A Pragmatic Technical Blueprint for the AI Scrum Assistant

This document provides the exhaustive architectural blueprint and implementation plan for building the "AI Scrum Assistant." The plan is designed to master next-generation technologies in AI-driven orchestration, focusing on a robust, full-cycle Agile tool.

Part 1: The AI Scrum Assistant - Architectural Blueprint & Core Stack

A. A High-Level Architecture for an Agentic Scrum Assistant

The system is architected as a **Compound AI System** (Multi-Agent System). A central **Orchestrator** (Node.js/Express) manages specialized **agents** and **tools** (LangChain, Jira, Chroma) to handle complex, multi-step tasks reliably.

The architecture is composed of five core components:

1. **React Frontend (HITL Dashboard):** The **Human-in-the-Loop (HITL)** interface for review and "approve/reject" actions.
2. **Node.js/Express Backend (Orchestration Layer):** The system's "brain" that orchestrates all calls.
3. **AI Orchestration Framework (LangChain.js):** The "nervous system" that manages multi-step LLM interactions and enforces structured output.
4. **External Service APIs (Jira & Google):** The "hands" and "senses" for data perception and action.
5. **Vector Database (Chroma):** The agent's **"semantic memory"** for Retrieval-Augmented Generation (RAG).

B. The Orchestration Layer: Your Node.js Backend

The Express server is an "intelligent API layer." Controllers manage requests but delegate all logic to modular service modules, strictly enforcing **Separation of**

Concerns.

Logic Flow Example (PRD-to-Ticket):

1. Frontend uploads a PRD file (PDF) to `POST /api/v1/scrum/suggestions`.
2. `scrum.controller.js` receives the file buffer.
3. The controller calls `await aiService.getSuggestionsFromPRD(prdFileBuffer)`.
4. `ai.service.js` uses `pdf-parse` to extract text, uses **LangChain.js** to build a prompt, calls the Gemini API (enforcing the Zod schema), and returns the clean data.

C. Foundational Stack Recommendations

Component	Technology	Rationale and Justification
Backend	Node.js + Express	Ideal for I/O-heavy orchestration of high-latency API calls.
AI Orchestration	**LangChain.js**	Best-suited framework for **agentic workflows**, chains, and complex tool-use.
LLM Provider	**Google (Gemini)**	`gemini-1.5-pro-latest` supports **Tool Calling** (Function Calling), essential for reliable structured JSON output.
Jira Communication	`jira.js` library	Provides a clean, class-based wrapper for the Jira v3 REST API.
File Processing	`multer` & `pdf-parse`	`multer` handles file upload; `pdf-parse` extracts raw text from the PDF buffer.
Vector DB	**Chroma**	Required for the RAG-based duplicate detection feature.
Data Validation	**Zod**	Used to define and validate LLM output schemas, enforcing reliability.
Frontend	React + Vite	Modern, high-performance frontend stack.
Frontend State	**React Query (TanStack)**	Standard for managing asynchronous **server state**. Its powerful `useMutation` hook is critical for the **HITL approval workflow**.

D. Phase 1 Core Capabilities (Complete Cycle)

Use Case	Description	AI Technique

1. PRD to Jira Ticket Generation	Automatically converts an uploaded **PDF/Text PRD** into structured Jira Stories and Tasks, ready for Human-in-the-Loop (HITL) review.	**Structured Output (Tool Calling)** enforced by Zod and LangChain.js.
2. AI-Powered Backlog Refinement	Finds **semantic duplicates** and suggests improvements (missing ACs, splitting oversized stories) on existing backlog items.	**Retrieval-Augmented Generation (RAG)** using Chroma Vector DB for semantic search.
3. Data-Driven Sprint Planning	Calculates historical **Velocity** and **Spillover** metrics from raw Jira data to provide data-backed recommendations for realistic sprint capacity.	**Manual Metric Calculation** via Jira API, followed by **Data-Rich Prompting**.
4. Sprint Retrospectives & Reporting	Analyzes a closed sprint's issues and metrics to auto-generate a summary of progress, burndown stats, and actionable retrospective insights.	**Data Aggregation and Qualitative Analysis** via dedicated, structured prompt.
5. AI-Generated Daily Standups	Automatically generates team and individual progress reports by synthesizing Jira status changes, comments, and work logs from the last 24 hours.	**Data Synthesis and Structured Narrative Generation** using a dedicated standup schema.

Part 2: Connecting the Assistant - Deep Dive into the Jira API

A. Authentication: The API Token-Based Approach

The system uses an **API Token with Basic Auth** for system-to-system integration. This is simpler and more appropriate than OAuth 2.0 for a standalone assistant. The `jiraClient` is instantiated in `backend/src/services/jira/jira_client.js` using credentials from the `.env` file.

B. Core Jira Service Layer Structure (Modular)

All Jira logic is split into specialized modules under `/services/jira`:

Service Module	Responsibility	Core Functionality

jira_client.js	Client Initialization	Exports the authenticated `jiraClient` instance.
issue_service.js	Issue CRUD & Search	`createTicket`, `updateTicket`, `search` (flexible JQL queries).
agile_service.js	Metric Calculation	`getBacklogIssues`, `getActiveSprint`, `calculateAverageVelocity`, `findSpillover`.

C. Querying for Metrics: The "Hidden" Logic

Since Jira's key Agile reports are not exposed via API, metrics are calculated manually:

- **Average Velocity:** Calculated in `agile_service.js` by fetching the last N closed sprints and summing the Story Points (`customfield_10002`) only for issues categorized as **'Done'**.
- **Spillover:** Calculated in `agile_service.js` using the JQL workaround: `sprint = {fromSprintId} AND sprint = {toSprintId}`.

Part 3: Implementation Guide - Use Case 1: From PRD to Jira Ticket

A. The PRD Input Handling (File Upload)

1. The Express controller uses the **`multer`** middleware to handle the `multipart/form-data` request, receiving the PRD as a **file buffer**.
2. **`ai.service.js`** uses the **`pdf-parse`** library to convert the binary buffer into a raw text string.

B. Defining the Output Schema with Zod (`schemas.js`)

The **Zod library** defines the exact nested JSON structure required for a Jira ticket. This schema is converted to JSON Schema by LangChain.js and sent to Gemini to enable **Tool Calling**, guaranteeing perfectly formatted output.

C. The LangChain.js Implementation (`ai.service.js`)

1. **Chain Setup:** A prompt template defining the "Senior Project Manager" persona is piped to the LLM.

2. **Structured Output:** The chain uses `llm.withStructuredOutput(PRDParserSchema)` to force the output into the defined schema.
3. **Result:** `chain.invoke()` returns a clean JavaScript object (`{ tickets: [...] }`) ready for the HITL dashboard.

Part 4: Implementation Guide - Use Case 2: AI-Powered Backlog Refinement

This use case uses **RAG** for duplicate detection and LLM analysis for ticket critiques.

- **RAG Pipeline:** An initial script indexes Jira tickets using **Google Generative AI Embeddings** and stores them in **Chroma**. The `ai.service.js` retrieves the top N similar tickets via `vectorStore.similaritySearch` to augment a new prompt that asks the LLM to confirm duplication.
- **Critique Task:** A dedicated "Ticket Analyzer" function in `ai.service.js` ingests the ticket JSON and uses a **Chain-of-Thought prompt** to check for INVEST principles and recommend splitting oversized stories into a structured JSON output.

Part 5: Implementation Guide - Use Case 3: Data-Driven Sprint Planning

This feature combines quantitative data from Jira with qualitative analysis from the AI.

1. **Data-Gathering:** `ai.service.js` orchestrates calls to `agile_service.calculateAverageVelocity` and `agile_service.findSpillover` to gather the key metrics.
2. **Prompt Construction:** All gathered metrics (e.g., `{avg_velocity}`, `{spillover_issues_list}`) are formatted and injected into a comprehensive prompt template.
3. **AI Recommendation:** The LLM acts as an "Expert Agile Coach" to acknowledge the data, calculate remaining capacity (`Velocity - Spillover Points`), and recommend the optimal backlog items to pull into the sprint.

Part 6: Implementation Guide - Use Case 4: Sprint Retrospectives & Reporting

This feature generates actionable insights by analyzing a closed sprint's issues.

1. **Data Source:** The controller fetches the full issue data for a closed sprint.
2. **Analysis:** The `ai.service.generateSprintRetrospective` function is called.
3. **Structured Output:** The LLM output is forced into the `RetrospectiveSchema` to provide clear, categorized results: `whatWentWell`, `whatDidNotGoWell`, and `actionableInsights`.

Part 7: Implementation Guide - Use Case 5: AI-Generated Daily Standups

This feature automates the creation of daily progress reports by synthesizing live Jira data.

A. Data Acquisition and Grouping

1. **Jira Query:** The system uses JQL to fetch all issues updated in the last 24 hours: `updated >= -1d AND sprint in openSprints()`.
2. **Group Work:** The controller or AI service groups the retrieved issues by the current assignee.
3. **Data Synthesis:** For each assignee, the system determines the issue's current state (Completed, In Progress, Blocked) by checking the issue's status and flags.

B. Structured Narrative Generation

1. **Prompt Persona:** The LLM is instructed to act as a **"Team Lead"** to convert the raw, synthesized issue data into a concise, natural-language standup update.
2. **Structured Output:** The output is forced into the `StandupSummarySchema`, which includes an array of updates (`individualUpdates`) for each team member and an overall `teamMetrics` summary. This ensures the output is instantly ready to be displayed in the UI or posted to a chat platform.

Part 8: Building the Human-in-the-Loop (HITL) Dashboard (LAST STEP)

The React frontend is the critical **Human-in-the-Loop** component, and its architecture must be robust. **It should be implemented after all core AI features (Parts 3-7) are complete and tested.**

A. Dashboard Architecture: React Query + Shadcn/ui

The frontend uses **React Query (TanStack)** for robust Server State Management.

- **Mutation Flow:** The **"Approve"** and **"Reject"** buttons are implemented with the `useMutation` hook.
- **Atomic Updates:** Upon successful mutation (ticket creation), the `onSuccess` callback runs `queryClient.invalidateQueries(['ai-suggestions'])`. This automatically re-fetches the list of suggestions from the backend, ensuring the UI is always in sync with the server state.

B. Core Hooks & Component Implementation

The application will be structured around these core TanStack Query hooks:

- `useAiSuggestions()`: Uses `useQuery` to fetch the array of draft tickets from the Part 3 endpoint.
- `useApproveSuggestion()`: Uses `useMutation` to call the backend API that creates the Jira ticket. It invalidates the `ai-suggestions` query on success to automatically update the list.
- The UI will use the `SuggestionReviewCard.jsx` component, built with **Shadcn/ui**, to render suggestions and trigger these mutations.

Part 9: Conclusions and Next Steps

This architectural blueprint provides a comprehensive, pragmatic, and modern path to building the AI Scrum Assistant, achieving all five core automation goals and establishing a robust foundation for Phase 2 expansion.

The next steps in the development flow focus on completing the backend intelligence: **Implementing the RAG indexing script and the remaining AI service functions (Parts 4-7).**