden on server). But if doing server: - Endpoint like `GET /api/export/deals?format=csv&filter=...` which returns `text/csv` content. The server can query deals (with any filters provided) and format CSV lines. This is straightforward since it's just text. - Similarly `/api/export/funds.csv`, `/api/export/allocations.csv`, etc., if needed. - It's often fine for client to do CSV by fetching data then converting, but large data might be heavy. However, likely not extremely large (maybe hundreds of deals at most). - We'll implement at least the deals export on the server for convenience. - **PDF Report Generation:** - Using a library like Puppeteer (headless Chrome) to generate PDFs from an HTML template, or a PDF library to compose content. - For a deal report PDF: We can create an HTML template with the deal's info: - Title: Deal Name, Date - Sections: Overview (desc, sector, source), Current Stage, Team (if we had any team info in notes), Fund Allocations, AI Memo content (if desired), and compliance checklist status. - Then convert that to PDF. - Or directly use PDF kits (like PDFKit for Node or ReportLab for Python) to draw text. That's more work for formatting but doable. -

Chart images: If including charts (like for analytics report), we might need the chart rendered as an image. We could have the front-end send the rendered chart image (e.g., charts can be rendered to canvas and then we get dataURL and send it to server as part of report generation). Or generate charts from data using a server-side chart library (some exist but might not be trivial). - For now, maybe exclude graphical charts from PDF or use simple textual summary in the PDF, to avoid that complexity. The user can always take a screenshot of the chart if needed, but since they specifically mentioned "optimized for Replit's code AI tools", focusing on implementing straightforward data outputs might suffice. - **Export Endpoints:** - `GET /api/export/deal/{id}.pdf` – generates a PDF for that deal. - `GET /api/export/memo/{memoId}.pdf` or maybe combine with deal export (like including memo). - `GET /api/export/analytics.pdf` – generates a PDF of key analytics. - Implementation would gather data, fill a template, output PDF bytes. - Ensure proper content type (application/pdf) and use content-disposition header to prompt download with a nice filename. - **Compliance Export:** - Possibly `GET /api/export/compliance?dealId=...` to get either a specific report or for all. Could be a CSV of all compliance tasks and whether done, or a PDF summary per deal. - Could be combined with deal report (like a section in deal PDF). - **Permissions:** Only authenticated can call these, which is fine. If there were user-specific restrictions, those apply to underlying data fetch (e.g., if user can only see certain deals, the export honors that). - **Testing:** - CSV: generate with sample data and verify formatting (commas, quotes). - PDF: might be tricky to test automatically; could check that endpoint returns a PDF (by checking header and maybe that content length >0). - Probably will need manual verification of PDF content layout. - **Replit Consideration:** Running headless Chrome for Puppeteer might be heavy on Replit; might need config and can slow things. Could use a simpler PDFKit approach which is lighter but yields less pretty output. - Possibly skip heavy formatting, just output a textual report or an HTML that user can print.

UI Download Behavior: Ensure when an export request completes, the browser triggers a download. If using fetch via JS, might need to programmatically create a blob and link or simply set window.location to the endpoint (which triggers download directly if response headers are right). Possibly simpler: normal anchor link to the API endpoint (with target=_blank or download attribute) could do it. But for authenticated, need token; could embed token in query param (not great) or use fetch and then blob. We'll likely do fetch -> blob -> createObjectURL -> download.

By enabling exports, Doliver's team can produce offline records and share information with parties not using the app, and also meet compliance record-keeping requirements. This reduces the friction in producing reports, which historically might be manual.

9. Compliance and Audit Tools (New Feature)

Description: This module ensures that the investment process complies with regulatory and internal policies. It introduces features to track required compliance tasks (like KYC, AML checks), maintain an audit trail of key actions, and enforce permission controls. It also provides a way to review and export these logs for audits.

User Stories: - As a compliance officer, I can define and assign a **compliance checklist** for each deal that must be completed before funding (e.g., KYC verification, legal review, conflict of interest check, etc.), and track their completion. - I can mark these compliance tasks as completed and attach evidence (like documents or notes). - The system prevents moving a deal to the final Funded stage until all compliance tasks are done (or at least warns). - All critical actions in the system (deal creation, stage changes, fund allocations, memo generation, etc.) are **logged with user, timestamp, and details** so we have an audit

trail. - I can view an **Audit Log** page listing these events and filter by deal or user or date. - The app supports **role-based access control (RBAC)** so that only authorized people can perform certain actions or view certain data. For example, only compliance role can mark compliance tasks, only admin can delete data, etc. - I can export compliance logs or checklist status (addressed in Export module).

Frontend Implementation: - **Compliance Checklist UI on Deal:** On each Deal Detail page, include a **Compliance** section/tab (visible to users with appropriate role, maybe read-only for others). This section lists required tasks (checklist items) with a checkbox or status dropdown (Not Started, In Progress, Completed). For example: - KYC/AML Verified – [ ] (checkbox or toggle, plus maybe who/when completed) - Legal Review Completed – [ ] - Compliance Approval – [ ] (for final sign-off) - Each item might have a space for notes or link to a document (e.g., attach KYC report PDF). - A compliance officer can check them off. When clicking an item, possibly a modal: “Mark as completed by [User] on [date]. Add notes or upload file if needed.” Then save. - Completed items show a checkmark and maybe greyed out or moved to bottom. - If any item is incomplete, and someone tries to move the deal to "Funded", the UI should warn: "Compliance checklist not complete. Are you sure?" and ideally block or require an override by compliance. - **Audit Log UI:** Possibly an admin page “Audit Logs” listing events in a table: - Columns: Date/Time, User, Action, Details, Object (like deal or fund reference). - e.g., "2025-08-01 10:45, jsmith, Stage changed, Deal ACME Corp from Screening to DD", or "Allocation added, Deal X: Fund A $2M". - Provide filters by date range, user, or free text search. - This can be large, so maybe backend pages it or we show last 100 events by default with a load more. - If needed, separate logs by category (deal changes vs user logins etc.), but one table with filter is fine. - **Role Management UI:** Perhaps a simple way to manage user roles: - On a User admin page (if we have one) an admin can set a user's role from a dropdown. Or at least during user creation. - Could be out-of-scope to build elaborate UI if user management isn't a focus. Could also just seed roles or require config, but ideally some UI. - At minimum, ensure roles are displayed (e.g., in a user profile, "Role: Analyst"). - **Permission Feedback:** The UI should hide or disable actions the user is not allowed to do: - e.g., if not compliance role, maybe you can't check off compliance tasks (just view them). - If not admin, you might not see the "Manage Users" or "Delete Deal" button. - If not part of deal team, maybe you can't edit a deal (if we implement such concept). - These rules mirror backend checks, but doing it in UI is for UX clarity.

Backend Implementation: - **Compliance Checklist Model:** - Possibly a `compliance_tasks` reference table listing tasks and maybe which stage they are required by. - A join table `deal_compliance` or `deal_tasks` with fields: id, deal_id, task_name (or task_id if referencing the ref table), status, completed_by (user id), completed_at, notes, document_id (if evidence file). - We might simplify and not have a ref table, just have a predefined list coded, but a table allows configuration. Perhaps seed tasks like "KYC", "Legal", "Approval". - Maybe an order or priority too. - **Enforcing on Stage Change:** In `DealService.advanceStage`, if target stage is a closed/won stage (Funded) and the deal has compliance tasks not completed, it should either: - Block and throw an error "Compliance tasks not completed", unless user role is admin and maybe override allowed. Could require a parameter like override=true, but easier to just block and require tasks completion first. - If we want to allow override in exceptional case, maybe allow an admin to force it (an admin could temporarily disable tasks or manually mark them done). - **Compliance Task API:** - `GET /api/deals/{id}/compliance-tasks` returns list of tasks and their status. - `PUT /api/deals/{id}/compliance-tasks/{taskId}` to update status (complete/incomplete) and add notes. - Or a specific endpoint `POST /api/deals/{id}/compliance-tasks/{taskId}/complete` with body containing maybe note or file reference, marking it done. - Possibly also allow adding a custom task on the fly (some systems let you add an extra requirement). Could be out-of-scope now; assume static list. - **Audit Log Model:** - A unified `audit_log` table: id, timestamp, user_id, action_type, object_type, object_id, details (text/json). - Every place where significant data changes,

we insert a log: - Deal created, updated, deleted. - Stage changed (with from/to stage). - Fund created, edited. - Allocation added/changed. - Compliance task marked done. - AI memo generated (maybe log that content was generated, though not sure if needed for compliance – but maybe log the fact generation happened, not necessarily storing content here). - User login success/failure (could log logins for security monitoring). - We might not log every view or minor thing; focus on changes. - Could also log when exports are done or when external integrations triggered events (less critical). - The log might store detail JSON for flexibility (like {"dealId":123,"from":"Screening","to":"DD"} for stage change). - **Permission (RBAC):** - Define roles: For instance, "Admin", "Analyst", "Partner", "Compliance". - The `users` table should have a role field (string or enum or separate table if we allow multiple roles per user). Simpler to allow one role per user for now. - In each service or controller, check role if the action is restricted: - Only Admin can manage users and funds (maybe). - Only Compliance can complete compliance tasks (or admin too). - Only certain roles can advance to certain stages (we mentioned possibly restrict final approval to Partners). - Regular Analysts might not delete deals, etc. - Could implement a simple check like:

```
if (action requires 'Compliance' and user.role != 'Compliance' && user.role != 'Admin') then throw
```

Or maintain a mapping of actions to allowed roles. - GraphQL: can use context with user role to authorize resolvers similarly. - **Data Security:** - Ensure that queries also respect any data partition if needed (for example, if someday deals are restricted by team; currently not mentioned, so likely all have access to all deals). - For multi-tenant scenario or more confidentiality, we'd add fields like each deal has an owner or team and restrict others. But if all one org, not needed. - **Document storage for compliance:** If compliance tasks need attachments (like a signed doc), we already have document storage plan from earlier. Just ensure linking possible. - **Testing:** - Try stage change to funded with tasks incomplete -> should fail. - Complete tasks then stage change -> succeed. - Ensure tasks are created for new deals (maybe auto-add tasks when deal reaches due diligence or creation? Could either create tasks at deal creation for all tasks, or only generate when needed. Perhaps simpler: create them upon deal creation so they exist as "not started". Or create when a deal reaches a certain stage. If tasks are always required by final stage, creating them at creation is fine, they can sit incomplete until needed). - Test log entries: create a deal, see an entry; change stage, see entry; etc. - Test role restrictions: e.g., a user with Analyst role calls compliance complete endpoint -> should get forbidden.

Compliance Checklist Trigger: - When to populate tasks for a deal? Options: - On deal creation, create all tasks (some might not apply if deal is dropped early, but that's okay). - Or when deal reaches a certain stage (like due diligence or IC) then populate tasks. But that's extra logic and if someone forgets to trigger, might miss tasks. - Simpler: create at start. If a deal gets closed early (Passed at screening), those tasks remain not done, which is fine (we might not care if passed deals didn't do KYC since they didn't invest; maybe we mark them N/A or leave incomplete but since the deal didn't proceed it's okay). - Or we can mark tasks as only required if deal funded; but to avoid confusion, perhaps tasks just stay incomplete on passed deals and we can filter them out when looking at open deals. - The compliance officer can ignore tasks for passed deals.

Additional Compliance Considerations: - **Data retention:** Ensure that if a deal is deleted or archived, we still retain logs. Perhaps we disable actual deletion of deals; instead mark them inactive or passed. - **PII & Privacy:** If storing personal info (like founder name, which we might under company contact), ensure appropriate handling (though likely minimal). - **Legal:** The app should adhere to any legal requirements for record-keeping. For SEC-regulated advisors, usually 5-year retention of investment decisions and communications is required. Our audit log and stored memos help satisfy that. We should ensure data can

be exported or archived (export covers that to an extent). - **Security Audits:** Ensure that password storage is secure (hashed with bcrypt or similar) – we'll mention in security model later.

By integrating these compliance tools, the platform not only streamlines workflow but also **enforces checks and balances**. The compliance checklist ensures that no investment slips through without required due diligence, and the audit trail provides transparency and accountability for every action (which is crucial in financial operations). These features collectively help Doliver maintain trust with investors and regulators, demonstrating a robust control environment (e.g., role-based access and documented approvals are common requirements in audits).

Data Model and Schema Details

This section defines the full database schema for the application, including table names, columns, data types, relationships, and constraints. The schema is designed to support all the features described. We describe each major entity (table) and how they link to each other. Primary keys (PK) and foreign keys (FK) are indicated, and important indexes or constraints are noted. The schema is relational (for e.g., PostgreSQL), but the concepts apply similarly if using another SQL database.

Note: In a GraphQL context, many of these tables correspond to GraphQL object types. We'll handle API in the next section, but this mapping is straightforward since GraphQL types often mirror the data model.

Table: `users`

- **user_id** – INTEGER, primary key, auto-increment.
- **name** – VARCHAR(100), user's full name.
- **email** – VARCHAR(255), unique, not null. (Used for login username as well).
- **password_hash** – VARCHAR(255), not null. (Hashed password for authentication, e.g., bcrypt output.)
- **role** – VARCHAR(50), not null, default 'Analyst'. (Role name, such as 'Admin', 'Analyst', 'Partner', 'Compliance'. Could also be an enum type.)
- **calendar_refresh_token** – TEXT, nullable. (Encrypted token for calendar integration, if any.)
- **calendar_account** – VARCHAR(255), nullable. (Email of linked calendar account for reference.)
- **created_at** – TIMESTAMP, not null, default NOW().
- **updated_at** – TIMESTAMP.
- **active** – BOOLEAN, default true. (If user is deactivated, set false instead of delete.)

Constraints & Indexes: - Unique index on (email). - Perhaps an index on (role) if querying users by role (rare). - When storing password_hash, use a strong hashing method (bcrypt with salt). No need for reversible encryption.

Description: Stores application user accounts. Each user has a role to determine permissions. Additional fields support calendar integration. `active` flag allows soft-deletion (not removing audit trail). Users will relate to deals through actions (created_by, etc.), but not directly in a foreign key sense except in audit logs and assigned tasks.

Table: `companies`

- **company_id** – INTEGER, PK, auto-increment.

- **name** – VARCHAR(200), not null.
- **sector** – VARCHAR(100), nullable. (Industry sector, e.g., "FinTech". Ideally this is standardized; we might have a separate sectors table, but we can also allow free text or a predefined list. Consider referencing a `sectors` table.)
- **sector_id** – INTEGER, FK to `sectors.id`, nullable. (If we have a sectors reference table for a controlled list. Alternatively, skip this if we just use sector string.)
- **website** – VARCHAR(255), nullable.
- **description** – TEXT, nullable. (Brief description of the company business.)
- **headquarters_location** – VARCHAR(100), nullable. (City/Country, if needed.)
- **contact_name** – VARCHAR(100), nullable. (Primary contact person at company.)
- **contact_email** – VARCHAR(100), nullable.
- **contact_phone** – VARCHAR(50), nullable.
- **created_at** – TIMESTAMP.
- **updated_at** – TIMESTAMP.

Constraints & Indexes: - Unique index on (name) if we want to avoid duplicate company entries. However, companies could have same name in rare cases; maybe not unique, but we likely treat name as unique or at least will caution when adding duplicates. - Index on sector for analytics grouping.

Description: Represents the entity being invested in. Many deals can reference one company (for multiple rounds). Contains general info about the company. We separate this to avoid re-entering company info for each deal (and to accumulate knowledge per company). Sector can be a string or a foreign key to a small `sectors` table (which might have a list of sectors we care about). If a sectors table exists, it might have columns: id, name; not much else, maybe a category grouping or description.

Table: `deals`

- **deal_id** – INTEGER, PK, auto-increment.
- **name** – VARCHAR(200), not null. (Short name or title of the deal, could be same as company name plus round, e.g., "ABC Corp Series A".)
- **company_id** – INTEGER, FK references `companies(company_id)`, not null.
- **stage_id** – INTEGER, FK references `stages(stage_id)`, not null. (Current pipeline stage of the deal.)
- **status** – VARCHAR(50), not null, default 'Active'. (Could be 'Active', 'Closed-Won', 'Closed-Lost'. Alternatively derive from stage, but having a status field can simplify filtering active vs closed. We can maintain it in sync with stage – e.g., if stage moves to a closed stage, set status accordingly.)
- **source** – VARCHAR(100), nullable. (Deal source, e.g., "Referral", "Inbound", etc., possibly an enum or ref table.)
- **source_details** – VARCHAR(255), nullable. (E.g., "Referred by John Doe" or details of source.)
- **amount_target** – DECIMAL(18,2), nullable. (Amount of capital sought or estimated deal size.)
- **description** – TEXT, nullable. (Deal description or notes.)
- **created_by** – INTEGER, FK to `users(user_id)`, not null. (User who added the deal.)
- **created_at** – TIMESTAMP, default NOW().
- **updated_at** – TIMESTAMP.
- **expected_close_date** – DATE, nullable. (Expected closing date for the deal's funding round, used in call forecasting.)
- **pass_reason** – TEXT, nullable. (If status is Closed-Lost/Passed, a reason for passing.)

- **last_stage_change** – TIMESTAMP, nullable. (Last time stage was updated, for quick reference; also in history table in detail.)

Constraints & Indexes: - Index on (company_id). - Index on (stage_id) for pipeline queries. - Index on (status) if frequently filtering active vs closed. - Possibly index on (expected_close_date) for sorting upcoming closes (used in forecasting). - Foreign keys ensure company and stage exist, user exists.

Description: Core table for deals (investment opportunities). Contains identifying info and current status. The separation of `stage_id` (which links to a stage definition) and `status` is optional; we could just infer status from stage, but we might keep status to quickly mark deals as active vs passed vs invested. Perhaps `status` could be an enum: 'Open', 'Passed', 'Funded' making it easy to filter active ones. But the stage also gives detail. We will keep `status` updated accordingly: - If stage is a final "Funded" stage, status = 'Funded'. - If stage is "Passed", status = 'Passed'. - Else status = 'Open'.

Example Data: A deal might be: *deal_id=101, name="ABC Corp Series A", company_id=10 (ABC Corp), stage_id=2 (Due Diligence), status='Open', source='Referral', amount_target=5,000,000, expected_close_date='2025-09-30'*.

Table: `stages`

- **stage_id** – INTEGER, PK, auto-increment (or could be a small static table with fixed IDs).
- **name** – VARCHAR(100), not null. (Stage name, e.g., "Screening", "Due Diligence", "IC Review", "Closed-Won", "Closed-Lost", etc.)
- **order_index** – INTEGER, not null. (Order for display and progression.)
- **is_closed** – BOOLEAN, not null, default false. (True if this stage represents a final closed state where the deal is no longer active in pipeline.)
- **is_won** – BOOLEAN, default false. (True if this stage is a successful close, i.e., invested. If is_closed and is_won=false, that implies closed-lost.)
- **description** – VARCHAR(255), nullable. (Short description of what this stage entails.)

Constraints & Indexes: - Unique index on (name) (since small set and we use names in logic sometimes). - Possibly pre-populate with fixed values and not allow modification easily without admin.

Description: Master list of pipeline stages. Allows customizing or reordering the pipeline without code changes (to some extent). The presence of is_closed and is_won flags helps in logic (e.g., to filter active deals, or to decide status). Example entries: 1. Screening (order 1, is_closed false) 2. Due Diligence (order 2, is_closed false) 3. IC Review (order 3, is_closed false) 4. Approved (order 4, is_closed false) – optional stage to indicate IC approved but not yet closed. 5. Funded (order 5, is_closed true, is_won true) 6. Passed (order 5 or 6, is_closed true, is_won false) – note: order might be equal for Funded and Passed if we consider them parallel end states, but for simplicity, one can have a higher number.

We will align order such that typically pipeline flows in increasing order_index.

Table: `deal_stage_history`

- **history_id** – INTEGER, PK, auto-increment.
- **deal_id** – INTEGER, FK to `deals(deal_id)`, not null.
- **from_stage_id** – INTEGER, FK to `stages(stage_id)`, nullable. (Null if initial creation.)

- **to_stage_id** – INTEGER, FK to `stages(stage_id)`, not null.
- **changed_by** – INTEGER, FK to `users(user_id)`, not null.
- **changed_at** – TIMESTAMP, not null, default NOW().
- **reason** – TEXT, nullable. (Reason for change, if provided, e.g., reason for passing.)

Indexes: - Index on (deal_id) to query history for a deal in order. - Possibly index on (changed_by) if we want to find all changes by user (not common). - We can order by changed_at to get chronological progression.

Description: Logs every stage transition for each deal. This provides a chronological record of the pipeline progression and is part of the audit trail. It captures who moved the deal and when. The initial creation of a deal can be logged here too, possibly with from_stage null and to_stage = initial stage. If we wanted to track all edits, we could include other changes, but we have separate audit log for general events. This table specifically focuses on stage transitions, because that's a key metric (time in stage, etc.). Reason is mainly for when a deal is closed-lost (passed) – user can note why. Might also be used if they revert stage or any significant note.

Table: `funds`

- **fund_id** – INTEGER, PK, auto-increment.
- **name** – VARCHAR(100), not null.
- **description** – TEXT, nullable. (Maybe fund strategy or notes.)
- **total_committed** – DECIMAL(18,2), not null. (Total capital committed by LPs to this fund.)
- **capital_called** – DECIMAL(18,2), not null, default 0. (Cumulative capital called so far from LPs.)
- **capital_invested** – DECIMAL(18,2), not null, default 0. (Total invested in deals so far – could be same as called minus fees, etc. We might equate called and invested for simplicity unless some capital called not deployed yet.)
- **currency** – VARCHAR(10), default 'USD'. (Assuming mostly USD.)
- **start_date** – DATE, nullable. (Fund launch date.)
- **end_date** – DATE, nullable. (Fund end of life date, if any.)
- **created_at** – TIMESTAMP.
- **updated_at** – TIMESTAMP.

Indexes: - Unique on (name) if needed (if names like "Fund I", "Fund II"). - Could index (start_date) if listing by chronology.

Description: Represents an investment fund or pool of money. Fields track how much money it has and has used. `capital_called` will increase as we mark capital calls executed (which correspond to deals being funded). We might update `capital_called` at the same time as marking deals funded, or derive it from sum of executed allocations. For simplicity, we might treat `capital_invested` as sum of all allocations with status executed. Could keep it derived or maintain it. Redundancy with allocation table is possible, but having it can ease quick calculations and allow tracking if any difference between called and invested (like if fees or reserves in play, but out of scope).

Table: `fund_allocations`

- **allocation_id** – INTEGER, PK, auto-increment.
- **deal_id** – INTEGER, FK to `deals(deal_id)`, not null.
- **fund_id** – INTEGER, FK to `funds(fund_id)`, not null.

- **amount** – DECIMAL(18,2), not null.
- **status** – VARCHAR(50), not null, default 'Proposed'. (e.g., 'Proposed', 'Committed', 'Called' or 'Executed'. We can say 'Committed' means approved to invest, 'Called' means capital call done/executed.)
- **allocated_at** – DATE, nullable. (Date when allocation was decided or committed.)
- **funded_at** – DATE, nullable. (Date when capital was actually transferred, if executed.)
- **created_at** – TIMESTAMP.
- **updated_at** – TIMESTAMP.

Constraints & Indexes: - Unique index on (deal_id, fund_id) if we want to prevent duplicate allocation entries for the same fund and deal. However, there might be cases for multiple tranches, but likely one entry per fund per deal is enough (we can adjust amount if changed). - Index on (fund_id) for quickly summing by fund. - Index on (status) maybe (if we often query all committed vs proposed). - FK constraints ensure valid deal and fund.

Description: This join table links deals to the funds investing in them. Each record is an investment allocation from one fund into the deal. The status field tracks where in the process that allocation is: - 'Proposed' = might do this investment (not final). - 'Committed' = formally approved to invest (like IC decision made, term sheet signed). - 'Called' (or 'Executed') = capital call done, money transferred (deal closed). We can simplify statuses to just 'Proposed', 'Executed' (or treat 'Committed' as similar to 'Proposed' since effect in pipeline is similar until executed). But having 'Committed' helps differentiate between plan vs final call.

This table is crucial for both tracking exposure (how much of each fund is used) and for forecasting (we look at 'Committed' deals not yet executed to forecast when 'Called' will happen).

When a deal moves to stage Funded, presumably all its allocations should be 'Executed'. We might update them as part of closing the deal. Similarly, maybe we set them to 'Committed' earlier when IC approves.

Table: `memos`

- **memo_id** – INTEGER, PK, auto-increment.
- **deal_id** – INTEGER, FK to `deals(deal_id)`, not null.
- **content** – TEXT, not null. (The text of the memo or note.)
- **is_ai_generated** – BOOLEAN, not null, default false. (True if content was generated by AI.)
- **created_by** – INTEGER, FK to `users(user_id)`, not null. (User who generated or wrote the memo. For AI, this might be the user who initiated it, or a system user id if we prefer.)
- **created_at** – TIMESTAMP, default NOW().
- **edited_at** – TIMESTAMP, nullable. (If someone edited the content after generation.)
- **title** – VARCHAR(200), nullable. (Optional title or subject of the memo, if we want to label it; possibly not needed if content is an entire memo.)
- **memo_type** – VARCHAR(50), nullable. (Could classify memos, e.g., "Investment Memo", "Note", etc. If we allow multiple notes in this table too.)

Indexes: - Index on deal_id (to fetch all memos for a deal). - Possibly index on is_ai_generated if we query specifically (not really needed). - Full text search index on content could be useful to search notes, but not required now.

Description: Stores textual notes or memos related to deals. We decided to unify AI memos and possibly manual notes here by differentiating via `is_ai_generated` and possibly `memo_type`. If the app only uses this for the AI investment memo, then each deal might have at most one main memo in this table (or multiple versions if regenerated). We could keep multiple (like history of memos), but likely one primary. If so, we might also mark one as "latest" or just take the most recent by date.

If the team also adds their own notes (like meeting notes), they could either be stored here with is_ai_generated false and a type 'Note'. However, if heavy note-taking is expected, we might want a separate `notes` table to avoid confusion. But to limit tables, we can repurpose this: - For AI memos: is_ai_generated=true, memo_type='AI Memo'. - For human notes: is_ai_generated=false, memo_type='Note' (or 'Meeting Note', 'Decision Rationale', etc., if needed).

We'll assume memos are primarily the AI memos and maybe a few manual text.

Table: `documents`

- **document_id** – INTEGER, PK, auto-increment.
- **deal_id** – INTEGER, FK to `deals(deal_id)`, nullable.
- **fund_id** – INTEGER, FK to `funds(fund_id)`, nullable.
- **type** – VARCHAR(50), nullable. (Type of document, e.g., "Pitch Deck", "Financials", "Term Sheet", "KYC Doc", etc.)
- **file_name** – VARCHAR(255), not null. (Original file name or a descriptive name.)
- **file_path** – VARCHAR(500), not null. (Path or URL to the stored file.)
- **uploaded_by** – INTEGER, FK to `users(user_id)`.
- **uploaded_at** – TIMESTAMP.
- **description** – VARCHAR(255), nullable. (Short description or notes about the document.)

Indexes: - Index on deal_id, on fund_id. - Possibly composite index on (deal_id, type) if we often fetch by type.

Description: Handles file attachments. A document can be linked to a deal or a fund or other entities (we have separate columns; in practice, one of deal_id or fund_id or maybe user_id could be set depending on context). For example: - A pitch deck PDF belongs to a deal (deal_id set, type="Pitch Deck"). - A subscription agreement might belong to a fund (fund_id set). - A compliance file (like KYC form) belongs to a deal (and possibly type="KYC"). - Could also have general documents not tied to specific deal or fund, but not needed now.

The `file_path` could be a local path or external URL if stored in cloud. If in Replit, maybe path to `/mnt/data/files/uuid.pdf`. Or we might use base64 (not recommended for large files). We won't detail file storage beyond noting path.

Table: `tasks` (Compliance and general tasks)

- **task_id** – INTEGER, PK, auto-increment.
- **deal_id** – INTEGER, FK to `deals(deal_id)`, not null.
- **name** – VARCHAR(100), not null. (Task description or name, e.g., "KYC Verification", "Send LOI". Could also reference a template table, but name is fine.)

- **category** – VARCHAR(50), nullable. (Perhaps "Compliance" or "General", to distinguish compliance tasks from general tasks or to group tasks by type.)
- **due_date** – DATE, nullable. (If there's a deadline for the task.)
- **completed** – BOOLEAN, not null, default false.
- **completed_at** – TIMESTAMP, nullable.
- **completed_by** – INTEGER, FK to `users(user_id)`, nullable.
- **notes** – VARCHAR(255), nullable. (Any note or link about the task result.)
- **created_at** – TIMESTAMP.
- **created_by** – INTEGER, FK to `users(user_id)`. (Who created/assigned this task; could be system for default tasks.)
- **assigned_to** – INTEGER, FK to `users(user_id)`, nullable. (If a specific user is responsible.)
- **order_index** – INTEGER, nullable. (Order of tasks if needed to display in a certain sequence.)

Indexes: - Index on deal_id (to get tasks for a deal). - Index on category if filtering compliance vs others. - Index on completed (to find incomplete tasks easily). - If we implement user-specific tasks list, index on assigned_to.

Description: This table serves the compliance checklist and potentially any other tasks. Essentially, for each deal we might have a predefined set of tasks (like compliance tasks all category "Compliance"). We can also allow adding tasks (like "Follow up with founder in 1 week" as a general to-do). This design merges both concepts: - Compliance tasks: inserted either on deal creation or reaching certain stage, with category "Compliance". Usually assigned_to might be a compliance officer. - Other tasks: user can add with category "General" or something. For example, "Schedule technical due diligence call" and assign to a specific team member. These tasks could then integrate with Calendar (a separate event can be created, but the task itself is just to track that it needs doing). - Completed tasks are checked off, recorded with who/when.

This approach avoids making a separate compliance table and a separate tasks table; it's unified. If that’s confusing, we can separate, but likely fine to unify with a category.

Table: `audit_log`

- **log_id** – INTEGER, PK, auto-increment.
- **timestamp** – TIMESTAMP, default NOW().
- **user_id** – INTEGER, FK to `users(user_id)`, nullable (system actions may have null or a special user).
- **action** – VARCHAR(100), not null. (A short code for the action, e.g., "DEAL_CREATE", "STAGE_CHANGE", "LOGIN_FAIL", etc.)
- **object_type** – VARCHAR(50), nullable. (What entity the action is on, e.g., "Deal", "Fund", "User", "System".)
- **object_id** – INTEGER, nullable. (ID of the entity, if applicable, e.g., deal_id if action on a deal.)
- **details** – TEXT, nullable. (Any additional info in text or JSON. Could store old/new values or reason text. For stage changes, maybe from/to as text; for allocation maybe the amount.)

Indexes: - Index on (object_type, object_id) to query logs for a specific entity (like all logs for deal 5). - Index on (user_id) to see actions by user. - Index on (action) if filtering by type. - Index on timestamp (most queries will sort by time or range by time).

Description: A unified log of noteworthy events. This will accumulate over time. It's primarily for internal audit and possibly for debugging/tracing what happened. We may not expose all log entries to all users; maybe only Admin or Compliance can see them in an Audit UI. But we keep them anyway. Examples: - "DEAL_CREATE" with object_type "Deal", object_id 101, user=Jane, details could include "Deal ABC Corp created". - "STAGE_CHANGE" with details "Stage Screening -> DD". - "ALLOC_ADD" for allocation added. - "TASK_COMPLETE" for compliance task done. - "LOGIN" or "LOGIN_FAIL" for security monitoring maybe (object_type "User" or none). - The details could be structured JSON or just a message string. JSON might be useful to parse for advanced queries, but string is simpler. Possibly do a message in details for quick writing.

We should log enough to trace critical changes. (We might not log view events or minor edits if not needed, but creation/deletion and stage changes, etc. definitely.)

Relationships Recap: - One `company` has many `deals`. - One `deal` belongs to one `company`. - One `deal` has many `fund_allocations` (through that, to many `funds`). - One `fund` has many `fund_allocations` (to many deals). - One `deal` has many `memos` (notes). - One `deal` has many `tasks`. - One `deal` has many `documents` (and a document can belong to a deal). - Stages are referenced by deals and stage history entries. - One `deal_stage_history` entry for each stage change (many per deal). - One `deal` has many `deal_stage_history`. - One `user` can be in many logs, tasks, etc. One user creates many deals, etc. - Compliance tasks are basically tasks with category=Compliance.

Potential GraphQL Schema mapping: (We'll detail next in API, but as a quick mapping) - Deal type would have fields corresponding to deals table + relations: company (Company type), allocations (list of Allocation type), memos (list of Memo type), tasks (list of Task type), documents (list of Document type), maybe history (list of StageHistory type). - Company type with fields: name, sector, deals (list of Deal type). - Fund type: name, total_committed, etc., allocations (list of Allocation type). - Allocation type: fund (Fund type), deal (Deal type), amount, status. - Stage type: name, etc. - Memo type: content, is_ai_generated, created_by (User type), deal (Deal type). - Task type: name, category, completed, due_date, etc., assigned_to (User). - Document type: file_name, type, link, etc. - User type: name, role (but careful not to expose password etc, obviously). - AuditLog type might or might not be exposed to front-end; if we have an Audit page, then we need a type and query for logs.

We also consider constraints: - Deletions: we likely won't allow deleting deals through UI, maybe just marking passed or removing if erroneous. But if needed, ensure foreign keys either cascade or prevent deletion. Better to prevent deletion if references exist (to preserve history). - Cascade: If a company is deleted (which we likely won't if deals exist), that should cascade or be prevented. It's probably better to prevent deletion of any record that has meaningful children or use soft deletes (mark inactive). - Ensure that if a deal is deleted (again, probably not in normal use), we cascade delete stage history, tasks, memos, allocations, documents referencing it. But likely we won't allow deletion in UI to avoid losing data inadvertently.

Given a Replit environment, an initial implementation could use SQLite for simplicity which supports these schema well (with slightly different column type handling but conceptually same). Or Postgres if connecting externally.

The above describes a comprehensive data model. It covers the expanded architecture with new features: e.g. expected_close_date in deals for forecasting, tasks for compliance, etc., integrating PRD 2.0 base with expansions.

API Design (REST & GraphQL)

The application provides both RESTful API endpoints and a GraphQL API. This dual approach offers flexibility: internal front-end and advanced clients might use GraphQL for efficient data fetching, while third-party integrations or simple clients can use REST. Below we outline the key API contracts.

All API calls require proper authentication (e.g., a JWT token in headers). Responses for successful calls return the requested data in JSON (or specified format for exports). Errors return appropriate HTTP status codes and error messages.

REST API Endpoints

We'll list endpoints by resource and function, including method, path, request, and response outline. (For brevity, not every minor endpoint is listed, but all major CRUD and actions are.)

Authentication & User: - `POST /api/auth/login` – Authenticate a user. - **Request:** JSON: `{ "email": "...", "password": "..." }`. - **Response:** 200 OK with JSON `{ "token": "JWT_TOKEN", "user": {id, name, email, role} }`. - **Errors:** 401 Unauthorized if credentials invalid. - `POST /api/auth/logout` – (If needed for stateful sessions; for JWT might be handled client-side by discarding token). - `POST /api/users` – Create a new user (likely admin-only). - **Request:** JSON with user data (name, email, role, temp password). - **Response:** 201 Created with new user JSON (without password). - **Note:** Might not expose via UI publicly. - `GET /api/users/me` – Get current user profile. - **Response:** user JSON (id, name, email, role, connected_calendar flag, etc.). - `PUT /api/users/me` – Update own profile (like name, password change). - `GET /api/users` – (Admin only) list users. - `PUT /api/users/{id}` – (Admin) update user (e.g., role, or deactivate). - `DELETE /api/users/{id}` – (Admin) delete/deactivate user.

Deals: - `GET /api/deals` – Get list of deals. - **Query params:** could include filters: `stage=<stage_name>` or `status=Active/Closed`, `sector=<sector>` etc., and pagination (`page`, `pageSize`). - **Response:** JSON array of deals (or an object with `deals: [...], totalCount: X` if paginated). - Each deal object might include basic fields and possibly embedded company name, stage name to avoid extra queries (depending on design). - `GET /api/deals/{id}` – Get details of a specific deal. - **Response:** JSON of deal object, ideally with related data embedded: e.g., include company details, list of allocations, maybe memos and tasks. Alternatively, the client could call other endpoints for those, but for convenience we might embed some or provide separate endpoints. - `POST /api/deals` – Create a new deal. - **Request:** JSON of deal info, e.g.: `{ "company": {"name": ..., "sector": ...}, "deal": {"name": ..., "description": ..., "source": ..., "amount_target": ...} }`. - We allow sending company info; server will create company if not exists. Alternatively, require `company_id` if using existing. - If we allow linking existing company: either client picks an ID, or we find by name. - **Response:** 201 Created with JSON of created deal (including assigned id and maybe created company id). - `PUT /api/deals/{id}` – Update deal (for editable fields). - **Request:** JSON with fields to update (e.g., description, expected_close_date). - **Response:** 200 OK with updated deal JSON. - Note: This is not used for stage change or adding allocation, which have separate endpoints, to enforce specific logic. - `DELETE /api/deals/`

`{id}` – Delete a deal. - **Note:** Possibly disabled or admin-only if allowed at all (soft deletion recommended). - **Response:** 204 No Content on success, or 403 if not allowed due to compliance rules.

Deal Actions: - `POST /api/deals/{id}/stage` – Advance or change stage. - **Request:** JSON `{ "stage": "<stage_name_or_id>", "reason": "..." }` (reason optional, mainly for passes). - **Response:** 200 OK with deal JSON updated (new stage). - **Behavior:** Performs all server-side checks (permissions, compliance tasks). - `GET /api/deals/{id}/history` – Get stage change history for the deal. - **Response:** JSON array of history events (each with from_stage, to_stage, user, date, reason). - `GET /api/deals/{id}/allocations` – Get all fund allocations for this deal. - **Response:** JSON array of allocations (with fund info, amount, status). - `POST /api/deals/{id}/allocations` – Add a new fund allocation to the deal. - **Request:** JSON `{ "fund_id": X, "amount": Y, "status": "Proposed" }`. - **Response:** 201 Created with allocation JSON (including allocation_id). - **Behavior:** Checks available fund capacity, etc. - `PUT /api/allocations/{id}` – Update an allocation (e.g., change status to Committed or adjust amount if not executed). - **Request:** JSON with fields to update (amount, status, funded_at if marking executed). - **Response:** 200 OK with updated allocation JSON. - `DELETE /api/allocations/{id}` – Remove allocation (if needed, maybe only if proposed and we decide not to invest). - **Response:** 204 No Content or appropriate status. - `GET /api/deals/{id}/memos` – Get memos/notes for a deal. - **Response:** JSON array of memos (likely one main AI memo and any other notes). - `POST /api/deals/{id}/memos` – Add a note (human-generated). - **Request:** JSON `{ "content": "...", "title": "...", "type": "Note" }`. - **Response:** 201 with memo JSON. - (AI generation has a separate endpoint below.) - `PUT /api/memos/{id}` – Edit a memo/note (perhaps allowed for notes or to save edits to AI memo). - **Request:** JSON `{ "content": "edited text..." }`. - **Response:** 200 with updated memo. - `GET /api/deals/{id}/tasks` – Get tasks for a deal (compliance or general). - **Response:** JSON array of tasks. - `POST /api/deals/{id}/tasks` – Create a new task for the deal. - **Request:** JSON `{ "name": "...", "due_date": "...", "assigned_to": userId, "category": "General" }`. - **Response:** 201 with task JSON. - (Compliance tasks are usually pre-created by system, but this allows manual addition.) - `PUT /api/tasks/{id}` – Update a task (mark complete or edit). - **Request:** JSON for fields e.g., `{ "completed": true, "completed_at": "2025-08-01", "notes": "Done." }`. - Or a specific endpoint `POST /api/tasks/{id}/complete` could mark complete with current time by current user implicitly. - **Response:** 200 with updated task. - `DELETE /api/tasks/{id}` – Remove a task (if not needed; maybe only for general tasks, not compliance defaults). - `GET /api/deals/{id}/documents` – List documents attached to a deal. - `POST /api/deals/{id}/documents` – Upload a document for the deal. - This could be a file upload (multipart/form-data). Possibly just handled by a generic upload endpoint or GraphQL uses base64. But in REST, likely an endpoint expecting multipart. - **Request:** Form-data with file and fields (type, description). - **Response:** 201 with doc metadata JSON (id, file_name, etc). - `DELETE /api/documents/{id}` – Delete a document (perhaps admin or if uploaded by mistake).

Funds: - `GET /api/funds` – List all funds. - **Response:** JSON array of funds (with maybe summary fields like percent invested). - `GET /api/funds/{id}` – Fund detail. - **Response:** JSON of fund, possibly including an array of allocations or deals it invested in. - `POST /api/funds` – Create a new fund. - **Request:** JSON `{ "name":..., "total_committed":..., "start_date":..., ... }`. - **Response:** 201 with fund JSON. - `PUT /api/funds/{id}` – Update fund (e.g., description or maybe adjust total_committed if needed). - **Note:** Changing committed after inception is unusual, but maybe if recording updated commitments or adjusting for recycling of capital. - `DELETE /api/funds/{id}` – (Probably not allowed if any allocations exist). - `GET /api/funds/{id}/allocations` – List allocations for a fund. - **Response:** JSON array of allocations (with deal info). - Could also be embedded in GET fund detail, but

separate if needed. - `GET /api/funds/{id}/forecast` – Get capital call forecast for that fund. - **Response:** JSON array of projected calls, e.g., `[ { "expected_date": "2025-Q3", "amount": 2000000, "details": [ {dealId, dealName, amount}, ... ] }, ... ]`. If we break down by quarter or month. - Or could be by specific date if we have exact dates. - `GET /api/forecast` – Combined forecast across all funds (or for all funds user has access to). - **Response:** Maybe an object keyed by fund or a list with fund included in each item.

AI Analyst: - `POST /api/deals/{id}/generate-memo` – Generate AI investment memo for a deal. - **Request:** (optional) JSON with parameters, e.g., `{ "refresh": true, "includeSections": ["market", "team"] }`. But most likely no body needed besides maybe option to regenerate. - **Response:** 200 with memo content or 202 Accepted with a job id if async. - For simplicity, do synchronous: hold connection until done and return memo as JSON { memo_id, content }. - If long running, consider asynchronous workflow. - `POST /api/deals/{id}/ask-ai` – Ask a question about the deal to AI. - **Request:** JSON `{ "question": "What are the biggest risks?" }`. - **Response:** 200 with `{ "answer": "..." }`. - This will use AI service with context of deal. Might not store anything, though we could log Q&A in audit or separate table if needed. - These endpoints require the AIService to be configured with API keys.

Analytics: - `GET /api/analytics/summary` – Get overall analytics summary (as described in Analytics module). - **Response:** JSON

```
{
  "pipeline": {
    "total_active": 30,
    "by_stage": [ {"stage": "Screening", "count":10}, ... ],
    "closed_won": 5,
    "closed_lost": 12,
    "conversion_rate_to_won": 10.0
  },
  "deals_by_sector": [ {"sector":"FinTech","count":8,"funded":2}, ... ],
  "deals_by_source": [ {"source":"Referral","count":10,"funded":3}, ... ],
  "invested_capital_ytd": 15000000,
  "calls_next_quarter": 3000000,
  "capital_deployed_timeline": [ {"period":"2025-Q1","invested":5e6}, ...]
}
```

(This is an example structure; we can refine keys.) - `GET /api/analytics/deals-by-sector` etc., if we choose to break it out. But one summary might be easier. - Possibly allow a query param for year or filter, like `?year=2025`. - `GET /api/analytics/sector/{sector}` – Detailed metrics for one sector (if implemented). - **Response:** JSON maybe with similar structure but filtered to that sector (counts, list of deals maybe). - `GET /api/analytics/fund/{id}` – Analytics specific to a fund, e.g., sector distribution in that fund's portfolio, etc. Could be a nice addition if needed.

Calendar Integration: - `GET /api/calendar/auth-url` – Get URL to start OAuth flow for calendar linking. - **Response:** 200 with JSON `{ "url": "<Google auth URL>" }`. - Alternatively, front-end can

have the URL embedded from config, not an API call. - `GET /api/calendar/callback` – OAuth callback endpoint (not directly called by client, called by Google redirect). - This would be handled server-side to capture the code and exchange for token. Not a typical API call by app user, so we may not list it in docs for them. - `POST /api/calendar/events` – Create a calendar event. - **Request:** JSON

```
{
  "deal_id": 123,
  "title": "Due Diligence Call - ABC Corp",
  "datetime": "2025-09-15T10:00:00Z",
  "duration_minutes": 60,
  "attendees": ["investor1@doliver.com", "ceo@abccorp.com"],
  "location": "Zoom",
  "description": "Discussion of technical due diligence findings."
}
```

- **Response:** 201 Created with `{ "event_id": "<google_event_id>", "status": "confirmed", "htmlLink": "<link to event>" }` or similar fields from Google API. - On error (not authenticated or Google API fail) return 400/500 with message. - Could also have `DELETE /api/calendar/events/{id}` to cancel, and `PUT` to update, but those might not be needed initially through app if editing can be done directly on Google.

Export: - `GET /api/export/deals.csv` – Download CSV of all deals (or filtered). - Might accept same filter params as GET deals. The endpoint would set `Content-Type: text/csv` and `Content-Disposition: attachment; filename="deals.csv"`. - The body is CSV text (with header row). - `GET /api/export/deal/{id}.pdf` – Get a PDF report for a single deal. - **Response:** PDF (application/pdf, with download header). The PDF might include deal details, allocations, etc., nicely formatted. - `GET /api/export/analytics.pdf` – PDF of dashboard analytics. - Could include key charts or numbers. - `GET /api/export/compliance.csv` – e.g., export of compliance tasks across deals. - Maybe include for each deal, each task and status. - Possibly other such endpoints as needed.

Audit Logs: - `GET /api/audit-logs` – (Admin only) fetch logs. - **Query params:** maybe filter by user, action, date range. - **Response:** JSON array of log entries (each with timestamp, user (with name), action, object_type, object_id, details). - If a lot, might include pagination.

This covers the main endpoints. All responses should have proper HTTP codes (200 for success GET, 201 for creation, 204 for deletion without content, 400 for bad input, 401 for unauth, 403 for forbidden, 404 for not found, 500 for server error, etc.).

GraphQL API

GraphQL provides a single endpoint (commonly `/graphql`) where clients send queries and mutations. The schema will define types corresponding to the data model, and inputs for creation/updating.

GraphQL Endpoint: `POST /api/graphql` (or `/graphql`) – handles all queries.

We'll outline key parts of the GraphQL schema (in SDL-like format) to demonstrate the contract:

Types:

```
type User {
  id: ID!
  name: String!
  email: String!
  role: String!
  # Possibly calendarLinked: Boolean
}

type Company {
  id: ID!
  name: String!
  sector: String
  description: String
  website: String
  deals: [Deal!]!  # all deals associated
}

type Deal {
  id: ID!
  name: String!
  company: Company!
  stage: Stage!
  status: String!
  source: String
  sourceDetails: String
  amountTarget: Float
  description: String
  createdBy: User!
  createdAt: String!
  expectedCloseDate: String
  passReason: String
  allocations: [Allocation!]!    # funds allocated to this deal
  memos: [Memo!]!                # all memos/notes for this deal
  tasks(category: String): [Task!]!  # tasks, optionally filter by category
  documents: [Document!]!
  history: [StageHistory!]!      # stage change history events
}

type Stage {
  id: ID!
  name: String!
  order: Int!
  isClosed: Boolean!
```

```
  isWon: Boolean!
}

type StageHistory {
  id: ID!
  deal: Deal!
  fromStage: Stage
  toStage: Stage!
  changedBy: User!
  changedAt: String!
  reason: String
}

type Fund {
  id: ID!
  name: String!
  description: String
  totalCommitted: Float!
  capitalCalled: Float!
  capitalInvested: Float!
  startDate: String
  endDate: String
  allocations: [Allocation!]!   # deals this fund invested in
}

type Allocation {
  id: ID!
  deal: Deal!
  fund: Fund!
  amount: Float!
  status: String!
  allocatedAt: String
  fundedAt: String
}

type Memo {
  id: ID!
  deal: Deal!
  content: String!
  isAIGenerated: Boolean!
  createdBy: User!
  createdAt: String!
  editedAt: String
  title: String
  type: String
}

type Task {
```

```
  id: ID!
  deal: Deal!
  name: String!
  category: String
  dueDate: String
  completed: Boolean!
  completedAt: String
  completedBy: User
  notes: String
  createdAt: String
  createdBy: User
  assignedTo: User
}

type Document {
  id: ID!
  deal: Deal
  fund: Fund
  type: String
  fileName: String!
  fileUrl: String!   # could be presigned URL or path
  description: String
  uploadedBy: User
  uploadedAt: String
}

type AuditLog {
  id: ID!
  timestamp: String!
  user: User
  action: String!
  objectType: String
  objectId: Int
  details: String
}
```

Queries:

```
type Query {
  me: User
  users: [User!]!                          # admin only perhaps
  deals(stage: String, status: String, sector: String, source: String): [Deal!]!
  deal(id: ID!): Deal
  companies(search: String): [Company!]!
  company(id: ID!): Company
  funds: [Fund!]!
```

```
  fund(id: ID!): Fund
  allocations(fundId: ID, dealId: ID): [Allocation!]!
  memos(dealId: ID!): [Memo!]!
  tasks(dealId: ID!, category: String): [Task!]!
  analyticsSummary(year: Int): AnalyticsSummary   # a custom type to wrap multiple stats
  auditLogs(limit: Int, offset: Int, userId: ID, action: String): [AuditLog!]!  # admin only
}
```

We would also define the `AnalyticsSummary` type:

```
type StageCount { stage: String!, count: Int! }
type SectorCount { sector: String!, count: Int!, funded: Int! }
type SourceCount { source: String!, count: Int!, funded: Int! }
type TimeSeriesPoint { period: String!, value: Float! }

type AnalyticsSummary {
  totalActiveDeals: Int!
  stageCounts: [StageCount!]!
  closedWon: Int!
  closedLost: Int!
  dealsBySector: [SectorCount!]!
  dealsBySource: [SourceCount!]!
  capitalDeployedTimeline: [TimeSeriesPoint!]!
  # etc., any other fields like conversionRate etc.
}
```

Mutations:

```
type Mutation {
  login(email: String!, password: String!): AuthPayload!   # returns token and user
  createDeal(company: CompanyInput!, deal: DealInput!): Deal!
  updateDeal(id: ID!, input: DealUpdateInput!): Deal!
  advanceDealStage(dealId: ID!, stage: String!, reason: String): Deal!
  deleteDeal(id: ID!): Boolean!

  createAllocation(dealId: ID!, fundId: ID!, amount: Float!, status: String): Allocation!
  updateAllocation(id: ID!, amount: Float, status: String): Allocation!
  deleteAllocation(id: ID!): Boolean!

  createFund(input: FundInput!): Fund!
  updateFund(id: ID!, input: FundUpdateInput!): Fund!
  # maybe deleteFund if needed

  addMemo(dealId: ID!, content: String!, title: String, isAIGenerated: Boolean): Memo!
```

```
  updateMemo(id: ID!, content: String!): Memo!
  # we might not allow deleting memos easily, so skip

  addTask(dealId: ID!, name: String!, category: String, dueDate: String, assignedTo: ID): Task!
  updateTask(id: ID!, completed: Boolean, notes: String): Task!
  deleteTask(id: ID!): Boolean!

  uploadDocument(dealId: ID, fundId: ID, file: Upload!, type: String, description: String): Docum
  deleteDocument(id: ID!): Boolean!

  generateMemo(dealId: ID!): Memo!    # triggers AI generation and returns the Memo (with conten
  askAI(dealId: ID!, question: String!): String!  # returns answer text (could also structure as

  # Calendar integration may not be exposed via GraphQL, might rely on REST or internal handling
  scheduleEvent(dealId: ID!, title: String!, datetime: String!, durationMinutes: Int, attendees:

  completeComplianceTask(taskId: ID!): Task!  # or use updateTask if integrated

  # Possibly user management:
  createUser(input: UserInput!): User!
  updateUser(id: ID!, input: UserUpdateInput!): User!
  # etc for user admin
}
```

Input types:

```
input CompanyInput {
  name: String!
  sector: String
  website: String
  description: String
  # contact info etc.
}

input DealInput {
  name: String!
  description: String
  source: String
  sourceDetails: String
  amountTarget: Float
  expectedCloseDate: String
}

input DealUpdateInput {
  description: String
  source: String
```

```
  sourceDetails: String
  amountTarget: Float
  expectedCloseDate: String
  passReason: String
  # we do NOT allow stage change here to force use advanceDealStage
}

input FundInput {
  name: String!
  totalCommitted: Float!
  startDate: String
  endDate: String
  description: String
}

input FundUpdateInput {
  description: String
  totalCommitted: Float
  # maybe allow adjusting committed if needed
}

# ... UserInput for creating user, etc.
```

AuthPayload:

```
type AuthPayload {
  token: String!
  user: User!
}
```

File Upload: GraphQL often uses a special scalar `Upload` for file uploads via multipart. If using Apollo Server, we can accept an Upload in `uploadDocument`.

Permissions: GraphQL doesn't inherently restrict by role out-of-the-box, but our resolvers will check roles similarly to REST. For example, `advanceDealStage` resolver will verify if user is allowed to change stage (if trying to move to Funded, check compliance tasks and role). `auditLogs` query would check user.role is Admin, otherwise return error.

Examples of GraphQL usage: - Query all active deals with their company name and stage:

```
query {
  deals(status: "Active") {
    id
    name
```

```
    company { name sector }
    stage { name }
    amountTarget
    expectedCloseDate
  }
}
```

- Get detail of one deal, including allocations and tasks:

```
query {
  deal(id: 101) {
    name
    description
    company { name }
    allocations {
      fund { name }
      amount
      status
    }
    memos {
      content
      isAIGenerated
      createdAt
    }
    tasks(category: "Compliance") {
      name
      completed
      completedBy { name }
    }
    history {
      fromStage { name }
      toStage { name }
      changedBy { name }
      changedAt
      reason
    }
  }
}
```

- Mutation to create a deal (with new company):

```
mutation {
  createDeal(company: { name: "ABC Corp", sector: "FinTech" },
             deal: { name: "ABC Corp Series A", description: "...", source: "Referral", amountTar
    id
    name
```

```
    company { id name sector }
    stage { name }  # should be initial stage by default
  }
}
```

- Mutation to advance stage:

```
mutation {
  advanceDealStage(dealId: 101, stage: "Due Diligence") {
    id
    name
    stage { name }
    history {
      toStage { name }
      changedBy { name }
      changedAt
    }
  }
}
```

- Mutation to generate memo:

```
mutation {
  generateMemo(dealId: 101) {
    id
    content
    isAIGenerated
    createdAt
  }
}
```

(This will run the AI and return the memo content.)

The GraphQL API provides a more flexible way for front-end to get exactly the data needed (for instance, one query can fetch a deal and its fund allocations and tasks together, rather than multiple REST calls). We must implement resolvers accordingly: - Many resolvers will just fetch from DB using the models (with possible joins for efficiency). - Some (like generateMemo) will trigger logic (calling AIService and then returning result). - Ensure N+1 query issues are mitigated (e.g., if querying deals and each company's name, we should batch or join rather than separate query per deal; ORMs and dataloaders can help).

Given the complexity, developers might choose GraphQL for the main app UI and use REST for integrations or specific tasks (like file upload, or simpler data pulls).

We have now specified the API contract comprehensively, covering both styles. Implementation can choose either to focus on GraphQL for core flows and implement minimal REST for external, or fully do both. The PRD ensures either path has the needed details.

Implementation Plan and Architecture Summary

(This section is for the developers to understand how to structure the code and the workspace on Replit, as well as how to test and deploy. It reiterates some architecture but with a focus on practical implementation steps.)

Project Structure and Replit Workspace

We recommend a **monorepo** structure with separate directories for frontend and backend, since the front-end is a React app and the back-end is an Express/Node app (assuming Node for both for synergy, though a Python backend is also possible – here we proceed with Node/Express for consistency, but structure would be similar if Python).

Example file/folder layout:

```
/project-root
│   replit.nix            # Replit environment config (if needed for custom setup)
│   .replit               # Replit run configuration
│   package.json          # Maybe a root package.json if we manage concurrently (or separate)
/frontend
│   package.json          # Frontend dependencies (React, etc.)
│   public/               # Static public assets
│   src/
│      index.js, App.js, etc.   # Main React entry and app
│      components/         # Reusable components
│      pages/              # Page components for each route (DealsPage, DealDetailPage, etc.)
│      services/api.js     # API helper (for REST/GraphQL requests)
│      ... (other such as context/ state mgmt files)
/backend
│   package.json          # Backend dependencies (Express, Apollo, ORMs, etc.)
│   index.js or server.js  # Entry point to start Express server
│   config/
│       default.json       # config files for DB connection, etc. (or use env)
│   models/               # ORM model definitions (Deal.js, Company.js, etc.) or Prisma schema i
│   resolvers/            # GraphQL resolver functions (if code-first)
│   schema.graphql        # GraphQL schema (if using SDL approach)
│   controllers/          # Express route handlers if using REST controllers separately from Gra
│   services/             # Business logic services (DealService.js, FundService.js, etc.)
│   middlewares/          # Auth middleware, error handlers, etc.
│   utils/                # Utility modules (for formatting, emailing, etc.)
│   ai/                   # AI integration code (like AIService and any prompt templates)
```

```
    cron/ or jobs/         # If any scheduled or background jobs (like periodic forecast refresh,
    test/                  # Backend tests (if using a testing framework for logic)
```

Replit Config: - The `.replit` file can specify the command to run both frontend and backend. Possibly using `npm run dev` if we have a script for concurrently starting both. - One approach: Run the back-end on a certain port and front-end on another using a proxy or concurrently. But Replit typically exposes one port publicly. So maybe simpler: serve front-end through back-end. - We can build the React app (production build) and have Express serve the static files from `/frontend/build` directory. - For development, one might run React dev server on port 3000 and Express on 3001 and configure Replit to show 3000. But let's assume we can test separately and then do a final build for single server. - In `.replit`, a run command could be: `npm install && npm run build && node backend/index.js` - Where `npm run build` in front-end builds static files, and backend/index.js serves them and starts API. - Or break into two phases if needed. - Alternatively, use a single `package.json` with workspaces and one start script controlling both. But separate is clearer.

File structure details: - Frontend: - Might use Create React App or Vite for initial scaffolding. The specifics of config aren't a PRD focus, but ensure to include any needed polyfills or environment (like how to specify the GraphQL or API URL). - Possibly a `.env` file for front-end to know API base URL (for development if separate). - It's typical to have an `api.js` that uses `fetch` or axios to call our backend endpoints. For GraphQL, we might set up Apollo Client at the top level (ApolloProvider, etc.). - Use modern React (functional components, hooks). - Possibly incorporate TypeScript if desired by team (makes it easier for AI to avoid type bugs, but up to them). - Backend: - If using an ORM like Prisma: - A `prisma/schema.prisma` file would define the models akin to our tables. Then generate JS/TS client. - That might reside under /backend as well. - If not, define model classes with something like Sequelize or use Knex query builder or even raw queries for simplicity. - Services: e.g., DealService in `services/DealService.js` implementing createDeal, advanceStage, etc., as per earlier logic. - Controllers: `controllers/dealController.js` defines Express routes mapping e.g., `createDeal(req,res)`, which calls DealService then sends result. - GraphQL: - Possibly use Apollo Server middleware on Express. Could keep schema in `schema.graphql` and load it, and define resolvers in code. - Use context to pass `user` info from JWT to resolvers. - Middlewares: - Auth middleware to verify token on protected routes (for REST). - Body-parser for JSON (if not included by modern Express by default). - Error handling to catch exceptions and send JSON error. - External integration code: - `ai/AIClient.js` to encapsulate calls to OpenAI (with API key from env). - `calendar/CalendarClient.js` for Google Calendar API calls and OAuth. - Could rely on libraries (e.g., googleapis npm). - `email/EmailClient.js` if emailing (SendGrid SDK or nodemailer for SMTP). - Config: - use something like dotenv or config files for DB connection string, API keys, etc. Replit secrets can supply environment variables.

Database Setup in Replit: - Replit might not have a persistent full Postgres instance by default. Options: - Use `sqlite` for dev. If using an ORM, it's easy to switch DB adapters with config. Could run with SQLite file in /mnt/data to persist. - Or connect to an external Postgres (maybe a free cloud one). That would require internet (which presumably we have if the code is running outside the restricted sandbox). - For simplicity, and given moderate data size, SQLite could suffice in initial dev. The PRD, however, wrote in terms of a SQL DB, so the code should be abstracted enough to later migrate to Postgres. - Possibly mention: "Use SQLite for development (file stored on /mnt/data/db.sqlite for persistence), and plan for PostgreSQL in production with connection string config."

Environment Variables (to document for Replit secrets): - `DATABASE_URL` – connection string for DB (if using Postgres, e.g., `postgres://user:pass@host:5432/dbname`; if SQLite, maybe `file:/mnt/data/sqlite.db`). - `JWT_SECRET` – secret key for signing JWT tokens. - `OPENAI_API_KEY` – for AI. - `GOOGLE_CLIENT_ID`, `GOOGLE_CLIENT_SECRET` – for calendar integration. - `SENDGRID_API_KEY` or `SMTP_PASS` etc if email integration. - Possibly `NODE_ENV` (production/development). - `PORT` – port on which Express listens (Replit typically uses 3000 or so, but could be environment variable). - If using Prisma, `DATABASE_URL` is also used by it.

Third-Party Libraries to consider: - Backend: express, apollo-server-express, jsonwebtoken (for JWT), bcrypt (for password hashing), body-parser (if needed), cors, prisma or sequelize, dayjs/moment for dates, nodemailer or sendgrid, openai library, googleapis, multer (for file uploads) or Apollo's file upload. - Frontend: React, perhaps Apollo Client, recharts or chart.js for charts, axios for REST if not using GraphQL, Material-UI or similar for UI components, form libraries (Formik or React Hook Form) to handle forms nicely, maybe state mgmt (but could keep it simple with context). - Testing: Jest for backend, maybe React Testing Library for front (if they go that far in initial build). - Deployment: If needed, something like Vercel or Heroku config if not staying on Replit.

Testing Strategy

Backend Testing: - Use a testing framework (Jest or Mocha/Chai) to write unit tests for service functions. For example: - Test DealService.createDeal with various inputs (valid, missing required fields). - Test DealService.advanceStage logic (e.g., trying to close without tasks done). - Test AIService with a stubbed OpenAI API (to not actually call external in test). - Test GraphQL resolvers possibly via apollo-server testing utilities (or skip if service tests cover logic). - Set up a test database (maybe an in-memory SQLite or a separate file) to run tests on. Use migrations or ORM sync to create schema, then tear down. Or use an approach like SQLite in memory for speed. - Could also test REST endpoints using supertest (spinning up Express and hitting endpoints). - Include tests for security: e.g., ensure a normal user cannot access admin-only endpoint (simulate with JWT of different roles). - If time, test the integration of Google Calendar by mocking the API calls (simulate that we got a token and that an event creation returns success). - Similarly, test that the forecast function calculates correct outputs for known data sets.

Frontend Testing: - Basic rendering tests for React components can be written. E.g., ensure DealsPage fetches deals (mock API) and displays them. Use React Testing Library and MSW (Mock Service Worker) or similar to simulate API. - Since Replit's code AI might do best with instructions, we ensure the code is modular for testing (e.g., separate logic from components where possible). - Could plan a Cypress end-to-end test later (not in initial implementation, but mention as future: E2E test that user can login and go through adding a deal etc., which could be run on a staging environment).

Manual Testing Plan: - We will perform manual tests in a development environment, going through all user stories: - Create a deal, update it, advance stages, allocate funds, generate memo, complete tasks, etc. and verify the expected outcome in UI and DB. - Prepare some seed data (maybe a script to insert a couple of funds, and a couple of sample deals) to have something to test UI with initially. - Cross-browser test the front-end (Chrome, Firefox). - Test on different screen sizes if possible (for responsiveness). - Ensure no console errors or network errors.

Deployment

Development Deployment on Replit: - During development, Replit will run the app. We might keep it in "Development mode" which might show detailed errors. But once near production, set `NODE_ENV=production` and test accordingly. - We need to make sure the Replit host is accessible for OAuth (e.g., Google OAuth redirect URIs must include the Replit app URL). - The domain might be something like `https://DoliverInvestmentLifecycleApp.username.repl.co` – we will register that for Google API. - Use Replit's secret management for keys.

Production Deployment: - If Replit usage is intended for production as well (they have Always-on), ensure to secure it: - Use HTTPS (Replit provides by default via *.repl.co). - Add any necessary environment config (like a more persistent database). - Alternatively, containerize: - Create a Dockerfile that sets up Node, copies code, installs, builds front-end, runs backend. Could deploy on a cloud VM or Heroku-like service. - Ensure environment variables are set in prod environment with secure values. - Database: If using an external DB for production, update `DATABASE_URL` accordingly and run migrations to that DB. - We should include a migration or schema initialization process: - If using Prisma, `prisma migrate deploy` in startup to apply migrations. - If using a different method, maybe a custom script to create tables if not exist (for SQLite). - Logging: In production, maybe less verbose logs, but ensure any critical errors are logged. - Monitoring: Set up uptime alert or integrate with a logging/monitoring service if needed.

Deployment Hooks/CI: - If using GitHub with Replit, could set up that pushing to main triggers a deployment (Replit might auto reflect if connected). - If not, perhaps use GitHub Actions to run tests on push and then deploy to a server. - On Replit, we can rely on manual starts, but an automated test run before deploy is ideal. Possibly integrate a GitHub Actions that runs `npm test` for both front and back (if separate). - Ensure to not commit secrets; use environment.

Security Checklist *(Recap from security model but in checklist form for developers to verify before go-live):* - [ ] All API routes are protected by authentication except those explicitly open (e.g., login). - [ ] Passwords are hashed, and on login we compare hash (no plaintext store). - [ ] JWT token is signed with a strong secret and a reasonable expiration (maybe 8-12 hours). Possibly implement refresh tokens for long sessions. - [ ] Input validation is in place to avoid injection (use parameterized queries via ORM, validate lengths to avoid overflow). - [ ] XSS protection: The front-end should escape or safely render any data that might contain user input (e.g., if description could contain HTML, we either sanitize or disallow HTML). Most inputs here are plain text, so risk is low. Still ensure e.g., if someone put a script tag in description, our UI would treat it as text not execute it (React by default escapes content unless using dangerouslySetHTML). - [ ] CSRF: For our API, since it's stateless with JWT and an SPA, CSRF is not a big issue (no cookies being automatically sent). If we were using cookies for auth, we'd need CSRF tokens. We'll rely on JWT in header, so should be safe from CSRF as long as we don't accidentally allow token to be read by third-party (we shouldn't as it's in Authorization header and same-site). - [ ] CORS configured to only allow our front-end origin (if front-end served separately). If served same domain, still maybe lock down as needed. - [ ] Rate limiting on auth endpoints (to mitigate brute force on login) – e.g., max 5 login attempts per minute per IP. Could implement via an Express middleware or simple in-memory count. - [ ] File uploads: Validate file types and size (to avoid storing huge files or malicious files). Possibly restrict to certain formats (pdf, docx for documents, images like png for something). Also scan if needed (maybe out-of-scope, but mention it). - [ ] External API keys: secure and not exposed to client. The front-end should never directly call OpenAI with our key or Google with client secret – all goes through backend. - [ ] Data access control: ensure users cannot query or mutate data they shouldn't (we assume one org's data, so it's mostly role-based not object-based. If multi-client,

would need tenant ID filters). - [ ] Backups: For production, ensure the DB is backed up or can be recovered (for compliance, data loss could be bad). Could mention schedule backups if using a managed DB. - [ ] Compliance data: If any personal data is stored (founder contact), ensure it's minimal and protected. Possibly add a privacy notice to the app if needed (internal app maybe not). - [ ] Test the compliance override scenarios to ensure the system indeed blocks what it should block.

Compliance Checklist: - All compliance tasks enumerated are present for each deal at appropriate time. - Before moving a deal to Funded, verify tasks done. - The audit log is capturing required events. (We may specifically want to log things like "Compliance Approved" if there's an explicit approval step. If needed, a special task "Compliance Approval" can serve as that, or we could add an explicit field like compliance_approved_by, but tasks approach is fine.) - The audit log should be immutable (we won't allow deletion of log entries via API). - Ensure the retention of data: if a user tries to delete a deal that was invested, maybe disallow or require archiving instead to not lose records. - If any regulatory requirements like anti-money-laundering checks, we have fields for KYC tasks etc. If needed, ensure integration or ability to attach results (we did allow attaching documents).

Documentation & Handover: - The code should have comments especially in complex logic (like in AIService explaining prompt structure, or in DealService for stage rules). - Provide a README with instructions to run (especially for someone launching in Replit, though in PRD likely covered). - Possibly include sample environment config in README (like which env vars need to be set). - Provide SQL or migration file to set up initial tables (if not using an automated migration tool). - Provide seed data for initial use (maybe a script or instruct to create a test user and a sample fund). - Summarize how to use the API (some of above could go into a smaller developer docs, or rely on self-documenting via GraphQL schema and such).

By following this PRD, a developer (or Replit's AI coding assistant) should be able to implement the system step by step: 1. Set up base project structure. 2. Implement authentication and user model. 3. Implement deals and basic pipeline (with stage model). 4. Implement funds and allocation logic. 5. Integrate AI memo using OpenAI API. 6. Build out front-end components for each part, tying into these APIs. 7. Add polish features (analytics charts, export, calendar). 8. Test thoroughly. 9. Deploy and monitor.

The PRD is comprehensive, covering both functional and non-functional requirements, and is intended to serve as a blueprint for implementation. Each feature has clearly defined expectations and underlying logic spelled out, minimizing ambiguity. With this, the Replit AI (or any developer) should have a clear path to build the Doliver Investment Lifecycle App version 2.0 that meets Doliver’s needs.

---

1 2 4 5 6 7 8 9 11 12 **How to Select Deal Flow Management Software | Seraf-Investor.com**
https://seraf-investor.com/compass/article/9-key-elements-deal-flow-management-software

3 10 **Behind the Curtain: Unveiling our AI-Powered Investment Memo Generator — Flybridge**
https://www.flybridge.com/ideas/behind-the-curtain-unveiling-our-ai-powered-investment-memo-generator

13 14 **Lens Store™ - Track and Manage Capital Calls with Copia**
https://copiawealthstudios.com/lens/copia/capital-calls