INTEGRATED AI DEVELOPMENT FRAMEWORK (Phased Hybrid Approach)

Overview

This framework provides a practical, phased hybrid methodology for developing new applications using prompt-based AI implementation. It builds on successful patterns from the PosalPro MVP2 project, emphasizing wireframe-driven development, user journey mapping, and parallel implementation tracks to accelerate prototyping without sacrificing data integrity or scalability

GitHub

GitHub

. Each development phase is defined with clear goals and strategies, and each phase includes a representative AI prompt template to guide code generation (no actual code) in a modular, reusable format. This ensures frontend architecture and user experience remain aligned with user stories, while the backend and data layers maintain robust type-safe contracts for long-term maintainability.

Key Principles

Documentation-Driven Development – Begin each feature or module with thorough documentation (requirements, user stories, wireframes) that serves as both specification and knowledge repository

GitHub

. This ensures clarity and captures lessons for future reference.

Contract-First (Type-Safe) Design – Define data contracts (TypeScript interfaces, types and Zod schemas) before implementation to enforce end-to-end type safety

GitHub

. This guarantees that rapid development never compromises data integrity, as every API and component adheres to strict schemas.

Parallel Track Development – Develop the data model and the UI in parallel using a “mock-driven” approach

GitHub

. Backend and frontend teams (or AI tasks) work simultaneously with agreed-upon contracts: the UI uses mock data conforming to the schemas while the backend logic is being built

GitHub

. This hybrid approach enables fast prototyping of the user interface without waiting on full backend implementation, while ensuring any eventual integration will not break the UI due to the strict contracts in place.

Atomic Component Composition – Build the frontend with an atomic design system (atoms, molecules, organisms) to ensure flexibility, modularity, and reuse

GitHub

. Complex UIs are constructed from small, well-defined components, improving maintainability and consistency across the app. For example, breaking down complex forms into reusable compound components improved code maintainability and uniform validation in the PosalPro project

GitHub

.

User Journey Alignment – Design and implement features in alignment with user journey maps and user stories. Every UI screen and component should trace back to a user story or workflow step

GitHub

. This ensures the development stays user-centric and that acceptance criteria are met. The framework encourages maintaining a component traceability matrix linking user stories to implementation components

GitHub

for full coverage and easier validation of user needs.

AI-Assisted Implementation – Leverage AI coding assistants with structured prompts to accelerate development

GitHub

. Each phase of development has an associated prompt pattern that guides the AI to generate code consistent with the project’s conventions and requirements. Human developers remain in the loop for reviewing AI output, guiding complex business logic, and refining prompts. This collaboration maximizes speed while maintaining code quality.

With these principles, the development process is divided into clear phases from project foundation to post-launch, each with specific goals, strategies, best practices, and an example AI prompt to assist implementation.

Phased Development Methodology

Phase 1: Project Foundation & Infrastructure

Goal: Establish the project’s foundation, development environment, and tooling necessary for a scalable application. This phase sets up everything needed for developers (and AI systems) to work efficiently. Strategy:

Initialize Base Project – Set up a new codebase with Next.js (using the latest App Router) in TypeScript strict mode

GitHub

. This provides a modern, robust starting point and enforces type safety from day one.

Development Tools & Standards – Configure essential tools for code quality: ESLint and Prettier for linting/formatting, and Husky git hooks for pre-commit checks

GitHub

. Establish a consistent directory structure and version control (Git) from the start.

Core Utilities Setup – Implement foundational utilities such as logging and performance monitoring early

GitHub

. For example, set up a structured logging system (with levels, timestamps, context) and basic performance timing utilities to track app behavior.

Environment & Configurations – Create an environment configuration system to manage different settings for development, testing, and production

GitHub

. Include a base API client setup that will be used for calling backend services (even if those services are not built yet).

Documentation Structure – Establish a documentation repository for the project. Create a central reference (e.g. PROJECT\_REFERENCE.md) and an Implementation Log to record decisions

GitHub

. Ensure there's a place to capture lessons learned as the project progresses (e.g. a LESSONS\_LEARNED.md to record insights and patterns

GitHub

).

Best Practices:

Knowledge Capture: Document all setup decisions and reasoning in the implementation log as you go

GitHub

. This provides context for future developers and for refining AI prompts.

Automation: Use scripts or tools to automate checks (linters, type checking) and environment validation

GitHub

. Early automation ensures consistency and prevents configuration drift.

Strict Standards: Keep TypeScript in strict mode and define comprehensive linting rules from the beginning

GitHub

. A strict foundation catches errors early and enforces best practices (leading to better scalability).

Reference Architecture: Maintain a project reference guide that cross-links all important setup docs

GitHub

. This could include a high-level architecture diagram, tech stack choices, and how each tool is used.

Lessons Learned: Start a lessons learned document in this phase

GitHub

and update it regularly. Capturing what worked or didn’t in the setup will inform improvements in later phases and future projects.

Prompt Template (Phase 1 – Foundation): Below is a representative AI prompt format for implementing an infrastructure or setup task in Phase 1. It guides the AI to create or configure a foundational component of the project:

\*\*Prompt [1.X]: [Infrastructure Component Implementation]\*\*

- \*\*User Story Alignment\*\*: [If applicable, link to any user story or project requirement driving this setup, e.g., "As a developer, I need a CI pipeline to deploy the app."]

- \*\*Goal\*\*: Set up \*\*[Specific Infrastructure Component]\*\* to establish the project's foundation. This will enable [capability] (e.g., continuous integration, consistent code style, etc.) for the development team.

- \*\*Prerequisites\*\*: [Any prior steps completed or tools installed, e.g., Node.js, initial Next.js app].

- \*\*Task\*\*:

1. \*\*Navigate\*\* – Open or create the relevant configuration files (e.g., `/` or `config/` directory).

2. \*\*Create/Configure Files\*\* – For example: \*"Initialize a Next.js project with TypeScript support. Add a `.gitignore` for Node modules and environment files. Set up `tsconfig.json` with strict mode."\*

3. \*\*Implement Settings\*\* – For example: \*"Configure ESLint and Prettier: add an `.eslintrc.json` extending recommended rules, and a `prettierrc` for formatting. Set up a Husky pre-commit hook that runs `eslint` and `prettier`."\*

4. \*\*Integrate\*\* – \*"Ensure the new configuration is integrated: e.g., verify `npm run lint` and `npm run dev` both work without errors."\* Connect the component with other parts of the system if needed (for example, ensure the API client is imported where it will be used).

5. \*\*Document\*\* – \*"Add documentation or comments explaining the configuration. Update the README with setup instructions. Log this setup in the IMPLEMENTATION\_LOG.md with reasons for chosen tools."\*

- \*\*Validation\*\*:

- \*Run development server and confirm it starts without errors.\*

- \*Run linting/formatting scripts to ensure they are picking up issues correctly.\*

- \*If applicable, run a simple test (or CI pipeline) to confirm the infrastructure works (e.g., a commit triggers the Husky hooks).\*

- \*Verify in documentation that all steps are recorded.\*

(This template is based on the PosalPro project's approach to setting up infrastructure

GitHub

GitHub

, guiding the AI through structured steps: navigate, implement, integrate, and validate.)

Phase 2: Data Architecture

Goal: Establish a robust, type-safe data architecture for the application. In this phase, we define all core data models, validation schemas, and data management utilities. The outcome is a set of well-defined contracts (types and schemas) that ensure any data processed by the system is validated and consistent. Strategy:

Define Data Contracts – Identify all core entities in the application (e.g., Users, Proposals, Products, etc., based on the project requirements or a blueprint) and create TypeScript interfaces or types for them

GitHub

. Include enumerations (enums) for any fixed sets of values (statuses, roles, categories) to use throughout the app for consistency

GitHub

.

Implement Validation Schemas – For each interface, implement a corresponding Zod schema (or similar validation library schema) that enforces runtime validation

GitHub

. These schemas will be used to validate data at the boundaries (user input, API responses) to maintain data integrity. Set up a centralized validation module or middleware to use these schemas for forms, API requests/responses, etc.

GitHub

GitHub

.

Data Relationships and Business Rules – Incorporate relationships and business logic into the data models. For example, if a Proposal contains multiple Products, ensure the schema reflects that (e.g., an array of Product IDs or objects) and enforce any business rules (like unique names, required fields, valid value ranges) in the validation layer

GitHub

.

Mock Data Generators – Create utilities to generate realistic mock data conforming to the schemas

GitHub

. These generators can produce sample objects for each entity (e.g., a function to create a fake User or fake Proposal) and will be invaluable for testing and for the UI team to use before the real backend is ready.

Data Utility Functions – Implement helper functions for common data tasks: e.g., formatting or parsing data, computing derived values, or aggregating collections. Also, include consistency checks or assertions that can run in development to verify data assumptions (for instance, a function to verify referential integrity of sample data)

GitHub

.

Best Practices:

Contract-First Approach: Ensure the client and server share the exact same data contract definitions. Mismatches between an API response and the client’s expected type can cause runtime errors, so treat the TypeScript interfaces and schemas as a source of truth

GitHub

. Every field name and type in an API response should match what the client code expects.

Validation at Boundaries: Enforce schema validation for data at all boundaries (on form submission, before sending data to an API, when receiving data from an API, etc.)

GitHub

. This catches errors early and prevents corrupt or unexpected data from propagating through the system.

Reusability: Use TypeScript generics and utility types for common patterns so that the data layer is flexible and doesn’t require duplicate code

GitHub

. For example, if many entities have an “ID” field or timestamp fields, define a generic base interface or use generic functions to handle those patterns.

Data Integrity Utilities: Maintain high data integrity by using automated checks. For instance, in PosalPro a set of collection validation functions and consistency checks were used to maintain data quality throughout development

GitHub

. Incorporate similar tools to continuously validate assumptions (e.g., a test that all mock data passes schema validation, ensuring your schemas and types remain correct).

Progressive Integration: Because this phase runs in parallel with UI development (Phase 3), make sure to version or document the contracts clearly. Any changes to interfaces or schemas should be communicated and updated in the mock data and UI usage. Maintaining type safety means UI code will immediately flag any contract changes – leverage that by updating the UI as soon as the types change to avoid integration gaps

GitHub

.

Prompt Template (Phase 2 – Data Architecture):

\*\*Prompt [2.X]: [Data Model/Schema Implementation]\*\*

- \*\*User Story Alignment\*\*: [Reference relevant user stories or requirements that this data model supports, e.g., "As an admin, I need to manage users (requires a User data model)".]

- \*\*Goal\*\*: Define and implement the \*\*[Name of Data Entity or Schema]\*\* to establish a type-safe data contract. This will ensure the application handles \*\*[Entity]\*\* data consistently and validates all inputs/outputs.

- \*\*Prerequisites\*\*: [Any prerequisite data structures or schemas, e.g., "BaseEntity interface", "Enumerations for status types" should be ready.]

- \*\*Task\*\*:

1. \*\*Navigate\*\* – Go to the data models or types directory (e.g., `src/types/` or `src/schema/`).

2. \*\*Create/Modify Files\*\* – \*"Create a new file `src/types/[Entity].ts` and `src/schema/[Entity]Schema.ts`."\*

3. \*\*Define Interface\*\* – \*"Define a TypeScript interface `I[Entity]` with fields: [list key fields with types]. Include documentation comments for each field."\*

4. \*\*Implement Schema\*\* – \*"Implement a Zod schema `[Entity]Schema = z.object({ ... })` mirroring the interface. Include validation rules (e.g., string length, email format, enums for specific fields, etc.)."\*

5. \*\*Integrate Mock Data\*\* – \*"If not already existing, update the mock data generator to produce `[Entity]` instances. Ensure the generated data passes the schema validation."\*

6. \*\*Document\*\* – \*"Add usage notes in the PROJECT\_REFERENCE or code comments, explaining where this data is used. Update any related documentation (like APPLICATION\_BLUEPRINT.md) if this fulfills a requirement."\*

- \*\*Validation\*\*:

- \*Type Checking\*: Ensure the interface and schema are in sync (you can `z.infer` the schema and compare to the interface type).

- \*Schema Tests\*: Write a few quick tests or use a REPL to validate that `[Entity]Schema.parse(mockEntity)` succeeds for valid data and throws errors for invalid data.

- \*Integration\*: Verify that any component or function expecting this entity type now recognizes it (e.g., no TypeScript errors in UI code using `I[Entity]`).

- \*Consistency\*: Run all tests and lints to ensure nothing else broke with the addition. Commit the changes with a message reflecting the new data model.

(This prompt format is adapted from the project’s hybrid phase plan

GitHub

GitHub

, focusing on defining data contracts and validation schemas.)

Phase 3: UI Foundation

Goal: Establish the frontend foundation by developing a design system and a library of reusable UI components, and by laying out the key screens of the application. This phase ensures the user interface is built in a scalable way, aligned with user experience design (wireframes and journeys), and ready to integrate with real data later. Strategy:

Design System & Styling – Implement a cohesive design system. Define design tokens for colors, typography, spacing, etc., and configure Tailwind CSS (or your chosen styling framework) with these tokens

GitHub

. Ensure themes or style guidelines (like dark mode, accessibility settings) are accounted for from the start.

Atomic Component Library – Develop a set of atomic components (small, reusable UI elements) and compose them into higher-level components. Start with atoms like buttons, inputs, icons, etc., then create molecules (combinations like form inputs with labels, modals) and organisms (navbars, sidebars, complex widgets)

GitHub

. Each component should be focused on a single responsibility and be reusable across screens.

Layout & Navigation – Set up the overall layout structure of the app, including a responsive grid or flex system for pages and a navigation system (header, menus, routing). Implement routing with appropriate public/protected route handling (if authentication is needed, which might tie into Phase 4) and ensure it aligns with the sitemap/user flows designed

GitHub

.

Wireframe-driven Screens – Develop the key screens as per the wireframes using the components and layouts. Use mock data from Phase 2 for any dynamic content so that the screens can be interacted with even without a real backend

GitHub

. For example, implement the Dashboard, Profile page, or other primary screens, populating them with fake data via context or hardcoded mocks, to validate the UI against the design.

Accessibility & Responsiveness – From the beginning, build components and pages to be accessible (ARIA labels, keyboard navigation, screen reader support) and fully responsive to different screen sizes

GitHub

. Use semantic HTML elements and test the UI in mobile and desktop views for each screen.

Best Practices:

User Story Mapping: Continuously refer to user stories and journey maps when building UI components and screens. This ensures the UI not only looks like the wireframes but also supports the intended user paths. For each screen, verify it covers all steps and scenarios from the user journey documentation (e.g., error states, edge cases)

GitHub

.

Traceability: Maintain a mapping of components to user stories (a component traceability matrix)

GitHub

. This helps in testing and ensures coverage — every user story should have UI representation, and every UI element should serve a story. PosalPro’s approach of linking user stories to components ensured nothing was built without purpose and nothing was omitted

GitHub

.

Consistency & Reuse: Adhere strictly to the design system. If a new need arises, update the design system rather than creating ad-hoc styles. Encourage reuse by documenting components and their variations. This improves maintainability as changes in design propagate systematically.

Accessibility & UX: Treat accessibility as a first-class citizen. Implement components with proper ARIA attributes and test with screen readers. Also, use accessible color contrasts and consider keyboard-only navigation. A good UI foundation means the app is usable by all users from the start, rather than trying to retrofit accessibility later.

Error Boundaries & Feedback: Implement error boundary components and loading states globally or per section

GitHub

. This ensures that if part of the UI fails (for example, a component throws an error), the rest of the app can still function and provide feedback. Similarly, uniform loading spinners or skeletons improve perceived performance during data fetches. Comprehensive form validation on the UI (using the schemas from Phase 2) with user-friendly messages is crucial for a good user experience

GitHub

.

Compound Components: For complex UI elements (like multi-step forms or configurators), use compound component architecture. In PosalPro, breaking down a complex product form into smaller sub-components (pricing selector, etc.) greatly improved code maintainability and allowed consistent validation

GitHub

. This approach can be applied generally: complex components manage state and context internally but render smaller specialized components, which keeps each piece simpler and easier to test.

Prompt Template (Phase 3 – UI Foundation):

\*\*Prompt [3.X]: [UI Component or Screen Implementation]\*\*

- \*\*User Story Alignment\*\*: [Reference the user story or journey that this UI element addresses, e.g., "As a user, I want to see a dashboard of my proposals at a glance."]

- \*\*Goal\*\*: Create the \*\*[Name of UI Component/Screen]\*\* using the established design system. This component/screen will fulfill [describe the user need or feature] and align with the wireframe specification.

- \*\*Prerequisites\*\*: Design tokens and basic components should be available (e.g., the design system from earlier). If implementing a screen, ensure atomic components needed (buttons, form fields, etc.) are ready or being implemented.

- \*\*Task\*\*:

1. \*\*Navigate\*\* – Go to the UI components directory (e.g., `src/components/` or appropriate folder for screens/pages).

2. \*\*Create Component/Screen\*\* – \*"Create a new file `src/components/[ComponentName].tsx` (or a page in `src/app/`). Use React (Next.js) with functional components. Add `'use client'` at the top if it's a client component."\*

3. \*\*Implement Structure\*\* – \*"Implement the JSX structure for the component according to the wireframe. Use Tailwind CSS classes (or CSS-in-JS) to apply design system styles. Make sure to compose existing atomic components if possible (e.g., use `<Button>` component from library instead of raw `<button>`)."\*

4. \*\*Props and State\*\* – \*"Define a `Props` interface for the component if needed, ensuring it receives all necessary data. Implement local state with React hooks for interactive parts (e.g., form inputs, open/close modal state). Manage state via context or props for parent-child communication as needed."\*

5. \*\*Data Integration (Mock)\*\* – \*"If this component displays dynamic data, integrate it with mock data: for example, use a context or a temporary data source that provides data in the shape defined by Phase 2 schemas. Ensure that it gracefully handles empty or loading states by showing spinners or placeholder UI."\*

6. \*\*Accessibility\*\* – \*"Add ARIA labels or roles where appropriate. Ensure the component is navigable via keyboard (e.g., for a modal, focus trap within it). Test responsive behavior by adjusting the container width."\*

7. \*\*Document\*\* – \*"Write JSDoc/TSDoc comments for the component and its props. Update the Storybook (if used) or a style guide with an example of this component. Also, log the completion in the implementation log, noting which user story it satisfies."\*

- \*\*Validation\*\*:

- \*Visual Check\*: Compare the rendered component/screen to the wireframe – they should match in layout and elements. Check in multiple screen sizes.

- \*Interaction\*: Simulate user interaction (clicks, form entries) and ensure the component behaves as expected (e.g., modal opens, validation messages show).

- \*Accessibility Test\*: Use browser dev tools or accessibility checkers to ensure no critical accessibility issues (and that all interactive elements have appropriate labels).

- \*Integration\*: If the component is part of a larger page, integrate it and make sure it does not break the overall page. All props/data should flow correctly.

- \*Commit\*: After manual testing, commit the component with a message referencing the user story (e.g., "Feat(UI): [3.X] Add DashboardCard component for proposals overview").

(This prompt format draws on the project’s approach to UI development

GitHub

GitHub

, emphasizing use of design system components, wireframe compliance, and integration with mock data.)

Phase 4: Application Logic

Goal: Implement the core application logic and state management, connecting the UI (Phase 3) with the data structures and services. In this phase, features like authentication, form handling, and any real-time updates are developed. The aim is to make the app functional (using either mock or real services) so that all user journeys can be executed end-to-end within the application. Strategy:

Authentication & Authorization – Implement user authentication flows and protected content access. For example, integrate a user authentication context or provider that handles login state, and guard certain routes or components so only authorized users can access them

GitHub

. This may involve creating a login page, a registration form, password reset, and token management (possibly still using mock APIs in this phase, unless a real auth service is available).

Form Workflows & Validation – Develop comprehensive form submission logic for all forms in the app (beyond basic field validation already handled by schemas). This includes multi-step form wizards, if any, and handling of form state transitions (e.g., from editing to submitting to success)

GitHub

. Integrate the Zod validation from Phase 2 on form submission to prevent invalid data from proceeding. Ensure proper user feedback (error messages, field highlights) for validation errors.

Data Connectivity (Frontend) – Build the frontend data layer that connects UI components to data sources. If a backend API is ready, this means implementing API calls (using the API client set up in Phase 1) to fetch or send data

GitHub

. If the backend is not ready yet, this may involve creating local mocks or utilizing a temporary in-memory store. In either case, set up the logic for data fetching, caching, and state updates when data changes (e.g., after a form is submitted, update the list view).

Real-time Updates (if needed) – If the application requires real-time features (such as live notifications, chat, or live updates of a dashboard), implement a mechanism for real-time data synchronization

GitHub

. This could be via webSockets, WebRTC, or polling, depending on the needs. Ensure the UI updates seamlessly with these real-time events and handle concurrency issues (like two users editing the same data).

Analytics & Monitoring – Integrate client-side analytics and logging for user interactions

GitHub

. For example, track page views, important button clicks, or usage metrics that are relevant to the business. This not only helps in post-launch analysis but also during development to trace how features are used. Ensure any user tracking respects privacy and can be toggled or configured as needed.

Best Practices:

State Management: Use a predictable state management approach for complex data. For global state (user info, app-wide settings), consider using React Context or a state management library; for local component state, continue to use React useState/useReducer. Avoid excessive global state – balance local vs global state to optimize performance and clarity

GitHub

. The PosalPro team found that combining local state for UI interactions with global state for shared data prevented unnecessary re-renders in complex forms

GitHub

.

Custom Hooks: Encapsulate reusable logic in custom React hooks

GitHub

. For instance, if multiple components need to handle polling a backend or managing a form step, a custom hook can provide that logic cleanly. This improves maintainability by avoiding code duplication and makes it easier to test logic in isolation.

Type-Safe API Integration: Ensure your API calls (even if to mock endpoints) use the TypeScript types and validation established earlier. Build a robust API service layer or client that handles errors gracefully and consistently (e.g., an interceptor to catch 401 Unauthorized globally)

GitHub

. By centralizing API interactions, you can enforce security (like attaching auth tokens) and handle performance (like caching or debouncing calls) uniformly.

Error Handling & Boundary Conditions: Implement comprehensive error handling for all logic flows. This includes handling network failures (maybe by showing a “retry” option to users), handling form validation errors, and catching any exceptions in user interactions so the app doesn’t crash. Logging these errors (possibly to an external monitoring service) is also recommended so that issues can be tracked and fixed.

Progressive Enhancement: Where possible, design the application logic to allow for graceful degradation. For example, if a real-time feature is unavailable (user is offline or server is down), the app should still function with fallback behaviors. Similarly, ensure the core functions (like creating an entry, saving data) work even if advanced enhancements (like on-the-fly suggestions via AI, etc.) are not present. This makes the application more robust and user-friendly

GitHub

.

Prompt Template (Phase 4 – Application Logic):

\*\*Prompt [4.X]: [Feature Logic Implementation]\*\*

- \*\*User Story Alignment\*\*: [Identify the user story or use-case that this logic will fulfill, e.g., "As a user, I want to reset my password so that I can recover access to my account."]

- \*\*Goal\*\*: Implement \*\*[Name of Feature or Logic Component]\*\* to enable [specific capability] in the application. This will connect the UI with the data layer for [feature] and ensure the workflow is functional and robust.

- \*\*Prerequisites\*\*: Ensure relevant UI components (Phase 3) and data schemas (Phase 2) are already in place. For example, "UserProfileForm component exists" or "User schema is defined".

- \*\*Task\*\*:

1. \*\*Navigate\*\* – Find or create the appropriate location for this logic. For UI-centric logic, this might be within a React component or hook; for global logic, maybe in a context provider or a controller module.

2. \*\*Implement Logic\*\* – Depending on the feature:

- \*Authentication Example\*: \*"Implement an `AuthContext` with React Context API in `src/context/AuthContext.tsx`. Include state for current user and functions `login`, `logout`, `register` that interact with the API client. Use `useEffect` to persist session (e.g., check localStorage for a token)."\*

- \*Form Handling Example\*: \*"Within `UserProfileForm.tsx`, implement a `handleSubmit` function that uses `UserSchema.safeParse(formData)` to validate input. If valid, call `api.updateProfile(formData)`, handle loading state and show success or error messages based on response."\*

- \*Real-time Example\*: \*"Implement a `useNotifications` hook that connects to a WebSocket at `[WS URL]` and listens for new notifications. Provide an interface for components to subscribe to notifications and update local state when a new message arrives."\*

3. \*\*Integrate Data\*\* – \*"Use the data models and API client to fetch or send data as needed. For instance, in `AuthContext.login`, use `ApiClient.post('/login', credentials)` and handle the response, parsing it into the `User` type defined in Phase 2."\* Ensure any data coming in/out is validated (e.g., wrap API calls with zod schema parse for responses).

4. \*\*Update UI State\*\* – \*"Make sure the relevant UI components respond to this logic. For Auth, context changes should re-render nav bars or protected routes. For forms, the UI should reflect loading or errors from the logic implemented."\* You may need to pass down context or use callbacks as props to link this logic to the UI.

5. \*\*Document\*\* – \*"Add comments or README notes explaining how this logic works, especially if it affects many parts of the app (like Auth). Include any assumptions or hacks for later review. Update the traceability matrix or user story docs to mark this story as implemented."\*

- \*\*Validation\*\*:

- \*Functional Test\*: Manually test the entire user flow associated with this logic. For example, try logging in with valid and invalid credentials to see that the happy path and error path work as expected.

- \*Unit Tests\*: If possible, write unit tests for the core logic (e.g., test that `AuthContext.login()` sets the right state on success or returns the correct error on failure, perhaps by mocking the API client).

- \*Integration Test\*: Ensure that the feature works when integrated in the app – e.g., after login, the protected routes now render, or after form submission, the new data appears on the dashboard.

- \*Edge Cases\*: Consider edge cases like expired sessions, network failure, validation edge cases (max field lengths, special characters) and test those.

(This prompt template is informed by the application logic tasks identified in the framework

GitHub

GitHub

, covering authentication, form handling, and data connection.)

Phase 5: Backend Integration

Goal: Integrate the application with backend services and APIs, transitioning from mock data to real data. This phase involves connecting the frontend to live backend endpoints (or implementing backend functions if this is a full-stack project) and ensuring that data flows securely and correctly between client and server. It lays the groundwork for a production-ready system by solidifying the data layer and service connections. Strategy:

API Client Configuration – Update or configure the API client (established in Phase 1) for real backend URLs and endpoints

GitHub

. This includes setting base URLs, authentication headers (e.g., attaching JWT tokens or API keys), and global error handling for requests (like refreshing tokens on 401 responses or logging out users).

Secure Communication – Implement security measures for all data exchanges. For example, ensure all API calls use HTTPS and include proper authentication tokens. If the backend provides CSRF tokens or other security tokens, make sure the client handles those (and the server is expecting them)

GitHub

. Validate inputs on the server side as well (the same schemas from Phase 2 can often be reused in backend).

Middleware & Server Extensions – If working in a full-stack environment (e.g., Next.js API routes or a Node backend), create middleware for common concerns like authentication, input validation, and rate limiting

GitHub

GitHub

. These ensure every request is checked for valid input (using Phase 2 schemas) and proper permissions. If it’s a pure frontend, coordinate with backend team to ensure they have implemented corresponding validation and security.

Third-Party Services – Integrate any external services (payments, maps, email, etc.) during this phase, as they are part of the “backend integration.” Use well-defined interfaces or SDKs for these, and wrap them in the app’s own API layer if needed for easier testing and swapping out later.

Data Migration & Seeding – If the app uses a database, this phase likely includes running database migrations for any new schema changes discovered during UI/logic development. It also involves seeding the database or preparing initial data (especially if transitioning from mock data – you might load some of those mocks into a dev database for continuity).

Best Practices:

Schema Validation for APIs: Continue to use the contract definitions. For each API response, parse it with the Zod schemas on the client to immediately catch any deviations

GitHub

. Likewise, on the server, validate incoming requests. This double validation (client and server) ensures robust data integrity and quickly highlights mismatches in assumptions.

Error Handling & Monitoring: Implement comprehensive error handling on all service calls

GitHub

. For instance, handle network timeouts, unexpected server errors, or partial failures (like one request succeeding and another failing in a batch). Use logging (and perhaps user-facing notifications) to make troubleshooting easier. Additionally, integrate monitoring (APM) for backend calls – track response times and error rates.

Security Hardening: Follow security best practices such as using parameterized queries or ORM for database operations (to prevent SQL injection)

GitHub

, encoding/escaping any user input that goes into responses (to prevent XSS via the API), enforcing authorization checks on every API (even if the UI hides a button, the backend should still verify permissions). Implement rate limiting and audit logging for sensitive operations

GitHub

.

Documentation & Communication: Document all API endpoints and their request/response formats (a tool like OpenAPI/Swagger can be helpful). This is useful both for internal reference and if any third-party or future services need to interface with your backend. In PosalPro, establishing clear API contracts and documenting them was key to keeping the front and back ends in sync.

Testing Integration: Begin writing integration tests at this phase if not already (overlapping with Phase 6). For example, test that calling the real API from the client returns data that matches the expected format. If possible, set up a staging environment where the frontend can hit a test backend, to validate everything works outside of a local dev setup.

Prompt Template (Phase 5 – Backend Integration):

\*\*Prompt [5.X]: [API Integration or Service Implementation]\*\*

- \*\*User Story Alignment\*\*: [If applicable, reference stories like "As a user, I want to see real data on my dashboard" or technical tasks like "Implement payment processing".]

- \*\*Goal\*\*: Integrate \*\*[Specific Backend Service or API]\*\* into the application. This will replace any remaining mock data for this part of the system and ensure secure, real-time communication with backend services for [feature].

- \*\*Prerequisites\*\*: The application’s front end for this feature is built and uses a mock or placeholder service. Backend endpoints (or third-party services) for this feature exist or are being developed. Configuration (API base URL, keys) should be available.

- \*\*Task\*\*:

1. \*\*Configure Client\*\* – \*"Update the API client to use the production/staging base URL for [Service]. Ensure that the authentication token is included in headers for each request (e.g., Authorization: Bearer token)."\*

2. \*\*Implement Calls\*\* – \*"In the [relevant module or context], replace the mock data usage with real API calls: use `apiClient.get('/endpoint')` to fetch data. Parse the response with `schema.parse` (from Phase 2 definitions) to validate."\*

3. \*\*Handle Responses\*\* – \*"Implement proper handling of API responses: on success, update state with the new data; on error, implement retry logic or surface an error message to the user. For example, if a 401 error occurs, redirect to login (if applicable), or if a 500 occurs, show a 'please try again later' message."\*

4. \*\*Security & Middleware\*\* – \*"Ensure that any required security steps are followed. For front-end only: include CSRF tokens or any needed headers. For backend routes: add middleware to check auth and validate request bodies using Zod. E.g., protect the `POST /proposal` route with an auth check and use `ProposalSchema.safeParse(req.body)` to validate input."\*

5. \*\*Integrate with UI\*\* – \*"Remove or toggle off the mock data source. Connect the UI components to the real data by calling the integration functions. E.g., in the dashboard component, call `useEffect` to fetch real proposals via the service instead of using dummy data."\* Ensure the UI reacts properly (loading indicators while fetching, etc., which should have been implemented).

6. \*\*Document\*\* – \*"Update documentation: mark the feature as using real data now. If any API contracts differed from assumptions, note them in LESSONS\_LEARNED.md. Update any .env sample files with new environment variables (like API keys or URLs) introduced."\*

- \*\*Validation\*\*:

- \*End-to-End Test\*: Run through the user functionality now using the real backend. For instance, log in and check that the dashboard loads actual user-specific data correctly.

- \*Error Simulation\*: Use network tools or test servers to simulate API errors (like 500 or 401 responses) to ensure your error handling in the app works and the user sees appropriate feedback.

- \*Security Check\*: Verify that secure pages indeed require authentication (e.g., accessing a protected route without a token should redirect to login). Ensure no sensitive data is exposed in client-side logs or error messages.

- \*Performance\*: Profile the real API calls in the app. Ensure that any long calls are done asynchronously with spinners, and consider adding caching if the same data is fetched frequently.

(This template is derived from the integration tasks and best practices observed in the project

GitHub

GitHub

, focusing on replacing mocks with real API calls and adding necessary security and error handling.)

Phase 6: Testing & Quality Assurance

Goal: Rigorously test the application (unit tests, integration tests, end-to-end tests) to ensure reliability, and set up quality assurance processes. By the end of this phase, the application should have high test coverage and a verified user journey flow, with automated tests integrated into the development workflow (e.g., continuous integration). Strategy:

Unit Testing – Write unit tests for all critical pure functions and utilities (e.g., data helpers from Phase 2, custom hooks from Phase 4). Ensure edge cases are covered. Use Jest or a similar framework for component-level tests (e.g., testing React components in isolation, perhaps with React Testing Library). Aim to isolate logic and verify that given certain inputs, the outputs or side-effects are as expected.

Component and Integration Testing – Create tests that render components and simulate user interactions to verify that components work together correctly. For example, test a form component by inputting data and clicking submit, then asserting that the expected API call was made and that error states appear for invalid data. Use mocking libraries or Mock Service Worker (MSW) to simulate API responses in integration tests

GitHub

, so the frontend can be tested in a near-real environment without needing a live backend.

End-to-End (E2E) Testing – Use an E2E testing tool (like Cypress or Playwright) to simulate real user scenarios in a headless browser. Write test cases for the most important user journeys (login, main workflows, etc.) to catch any integration issues between frontend, backend, and anything in between. These tests ensure that the entire stack works as intended when deployed.

Testing Pyramid & Coverage – Follow a balanced testing approach. Emphasize a testing pyramid where you have many unit tests, a moderate number of component/integration tests, and fewer E2E tests

GitHub

. Unit tests are fast and isolate issues, integration tests cover interaction between modules, and E2E tests cover full scenarios. Aim for a high overall coverage (e.g., >80% of critical code) but focus on meaningful coverage (covering different branches and edge cases, not just hitting lines).

Continuous Integration (CI) Integration – Integrate tests into the CI pipeline. Configure the CI server to run the test suite on each pull request or commit to main. Set up thresholds for coverage (e.g., fail the build if coverage drops below a certain percentage)

GitHub

. Also, incorporate other quality checks here (linters, type checks, etc., from Phase 1) so that the pipeline ensures code quality gates are all passing before merge/deploy.

Best Practices:

Reusable Test Utilities: Create utilities for common testing tasks (rendering components with context providers, resetting state between tests, etc.)

GitHub

. Use typed, reusable mock implementations for things like browser APIs, network requests, or any global singletons. In the PosalPro project, a structured mock design pattern (with typed jest mocks and reset utilities) significantly reduced test setup complexity and improved maintainability

GitHub

.

Mock External Interactions: Use tools like Mock Service Worker (MSW) to intercept network calls in tests

GitHub

. This provides a reliable and flexible way to simulate backend responses for different scenarios (including error cases) without flakiness. It ensures your integration tests run consistently and can cover scenarios like authentication flows or error handling paths that might be hard to reproduce with a live backend.

Snapshot vs Assertion: Use snapshot testing sparingly and intentionally. Snapshots can catch unintended changes in UI, but they should be limited to stable UI components and not be a substitute for logical assertions

GitHub

. Prefer explicit assertions about behavior (e.g., “after clicking submit, the form should display X message”) over large snapshots, which can become brittle.

Quality Gates: Treat tests as a first-class deliverable. Set up quality gates where a merge is blocked unless tests pass and coverage is sufficient

GitHub

. Enforce running a subset of quick tests on each commit (via pre-commit hook or local script) for fast feedback, and the full test suite in CI for thorough verification.

Test Documentation: Document how to run tests and what the testing strategy is (possibly in a TESTING\_GUIDELINES.md or similar). This helps new developers (or contributors) understand how to write and maintain tests. It also helps AI assistants – if using AI to generate tests, having a clear guideline can be part of the prompt to ensure the AI follows project conventions.

Prompt Template (Phase 6 – Testing & QA):

\*\*Prompt [6.X]: [Testing Scenario Implementation]\*\*

- \*\*Scope\*\*: [Specify what level and scope this test covers: unit, integration, or E2E. For example, "Unit test for the Proposal data model", or "Integration test for the login flow".]

- \*\*Goal\*\*: Ensure \*\*[Component or Functionality]\*\* works as expected under various scenarios. This involves writing tests that cover [list key scenarios, e.g., "valid input", "invalid input shows error", "API error returns fallback data"].

- \*\*Prerequisites\*\*: The feature implementation is complete. Testing framework is set up (e.g., Jest, Testing Library, Cypress). Any required test data or configuration (e.g. mock server, test user accounts) is available.

- \*\*Task\*\*:

1. \*\*Set Up Test Case\*\* – \*"Create a test file `[ComponentName].test.tsx` in the `\_\_tests\_\_` directory. Import the component and necessary utilities (like `render` from our test utils, and `screen`, `fireEvent` from Testing Library)."\* If writing an E2E test, set up the test suite with the initial app state (e.g., start the dev server or use the testing URL).

2. \*\*Write Test for Normal Case\*\* – \*"Write a test (using `it` or `test` block) for the primary expected behavior. For example: render the `<LoginForm>` component, input a valid email/password, click submit, then assert that a success message appears or navigation to dashboard occurs."\*

3. \*\*Write Test for Edge/Error Case\*\* – \*"Write another test for an error scenario. For example: input an invalid email format and assert that a validation error message is shown and no API call is made. Or simulate an API failure (using MSW to return 500) and assert the UI shows an error notification."\*

4. \*\*Repeat for Other Critical Scenarios\*\* – Cover additional branches, e.g., \*"password too short shows specific message," "network timeout triggers retry mechanism," etc.\* Aim to cover all branches of logic in the component or feature.

5. \*\*Use Mocks and Utilities\*\* – \*"Use the provided test utilities to simplify setup: e.g., wrap the component in the required context providers by using our custom `renderWithProviders` utility."\* If testing a hook or utility, mock external dependencies (e.g., for a date utility, set a fixed Date).

6. \*\*Run and Refine\*\* – \*"Run the test suite and ensure all tests pass. If some tests are flaky or failing, adjust either the code (if it revealed a bug) or the test (if the test was not written correctly). Add comments to tests for clarity if needed."\*

- \*\*Validation\*\*:

- \*Test Accuracy\*: Temporarily introduce a small change or bug in the feature (if feasible) to confirm a test catches it. This validates that the test would indeed catch regressions (and is not a false positive).

- \*Coverage\*: Check the coverage report to ensure this test increases coverage for the target component/feature significantly. Make sure key lines/branches are hit.

- \*Code Review\*: Have another developer or reviewer look at the tests to see if scenarios are missing or if any test is unclear. Clear, maintainable tests are important for long-term quality.

- \*CI Integration\*: Ensure the new tests run in the CI environment and pass consistently. If any timing issues or flakiness appear in CI, address them (e.g., by adding waits or adjusting asynchronous code).

(This prompt format is informed by the testing patterns from the project

GitHub

GitHub

, guiding AI to generate meaningful tests for various scenarios and levels.)

Phase 7: Deployment & Delivery

Goal: Prepare the application for deployment and set up the infrastructure for continuous delivery. This phase covers everything needed to reliably build, deploy, and run the application in production (or staging) environments, including CI/CD pipelines, environment configuration for production, and deployment automation. The outcome is a repeatable deployment process and an application ready for launch. Strategy:

Continuous Integration/Delivery Pipeline – Establish a CI/CD pipeline if not already. This includes automated build, test, and deploy steps

GitHub

. For example, use GitHub Actions, Jenkins, or another CI service to run tests (Phase 6) on each push and to build and deploy the app to a staging/production environment upon merge to main. Incorporate approvals or manual gating as needed for production deployment.

Environment Configuration – Set up separate configurations for each environment (development, staging, production)

GitHub

. Manage secrets (API keys, database URLs, etc.) securely, using environment variables or a secrets manager. Ensure that the build process can inject the correct configuration (for instance, using Next.js environment vars or a config file) so that things like API endpoints and feature flags point to the right services in each environment.

Deployment Automation – Write deployment scripts or use infrastructure-as-code (like Terraform, AWS CloudFormation, etc., depending on context) to automate provisioning of resources and deployment. For a web app, this might mean configuring Vercel/Netlify for Next.js, or Dockerizing the app and deploying to a cloud provider’s container service. Automation reduces human error and makes deployments consistent.

Health Checks & Monitoring – Implement health checks for the application in the deployed environment

GitHub

. For a web service, health check endpoints can be used by the cloud infrastructure to verify the app is running (e.g., an /health API route). Also set up monitoring and alerting: integrate with a monitoring service (like Azure Application Insights, NewRelic, etc.) to track uptime, performance metrics (CPU, memory), and error rates in production. Ensure alerts are configured for critical issues (like downtime or error spikes).

Rollback Strategy – Develop a clear rollback plan in case a deployment goes wrong

GitHub

. This could mean having the ability to redeploy the previous stable version easily, or using blue-green deployments or feature flags to switch off a failing feature. Document how to trigger a rollback and test this process (e.g., ensure the previous release artifacts are retained).

Best Practices:

Automate Everything: Manual deployment steps should be minimized. Use scripts for common tasks like running migrations, seeding data, clearing caches, etc., during deployment. Automation and infrastructure-as-code ensure that you can recreate environments reliably and that scaling up is easier.

Security in Deployment: Make sure that production environment is secure – e.g., environment files not checked into version control, secure handling of secrets (through CI secret stores, not plain text), and that the build artifacts do not accidentally include dev-only information. Also, verify that security-related headers or settings are enabled in production (for Next.js, ensure that it’s using production build optimizations, etc.).

Performance Settings: Enable optimizations for production: e.g., turn on CDN caching for static assets, use a content delivery network for assets if applicable, ensure that debug logs are disabled or minimized. Monitor performance metrics after deployment and consider enabling features like code-splitting or caching if not already (some of these might have been addressed in earlier phases, but now is the time to double-check under real load).

Documentation & Training: Document the deployment process in a DEPLOYMENT.md or similar. Include how to set up a local production-like environment, how to run the build, any cron jobs or scheduled tasks needed, etc. This is useful for new team members or if an AI agent is being instructed to assist in deployment. Having a clearly written deployment runbook (even if most steps are automated) is invaluable.

Verify Post-Deploy: Always verify the application post-deployment. This includes running a quick smoke test (either manually or automated) after each deployment to ensure the major functionalities are working on production. Many CI/CD setups can include post-deploy test jobs. Also, monitor logs immediately after deploy for any errors introduced by the new release.

Prompt Template (Phase 7 – Deployment & Delivery):

\*\*Prompt [7.X]: [Deployment Automation or Config Setup]\*\*

- \*\*Context\*\*: The application is ready to be deployed. We need to configure and automate the deployment process for environment: \*\*[Staging/Production]\*\*.

- \*\*Goal\*\*: Implement \*\*[Specific Deployment Task]\*\* to enable reliable releases. This could be setting up a CI/CD pipeline, writing infrastructure-as-code for provisioning, or configuring environment settings for production.

- \*\*Prerequisites\*\*: Ensure all tests are passing (from Phase 6) and the application builds successfully. Have access to the target hosting environment or CI system. Gather necessary credentials (but do not expose them in prompts).

- \*\*Task\*\*:

1. \*\*Configure Build\*\* – \*"Set up the build process for production: e.g., define a Dockerfile for the application with multi-stage build (builder and runtime stages). Ensure that the production build uses environment variables for secrets."\* Or \*"Configure Next.js export settings if using static export, adjust base paths if needed for the hosting platform."\*

2. \*\*CI Setup\*\* – \*"Create a GitHub Actions workflow file (`.github/workflows/deploy.yml`) that triggers on push to `main`. Add steps to checkout code, set up Node, install dependencies, run tests, and build the app. If tests pass, add a step to deploy: e.g., use the Vercel Action or AWS CLI to upload the build artifact to the cloud."\*

3. \*\*Infrastructure\*\* – \*"If using a cloud service, define infrastructure: e.g., a Terraform script or cloud config to create a hosting service, database, etc. Ensure it’s idempotent so it can run on each deploy if needed."\* If infrastructure is already there, ensure the new release is properly pointed to (maybe using a versioned release tag).

4. \*\*Environment Variables\*\* – \*"Set up environment config for production: e.g., in Vercel, define `NEXT\_PUBLIC\_API\_URL` to the production API endpoint, etc. Locally, create a `.env.production` file as a template for what needs to be set in production."\* Ensure that sensitive values (API keys, DB passwords) are referenced but not hardcoded.

5. \*\*Health Check & Monitoring\*\* – \*"Implement a simple health check endpoint in the app (if not existing) that returns status. Then, set up an uptime monitor (perhaps via a service or simple script) to regularly ping this endpoint."\* Also, \*"Integrate Sentry or another error tracking service for the production build if not already, by initializing it in the app startup."\*

6. \*\*Dry Run\*\* – \*"Do a dry run of the deployment: e.g., deploy to a staging environment first. Verify everything works, then proceed to production. Document any manual steps that had to be done and consider automating them next time."\*

- \*\*Validation\*\*:

- \*Pipeline Test\*: Push a dummy commit (or use an existing one) to trigger the CI/CD pipeline. Ensure it runs through all steps successfully and deploys to the target environment.

- \*Smoke Test\*: After deployment, perform a quick test of major functionalities on the deployed site (can be automated or manual). Check logs for any errors that might have occurred during startup or after deployment.

- \*Rollback Test\*: Intentionally deploy a known bad version or simulate a failure to ensure your rollback procedure works. For example, if using blue-green deployment, verify that you can switch back to the previous version quickly.

- \*Team Review\*: Have another team member review the deployment configuration and scripts. They might catch potential issues or edge cases. A second pair of eyes (or an AI review) can validate that secrets are handled safely and that the process aligns with organizational practices.

(This prompt template incorporates tasks from the production readiness checklist

GitHub

and CI/CD setup mentioned in the project

GitHub

, guiding the AI to establish a deployment pipeline and production environment configuration.)

Phase 8: Post-Launch & Maintenance

Goal: Monitor the application in production, gather feedback, and continuously improve the system after launch. This phase is about ensuring the app remains stable, performant, and user-aligned over time. It includes maintenance tasks like bug fixes, performance tuning, scaling, and adding minor enhancements based on real user feedback, all while capturing new lessons and updating the development process for future projects. Strategy:

Monitoring & Alerting – Actively monitor the production application. Use logging and monitoring tools to keep track of errors, response times, and usage patterns in real time. Set up alerts for critical issues (e.g., application downtime, errors above a threshold, or significant performance degradation) so the team is notified quickly and can respond

GitHub

.

Analytics & User Feedback – Analyze user behavior through analytics (which were integrated in Phase 4). See what features are most used or where users drop off in a flow. Additionally, gather direct user feedback via surveys or support channels. Use this insight to prioritize fixes or new features. For example, if analytics show many users abandon a certain form, investigate and address potential UX issues.

Performance & Scaling – As usage grows, be ready to optimize. This could involve profiling the application to find slow parts and optimizing code or queries, scaling the infrastructure (adding more instances, database tuning), and employing caching or CDNs more aggressively. Continuous performance testing (perhaps reusing Phase 6 tests in a load testing context) can help identify bottlenecks before they become problems.

Regular Maintenance – Schedule regular maintenance windows or cycles to update dependencies, fix bugs, and refactor code as needed. Technologies evolve, and security patches come out; staying up-to-date is important for longevity. Also, address any technical debt incurred during fast development phases. Refactoring for better modularity or readability now will pay off in the long run.

Knowledge Capture & Process Improvement – Continue updating documentation with any new insights. After launch, development doesn’t stop; ensure that any new patterns or pitfalls discovered are recorded. Update the LESSONS\_LEARNED.md with post-launch findings and the PROMPT\_PATTERNS.md with any adjustments to how prompts should be structured for future tasks

GitHub

. This phase often reveals how well the AI-assisted approach worked, so document successes and failures to refine the methodology in future projects.

Best Practices:

User-Centric Iteration: Keep the user at the center of post-launch improvements. When bugs are reported or features requested, tie them back to user stories and journeys. This maintains the alignment we established from the beginning, ensuring even maintenance work fits into a user scenario or need, rather than being just technical for its own sake.

Rapid Response: Develop a protocol for hotfixes or patches. Despite thorough testing, issues might appear in production. Having a streamlined process (and possibly AI prompt templates ready) for generating and deploying quick fixes (while still testing them) can minimize downtime or user frustration.

Continuous Testing: Even after launch, continue writing tests for new features or bugs. When a bug is found in production, write a test that reproduces it (if possible) and then fix it – this ensures the bug stays fixed going forward. Also, consider adding production tests or synthetic monitoring (scripts that run against the live app periodically to simulate user actions).

Scaling Plan: Have a plan for scaling the system if user load increases. This might involve adding more servers, optimizing the database, or partitioning services. Use data from monitoring to know when to execute the scaling plan (e.g., when average CPU usage stays above a threshold or response times start to degrade).

Refinement of AI Prompts: Reflect on where AI assistance was most effective and where it struggled. Improve the prompt templates or add new ones in your framework doc (this document) for scenarios that were encountered. For example, if generating test cases via AI turned out well, document how those prompts were written. If some code generation had issues (say the AI produced insecure code), note that in prompt guidelines (e.g., explicitly remind to avoid certain insecure patterns). This continuous learning will make the next project start on an even stronger footing

GitHub

.

Prompt Template (Phase 8 – Post-Launch Maintenance):

\*\*Prompt [8.X]: [Post-Launch Improvement or Fix]\*\*

- \*\*Context\*\*: The application is live. We have observed [describe issue or improvement] through [monitoring/feedback]. We want to address this in a timely manner.

- \*\*Goal\*\*: Implement \*\*[Bug Fix / Enhancement]\*\* to improve the live application. This will [reduce errors/improve performance/enhance UX] for the user.

- \*\*Details\*\*:

- \*Issue Description\*: Summarize what the issue is or what feedback was received (e.g., "Users report the dashboard loads slowly when they have many items" or "Error logs show a frequent null pointer exception on submit").

- \*Impact\*: Describe the severity (e.g., "affects all users occasionally", "only admin feature", "performance impact on high load").

- \*Root Cause Hypothesis\*: If known, state what part of the code or architecture might be causing it (e.g., "likely inefficient database query in getDashboardData").

- \*\*Task\*\*:

1. \*\*Investigate\*\* – \*"Examine the relevant code (e.g., `DashboardService.getData` function). Identify any obvious inefficiencies or errors. Write a unit test or use logging to reproduce the issue if possible."\*

2. \*\*Implement Fix or Improvement\*\* – \*"Optimize or correct the implementation: e.g., add an index to the database query or modify the loop logic in `DashboardService` to avoid unnecessary calculations. If it's a bug like a null pointer, add a check or adjust the data flow to ensure the value exists."\*

3. \*\*Test the Change\*\* – \*"Write tests if they don't exist: e.g., a test with a large number of items to ensure performance is improved, or a unit test for the function that was throwing an error. Run the full test suite to ensure nothing else broke."\*

4. \*\*Deploy\*\* – \*"Deploy the fix to a staging environment if possible, and replicate the scenario (lots of data or the specific user action) to confirm the issue is resolved. Then proceed to production deployment following the established process."\*

5. \*\*Monitor Post-Fix\*\* – \*"After release, closely monitor the relevant metrics or logs to confirm that the issue is indeed resolved and no new issues have been introduced."\*

6. \*\*Document\*\* – \*"Update the LESSONS\_LEARNED.md with a note about this issue and fix (what was learned). If this uncovered any new best practice (e.g., need to paginate large data sets), update the relevant section of documentation. Also, if this fix requires a change in how AI prompts should be written (like avoid generating unpaginated queries), update the prompt framework."\*

- \*\*Validation\*\*:

- \*User Confirmation\*: If the issue was user-reported, reach back out (or observe via analytics) to ensure user satisfaction improved (e.g., users are now completing the previously problematic flow without dropping off).

- \*Performance Metrics\*: Check before-and-after metrics for performance improvements (if relevant). For a bug, ensure error counts in logs have dropped to zero for that particular error.

- \*Regression Testing\*: Keep the new tests as part of the suite to ensure the issue doesn’t recur in future changes. Run a full regression test if the fix was complex or touched many areas.

- \*Continuous Improvement\*: Consider if this fix suggests further enhancements (e.g., if dashboard is still a bit slow, plan a Phase 2 of improvements or create a backlog item). Use this as input for the next iteration of development.

(This prompt template highlights the approach to addressing issues post-launch and is informed by the project’s emphasis on capturing lessons and refining patterns after implementation

GitHub

. It ensures AI assistance can be used not just for initial development but also for ongoing maintenance tasks.)

Prompt for Implementing Prompt 2.2: Database Integration & Data Models

Goal: Transition the PosalPro application from mock data to a fully integrated PostgreSQL database using Prisma. This involves defining all necessary data models, establishing relationships, creating a comprehensive data access layer, and ensuring data integrity.

Current State:

* Core enums (UserType, etc.) and shared types (BaseEntity) are in src/types/.
* Prisma is set up with a User model and related auth tables.
* Some API routes (e.g., /api/proposals) exist but might be using mock data (mockProposalsDB).
* DATABASE\_URL is configured, and basic database connectivity is established.

Key Reference Documents:

* front end structure /implementation/DATA\_MODEL.md: PRIMARY SOURCE for all data entities, fields, types, and relationships.
* prisma/schema.prisma: Current Prisma schema (needs expansion).
* src/types/: Existing TypeScript definitions (may need to be reconciled with Prisma models).

Phase 2.2.1: Comprehensive Prisma Schema Definition

1. Analyze DATA\_MODEL.md:

* Thoroughly review DATA\_MODEL.md to identify all core entities required for the application (e.g., Proposal, Client, Product, RFP, Section, ContentBlock, SMEInput, AuditLog, Notification, etc.).
* Pay close attention to fields, data types, required/optional status, default values, and unique constraints for each entity.
* Map out all relationships (one-to-one, one-to-many, many-to-many) between entities, including foreign keys and relation names.

1. Update prisma/schema.prisma:

* Define All Core Models: Implement all identified entities from DATA\_MODEL.md as Prisma models in prisma/schema.prisma. Ensure field names and types align precisely.
* Example for Proposal:  
  prisma  
    
    
    
    
  Apply to route.ts  
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
              model Proposal {  
    
                id            String        @id @default(cuid())  
    
                title         String  
    
                status        ProposalStatus @default(DRAFT) // Ensure ProposalStatus enum exists  
    
                clientId      String  
    
                client        Client        @relation(fields: [clientId], references: [id])  
    
                // ... other fields based on DATA\_MODEL.md  
    
                sections      Section[]  
    
                createdAt     DateTime      @default(now())  
    
                updatedAt     DateTime      @updatedAt  
    
                // ... audit fields, versioning, etc.  
    
              }
* Establish Relationships: Define all relationships between models using Prisma's relation syntax (@relation). Use explicit relation names for clarity, especially for self-relations or multiple relations between the same two models.
* Implement Enums: For any enumerated types defined in DATA\_MODEL.md (e.g., ProposalStatus, ClientTier, ProductType), define them as Prisma enums in schema.prisma if they are not already present or cannot be directly mapped from existing TypeScript enums.
* Add Indexes: Define indexes (@index, @@index) for frequently queried fields or combinations of fields to optimize database performance, as indicated in DATA\_MODEL.md or based on anticipated query patterns.
* Map Table Names (Optional but Recommended): Use @@map("table\_name") for models and @map("column\_name") for fields if you want database table/column names to differ from Prisma model/field names (e.g., snake\_case in DB, camelCase in Prisma).
* Review Existing User Model: Ensure the existing User model and its relations are compatible with any new models being added.

1. Generate Prisma Client:

* After updating the schema, run npx prisma generate to update the Prisma Client with the new models and types. This will provide type safety for the data access layer.

1. Run Database Migration:

* Create a new migration to apply the schema changes to the database:  
  bash  
    
    
    
    
    
    
    
    
    
    
    
    
          npx prisma migrate dev --name "add-core-data-models"
* Review the generated SQL migration script for correctness before applying.

Phase 2.2.2: Data Access Layer (Service Functions)

1. Create Service Directory Structure:

* Organize service functions by domain/entity within src/lib/services/.
* Example: src/lib/services/proposalService.ts, src/lib/services/clientService.ts, etc.

1. Implement CRUD and Business Logic Functions:

* For each core entity defined in prisma/schema.prisma, create a corresponding service file.
* Implement standard CRUD (Create, Read, Update, Delete) operations using Prisma Client.
* Create: e.g., createProposal(data: CreateProposalInput): Promise<Proposal>
* Read: e.g., getProposalById(id: string): Promise<Proposal | null>, getAllProposals(filters?: ProposalFilters): Promise<Proposal[]>
* Update: e.g., updateProposal(id: string, data: UpdateProposalInput): Promise<Proposal>
* Delete: e.g., deleteProposal(id: string): Promise<void>
* Type Safety: Use Prisma-generated types for inputs and return values. Define custom input types (DTOs) if necessary (e.g., CreateProposalInput).
* Include Related Data: Utilize Prisma's include or select options to fetch related data as needed (e.g., fetching a proposal with its client and sections).
* Implement Business Logic: Encapsulate any specific business rules or complex queries within these service functions (e.g., calculating proposal value, fetching proposals by status and user).
* Error Handling: Implement robust error handling. Catch Prisma errors (e.g., PrismaClientKnownRequestError for unique constraint violations like P2002) and throw custom, more meaningful errors or return appropriate null/error responses.
* Data Validation/Sanitization (Basic): While Zod handles API-level validation, service functions should ensure data conforms to database constraints before saving. Prisma handles some of this (e.g., type mismatches), but consider edge cases.

Phase 2.2.3: API Route Integration

1. Refactor Existing API Routes:

* Identify all API routes currently using mock data (e.g., mockProposalsDB in /api/proposals).
* Replace mock data interactions with calls to the newly created service functions.
* Example in /api/proposals/route.ts (for GET):  
  typescript  
    
    
    
    
  Apply to route.ts  
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
    
          // Before  
    
          // return NextResponse.json(mockProposalsDB);  
    
    
    
          // After  
    
          import { getAllProposals } from '@/lib/services/proposalService';  
    
          // ...  
    
          const proposals = await getAllProposals();  
    
          return NextResponse.json(proposals);
* Ensure request bodies (for POST, PUT) are validated using Zod schemas and then passed to the appropriate service functions.
* Map service function responses (or errors) to appropriate NextResponse objects.

1. Create New API Routes:

* If DATA\_MODEL.md implies new API endpoints are needed for the new entities, create these under src/app/api/.
* Follow the same pattern: Zod validation -> service function call -> NextResponse.

Phase 2.2.4: Validation and Testing

1. Data Integrity:

* Test creating, reading, updating, and deleting instances of each new model through your API endpoints to ensure data is stored and retrieved correctly.
* Verify that relationships are established and queried correctly.
* Manually inspect the database (using npx prisma studio) to confirm data integrity.

1. Migration Verification:

* Ensure the migration applied successfully and the database schema matches prisma/schema.prisma.

1. Type Consistency:

* Ensure types are consistent from API request (Zod) -> Service function (Prisma types/DTOs) -> Database schema.

Final Steps & Reminders:

* Documentation:
* Update PROJECT\_REFERENCE.md if new major architectural patterns emerge for data access.
* Log this implementation in IMPLEMENTATION\_LOG.md.
* If significant challenges or solutions are found, document them in LESSONS\_LEARNED.md.
* Code Quality:
* Adhere to existing ESLint rules and TypeScript best practices.
* Ensure all new service functions and API routes have clear, concise JSDoc comments.
* Iterative Approach: Tackle one entity or a small group of related entities at a time to keep the changes manageable.

General Instructions for Your Implementation Work:\*\*

\*

\* \*\*Code Quality:\*\* Strictly adhere to ESLint, Prettier, and TypeScript strict mode.

\* \*\*Redundancy Avoidance:\*\* Before creating new functions or types, always check `src/lib/`, `src/types/`, and `src/hooks/` for existing reusable elements.

\* \*\*Error Handling:\*\* Implement robust and user-friendly error handling as per project standards.

\* \*\*Wireframe & Document Adherence:\*\* Follow specifications from wireframes (`WIREFRAME\_INTEGRATION\_GUIDE.md`), `COMPONENT\_STRUCTURE.md`, and `DATA\_MODEL.md`.

\* \*\*Component Traceability:\*\* Ensure new UI components and features align with the Component Traceability Matrix requirements.

\* \*\*Commits:\*\* Use small, atomic commits with clear messages.\* \*\*Error Handling:\*\* Implement robust and user-friendly error handling as per project standards.

\* \*\*Wireframe & Document Adherence:\*\* Follow specifications from wireframes (`WIREFRAME\_INTEGRATION\_GUIDE.md`), `COMPONENT\_STRUCTURE.md`, and `DATA\_MODEL.md`.

\* \*\*Component Traceability:\*\* Ensure new UI components and features align with the Component Traceability Matrix requirements.