ChatGPT

The Wizard V1 – Technical Design Document

Executive Summary

The Wizard V1 is a **pluggable multi-workflow AI-assisted development system** focused on a **“Spec-Then-Code”** paradigm. It addresses the trust gap in AI coding by introducing **Diagram-as-a-Spec (DAAS)** – using structured diagrams as binding design specifications between the AI and developer. In the primary V1 workflow (codenamed **RIPER-5**), the AI first proposes a design blueprint in Mermaid diagram form, the user reviews/edits and confirms it, and *only then* does the AI generate test cases followed by code strictly according to the confirmed diagram. This forward-DAAS workflow (diagram → code) ensures transparency and developer control at each step, mitigating “black-box” AI behavior and hallucinations [1] [2].

V1 delivers an end-to-end implementation of the Spec-Then-Code workflow within a web-based UI integrated into the development environment. Every stage of the interaction – **Read, Insight, Plan, Execute, Refine** (RIPER) – is explicitly guided and gated by user approval [3]. The system is designed with **extensibility in mind**: although RIPER-5 is the first example “routine”, the architecture allows adding new AI workflows in the future via configurable templates. This document details the V1 technical design, covering terminology, scope, architecture (with layered components and control flows), interface contracts, front-end design (modal editors, etc.), data management, security, logging, performance considerations, and a development plan. By the end of the document, stakeholders should have a clear understanding of how The Wizard V1 will be built to achieve a highly controllable AI coding assistant that aligns with developer intent through DAAS.

Terminology & Source of Truth

- **Workflow Routine (Workflow / “套路”)**: A predefined multi-step AI/user collaboration process to accomplish a coding task. Each routine breaks the task into **phases (stages)** with distinct goals and UI treatments. **RIPER-5** is the primary example routine in V1, representing a five-stage Spec-Then-Code workflow [3].

- **RIPER-5**: Acronym for the five interaction stages of the Spec-Then-Code workflow [4]:
  Read – AI reads and understands context and requirements.
  Insight – AI aligns on the task pattern or strategy with the user (e.g. clarifying which workflow quadrant to use).
  Plan – AI proposes a **visual plan** in the form of diagrams (the “blueprint”), for user review and approval.
  Execute – AI strictly executes the approved plan: first generating test cases, then generating code that fulfills the plan and passes the tests.
  Refine – AI and user verify the output against the plan; AI can analyze its output to prove consistency with the plan, and the user gives final confirmation [3]. (In V1, the Refine stage is present conceptually, but some automated aspects are postponed – see *Scope & Non-Goals*).

- **DAAS (Diagram as a Spec)**: The principle of using a **diagram as a formal specification** of the solution design. In The Wizard, a diagram isn't just documentation – it is the **source of truth** for subsequent generation steps [5]. The AI's plan output is a structured diagram (e.g. Mermaid flowchart/UML), which the user edits and then "locks in" as the agreed blueprint. All code and tests are then derived from this diagram, treating it as an authoritative spec that the AI must follow exactly [6] [2]. This ensures the AI's implementation cannot diverge from what the user approved, effectively placing the AI "on rails" defined by the diagram.

- **Blueprint**: The **design diagram** produced in the Plan stage (and potentially other supplementary diagrams/views). It captures the key components, modules, or sequence of logic for the task. In V1, the blueprint is typically a Mermaid diagram (e.g. flowchart, sequence diagram, class diagram) that is human-readable and machine-parsable. Once confirmed, the blueprint serves as the contract between user and AI – it is the **frozen spec** that drives code generation [5]. The blueprint's Mermaid **text** is treated as the single source of truth for the implementation.

- **Source of Truth**: In this context, the term refers to the **final confirmed diagram artifact** (the blueprint text) that is considered the ground truth specification. All subsequent outputs (test cases, code) must trace back to this source. If there's any discrepancy, the blueprint is the reference against which corrections are made. This approach ensures consistency from design to code and keeps the AI accountable to the agreed plan [2].

- **Spec-Then-Code**: The overall methodology enabled by the above concepts – a guided workflow where specification (in diagram form) is created and agreed upon *before* any code is written. It contrasts with end-to-end code generation by inserting a human-verifiable spec step, thereby increasing trust and correctness. "Spec-Then-Code" is the core mode The Wizard V1 implements (as opposed to other modes like quick scripting or traditional TDD, which are out of scope in V1). In the PRD it's called "规格先行开发" [7].

- **Routine Engine / Guide Engine**: The back-end component that manages workflow routines. It loads the definition of each routine (stages, prompts, etc.), steps through the stages in order, and handles transitions based on user input and AI outputs. The routine engine ensures that each stage is executed with the correct prompt and that user approvals are obtained before moving on. It is designed to be **pluggable** – new routines can be added via configuration or code without altering the engine's core logic [8] [9].

- **Prompt Template**: A parameterized prompt script associated with a given stage of a routine. Prompt templates encode how the AI is instructed at that stage – for example, the Plan stage template will prompt the AI to output a design diagram, whereas the Execute stage template will prompt it to output code, given the blueprint. Templates often include placeholders for dynamic content (like the current **blueprint text** or **test cases**) and enforce certain style or rules (e.g. *"You must not introduce steps not in the diagram"*). They serve as the primary means of guiding the LLM's behavior in each stage.

- **Mermaid**: A text-based diagram notation (commonly used for flowcharts, sequence diagrams, etc.). The Wizard uses Mermaid as the initial medium for DAAS because of its simplicity and wide adoption in developer tools. The AI outputs Mermaid code for diagrams, and the front-end renders it. Mermaid diagrams are easy to edit as plain text by the user and easy for the AI to parse or regenerate, fitting the DAAS need for a machine-human shared language [10]. (Future versions may support other diagram DSLs, but Mermaid is the focus for V1 [11].)

Scope & Non-Goals

In Scope (V1): The Wizard V1 will implement a **complete end-to-end "Spec-Then-Code" workflow** with all primary stages and features needed for a basic but usable experience [12]. Specifically, V1 will deliver:

- **Full RIPER-5 Workflow** – The five-stage guided interaction (Read, Insight, Plan, Execute, Refine) for spec-first development, including UI and logic for each stage and transitions through to completion [12]. This includes generating a diagram, getting user confirmation, generating tests, then code, and final user approval of code.

- **Single Routine Focus** – The **Spec-Then-Code** routine (RIPER-5) is implemented in depth. Other collaboration modes (e.g. rapid scripting, prototyping, TDD) will be available only as stubs or simple pass-throughs (the user can select them, but they won't have specialized multi-stage flows in V1) [13]. The architecture will support multiple routines, but full-featured support is **in-scope only for the spec-first routine** in this version.

- **Pluggable Workflow Engine** – The system will include a **config-driven routine registration** mechanism [14]. In V1, this may be as simple as loading a JSON/YAML or using a hardcoded definition for RIPER-5, but the design accommodates adding new routines by providing their stage definitions and prompts without modifying core logic. This fulfills the "workflow container" concept: The Wizard can host multiple guided workflows.

- **Diagram Editing & Confirmation UI** – A dedicated UI for the Plan stage where the AI-generated **DAAS blueprint** is displayed and can be edited by the user (Mermaid source editor with real-time preview) [15]. The user **must explicitly confirm** the diagram (e.g. via a "Confirm Blueprint" action) to proceed [16].

- **Test-first Code Generation** – After blueprint confirmation, the AI will generate key unit **test cases first**, present them to the user (in an editable tabular form), and only proceed to code generation after the user approves the tests [17]. This ensures an additional checkpoint and that code is written to satisfy agreed tests ("tests as spec" aligning with the blueprint).

- **Stepwise User Approvals** – Every critical transition requires user action: e.g. confirming the diagram, approving the tests, accepting the final code [18]. The workflow can also handle user requesting changes at a stage (e.g. regenerate or modify diagram) before moving on. **Ability to go backwards** one step is supported – e.g. after seeing tests or code, the user can decide to go back to the Plan stage to adjust the blueprint (with appropriate state handling) [14].

- **Session State & Versioning** – The system will maintain state across stages, including the original AI outputs and user-modified versions. Particularly, **diagram versioning** is in scope: the system will keep track of the initial AI-proposed diagram vs. the user-edited confirmed diagram (and potentially prior versions if the user iterates) [19]. This version control is session-scoped and allows comparing or reverting changes within the workflow.

- **Graceful Failure Handling** – If the AI cannot fulfill a request or hits an error (e.g. “I don’t have enough info to do this”), the system will handle it gracefully rather than crashing the workflow [20]. This may include showing an error message with suggestions, allowing the user to adjust input or abort the routine, rather than leaving the AI in an undefined state. V1 relies on underlying platform stability, but we will catch and present model errors or unfulfillable requests in a user-friendly manner.

- **Basic Logging for Metrics** – V1 will implement minimal logging necessary to compute the **core success metric**: the **Spec-Then-Code happy path completion rate** [21]. This means logging events such as when a session starts, when the user confirms the blueprint, when code generation completes, and whether the user ultimately clicks “Accept Code”. These logs will allow calculating what fraction of initiated workflows result in a completed flow. No full analytics dashboard is built (see non-goals), but the data to measure success is collected [20].

Out of Scope (V1 Non-Goals): To focus on core functionality, several features and optimizations are **not included in V1** (some were considered in earlier plans but deferred – these are noted as historical adjustments to scope):

- **Other Collaboration Modes** – Aside from the spec-first (Spec-Then-Code) workflow, the other three AI programming modes mentioned (quick scripting, exploratory prototyping, test-driven development) will **not have detailed guided flows** in V1 [13]. The UI may present the options, but choosing them will likely result in a simple fallback behavior (e.g. direct one-shot answer) or a message that they are not fully available. The focus is exclusively on perfecting the spec-first guided workflow.

- **Performance Optimizations** – V1 will not spend effort on heavy performance tuning or scaling beyond the baseline provided by the underlying platform [22]. For example, we won’t implement caching of LLM calls, extensive client-side optimizations for diagram rendering, etc., unless needed for basic functionality. The initial target is a functional prototype; performance will be revisited in later versions if needed.

- **Advanced Reliability & Security Hardening** – We will largely rely on the security and reliability features of the host platform (Cursor IDE and any MCP/LLM services) for V1 [22]. This means no custom sandboxing beyond basic file access restrictions, no special guardrails beyond prompt-based controls, and no elaborate recovery mechanisms if the backend crashes. Basic compliance (no code auto-writing without confirm, etc.) is in scope, but things like sandboxed code execution or complex security audits are not in V1’s scope.

- **“Guide Store” Marketplace** – The idea of a marketplace or library for sharing/swapping new workflow routines (so that users can download community-created guides) is a future concept and not part of V1 [23]. V1 will have the capability to add new routines *for developers internally*, but there is no UI for end-users to add/download workflows at runtime.

- **Fully Automated Code Verification (Reverse DAAS)** – V1 will implement **forward DAAS** (diagram → code) but **will not implement automated reverse-DAAS** (code → diagram consistency checks) in this initial version. Originally, a **Refine stage** was conceived where the AI would generate a diagram from the produced code and compare it to the original blueprint [24]. This is **no longer in V1 scope** due to complexity and time constraints. Instead, the Refine stage in V1 will be minimal (perhaps just a manual review step where the user can compare the blueprint and code by eye). Any automated generation of a “code traceability diagram” or diff highlighting differences is deferred to a future release [25].

- **“Accept Code” Integration** – While the UI will have an **“Accept Code”** button to finalize the workflow, in V1 this action will **not be deeply integrated** with project version control or automated test execution. Its function is largely to mark the end of the workflow and maybe copy the code into the editor, but it will not, for example, auto-merge changes into a repository or run a full test suite. Integration points such as automatically inserting the code into the user’s file system or committing to git are out of scope for now. The developer will manually review and integrate the code output.

- **Desktop/IDE Native Interface** – The Wizard V1 will run as a **web-based UI** (likely a web view embedded in Cursor IDE or a browser-based panel). A native desktop UI or deep IDE plugin integration is not targeted in V1 [26]. The architecture is designed such that a desktop front-end could be added later, but for now we focus on the web front-end. Consequently, offline use or integration into other IDEs (VS Code, JetBrains, etc.) is not included, though we anticipate adding those in future versions [26].

- **Subagent / Multi-Agent System** – V1 uses a single AI agent (the primary LLM) to handle all stages of the workflow. Although the architecture document envisions a future **subagent system** (multiple specialized AI agents for different tasks) [27] [28], none of that is implemented in V1. There will be no orchestrating of multiple agents or roles; the complexity of coordinating an “AI team” is beyond the initial scope. However, the modular design (tools interface, etc.) lays groundwork that could allow adding subagents later without redesign [29].

By constraining scope in these ways, we ensure that The Wizard V1 can be delivered in a timely manner with high quality for its core functionality. Future versions will address the non-goals once the viability of V1’s approach is proven in practice.

System Architecture

High-Level Architecture

At a high level, The Wizard V1 follows a client-server architecture with a modular design. The main components are:

- **Frontend Web UI** – A browser-based application (embedded in the IDE or Electron container) that provides the user interface: chat interactions, stage navigation, diagram editor modal, test review table, etc. It is built with modern web tech (likely a React-based UI with custom JS/TS) [30]. The front-end communicates with the backend via a standardized API (HTTP/REST and/or WebSocket for real-time updates).

- **Backend (Wizard Engine)** – The backend orchestrates the workflow. It maintains the state machine for the current session, invokes the AI model with appropriate prompts for each stage, and handles user feedback loops. It encapsulates several sub-modules:

- **Workflow Engine** – Manages the progression of stages as defined by the active routine [31]. It knows the sequence of stages (for RIPER-5, the order and criteria for moving from one to next) and handles branching or looping logic (e.g. going back to Plan if needed).
- **Prompt/AI Interface Layer** – Responsible for communicating with the Large Language Model (LLM) or AI agent [32]. This layer formats the prompt using the stage's template, sends it to the model (via an API or through an MCP protocol if running an agent), and obtains the AI's output. It also may invoke **tools** or external APIs if the AI needs (e.g. a code execution sandbox, a search engine) – though in V1, such tool use is limited.
- **Stage Handlers/Tools** – Specific support for certain stages that require external operations. For example, if we had a stage that needed to run code or retrieve context from files, a handler in this module would do that. In V1, the Plan stage uses a Mermaid renderer (on frontend) but no special backend tool; Execute stage might interface with a code compiler or test runner (though likely minimal in V1). The architecture is modular so each stage's special needs can be handled in isolation [33].
- **Workflow Definition Registry** – The logic to load and register available workflow routines [34]. In V1, this will include the built-in definition for RIPER-5. The system can be configured to load additional routines from config files or plugins, though this is primarily for future use. The registry provides the metadata for each stage (name, description, prompt template, UI type, etc.) to the engine [8].
- **Session State Store** – Data storage for the current workflow's context and outputs [35]. This keeps track of things like: the initial problem statement, AI outputs at each stage (e.g. the diagram text), user edits (the final diagram), test cases generated, and so on. In V1 this can be in-memory (since each Wizard session is likely short-lived and single-user), possibly with a simple data structure or a lightweight database if needed for persistence across app reloads. It ensures that information flows from one stage to the next and remains consistent.
- **AI Model** – The large language model or AI agent used to generate content. In V1, this could be an API call to a service like OpenAI GPT-4 or an internal model accessible via the MCP (Model-Context Protocol) interface. The AI model is external to The Wizard but is invoked through the AI Interface layer. It's effectively a dependency providing the "intelligence" for generating diagrams, tests, and code according to our prompts.

Below is a high-level component diagram illustrating The Wizard's architecture and data flow:

```
flowchart LR
    subgraph Browser (Frontend)
        UI[Wizard Web UI<br/>(React app)]
    end
    subgraph Server (Backend)
        WEngine[Workflow Engine<br/>(Routine Orchestrator)]
        AIProxy[AI Interface Layer]
        Tools[Stage Tools/Helpers]
        DB[(Session State Store)]
    end
    AIModel[LLM AI Model]

    UI -- API calls / WebSocket --> WEngine
    WEngine -- prompt --> AIProxy
    AIProxy -- request--> AIModel
    AIModel -- response--> AIProxy
    AIProxy -- stage output --> WEngine
    WEngine -- push update --> UI
    WEngine --> DB
    UI -- user actions --> UI
    style UI fill:#f0f9ff,stroke:#66b1ff,stroke-width:1px
    style WEngine fill:#fdf5e6,stroke:#e1b12c,stroke-width:1px
    style AIProxy fill:#fdf5e6,stroke:#e1b12c,stroke-width:1px
    style Tools fill:#fdf5e6,stroke:#e1b12c,stroke-width:1px
    style DB fill:#eee,stroke:#888
    style AIModel fill:#e8f7e4,stroke:#52b788,stroke-width:1px
```

Diagram Description: The front-end **UI** communicates with the back-end **Workflow Engine** via API or WebSocket. The Workflow Engine loads routine definitions and manages stage progression. At each stage, it uses the **AI Interface Layer** to formulate a prompt and call the external **AI Model**. The model's output is returned to the engine, possibly passing through stage-specific **Tools** if needed (e.g. processing or validation). The engine then stores relevant data in the **State Store** and sends the result to the UI. The UI presents the result to the user (e.g. showing the diagram or code) and awaits user actions (edits, confirmations), which feed back into the engine to trigger the next stage. This modular design allows the front-end and back-end to evolve independently and makes it easier to swap out components (for example, use a different AI model or add new tools) without affecting the whole system [36] .

Key design considerations from this architecture:

- **Layered Separation:** We separate concerns into frontend vs backend, and within backend into workflow control vs AI communication vs specialized operations. This makes the system flexible – e.g., the AI interface could be replaced to support a different LLM provider or even multiple models (in future), without changing the workflow logic [36] .

- **Extensibility:** Adding a new workflow routine primarily means adding new config (or classes) in the workflow registry and providing appropriate prompt templates and UI components. The engine logic is generic over any sequence of stages with defined transitions [8] . Similarly, new tools (say a static analysis tool for a new stage) can be plugged in via the Tools module without touching core logic.

- **Frontend/Backend Decoupling:** They communicate via **standard protocols** (HTTP/WS with JSON). This decoupling means we could run the backend as a local service or a cloud service, and the frontend could be embedded in different host apps (web app, desktop app) in future with minimal changes [26] . It also allows the possibility of integrating The Wizard into different IDEs via an API, by treating the backend as a service.

- **State Management:** The use of a session store (in-memory or persistent) ensures that multi-step workflows have continuity. For example, the Mermaid diagram text produced in Plan stage is stored so it can be passed into the Execute stage prompts; user edits are captured and not lost on refresh; if a user goes backward to Plan, the previous state is retrieved (or an earlier version of the diagram can be restored) [19] .

Data & Control Flow (Forward DAAS Workflow)

The sequence of operations in the Spec-Then-Code workflow (RIPER-5) can be viewed as a **state machine** with user-triggered transitions. The system moves through states corresponding to the five stages, with possible loops for refinement. Below is a state diagram of the RIPER-5 routine including key transition triggers:

```
stateDiagram
    [*] --> Read : User starts Spec-Then-Code routine
    Read --> Insight : (automated after context analyzed)
    Insight --> Plan : User selects "Spec-First" mode (task alignment done)
    Plan --> Execute_Tests : User confirms the Blueprint diagram [16]
    Execute_Tests --> Execute_Code : User approves generated Tests [17]
    Execute_Code --> Review : Code output delivered
    Review --> [*] : User accepts code (workflow complete) [24]
    Review --> Plan : User rejects output / needs changes (go back to Plan to
refine)
    note right of Plan
        (User can edit diagram or
         request AI adjustments
         before confirming)
    end note
    note right of Execute_Tests
        (AI generates test cases
         based on Blueprint;
         user can edit tests or approve)
    end note
    note right of Review
        (In V1, Review is manual check;
         user can loop back to Plan if issues)
    end note
```

Workflow States & Transitions:

- **Read:** Initial state where the AI gathers context. In practice, this might happen automatically when the routine starts (the AI may summarize the problem or relevant code context). Transition: once the AI has provided initial analysis (or if none needed), proceed to **Insight**.

- **Insight:** The AI (and user) establish the approach. In V1, this stage is lightweight – essentially the user's choice to engage the Spec-Then-Code workflow. (If the user invoked The Wizard and selected the Spec-First mode, the system is already aligned.) Transition: move to **Plan** either immediately or after any clarifying Q&A.

- **Plan:** The core planning stage where **DAAS comes into play**. The AI produces an initial **Blueprint diagram** (Mermaid code) representing the solution plan [37]. The front-end displays this in the dual-pane editor. The user can modify the diagram (or ask the AI to refine it via natural language input) until satisfied [38]. **Transition:** When the user clicks “Confirm Blueprint”, the final diagram text is sent to the backend [39]. The state machine then advances to the Execute phase.

- **Execute – Tests:** The first part of Execute stage. Upon entering Execute, the backend uses the now-confirmed blueprint to prompt the AI to generate **critical unit test cases** (e.g. happy path scenarios) that the implementation should satisfy [17]. The AI’s proposed tests are sent to the frontend, which displays them in an editable table. The user reviews and can edit these test cases (or add any crucial ones they feel missing). **Transition:** The user presses “Approve Tests & Generate Code” (or equivalent) to signal that the tests are acceptable [17]. This locks in the test specifications and triggers the next sub-stage.

- **Execute – Code:** The second part of Execute. The backend now prompts the AI to generate the actual **implementation code**, explicitly instructing it to follow the approved blueprint and to make sure the code will pass the approved tests [17] [6]. The AI returns code (for one or multiple files or functions), which is delivered to the user, typically in the chat or a code viewer. **Transition:** Once code is provided, the system moves into the Review stage. (Note: If the AI fails to produce code or tests due to error or refusal, the system would handle that as a failure case – possibly letting the user go back or abort gracefully, per the failure handling in scope.)

- **Review:** In an ideal scenario, this is the final verification stage. In the envisioned design, the AI would analyze the produced code and generate a “traceability” diagram to compare with the original blueprint [24], highlighting any deviations (this would be *reverse DAAS*). However, as noted, V1 will not automate this. Instead, in V1 the Review stage will present the original Blueprint alongside the code output for the **user to manually compare**, possibly with a checklist of things to verify (or simply an instruction to run tests). The UI will provide an **“Accept Code”** button here [40]. The user can at this point either accept the code as satisfactory or decide that changes are needed.

- **Completion or Refinement:** If the user clicks **Accept Code**, the workflow ends successfully (final state). If the user is not satisfied – e.g. they spot a discrepancy or a missing piece – they have the option to **go back to the Plan stage** (or potentially an earlier stage) to refine the design and implementation. The system will support a backward transition from Review to Plan (possibly skipping earlier stages as needed) to allow iteration: the user could edit the diagram to address the issue and then proceed forward again through Execute. In V1, we will implement at least the ability to return to Plan from the final stage; more complex rollback (like jumping back multiple stages) may be limited, but the architecture allows it.

This state machine ensures that data flows in one direction (forward) under normal operation, always gated by user approval at key points, which embodies the forward-DAAS principle. **Data artifacts flow** as follows: initial user query/context → AI’s draft diagram → user-edited diagram (final blueprint) → AI’s test cases → user-approved tests → AI’s code output → (user final check). Each artifact flows into the next step, and nothing is used for generation unless it has been confirmed by the user. This guarantees that the final code is grounded in user-approved specifications [6].

Extensibility Points: The architecture is built to be **extensible** both in terms of adding new workflows and swapping components:

- *Multiple Workflows:* The **routine registry** can hold multiple workflow definitions. In V1, RIPER-5 is the only fully implemented routine, but others can be registered in a configuration file or by code. The engine doesn't hardcode any RIPER-5 specifics except loading its definition. To add a new routine (say, a TDD workflow with stages like "Write Test → Write Code → Refactor"), a developer would create a config (JSON/YAML or Python class) listing the stages, their order, prompt templates, and any special UI settings. The Wizard can then load it and present it as an option to users. Each routine can define its own sequence of stages and thus orchestrate a different pattern of AI-user interaction [8] [41].

- *Stage Templates & Tools:* New **prompt templates** can be incorporated easily for different domains or updated as we refine our AI instructions. For example, if we find a better way to prompt the AI to generate diagrams, we can update the Plan stage template without touching other code. Similarly, the design allows plugging in new **tools** or adapters for specific stages. If future workflows require, say, a static analysis step or a documentation generation step, we can add a stage with an associated handler module. The main engine will call it at the right time. As an example, the architecture already envisions possibly calling external code execution in Execute stage or search in Research stage [33], though in V1 we may not fully utilize these.

- *Replaceable AI Backend:* The AI interface is abstracted such that we could switch between different LLM providers or even a set of models. For instance, in a future with multiple subagents, the Orchestrator agent could query different specialized models (one for coding, one for architecture) and combine results [42]. While V1 sticks to a single model pipeline, the loose coupling (via an interface and message protocol) means we could upgrade the AI component (to a newer model or a self-hosted model) with minimal changes elsewhere.

- *Frontend Modularization:* The front-end is also built in a modular way. Key UI components like the **diagram editor modal** are encapsulated so they can be reused or replaced [43]. If, for example, we later integrate a more advanced diagramming tool or add support for a new diagram type, we could swap out that component. The staged navigation is also data-driven (rendering stages based on definitions), which makes adding a new stage mostly a matter of providing the UI for it. The UI design for each stage is tailored, but using a common framework (React) with shared state management, so additional stages or adjustments to flow can be made without rewriting the entire UI.

In summary, the system architecture provides a robust foundation for the Spec-Then-Code workflow in V1, while being flexible enough to accommodate future enhancements such as new workflows, additional AI capabilities, and different deployment environments.

Interface Design

This section describes how different parts of the system interact through defined interfaces and data contracts: how routines are specified/registered, how the front-end and back-end communicate, and how prompt templates are structured.

Routine Registration & Configuration

Workflow routines in The Wizard are defined by a set of stage metadata and associated behavior. In V1, the definition for RIPER-5 is likely hard-coded for expediency [34], but we treat it as if it were loaded from a config to illustrate the design. A routine definition includes:

- **Routine ID and Name:** e.g. `"id": "RIPER-5", "name": "Spec-First Development (RIPER-5)"`.
- **Stages:** an ordered list of stage definitions. Each stage has:
- An **ID/Key** (e.g. "Research", "Innovate", ...),
- A **Human-Friendly Name** (could be same as ID or localized),
- A **Description/Purpose** (what the stage is for, used for tooltips or logs),
- The **Prompt Template** or a reference to it,
- **UI Type/Config** indicating how the front-end should present this stage (e.g. `"ui": "chat"` for normal chat interaction, `"ui": "mermaid_editor"` for the Plan stage diagram editor, `"ui": "test_table"` for test review, etc.),
- Any **special parameters** (for instance, a stage might have `allowUserEdits: true` if the user can edit the AI output directly, as in Plan and Execute/Tests).
- **Transitions & Rules:** Optionally, some rules about how to transition (though generally the engine assumes linear unless a user action triggers a loop).

For example, a simplified JSON-like representation for RIPER-5 might look like:

```
{
  "id": "RIPER-5",
  "name": "Spec-Then-Code (RIPER-5)",
  "stages": [
    {
      "id": "Read",
      "name": "Read",
      "description": "AI reads context and requirements.",
      "promptTemplate": "prompt_read.txt",
      "ui": "chat",
      "requiresUserInput": false
    },
    {
      "id": "Insight",
      "name": "Insight",
      "description": "AI and user align on approach (choose workflow).",
      "promptTemplate": "prompt_insight.txt",
      "ui": "chat",
      "requiresUserInput": true  /* e.g., user selects an option here */
    },
    {
      "id": "Plan",
      "name": "Plan",
      "description": "AI proposes a solution blueprint (diagram).",
      "promptTemplate": "prompt_plan.txt",
      "ui": "mermaid_editor",
      "allowsEdit": true,
      "confirmAction": "Confirm Blueprint"
    },
    {
      "id": "Execute",
      "name": "Execute",
      "description": "AI generates tests, then code according to blueprint.",
      "promptTemplateTest": "prompt_execute_tests.txt",
      "promptTemplateCode": "prompt_execute_code.txt",
      "ui": "code_generation",
      "confirmAction": "Approve Tests & Generate Code"
    },
    {
      "id": "Review",
      "name": "Review",
      "description": "User reviews output vs. plan, final acceptance.",
      "promptTemplate": "prompt_review.txt",
      "ui": "review",
      "confirmAction": "Accept Code",
      "allowsEdit": false
    }
  ]
}
```

> **Note:** This example is illustrative – the actual implementation may not literally use a JSON file like this in V1, but the design keeps it conceptually similar. The **Execute** stage in this example shows two prompt templates, reflecting that it has two sub-steps (test generation and code generation). Alternatively, this could be modeled as two separate stages (Execute-Tests and Execute-Code) in the list. The Wizard engine is flexible enough to handle either approach. We anticipate possibly representing them as one stage with an internal sequence to preserve the "5" in RIPER-5 for user messaging, but internally it will execute two prompts.

Routine Registration: On backend initialization, the system will load the routine definitions (either by reading config files or by constructing them in code) [34]. Each routine is registered with the Workflow Engine. If using configuration files, we'll likely have a folder like `routines/` containing `riper5.json`, etc., which the engine parses. The PRD indicated a preference for JSON/YAML based definitions for portability [14].

The front-end likely queries the backend for available routines when The Wizard is invoked, in order to display the initial mode selection UI (for the four quadrants of AI programming). In V1, since only spec-first is fully implemented, the other options might be present but disabled or marked beta.

Frontend-Backend API Contracts

The front-end and backend communicate through a clearly defined interface. We expect to use **WebSocket** for real-time bidirectional communication (so the backend can push stage outputs to the UI as soon as they're ready), possibly combined with REST endpoints for initiating or controlling sessions.

Key API interactions and their data formats include:

- **Start Workflow**: When the user invokes The Wizard on a task (e.g. via an `@TheWizard` command in the IDE), the frontend sends a request to start a new session. For example:

Request: `POST /wizard/start` with JSON body `{"routine": "RIPER-5", "userQuery": "<User prompt or context>"}`.

Response: The backend responds with a session identifier and possibly the initial stage output if it's ready immediately. Alternatively, the response might just confirm session creation, and the actual content from the first stage (Read/Insight) will arrive via WebSocket events.

- **Stage Output Event**: When the AI completes a stage output (e.g., finishes generating the diagram for Plan stage), the backend pushes an event over WS: e.g. an event type `"stage_result"` with payload `{"stage": "Plan", "content": "<Mermaid code or other>", "render": "mermaid"}`. The front-end receives this and renders the appropriate UI (here, open the diagram editor with the Mermaid content).

- **User Feedback/Action**: When the user performs an action like confirming a blueprint or editing something, the front-end sends this to the backend. For instance:

- If the user edits the Mermaid diagram and clicks **Confirm Blueprint**, the front-end sends an event or request like: `{"action": "confirm_stage", "stage": "Plan", "data": {"finalDiagram": "<Mermaid text>"}}`. The backend then takes that diagram text, stores it, and proceeds to the next stage (Execute tests generation).

- If the user instead requests a change before confirming (say they type a comment "Please add an error handling step" in a chat input during Plan stage), the front-end might send something like `{"action": "user_feedback", "stage": "Plan", "message": "Add error handling for XYZ."}`. The backend would then incorporate that into a prompt to the AI to regenerate or adjust the diagram (essentially staying in the Plan stage).

- **Error Handling**: If something goes wrong (AI fails or any internal error), the backend will send an error event to the UI. For example: `{"event": "error", "message": "Unable to generate diagram. Please refine the request or try again."}`. The UI will display this gracefully (perhaps as a message in the chat or a notification bar) [20]. The user can then decide to adjust input or cancel.

- **Progress & Status**: Optionally, the backend might send intermediate status updates (like a "thinking…" or progress percent) to the UI, especially if an operation is taking a while (for instance, generating a large code file). This can be as simple as an event `{"event": "status", "stage": "Execute", "status": "Generating code..."}` so the UI can show a loading indicator.

- **Session End**: When the user clicks **Accept Code** or otherwise ends the session, the frontend may call an endpoint like `POST /wizard/finish` or the backend might infer it from the confirm action on Review. The backend can then finalize logs, free resources, etc. The UI might receive a final confirmation event or simply close the Wizard UI.

Data Types & Examples:

- **Diagram Data**: The Plan stage output is essentially a text blob (Mermaid syntax). It may also include metadata like diagram type. Example payload excerpt: `{"stage": "Plan", "diagramType": "flowchart", "diagramSource": "flowchart TD; A-->B; ...", "notes": "Diagram of module interactions."}`.

- **Test Cases Data**: We plan to represent tests in a structured way. Possibly as a list of test case objects, each with a description or given/when/then fields. For UI ease, the backend might send tests as an HTML or Markdown table to render. But a cleaner way: backend sends JSON like:

```
{
  "stage": "Execute",
  "substage": "tests",
  "tests": [
     {"id": 1, "description": "Given valid input X, the system returns Y"},
     {"id": 2, "description": "If X is null, an error is logged and no
crash"}
  ]
}
```

  The front-end can use this to populate a table where the user can edit the descriptions.

After user edits, the front-end sends back perhaps the same structure with modifications. The backend doesn't need to deeply understand the semantics of tests; it can simply incorporate the final text of each test case into the code-generation prompt.

- **Code Data**: The code output might be sent as one or more code blocks. If the AI returns markdown with triple backticks, the backend could forward that as-is (the UI can render it). Or the backend might structure it with filenames. For example, if multiple files are generated, the backend could send:

```
{
  "stage": "Execute",
  "substage": "code",
  "files": [
    {"path": "src/Foo.java", "language": "java", "content": "public class
Foo { ... }"},
    {"path": "src/FooTest.java", "language": "java", "content":
"@Test ..."}
  ]
}
```

If integration with the IDE is possible, these could even be suggestions to open editors with that content. In V1, likely we'll just display the code in the chat area or a modal.

- **Confirmation Actions**: For each stage that requires confirmation, the UI and backend need to agree on how to represent it. Likely, the backend when sending a stage result that needs confirmation will include a flag like `"awaitingUser": true` and maybe an action label. The UI will then enable the appropriate confirm button. When the user clicks it, the backend expects a `confirm_stage` action as described. We will standardize this to avoid confusion.

Error Handling & Edge Cases: - If the AI returns output that doesn't parse (e.g., a Mermaid syntax error), the backend could catch this by attempting to render or validate Mermaid (possibly using a Mermaid parser library). If invalid, the backend can either retry with a refined prompt or notify the UI so the user knows the diagram might be incomplete. In V1, we lean on the user to spot issues, but we'll try to at least not crash. The diagram editor might highlight syntax errors (Mermaid parsing errors) client-side as well [19]. - If the user becomes unresponsive (e.g., doesn't confirm or provide input), the backend will essentially idle. We might implement a timeout or ping to keep the session alive, but no drastic action (the user is always in control, so the AI waits). - If the user cancels the workflow mid-way (closes the Wizard UI), the front-end should notify the backend to terminate the session to free resources (especially if the AI was in the middle of a long generation). - If network disconnects, the system should allow resuming the session if possible. This may not be fully handled in V1 beyond relying on the user to restart a new session if things break.

Overall, the front-end/back-end contract is designed to be **explicit and stateless between steps** (each message contains the info needed to understand it, using the session context on the backend). By clearly delineating message types and payloads, we ensure the two can evolve somewhat independently and ease debugging (we can log or inspect the JSON messages flowing through).

Prompt Templates Design

Prompt templates are central to guiding the LLM's behavior in each stage. Each template is basically a text with placeholders, written to imbue the AI with the right instructions, context, and formatting requirements.

General principles for templates in The Wizard V1:

- **Stage-Specific Role**: Each template sets the context of the stage, often by explicitly telling the AI what its goal is in that step (e.g. "You are now in the Plan stage. Produce a system design diagram..."). This aligns the AI's output with the expected outcome of that stage [44] [45].

- **Context Injection**: The template will include placeholders for relevant context from previous stages. For example:

- In Plan stage, the template includes the **user's requirements or prompt** (from Insight stage or initial query) and any insights gathered.
- In Execute stage, the template includes the **final Blueprint diagram text** and the **approved test cases**.

- In Review stage (if it were automated), it would include both blueprint and code.

- **Format & Style Instructions**: We instruct the AI on how to format its answer. For Plan stage, we'll instruct it to output Mermaid syntax (and possibly nothing else extraneous, maybe a brief explanation if needed but likely we want primarily the diagram code) [46] . For Execute stage, we instruct the AI to output code in proper fenced blocks, including file names if multiple (maybe using the markdown syntax extension like `` ```java:FileName.java `` for clarity).

- **Strong Constraint Language**: Especially for the Execute stage, the prompt will **strongly emphasize that the AI must strictly follow the blueprint and tests** and not introduce any unwarranted changes [2] . For example, the prompt might say: *"You are to implement the solution exactly as per the confirmed design. DO NOT add new steps or modules not present in the design. Ensure all of the following tests will pass:"* and then list the test cases. This acts as a safeguard against hallucination or scope creep. It effectively turns the blueprint into a contract that the AI is bound to [47] .

- **Blueprint and Test Injection**: The blueprint text will be injected in a way that the AI can easily reference it. We might format it as a code block or as a preserved section. For example, the Execute-code template could be:

~~~
You have the following design specification (Mermaid diagram) approved by
the user:
```mermaid
(diagram text)
~~~

  And the following unit tests to satisfy:

- Test1: ...
- Test2: ...

Write the complete implementation in code that follows the design and passes all the tests. ``` By providing the diagram in the prompt, the model can parse it or at least use it as a structured guide. We will also instruct the model to make sure every element in the diagram is implemented and nothing outside of it is implemented.

- **Mode Declarations (if any)**: The internal design of the AI agent might require special mode syntax (for instance, if using an agent like in the `RIPER-5-EN.md` instructions, it had things like `[MODE: PLAN]` prefixes [48]). In V1, since we control the outputs, we might not need to expose those mode tags to the user, but the prompt may include certain tokens to keep the AI on protocol (e.g., telling it which mode it is in to prevent it from doing out-of-scope actions). Ensuring the AI does *not* deviate from the plan is paramount – if the underlying AI was fine-tuned or prompted with a system message containing the protocol (like in `RIPER-5-EN.md`), we will leverage that. The templates will likely reference that underlying guidance, for instance reminding: *“(You are in EXECUTE mode; follow the plan checklist exactly)”* [49] [50] .

- **User Language & Tone**: The PRD noted that responses should be in Chinese for the user-facing parts, with certain technical outputs in English for consistency (like mode names, code, etc.) [51] . The prompt templates will enforce this: for example, the Plan stage might generate the diagram (which is language-agnostic), but any explanatory text AI gives might need to be in Chinese (if it describes the diagram). We’ll instruct the AI accordingly on language. Similarly, test case descriptions could be generated in Chinese to align with user language, or in code form if it’s actual code. We ensure the templates cover this localization aspect (e.g. *“Answer the user in Chinese. Only use English for code or mode tags.”* as needed) [51] .

- **Examples and Few-Shot**: We might include example formats if necessary to guide the AI. For instance, in the Plan prompt, we might provide a mini example of a Mermaid diagram format so it knows to produce just the code. In the Execute prompt, if multiple files are expected, show an example format using triple backticks with file names. This helps ensure the output is parseable by our UI.

- **Testing Prompt**: For test generation (Execute-tests substage), the template will prompt the AI to output a set of test cases likely in a simple descriptive form. We’ll likely instruct it to focus on *“critical happy path and edge case scenarios, without code, just descriptions or pseudocode of tests”*. This output should be concise and clear, as it will be shown in a table for user approval.

Prompt Template Example Snippets:

- *Plan Stage Template (pseudo):*

```
[System role:] You are an expert software architect helping to plan the
solution.
[Instruction:] Propose a design for the following requirements in the form
of diagrams.
Include a sequence of steps or component interactions. Use Mermaid syntax
to create:
1. A sequence diagram or flowchart showing the process flow.
2. A class or module diagram showing key components (if applicable).
Ensure the diagrams cover all key requirements identified.
Only output the Mermaid code for the diagrams, prefaced by a short title if
necessary.
[Context:]{requirements_and_context}
```

Rationale: This would yield something like a flowchart and maybe a class diagram text. (We might actually use two separate prompts to get two diagrams, or one prompt that yields two diagrams in one go, depending on model capability. FR-2.1 suggests two views [52] , but implementing both might be ambitious. At least one diagram is certain.)

- *Execute (Tests) Template:*

```
[System:] You are a meticulous test engineer.
[Instruction:] Based on the approved design below, enumerate the key unit
tests (in plain language or pseudocode) that the implementation must pass.
Focus on the core functionality and one or two edge cases.
Format as a list of test case descriptions.
Design Blueprint:
"""mermaid
{diagram_text}
"""
```

Output: The AI might list tests: “1. When X happens, the system should do Y. 2. If input is invalid, the system should log an error and not crash. …”

- *Execute (Code) Template:*

```
[System:] You are now an implementation assistant.
[Instruction:] Implement the solution **exactly** according to the design
and tests below.
- **Do not deviate** from the design. Do not add new functionality not
described.
- Ensure the implementation will satisfy all the tests.
Design Blueprint (for reference):
"""mermaid
{diagram_text}
"""
Approved Tests (for reference):
{test_list_text}
Now, produce the complete code. Include all necessary classes/functions.
Use proper code blocks with language annotations.
If multiple files are needed, separate them clearly (e.g., one code block
per file with the file path in the header).
```

This template explicitly reminds the model to follow the blueprint structure and cover the tests. It provides the blueprint and tests as references. With this, we expect the model to translate each part of the diagram into code, and ensure, for example, if a step “Validate input” is in the diagram, the code indeed has input validation logic, etc. The tests listed also guide what conditions the code must handle.

- *Review Template (for completeness):* In V1, this might not be heavily used because we’re not auto-generating a diagram from code. But if we had one:

```
[System:] You are now in code review mode.
[Instruction:] Compare the final code with the design. Generate a list of
any discrepancies.
Design:
"""mermaid
```

~~~
{diagram_text}
"""
Code:
``` (code could be inserted here, though likely too large – maybe a
summary)
~~~

If any part of the code does not match the design or tests, highlight it. Otherwise, confirm that the implementation matches the plan exactly. ```

However, since automatic review is out of scope, we might not implement this prompt. Instead, the "Accept Code" step is simply user-driven.

All prompt templates will be stored as separate files or constants for easy maintenance. They will undergo iteration and tuning, as prompt quality is crucial for the system's success.

Finally, to enforce the protocol, the system might prepend a high-level system message to all conversations with the AI that includes core rules (like those in the context primer: no unsolicited changes, require explicit user approval to move mode, etc. [53] [50]). This acts as a guardrail around all stage-specific prompts.

In summary, the interface and prompt design ensures that: - The **routines are configurable** and understandable (making future extension feasible). - The **frontend-backend contract** covers all user interactions clearly (ensuring smooth UX). - The **prompts are crafted to tightly align the AI's output** with the blueprint/test spec, reinforcing the DAAS approach and the principle of developer control.

Frontend Design

The Wizard's frontend is responsible for delivering a fluid, guided experience through the multi-stage workflow, all within the developer's IDE environment. Key aspects of the frontend design include its overall information architecture, the specialized UI for certain stages (especially the diagram editing), and accessibility considerations.

Information Architecture & Layout

The UI follows a **wizard-style, step-by-step interface** embedded in the IDE. The general layout consists of:

- A **main interaction panel** (which could be the existing chat panel in Cursor IDE, or a dedicated sidebar/pane) where the conversation and outputs appear.
- A **stage indicator/progress bar** at the top or side, showing the current stage (Read, Plan, Execute, etc.) and possibly the upcoming stages [54] . Completed stages might be marked with a check, the current stage highlighted, and future stages greyed out. This gives users a mental model of where they are in the process at all times.
- The main panel content changes depending on the stage:
- For text-centric stages (Read, Insight, parts of Execute), it appears as a chat-like interface with AI messages and user messages.
- For the Plan stage, a special **dual-pane diagram editor** appears (likely as a modal overlay to give more space) [15] .

- For the Execute tests stage, an **editable table** or list of test cases is shown.
- For the Execute code stage, a code output view (which might just be the normal chat with code blocks, or a slightly enhanced code viewer).
- For the Review stage, possibly a side-by-side view of diagram and code, or a checklist UI for verification.

- **Controls and Navigation**: Each stage view has appropriate controls:

- A **"Next" or "Confirm" button** when user input/approval is required to advance [55] .
- A **"Back" or "Edit" option** if going backward is allowed (e.g. in Review stage, a "Revise Plan" button to jump back).
- In Plan stage, tools to either edit text or perhaps shape tools (though initially just text edit).
- In text/chat stages, a standard text input box for the user to type questions or feedback to the AI.

- The UI is largely **embedded**, meaning it doesn't take the user out of their coding context. For instance, the Plan stage editor might be a pop-up modal on top of the IDE, rather than a separate window, so that the user can still see the code behind if needed. Once they confirm, it goes away and returns to the main panel.

Stage-Specific UI Behaviors: - **Research/Read & Insight Stages**: These early stages are mostly conversational. The AI might display some findings or ask a question, and the user can respond via the chat input. The UI might show a summary of context in a subtle way (for example, if the AI identified key points from the user's problem, it might bullet them). Largely, these look like a typical chat with Q&A format, possibly with an icon or label indicating "Research Mode" or "Insight Mode".

- **Plan Stage (Diagram Editor)**: This is the most UI-intensive part:
- Implemented as a **dual-pane editor** – left side has a text editor with the Mermaid source, right side shows a live-rendered diagram preview [38] . The preview updates as the text changes (with a short debounce to avoid overly rapid re-render).
- The editor should have syntax highlighting for Mermaid, and possibly line numbers. If the Mermaid syntax has an error, we could highlight it or show an error message (helping the user fix any mistakes).
- There may be **two diagram tabs/views** if we generate both a sequence diagram and a class diagram (as per FR-2.1) [52] . For example, the modal might have tabs: "Flowchart View" and "Structure View". The AI might generate both diagrams, and the user can toggle between them to cross-verify consistency. This is a stretch goal; if only one diagram is present, we will just show that.
- There is a **natural language feedback box** in this modal as well (if screen space permits) [52] . This would allow the user to type something like "Looks good, but add a step to validate input before processing" – upon pressing Enter, this feedback is sent to the AI, which could then update the diagram accordingly (the AI's updated diagram would replace the text in the editor). This complements direct editing: users less comfortable with Mermaid can describe changes in NL and let AI do it.
- A prominent **"Confirm Blueprint"** button is available (likely bottom-right of the modal) [16] . It remains disabled until at least one diagram is present and possibly until the user made some acknowledgment. When clicked, the modal closes and the workflow proceeds.

For accessibility and usability, the diagram editor modal should be resizable or large enough by default since diagrams benefit from space. We will also allow copy-pasting the diagram text in case the user wants to save it.

- **Execute Stage (Tests)**: After confirming the diagram, the UI transitions to a test case view. This might be integrated in the chat or as another modal. Likely, a simple approach:
- Present a list or table of test cases. For instance, a two-column table: "Test Case Description" and perhaps "Expected Outcome".
- Each test case might be editable inline (user can click and edit the text).
- The UI might allow adding or deleting test cases (a small "+ Add test" button and a trash icon for removal).
- A **"Approve Tests & Generate Code"** button is shown above or below the test list [17]. The user clicks this when done reviewing.
- If the user makes changes, those should reflect in what we send to the backend (so we collect the edited text).

Optionally, if this is too heavy, we might simplify by having the AI's tests appear in the chat as a message and then ask user "Do you approve? (Yes/No)". But the PRD explicitly calls for an editable table UI, so we aim for that.

- **Execute Stage (Code)**: Once tests are approved, the AI will stream/generate code. The UI here likely reverts to a chat-like view where the AI's message contains code blocks. If the code is long, we might put each file's code in collapsible sections or provide a "copy to clipboard" for each block. The user can scroll through the code. Since this is a critical moment, we might highlight portions that correspond to parts of the diagram (though doing that automatically is advanced; more likely, the user will manually cross-ref with the open diagram if needed).
- If the code is generated as part of the chat, the normal chat input might be disabled here because we don't expect the user to type arbitrary messages while code is being output. Instead, they should move to the review/accept stage.
- We may show a small note like "Code generated based on the blueprint above. Please review."

- If the integration allows, perhaps an "Open in Editor" button to send this code into the actual file (but that's more of an integration task possibly out-of-scope due to no auto-write).

- **Review Stage (Verification)**: In V1, since automated diff is not there, the UI for review could be minimal:

- Possibly reopen the diagram editor in a read-only mode alongside the code for side-by-side view. For example, a split view: left side shows the final blueprint diagram (rendered, with maybe the text beneath), right side shows the code (maybe one file at a time or summary).
- Or, simpler: present a checklist of what to verify (like: "All steps in the blueprint are implemented in code", "All tests pass (if you run them)", etc.), and then an **"Accept Code"** button.
- When the user clicks Accept Code, perhaps we show a success message "Workflow complete!" and close the Wizard panel.

In case the user is not happy, we should allow a way to go back. Perhaps a **"Revise Blueprint"** button that jumps back to the Plan stage with the previous diagram loaded (or maybe they'll have to re-describe their issue to the AI). We have to manage state carefully for that scenario (the code and tests might become stale once we go back, but that's fine since we'll regenerate them after the diagram change).

- **General UI Behavior**: The **natural language input box** (chat box) may not always be active:
- During Plan stage when the diagram editor is open, we might disable or hide the main chat input to focus interaction through the diagram modal and its own feedback field. This avoids confusion (the user shouldn't be sending general chat messages while the blueprint modal is up) – they should either edit directly or use the modal's feedback input. Thus, we **disable the regular NL input** during certain phases to enforce the structured flow.
- Similarly, when test cases are shown awaiting approval, we likely disable the chat input (to prevent the user from skipping ahead with a random query or instruction). They should either approve tests or edit them. Only after that, the next stage (code) will proceed.
- We re-enable the chat input in review stage if the user wants to discuss issues with the AI. But if they do, that essentially means going back in the workflow. Perhaps if user types "The code missed a step", the system could interpret it as wanting to refine – ideally we funnel that into an action like "Alright, let's go back to Plan and address that." In V1, it might be simpler to require pressing the explicit back button rather than free chat in review.

These design choices align with the first principle that *the developer is in control but within a guided structure* – by limiting what they can do at a given time (for example, can't ask the AI to do something unrelated in the middle of the guided flow), we maintain focus and prevent the conversation from derailing the workflow.

- **Styling and Theming**: The Wizard UI should blend with the IDE's look as much as possible. Likely we'll adhere to Cursor's theme (dark mode, etc.). Diagrams will use default Mermaid theming (which has a dark theme variant if needed). We must ensure good contrast for text and diagram lines (especially in dark mode).

Accessibility (a11y)

We are committed to making The Wizard UI accessible:

- **Keyboard Navigation**: All interactive controls (buttons like Confirm, Approve, Accept, as well as text fields in the test table) will be reachable via keyboard. We will implement a logical tab order through modal content. For instance, in the diagram editor modal, tabbing might go: diagram text area → maybe NL feedback field → Confirm button. Hitting Enter in text areas should not accidentally trigger confirm unless explicitly intended. Also, keyboard shortcuts like `Esc` to close modals, or arrow keys to navigate test list rows, will be supported where intuitive.

- **Screen Reader Support**: We will use proper semantic HTML or ARIA roles. For example:

- The stage indicator can be marked as an `ARIA list` of steps, with the current item indicated via `aria-current`.
- Modals will have `role="dialog"` and an `aria-labelledby` pointing to the modal title (like "Plan Stage – Edit Blueprint").
- The Mermaid diagram, being visual, is tricky for screen readers. However, since the diagram has a text source, that text is actually meaningful. We will ensure the text editor portion is accessible (it's

just text, so screen readers can read the Mermaid description which is at least somewhat descriptive of the flow). Additionally, we might provide a hidden textual summary of the diagram if possible (e.g., the AI could provide a one-sentence summary like "Diagram showing A calls B then B calls C", which we can put in alt text).
- For code outputs, we'll use code block elements which screen readers can handle (though reading code line by line is tough, but developers with screen readers often use IDE-specific tools, we'll do our best with labeling).
- Icons (like any edit/delete icon for tests) will have `aria-label` attributes (e.g. `<button aria-label="Delete this test case">🗑</button>`).

- **Color and Contrast**: We will ensure any color used in diagrams or UI has sufficient contrast. Mermaid's default shapes (e.g., nodes are usually light with dark text) should be okay, but if not, we'll override styles for a11y. Also, any status indicators (like green vs red) will not rely solely on color – e.g. an error message will have an icon or text, not just a red border.

- **Font Size and Zoom**: The modal editors should be resizable or at least not fixed tiny sizes. Users should be able to zoom (the browser/IDE zoom will naturally enlarge our UI as well). We'll use responsive design so that if the panel is larger or smaller, content adjusts (for example, the diagram preview might stack under the text editor on a narrow screen, or side-by-side on wide).

- **No Time Limits**: We won't impose timeouts that a user can't control. The user can take as long as needed on each stage. If we have any session expiration (maybe after hours), we'll warn the user. But likely not needed as one would finish or cancel explicitly.

- **Accessible Feedback**: When new content appears (like a stage result), we may need to ensure it's announced. For screen readers, we might set `aria-live="polite"` on the chat content area so that new messages (diagram ready, etc.) are read out. Similarly, when switching stages, it should announce "Plan stage" etc., perhaps through updating the document title or an ARIA alert.

By following these practices, we aim for The Wizard to be usable by developers of varying needs, aligning also with the principle of minimizing cognitive load – a well-structured, accessible interface benefits everyone.

Data & Persistence

Handling data consistently across the multi-step workflow is crucial. While The Wizard V1 doesn't require heavy long-term persistence (we are not yet saving project artifacts automatically), it does maintain a variety of data during a session. We also want a record for possible debugging or audit (especially of decisions made at each step).

Session State Management: The backend will maintain a **session context object** for each active Wizard session. This context includes:

- **User Input Context**: The initial query or task description from the user, plus any relevant project context we pulled in (for example, if the user invoked The Wizard with some code selection or open file, we might store that context so the AI can refer to it in Read stage).

- **Stage Outputs**: For each stage, we store what the AI produced and any subsequent user modifications:

- *Read/Insight:* store the AI's analysis or the chosen mode.
- *Plan:* store the initial AI-generated Mermaid diagram text, and the final user-confirmed Mermaid text (these could differ if the user edited). We might also keep incremental versions if the user iterated (e.g., AI made diagram v1, user said "change X", AI made v2, etc.) – possibly store those in a list or at least the last one before user edits.
- *Execute Tests:* store the list of test cases proposed, and the final list after user edits (if any). Even if the user didn't edit, mark them as confirmed.
- *Execute Code:* store the code output. This might be large, so we can either store references (like file pointers if code is written to a temp file) or just keep it in memory as a string. At least until the session ends, we keep it to possibly enable review comparisons. (We may not persist code to disk automatically due to safety, but we keep it in memory.)

- *Review:* store the outcome (did user accept? did they go back for revision?).

- **Metadata**: timestamps for each stage transition (when did AI finish, when did user confirm) – useful for logs and potential future performance tuning.

- **Versioning**: Specifically for **diagram versioning**, as required, we can implement a simple version history. Each time a new diagram is generated or the user edits it, we create a new version entry in an array (with timestamp and maybe who (AI or user) made it). This could be used to implement an "undo" or just for record. In V1, we might not expose multiple undo levels, but we do want to at least keep the original AI proposal and the user's final version [19]. If the user re-enters Plan after a Review rejection, that effectively starts a new version thread (we might then start version 2 of the diagram).

- **Storage Mechanism**: Likely, this session state can live in memory (as part of a Python object or JS object in Node backend). If the Wizard backend runs inside the IDE, memory persistence is fine. If we wanted to persist across sessions (not required in V1), we could serialize it to a JSON file in a `.wizard` folder in the project or a database. V1 probably doesn't need disk persistence beyond logging events (i.e., when the user closes the IDE, the session is gone).

- **Concurrency**: It's expected a developer might run one Wizard session at a time (or a few). Each gets its own context. The design doesn't require shared data between sessions, except some caches possibly. But we can treat them separately.

Diagram as Source of Truth: Once confirmed, the **diagram text is stored as the authoritative spec**. We treat it as read-only spec for Execute stage and beyond [5]. If the user goes back to Plan, we unlock it again (and presumably mark code as stale). We ensure only the latest confirmed diagram is used for generation, to avoid any mix-ups.

Test Cases Storage: The tests, once approved, are similarly stored as part of the spec for code generation. Interestingly, these test cases *could themselves be considered an artifact worth saving*. In a future version, we might output them as actual test code files. In V1, we don't automatically write them out; however, we may keep them such that the user could copy them if desired. If the AI provided them in plain language, the user might then ask the AI (or manually) to translate into actual unit test code. But that's outside the main workflow.

If we were to persist any of these beyond the session, likely candidates are: - The final blueprint diagram (maybe stored in a file like `Design_123.mmd` or as an attachment in the IDE) – but not doing in V1 automatically. - The test cases (perhaps in a Markdown in a `.wizard` or `.tasks` folder). The design doc hints at maybe storing task files, but that's more related to the R-I-P-E-R agent concept outside our UI. V1 probably won't drop files in the project unless user does so manually.

Event Logs: In addition to session state, the backend will produce log entries for observability (discussed later). These logs might also double as a lightweight persistent record of what happened (though not intended to reconstruct full state). For example, a log line could be "2025-10-19T12:05: User confirmed blueprint (version 3) with 1 edit". That plus storing the diagram text in memory suffices.

No Auto-Save of Code: A key compliance decision is that **we do not automatically write the generated code to the user's filesystem** (or modify files) without explicit user action [40]. So, while we hold the code in memory to display and possibly for diffing, we won't, for instance, open the actual file and insert it. The user must manually take that code and put it into their codebase (or maybe use an IDE action like "apply suggestion" if available). This avoids any unwanted overwriting of their existing code and aligns with the principle of developer control.

If user does Accept Code: We might integrate with the IDE by, say, inserting the code into the editor at the appropriate location if it's clear. But since Accept Code integration is not in V1, Accept Code likely just ends the session. We might give a prompt like "Code accepted. Please integrate the above code into your project as needed." For future or for the user's convenience, we could consider generating a patch or diff that they can apply manually – not in V1 scope, but storing the code output means we could generate a unified diff against existing files if we had that context. For now, we simply keep the code available on screen or in the session state for the user to copy.

Versioning & Rollback: If user goes backward (e.g., after seeing code, goes back to change the diagram), how do we treat the intermediate data? - The system should invalidate or discard the now-obsolete test cases and code, since they may no longer align with the new blueprint. We might keep them in memory for reference (maybe the user changed their mind and wants to see what the previous code was – but that's probably not needed). We likely start a new Execute phase fresh. Possibly, for safety, we create a new session context or branch. - Version the blueprint such that blueprint v1 had code v1. If blueprint v2 is made, we could keep code v1 around only in logs. - The UI will be reset to the Plan stage UI when going back, and the Execute outputs cleared or archived.

Persistent Settings: Some user preferences might be stored if relevant: - For example, if the user resizes the diagram modal or chooses a different layout, we might remember that in local storage. - If the user decides to skip certain steps in future (maybe not in V1, but if they always want to skip Research stage, a setting could control that). - This is minor and not core to design, just noted.

Data Privacy: If any of the data is sensitive (user's code, etc.), we ensure it's not logged externally. The context we send to the AI obviously leaves the system to an LLM service, but that is inherent. We won't store user's proprietary code anywhere permanent. The ephemeral session memory is cleared after session. Logs might contain minimal references (like "function names"), we should avoid logging full code or prompts to be safe.

In summary, data in The Wizard V1 is primarily managed in-memory through the workflow, with careful versioning for the blueprint spec and tests to ensure consistency. Persistence is minimal and just enough to support the user's session and measure outcomes, aligning with V1's focus on a single-session guided flow rather than long-term artifact management.

Security & Compliance

The Wizard V1 must uphold strong security and compliance practices, especially as it operates on potentially sensitive codebases and uses AI generation. Below we outline key security measures and how compliance requirements are met:

- **Developer-in-Control (No Autonomous Writes)**: As stated, the system **never modifies the user's code without approval** [56] . In fact, in V1 it doesn't modify files at all on its own. All code that's generated is presented to the user for review. If the user wants to apply it, they will copy it or use an editor action. This prevents any accidental overwriting of critical code or insertion of malicious code. The "Accept Code" action is the final gate, and even that in V1 just ends the flow without auto-applying changes [40] .

- **Scoped File Access**: If the AI needs to read parts of the codebase (for context in Read stage perhaps), we will scope what it can access. Likely it will only get to see open files or a summary of relevant files instead of entire repository dumps. This is both to protect IP (not sending too much to the model) and to reduce risk of the AI messing with something unrelated. Moreover, since no write access is given, the AI could at most *read* context. We will ensure that any file reading operations by the AI (if using an agent that can use tools to read files) are sandboxed to the workspace and possibly read-only mode.

- **Network Communication**: The front-end and backend likely run on the same machine (if integrated in IDE) or within a trusted environment. We'll use secure channels (in a web context, `wss` and `https`). If the backend is local, it might not even expose a network port externally. We'll ensure that an attacker in a different project cannot easily intercept another developer's Wizard session, etc. Standard web security applies (CSRF not a big issue if only local, but we can include tokens if needed).

- **LLM Privacy**: If using a cloud LLM, we must be mindful of sending proprietary code to it. This is more a policy decision – ideally we'd use an on-prem or local model for high security. In V1, it might use OpenAI or similar, which means code and design details do leave the local environment to the model provider. Users should be informed of this if applicable. If required by compliance, The Wizard could restrict how much code context to send or allow an opt-out. However, addressing that deeply might be beyond our immediate scope except to document it.

- **No Execution without Consent**: The Wizard itself doesn't execute code in V1 (we considered maybe running tests, but likely not implemented). If in some scenario we integrate running the generated tests, we must ensure it's done safely (in an isolated environment) because executing AI-generated code can be risky. V1 probably avoids this by not automatically running anything. If a future iteration did run tests, we'd do it in a sandbox and not allow destructive operations.

- **Input Validation**: We have to be cautious of prompt injection or malicious inputs. Since the user can edit the Mermaid diagram or test text, they could theoretically input something that might break our parser or cause the LLM to behave unexpectedly (though likely just text). We should sanitize inputs we send to the model – e.g., ensure the Mermaid text is within reasonable length, escape any backticks in it when putting in prompt to not confuse the model's markdown context. Also, guard against any HTML/JS injection in the web UI (Mermaid preview should be safe as it's rendered by Mermaid lib, but we'll keep libraries updated and maybe run Mermaid in a sandbox iframe if concerned).

- **Authentication & Access**: If The Wizard is integrated with an online service, ensure only authorized users can start sessions (in an IDE context, the user is already authenticated to their environment). If we had a multi-user server, we'd need auth tokens on the API. V1 likely runs in a single-user context (the developer's IDE), so not a big issue.

- **Logging and Privacy**: Logs will avoid sensitive data. We will log events like "Blueprint confirmed" but not log the actual blueprint content (or if we do, perhaps optional debug mode). If logs are stored, treat them as internal and protect them as you would any app log (which may contain some info).

- **Third-party Compliance**: The Wizard uses Mermaid (open source) – ensure we comply with its license (Mermaid is MIT license, okay to embed). If we use any libraries (React, etc.), standard compliance. Also ensure the approach aligns with company policies (like not introducing GPL code unless allowed, etc.).

- **GDPR/PII**: Unlikely relevant, but if a user's code or spec had personal data, we'd treat it carefully. The AI might output code or text that includes such data only if it was in input. The Wizard doesn't store anything long-term, so minimal risk. Just ensure any telemetry is anonymized.

- **Security Testing**: We will test that:

- The user cannot break out of the intended workflow via UI (like clicking disabled buttons via HTML hacks, etc.).
- Large inputs (like an extremely large diagram text) won't crash the system (maybe we set some size limits).
- The AI's output is rendered safely – e.g., if the AI by mistake output some HTML, our UI should treat it as text (which by default if we only expect code and we sanitize, we should be fine).

- No sensitive tokens (like API keys for LLM) are ever exposed in the front-end; they remain on backend.

- **Compliance with Host IDE**: Cursor IDE likely has guidelines for extensions or internal tools. We ensure our integration (e.g., launching a webview) meets those guidelines and doesn't degrade the IDE's performance or violate any sandbox rules.

In summary, V1's design prioritizes safety by **limiting automated actions** and keeping the developer in the loop. By not writing files or executing code without user initiation, we eliminate a large class of potential issues. The architecture's modular nature also means any future capabilities (like multi-agent or code execution) can be sandboxed within the Tools module carefully when added. For now, V1 stays on the conservative side, focusing on providing guidance and suggestions while leaving final decisions and actions to the developer.

Observability

To ensure we can monitor The Wizard's usage and troubleshoot issues, we include basic observability features in V1, primarily through event logging. The goal is to gather enough data to understand how the workflow is performing (especially the **completion rate**, our success metric) [57] , without overly burdening the system or violating user trust by logging sensitive details.

Logging Strategy: - We will instrument the backend to log **key events** in the workflow. Each log entry will include a timestamp, a session identifier, the event type, and relevant metadata. - Important events to log: - *Session start*: user initiated the Wizard, and which routine they chose (e.g., "Session 123 start – routine=RIPER-5, userPrompt='Implement feature X'"). - *Stage completion*: when each stage's AI output is delivered and when the user approves it. For example, "Session 123: Plan stage output delivered (diagram length 10 lines)" and then "Session 123: Plan stage confirmed by user". We may also log if the user requested a redo at any stage (like "Session 123: Plan stage redo requested"). - *Transitions*: such as "Transitioned to Execute Tests stage", "Entered Review stage". Some of these overlap with stage completion/confirmation events. - *Session end*: either "Session 123 completed – code accepted" or "Session 123 terminated – user cancelled at Plan stage" or "Session 123 failed – error at Execute stage". - *Errors*: any error or exception. E.g., "ERROR: Failed to get response from AI at Plan stage, error=XYZ". Also if the AI returns something like it cannot fulfill the request, we log that as a handled event ("AI could not complete request at stage X").

- The logs will be structured (JSON lines or something) to allow easy analysis. Since V1 might not have a full dashboard, we might just log to console or a file. But we intend these to be machine-parseable so we can later aggregate metrics like: what percentage of sessions reach Accept Code, how long each stage takes on average, etc.

- **Core Metric – Completion Rate**: We specifically will measure "happy path completion rate" [57] , defined as (sessions with Accept Code pressed) / (sessions started, where spec-first was chosen). We will compute this offline by analyzing logs. For example, by counting "Session completed" vs "Session started".

- **Latency & Performance Metrics**: We can also log timestamps for each stage to later assess how long the AI is taking, and where potential delays are. This can be done by including in the stage completion logs the time elapsed since last stage or since start. E.g., "Plan stage completed in 15s", "Execute code generation completed in 30s". This isn't a direct requirement, but it's useful to gauge user experience.

- **Limited Content Logging**: We deliberately avoid logging the full content of user's code or diagrams to reduce noise and privacy concerns. We might log summary info like number of nodes in the diagram, number of lines of code generated, or certain identifiers (like we could log the titles of steps in the diagram if needed for debugging, but it might not be necessary). If a failure occurs due to content (like Mermaid parse error), we might log the problematic snippet or at least note "Mermaid syntax error at line 5 in diagram". But generally, keep content out of logs unless necessary for troubleshooting.

- **User Feedback Logging**: If a user explicitly uses a feedback mechanism (e.g. a "I don't like this suggestion" button if we had one), log that too. In V1, the main feedback is implicit (they either proceed or redo). We capture redo actions as negative feedback implicitly.

Monitoring: - Since V1 likely runs in a local context, we won't have a cloud monitoring dashboard. But during development and internal testing, developers can tail the log to see what's happening. - If integrated in an IDE extension, perhaps logs go to the IDE's output console for the extension (so the user/dev can see it if needed). - For a production or team setting, we could output logs to a central place (or the IDE could report telemetry back). This depends on product decisions about data collection. Given the PRD's emphasis on proving the concept, we likely want to gather some telemetry (with user consent if needed) about usage.

Debug Mode: - Possibly include a verbose or debug mode (maybe triggered by a setting or environment variable) that logs more info (like full prompts and outputs) to help developers of The Wizard troubleshoot. Not for end users typically. This would be invaluable during development to see what prompt was sent and what came back to diagnose any prompt issues or logic bugs. We just need to ensure this is off in normal usage or sanitized if on.

User Visibility: - We might incorporate some minimal UI indicator for logging/telemetry, but likely not. However, if something goes wrong, we should show the user an error (like "The Wizard encountered an error. Check logs for details."). If they are curious, having a way to see a log or report could be nice (maybe not in V1, except telling them to check a console).

Analytics and Improvement: - Using the logs, after some usage, we can answer questions like: - Where do users most often drop out? (e.g., many may start but not confirm blueprint, indicating maybe blueprint wasn't useful or too complex). - How often do users require multiple AI attempts at Plan stage or others? - Average time between stages (which might indicate if the user spent long editing the diagram – showing engagement). - Did any error conditions happen frequently?

- This observational data will guide future improvements (maybe tweak prompt if many fail at test generation, etc.).

Operational Alerts: - If we had many users or long-running service, we might set up alerts (e.g., if error frequency spikes). In V1 with likely single-user dev usage, not applicable.

Minimal overhead: - Logging will be kept lightweight to not slow down the workflow. We mostly log textual events asynchronously (writing to file should be non-blocking or quick). We will avoid any heavy synchronous logging that could delay UI updates.

In sum, observability in V1 is about capturing the lifecycle of the workflow in logs to verify that it's working as intended and to collect evidence of success. It's deliberately minimal – just enough to get insights – as a full telemetry system or UI is out of scope [23] .

Performance & Capacity

While performance optimization is not the top focus of V1, we still need to ensure the system performs reasonably for users and can handle expected usage patterns. Below we address the main performance considerations and how the design handles them:

- **Target Environment**: The Wizard V1 runs in a developer's IDE (likely on a standard dev machine) and possibly calls out to an external LLM service. We assume a modern computer with a capable browser environment for the UI. The typical usage is interactive (one user stepping through a workflow). We do not anticipate heavy concurrent usage (like many users or many parallel sessions) in V1. Thus, our performance focus is on *interaction latency* and *UI responsiveness* for that one user.

- **LLM Call Latency**: The largest delays will come from the AI model's responses. Diagram generation, test generation, and code generation can each take several seconds (depending on model and complexity). For instance, GPT-4 might take 5-15 seconds to produce a diagram for a non-trivial problem, and code generation could be 10+ seconds. We can't fully control this, but we mitigate user impatience by:

- Providing visual **loading indicators** and status messages (so the user knows it's working and hasn't frozen).
- Possibly using streaming: if the model supports streaming tokens, we could display partial output for code as it arrives (e.g., show code incrementally). For diagrams, streaming is less useful since the whole Mermaid text is needed to render; we might wait for completion before showing it to avoid flicker from partial diagram.
- The user is also actively involved between these calls (editing diagram, reviewing tests), which naturally introduces some wait time that they control, making the AI wait rather than the reverse. So the perceived flow might not feel too slow if the user is busy thinking/ editing during those steps.

- If using an API with rate limits, ensure we queue requests properly (shouldn't spam multiple calls at once; only one stage at a time typically).

- **Browser UI Performance**:

- **Diagram Rendering**: Mermaid can render quite complex diagrams. For typical use (maybe up to, say, 20-30 nodes/steps), it's fine. If a diagram gets extremely large, rendering could slow down or the preview might lag. In such cases, our approach:
  - We throttle re-renders – e.g., only update preview when the user stops typing for 500ms.
  - If a diagram is truly huge (which likely indicates a very complex task – might not be suitable for one session anyway), the user could still edit but the preview might be slow or we provide a note "Diagram is large, live preview disabled; click Refresh to update." This is a degradation strategy to maintain UI responsiveness.

- **Dual Pane Editor**: We should ensure editing the text doesn't freeze the browser. Mermaid parse is the heavy part. Maybe do it in a web worker if needed (Mermaid may not support that natively, but we could attempt).
- **Code Display**: Large code outputs (hundreds of lines) might also be heavy in a chat. We should use a virtualized list if output is extremely long or allow collapse. Syntax highlighting for large code can also be slow. We might use a efficient highlighter or none at all for very big blobs. If needed, break code into multiple blocks (like per file) to scope highlight work.

- **Memory usage**: The biggest memory consumer could be the code string and diagram. But a few hundred KB of text is trivial for modern systems. The front-end should be fine unless we accidentally keep accumulating stale data. We will ensure to discard or reuse components rather than keeping every intermediate result mounted.

- **Backend Performance**:

- The backend does orchestration – minimal CPU work (some JSON handling, maybe some light parsing or diff). Memory footprint per session is low (some strings, state objects).
- It will wait on I/O (AI calls).
- If a model is self-hosted (not likely in V1), then performance would depend on that model's infra.

- Concurrency: If somehow multiple sessions or multiple user requests hit at once, the engine can queue or process them sequentially because one user likely doesn't run two parallel flows. Even if two flows started, they are independent (just requiring separate state and perhaps careful threading of AI calls if multi-threading is used). We can allow concurrency but it's not a primary goal to optimize for it now.

- **Capacity Planning**: Since V1 is an MVP, we assume:

- **One session at a time** per user.
- Possibly the user can do multiple flows in a day, but that's sequential.

- The LLM usage might be rate-limited by API (like maybe 50 calls per hour or cost limited). The design tries to minimize calls: with five stages, if each calls the model once, that's five calls per session (maybe a couple more if user requests rework at Plan or if test and code are separate calls). That's acceptable usage. If the user uses it heavily, the main limit is API cost; our system doesn't impose further limits aside from perhaps some built-in cooldown if needed to avoid flooding (shouldn't happen with sequential flow).

- **Client-Side Degradation**:

- If the user's browser is slow or they open on a less powerful laptop, the heavy part (Mermaid and React rendering) is client-side. We ensure to test on moderately powered machines. If we find performance issues, as mentioned, we can disable live preview or heavy features.

- Also, we avoid memory leaks – make sure modals unmount properly, etc., so that after a session the memory is freed (especially important if the user does multiple sessions or keeps Wizard open; we don't want orphan DOM nodes from previous diagrams lingering).

- **Progress Feedback**:

- A performance principle is to always give feedback. So even if something takes time, the user sees a spinner or progress bar. For instance, when generating code, we might show a progress bar if we know how many tokens or a generic spinner with messages like "Generating code, this may take up to a minute for complex tasks…".
- If test generation is quick but code is slow, we might consider running them as a single combined step to avoid an extra round trip – but separating them is a feature. The user will likely find code gen the slowest step. As long as we set expectations ("running AI to produce code…"), users can wait a bit.

- **Browser Compatibility**: We assume a modern Chromium-based environment (Cursor IDE is likely Electron/Chromium). We won't optimize for old IE or anything. But we ensure it works in Chrome/Edge and ideally in Firefox if applicable.

- **Scalability**: If we did need to handle multiple users (like if this was deployed on a server), the architecture can scale by having multiple backend instances because sessions are independent. The stateless API helps in that we could have load balancing, though session affinity might be needed if state is in memory. But since this is likely local, not a concern now.

- **Resource Cleanup**: After a session finishes, ensure to clean up (close any AI streaming connections, free session objects after some time). This prevents buildup if someone does 100 sessions in a row.

In short, for V1, as long as we use efficient rendering techniques and handle the large content carefully, performance should be acceptable. The biggest factor is the LLM's speed, which we manage via UI feedback. Should performance issues arise (like slow diagram rendering), we have fallback options (disable auto-preview, etc.) to ensure the tool remains usable even under less-than-ideal conditions.

Risks & Tradeoffs

While The Wizard V1 brings a novel guided approach to AI-assisted development, there are several risks and trade-offs inherent in the chosen scope and design. We outline them below along with rationale:

- **No Automatic Consistency Checking between Code and Diagram**: In V1, once the code is generated, the system does not automatically verify that the code fully matches the diagram (i.e., no implemented reverse-DAAS to cross-check) [25]. This is a risk because if the AI accidentally strays from the blueprint or misses implementing a part of it, there is no programmatic way to catch it. We rely on the user's careful review to catch discrepancies. Trade-off: We chose to omit automated review to keep V1 manageable; implementing a robust code-to-diagram generator is complex and time-consuming. The risk is mitigated by the strong prompts and the test generation step – if tests are well-chosen, any major deviation should cause a test to fail, which the user would notice (assuming they run the tests). Still, a subtle blueprint deviation might slip by. We communicate to users that final validation is their responsibility in V1.

- **No "Accept Code" Integration with IDE/VC**: The Accept Code step in V1 is not integrated to do anything beyond mark completion [40]. This means after going through the whole workflow, the user still has to manually transfer the code into their project (copy-paste or apply changes themselves). Trade-off: This was done to prioritize safety and due to technical complexity of auto-inserting code correctly. The downside is a hit to convenience – some users might ask "why can't it just put the code in for me?". It also means the value of pressing Accept Code is mostly psychological/logging. In future, we'd likely integrate this with, for example, automatically opening a diff that the user can accept. For now, we accept the slight user effort required as a trade-off for not potentially messing up files or needing complex merge logic.

- **No Reverse DAAS (Diagram from Code) in V1**: As mentioned, the refine stage's ideal feature – generating a diagram from the AI's code and comparing it to the plan – is not implemented. This was a conscious trade-off to reduce complexity, but it sacrifices one of the checks that would have truly "closed the loop" [25] . The risk is that our story of full alignment is incomplete; if stakeholders expect that final comparison, we have to clarify it's planned but not ready. It also means we don't fully leverage the potential of diagrams for verification, leaving some potential errors to go unnoticed by less diligent users. We mitigate by planning this in the roadmap and ensuring the architecture could accommodate it later (the data flows and UI design take it into account, e.g., side-by-side view considered in design, even if manual).

- **Web UI Only (No Desktop/Offline)**: V1's reliance on a web interface (likely requiring an internet connection for the AI and a browser environment) means it cannot be used in fully air-gapped environments or those where a developer might prefer a native experience [26] . Trade-off: We prioritized rapid development via web tech and decided not to implement a parallel Qt desktop UI or deep VSCode plugin integration at this stage. The risk is some users (especially those who expected it inside their primary editor without a web view) might be hesitant or might experience slightly less seamless workflow. Also, performance in a browser might be a bit slower than a native component. We consider this acceptable for V1, focusing on proving the concept. The architecture notes plan for local/IDE integration later, which can alleviate stakeholder concerns if they know it's on the horizon [26] .

- **User Learning Curve and Process Compliance**: The Wizard enforces a strict multi-step process (which is unusual for those used to one-shot AI solutions). There's a risk that some users will find it cumbersome or might try to "fight" the workflow (e.g., attempt to skip steps, or ignore the blueprint and ask for changes after code is generated). The system attempts to guide them properly, but user experience risk is that if it feels too rigid, some might abandon it. Trade-off: We optimize for the skeptical senior developers who actually value the structure; they are more likely to appreciate it. We may alienate those who wanted quick wins. This is a conscious positioning decision more than a flaw, but it's a risk if our target user isn't as receptive as assumed. We mitigate by making the UI as smooth and intuitive as possible, and by clearly communicating the benefits of each step (maybe through onboarding or tooltips).

- **Quality of AI Output**: The system's performance hinges on the AI quality. If the AI generates a poor diagram (missing elements) or incorrect tests, the whole flow could collapse (garbage in, garbage out). We have the user in the loop to catch this, but it's possible that the AI's suggestions might be consistently subpar for certain complex tasks, leading to frustration. This isn't a design flaw per se, but a risk to user trust in the system. We address it by prompt tuning and possibly model selection (maybe using a model known for better adherence to instructions). Also, the iterative nature means if the first diagram is bad, the user/AI can refine it, which is better than one-shot code that's bad – but it does cost more time.

- **Handling Large/Complex Projects**: V1 might struggle with very large contexts. For example, if a user tries to use it on a massive feature that involves many components, the diagrams and code could be huge. The AI context window might be a limiting factor – we can't feed entire large codebase details in. We expect V1 to be used for contained features. This is a trade-off: we focus on relatively self-contained tasks (maybe a few modules). Using The Wizard on something that spans dozens of files or requires full knowledge of a huge system could hit limitations (either the AI forgets context or the diagram becomes unwieldy). We mitigate by possibly encouraging users (through docs) to use it on one feature at a time and by eventually improving context handling in future (e.g., pulling high-level summary from code instead of raw code).

- **No Multi-Agent Specialization**: All responsibilities (architecture suggestion, coding, testing ideas) are on one AI agent. The design docs foresee value in having specialized agents (Architect, Coder, Tester) collaborating [42]. Not having that in V1 is simpler but could be seen as a missed opportunity for quality: a single model might not be equally good at test generation and coding, for instance. If those tasks conflict (some models might not thoroughly test their own code), the output could suffer. We accept this risk in V1 because multi-agent orchestration is complex and unproven. If we see particular weakness (e.g., tests always trivial), we might consider using a different prompt or model for that stage as a quick fix rather than a full subagent system.

- **Potential UI/UX Issues**:

- The dual editor and multiple modals can be considered heavy. There's a risk of UI bugs (like modals not sizing right, etc.) or that context switching (even within the wizard, the user moves from chat to modal back to chat to table, etc.) might ironically create some context switching overhead (contrary to our principle of minimizing context switching [58]). It's a delicate balance – we break the task into steps to reduce cognitive load, but each step uses a different UI component which might be jarring. We've tried to design it cohesively (common styling, stage indicator to orient them). We'll gather feedback in testing to refine this.
- Accessibility risk: if we don't get the a11y right, some users might find the specialized UI hard to use (e.g., keyboard navigation might be tricky in the diagram editor). We plan to test those flows.

- **Integration with Existing Workflows**: Some developers may have their own way of using AI or writing tests. The Wizard imposes its pattern (like always writing tests first, maybe they don't usually do that). If they diverge, The Wizard might not accommodate well (e.g., user might want to skip test generation stage – currently not really allowed). This inflexibility is a known trade-off to enforce best practices and trust. It's a risk if it annoys users who feel constrained. In future, perhaps optional shortcuts can be given (skip tests if not needed), but for V1, we stick to the ideal flow.

- **Dependency on Cursor IDE**: If The Wizard is tied to Cursor IDE specifically and not available outside, its adoption is limited to Cursor users. If Cursor's environment has any issues or if user doesn't use Cursor, they can't use The Wizard. It's a product decision (embedding in Cursor to leverage that environment). We accept this as V1 target environment and not try to make it standalone or in other editors just yet.

Each of these risks is either accepted for V1 or mitigated by design choices. We will document these to users or stakeholders so expectations are clear. Many of the trade-offs (like no reverse checking, no auto-apply) will be revisited in V2 after we validate the core concept with V1.

Monitoring usage and user feedback (via our logs and possibly direct user studies) will help determine which risks actually manifest as problems (maybe users actually don’t mind manual accept, but maybe they do mind the time it takes, etc.). We consider V1 a learning release as much as a delivering one, so those insights will drive where to invest effort next.

Milestones & Acceptance Criteria

To successfully deliver The Wizard V1, we break down the implementation into milestones with a timeline and define clear acceptance criteria for each. Below is the proposed delivery plan by week, assuming development starts immediately:

Week 1: Framework Setup & Core Engine
- *Milestone:* **Backend Skeleton & Routine Engine.** - Implement the basic workflow engine that can load a routine definition (start with a stub for RIPER-5) and manage stage progression logic (sequence of calls, waiting for user input). - Integrate the LLM API (use a placeholder if actual model not ready) and be able to send a dummy prompt and receive a response. - Define data models for session state, stage outputs, routine config structure. - *Acceptance Criteria:* A command-line test where calling the engine with a hardcoded user prompt goes through all stages with dummy content. For example, simulate each stage by printing “<<Research Stage>>”, “<<Plan Stage Diagram>>”, etc., and requiring a simulated user confirmation in between. This proves the state machine flows from start to finish in the correct order.

Week 2: Frontend UI Prototype
- *Milestone:* **UI Layout & Stage Navigation.** - Create the React front-end with the multi-stage wizard flow. Implement the stage indicator component and basic screens for each stage. - Specifically, implement the Plan stage dual-pane Mermaid editor modal with real-time preview using the Mermaid library. - Implement placeholders for test table and code display. - Connect front-end to back-end via WebSocket or HTTP for a couple of stages (maybe Plan stage end-to-end: send prompt, get diagram, show in UI). - *Acceptance Criteria:* The front-end can start a session (with a dummy backend or the real backend if Week1 milestone achieved), display a sample diagram in the editor, allow an edit, and capture the confirm action. The diagram editor should show a simple Mermaid graph (e.g., “A-->B”) and reflect text changes. The stage navigation bar updates when moving to next stage. No actual AI calls needed if using stub, but the plumbing should be in place (able to receive an object representing a diagram).

Week 3: Integrate AI Prompts for Plan & Execute (Partial)
- *Milestone:* **Diagram Generation and Confirmation Loop.** - Finalize the prompt template for Plan stage and connect to the actual LLM. Get it to generate a Mermaid diagram given a test input. - Implement handling of user feedback in Plan: if user edits the diagram manually or requests changes via NL, ensure the system can either accept the manual edit or send a refinement prompt to AI. - Implement prompt template for Execute tests and code generation, but focus this week on test generation output. - Build the test case table UI and allow editing/approving. - *Acceptance Criteria:* End-to-end demo of a **forward DAAS** loop: The user enters a simple request, the AI (via backend) returns a Mermaid diagram, the user confirms it, then the AI returns a couple of test case descriptions, the user approves, and the AI returns a trivial code snippet (could be pseudo-code). The code might be nonsense if model isn't tuned yet, but the flow should

work fully across front and back. Essentially, one can simulate developing a trivial “HelloWorld” feature through all stages. Key acceptance: the Mermaid diagram from AI shows up and can be confirmed, tests show up in the UI table, code appears in chat.

Week 4: Execute Stage Code Generation & Review Stage
- *Milestone:* **Complete Execute & Basic Review.** - Refine the Execute code generation prompt to produce actual code for our targeted scenarios and ensure it uses the blueprint and tests (adjust prompt as needed from initial results). - Implement the code display nicely (with syntax highlighting, file separation if needed). - Implement the Review stage UI: perhaps show the original diagram next to code (if easily done) or a checklist and the Accept Code button. - Enable the backward transition: if user rejects at Review, allow them to go back to Plan (e.g., via a “Revise” button that triggers the front-end to reopen the blueprint editor with the last diagram). - Add logging of events throughout the backend (key stage events). - *Acceptance Criteria:* A realistic scenario is run through: e.g., “User wants to add a new API endpoint function.” The AI generates a diagram with a few steps (e.g., validate input -> call service -> return response), user tweaks maybe one step, confirms. AI generates 2-3 test cases (happy path, one error case), user approves (maybe edits a test text). AI generates code (some pseudo-code implementing that flow). The review screen shows the diagram and the code output. The tester (us) intentionally spots a discrepancy (e.g., code didn’t implement an error check), clicks “Revise Plan”. The wizard goes back to Plan stage, where the user adds an “error handling” node to the diagram, confirms again. Then Execute re-runs (tests maybe update or new one added for error, code regenerates). Finally, user clicks Accept Code. **Criteria:** The system handled the backward jump gracefully (no crashes, state updated), and logs show the entire flow including the revision cycle. The final accept event is logged.

Week 5: Polish, Performance, A11y, and Testing
- *Milestone:* **Quality Improvements.** - Conduct thorough testing and bug fixes: fix any UI glitches, ensure modals work on different screen sizes, make sure Mermaid errors are handled (e.g., put a try-catch around rendering). - Optimize performance: add loading spinners for AI calls, throttle diagram preview updates if needed, test with a larger example to see if UI slows. - Implement accessibility features: test navigating via keyboard, add ARIA labels, ensure screen reader reads stage changes. - Fine-tune prompt templates based on testing feedback (e.g., adjust wording if the model was going off-track, add few-shot examples if needed). - Ensure security: test that no file writes happen, try some malicious inputs to ensure they don’t break out (e.g., in Mermaid, a script injection attempt shouldn’t succeed). - *Acceptance Criteria:* - All high-priority bugs from previous weeks are resolved. - The system feels reasonably responsive (diagram render under, say, 2 seconds for a moderate diagram, code generation maybe streaming). - Accessibility checks: navigate the entire wizard using only keyboard successfully and have important elements announced by a screen reader (we can use something like ChromeVox to test). - A test with a slightly more complex case (maybe a 2-step business process) runs without performance issues. - Observability: ensure that logs are being written for all key events and can be interpreted to compute completion rate. Do a simulated few runs and calculate the metric manually from logs.

Week 6: Documentation & Acceptance Testing
- *Milestone:* **Final Testing and Handover.** - Prepare user-facing documentation (or at least internal docs) for how to use V1, including its limitations (like we will note “please verify the code as the system doesn’t auto-verify” etc.). - If possible, conduct a small user acceptance test: have a couple of target users run through a scenario with The Wizard and gather feedback. - Fix any minor issues discovered in acceptance testing. - Ensure all **mandatory V1 checklist items are met**: - End-to-end spec-first workflow works (Yes, demonstration done). - Can go back in flow with state preserved (Yes). - Config-driven engine (Yes, show

that we could load the routine from a JSON by maybe actually doing it, or at least the structure is present). - Graceful failure: simulate an "AI cannot do X" scenario by perhaps forcing an error, see that UI shows a message and doesn't break. - Event logging implemented (Yes, verify that). - Sign off that success criteria are met. - *Acceptance Criteria:* Product manager and stakeholders review the working system against requirements: - They trigger a sample session and complete it successfully. - They intentionally cause a failure (maybe ask the AI something nonsensical to see how it fails) and see that it degrades gracefully ("I cannot complete this request" message shown, and user can cancel or modify input). - They inspect the configuration (maybe show how one could add a dummy new routine in config and that the system would list it, even if not fully functional, proving pluggability). - All scope items from PRD Section 6.1 are demonstrated as implemented [12], and none of the out-of-scope items are present. - Stakeholders agree that the "North Star" metric (happy path completion) is at least measurable, and initial internal tests indicate users can complete the flow (we might not have a percentage yet, but anecdotal success in testing).

If all of the above criteria are satisfied by end of Week 6, The Wizard V1 can be considered ready for release (or internal pilot). We will have delivered a fully functional implementation of the Spec-Then-Code workflow, complete with documentation of how to run it and an initial evaluation of its efficacy.

---

[1] [3] [4] [7] [12] [13] [14] [16] [17] [18] [20] [21] [22] [23] [24] [40] [52] [56] [57] [58] The-Wizard-V1-PRD.md
file://file_00000000c07c61f6bfd6ec9ccdd34c96

[2] [5] [6] [8] [9] [10] [11] [15] [19] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [41] [42] [43] [44] [45] [46] [47] [54] [55] 设计文档版本 4 - 更正.pdf
file://file_000000009aac61f689bb19762c82ece6

[48] [49] [50] [51] [53] RIPER-5-EN.md
file://file_000000001c6c61f6bd2bcf06e1e3c10f