A Pragmatic Technical Blueprint for the AI Scrum Assistant: RAG Expansion and Conversational Intelligence

Executive Summary: Full-Cycle Agile Automation with Real-Time RAG

This comprehensive blueprint details the architecture for the AI Scrum Assistant, which automates the full Agile cycle—from PRD creation to retrospectives—now powered by a high-availability, real-time Retrieval-Augmented Generation (RAG) pipeline. This foundation ensures all generative outputs are grounded in the freshest Jira data and proprietary documentation.

The system is augmented with a new core feature, the **AI Scrum Assistant Chatbot**, which uses advanced conversational RAG techniques to provide instant, verifiable answers to complex "scrum master doubts," effectively "cashing" organizational knowledge into a long-term semantic memory.

The core stack remains Node.js/Express as the Orchestrator, LangChain.js for AI orchestration, Gemini for LLM Tool Calling, and Chroma as the multi-collection Vector Database.[1]

Part 1: Architectural Blueprint & Core Stack

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

The system is architected as a Compound AI System, where a central Orchestrator (Node.js/Express) manages specialized agents and tools (LangChain, Jira, Chroma) to handle complex, multi-step tasks reliably.[1]

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 and houses the core business logic.
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 expanded "semantic memory" for Retrieval-Augmented Generation (RAG).[1]

B. The Orchestration Layer: Node.js/Express

The Express server serves as an intelligent API layer. Controllers manage requests but delegate all logic to modular service modules, strictly enforcing Separation of Concerns.[1]

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.[1]

C. Foundational Stack Recommendations

Component	Technology	Rationale and Justification
Backend	Node.js + Express	Ideal for I/O-heavy orchestration of high-latency API calls.[1]
AI Orchestration	LangChain.js	Best-suited framework for agentic workflows, chains, and complex tool-use.[1]
LLM Provider	Google (Gemini)	Gemini-2.0-flash supports Tool Calling (Function Calling), essential for reliable structured JSON output.[1]
Jira Communication	jira.js library	Provides a clean, class-based wrapper for the Jira v3 REST API.[1]
File Processing	multer & pdf-parse	multer handles file upload; pdf-parse extracts raw text from the PDF buffer.[1]
Vector DB	Chroma	Expanded to serve as the primary RAG semantic memory.[1]
Data Validation	Zod	Used to define and validate LLM output schemas, enforcing reliability.[1]
Frontend	React + Vite	Modern, high-performance frontend stack.[1]
Frontend State	React Query (TanStack)	Standard for managing

		asynchronous server state, critical for the HITL approval workflow.[1]

D. Deep Dive into Jira Communication and Metrics

All Jira logic is modularized under /services/jira to enforce separation.[1]

- **Authentication:** The system uses an API Token with Basic Auth for secure system-to-system integration.[1]
- **Core Services:** Dedicated modules include issue_service.js (Issue CRUD & Search) and agile_service.js (Metric Calculation).[1]
- **Metrics Calculation:** Since Jira's key Agile reports are not exposed via API, metrics like **Average Velocity** (summing Story Points for 'Done' issues in the last N closed sprints) and **Spillover** (using JQL workarounds to identify issues moved to the next sprint) are calculated manually within agile_service.js.[1]

E. Elevating RAG to Foundational Status and Semantic Isolation

The vector database (Chroma) is elevated to the system's primary semantic memory, enabling continuous context grounding for all generative features.[1]

- **Semantic Noise Reduction:** To prevent retrieving irrelevant context (e.g., pulling PRD text when checking a bug status), the knowledge base is segmented. The architecture utilizes Chroma's capability to manage multiple isolated **Collections** based on data type, vastly improving retrieval speed and accuracy.[2]

Part 2: Data Ingestion and Synchronization Pipeline (RAG Core)

This section details the mechanisms required to ensure the vector database remains continuously updated and non-stale, adhering to required data freshness.

A. Multi-Collection Strategy for Isolation and Performance

The knowledge base is formally segmented into three distinct collections:

Collection Name	Primary Data Type Stored	Indexing Frequency	Primary Retrieval Goal
tickets_current	Current Jira Issue Content (Description, ACs, Comments)	Real-Time (Event-Driven)	Semantic Search & Duplication Check
prds_docs	Chunked Technical Documentation (PRDs, Specs, Architecture)	Event-Driven (On Upload/Edit)	Contextual Grounding for LLM Generation
sprints_metrics	Structured Metrics, Retrospective Analysis, Sprint Summaries	Scheduled (Sprint Close)	Quantitative Filtering for Sprint Planning & Retrospectives

This isolation allows the retrieval step to apply strict, pre-retrieval filtering before engaging in vector similarity search, which drastically reduces the search space and improves RAG effectiveness.[2]

B. Comprehensive Metadata Schema Design for Retrieval Efficiency

Rich metadata stored alongside vectors supports hybrid search (combining filtering and semantic search).[3]

Field Name	Data Type	Source	Purpose (Hybrid Search/Filtering)
source_type	String	Static Tag (ticket, prd, metric)	Essential for collection filtering integrity.[4]
jira_key	String	Jira API (Issue Key)	Direct link to the source Jira item for citation.[5]
jira_status	String	Jira API (Status Name)	Filtering by workflow state (e.g., 'To Do', 'In Progress', 'Done').
sprint_id	String	Jira API (Current Sprint ID)	Filter by active or closed sprint context.
doc_version	String	PRD Parser / Jira Link	Critical for document versioning and consistency checks.

C. Real-Time Indexing: Jira Webhooks Service Design

For high-churn data, such as Jira tickets (status changes, comments, AC updates), an event-driven mechanism via webhooks is used to maintain real-time data freshness, which is superior to periodic polling.[6]

- **IADP (Immediate Acknowledge, Deferred Process):** The Node.js Orchestrator implements a secure webhook endpoint that immediately performs basic validation and pushes the full Jira payload onto an internal message queue (e.g., Redis Queue) before returning a 200 OK response. This prevents high-latency vector operations from blocking Jira's threads.[8]
- **Worker Service:** A dedicated indexing_service.js asynchronously consumes the message queue, performs embedding, and executes the vector upsert/update in Chroma, using the Jira Issue Key (e.g., JIRA-123) as the unique ID for update/replace operations to prevent duplication.[8]

D. Scheduled Indexing: Cron Job Architecture for Batch Updates

For data that changes less frequently (sprint metadata, historical metrics), scheduled indexing via cron jobs is used.[1] The system must employ **State-Aware Indexing**, where a full metrics calculation and vector upsert/update is only triggered if a sprint's state has explicitly transitioned (e.g., from 'ACTIVE' to 'CLOSED').[9]

E. Unstructured Data Handling: PRD Chunking and Enrichment

Product Requirements Documents (PRDs) require sophisticated handling:

- **Parsing:** The ai.service.js uses pdf-parse to convert the binary buffer from the multer middleware into raw text.[1]
- **Hierarchical Chunking:** Instead of fixed-size chunks, a **Hierarchical Chunking** strategy is recommended, splitting the document based on its internal structure (headers, sections) to preserve semantic coherence. A moderate chunk size (e.g., 512 tokens) with a high overlap (10–20%) helps maintain continuity across boundaries.[10]
- **Metadata Extraction:** Essential metadata (e.g., document_title, version, author) is extracted during chunking and stored alongside the vector embedding.[13]

Part 3: Core Feature Implementation: PRD to Jira Ticket Generation

This feature automates the conversion of technical requirements into actionable work items, ready for Human-in-the-Loop review.[1]

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.[1]
2. ai.service.js uses the pdf-parse library to convert the binary buffer into a raw text string.[1]

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.[1]

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.[1]
2. **Structured Output:** The chain uses llm.withStructuredOutput(PRDParserSchema) to force the output into the defined schema.[1]
3. **Result:** chain.invoke() returns a clean JavaScript object ready for the HITL dashboard.[1]

Part 4: Core Feature Implementation: AI-Powered Backlog Refinement

This feature finds semantic duplicates and suggests improvements on existing backlog items.[1]

Implementation & RAG Enhancement

- **RAG Pipeline:** The initial indexing script stores existing Jira tickets (content and comments) using vector embeddings in the tickets_current collection.[1]
- **Duplicate Detection:** ai.service.js retrieves the top N semantically similar tickets via vector search to augment a new prompt that asks the LLM to confirm duplication.[1]
- **Critique Task:** A dedicated "Ticket Analyzer" function 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.[1]
- **RAG Augmentation:** During duplication checks, the RAG query is highly constrained, leveraging metadata (e.g., project_id, issue_type) to search only within the relevant subset of the tickets_current collection, avoiding semantic noise and improving precision.[3]

Part 5: Core Feature Implementation: Data-Driven Sprint Planning

This feature calculates metrics and uses AI to recommend optimal sprint capacity.[1]

Implementation & RAG Enhancement

1. **Data-Gathering:** ai.service.js orchestrates calls to agile_service.calculateAverageVelocity and agile_service.findSpillover to gather key metrics (Part 1.D).[1]
2. **Prompt Construction (Context Enrichment Pattern):** All gathered metrics

(avg_velocity, spillover_issues_list) are formatted and explicitly injected into a comprehensive prompt template.[14]

3. **AI Recommendation:** The LLM acts as an "Expert Agile Coach" to calculate remaining capacity (Velocity - Spillover Points) and recommend optimal backlog items to pull into the sprint.[1]
4. **RAG-Enabled Forecasting Confidence:** By indexing detailed historical metric records into the sprints_metrics collection (Part 2.F), the RAG system can analyze the standard deviation across retrieved historical velocity metrics. This allows the LLM to temper its capacity recommendations with a confidence interval, providing more realistic recommendations during planning.[15]

Part 6: Core Feature Implementation: Sprint Retrospectives & Reporting

This feature generates actionable retrospective insights by analyzing a closed sprint.[1]

Implementation & RAG Enhancement

1. **Multi-Source Aggregation:** The ai.service.generateSprintRetrospective function aggregates data from three distinct, grounded sources: Raw Issue Data (comments, history) for qualitative analysis; the sprints_metrics collection (retrieval filtered by sprint_id) for quantitative context (Velocity, Spillover points); and relevant prds_docs chunks to assess alignment between requirements and execution.[17]
2. **Structured Output:** The LLM output is forced into the RetrospectiveSchema to provide clear, categorized results: whatWentWell, whatDidNotGoWell, and actionableInsights.[1]
3. **LLM Prompting (Data-Rich Grounding):** The prompt template embeds the retrieved metrics as foundational data (e.g., "**CONTEXT:** The team's average velocity was 25 points..."). This data-rich prompting forces the LLM to ground its qualitative assessment in measurable facts before generating the retrospective summary.[18]

Part 7: Core Feature Implementation: AI-Generated Daily Standups

This feature automates the creation of daily progress reports by synthesizing live Jira data.[1]

Implementation & RAG Enhancement

1. **Data Acquisition:** The system uses JQL to fetch all issues updated in the last 24 hours: updated >= -1d AND sprint in openSprints().[1]
2. **RAG Context Augmentation:** Before narrative generation, for each key issue, a low-latency RAG query is executed against tickets_current and prds_docs to fetch relevant Acceptance Criteria (ACs), comments, or design notes (filtered by jira_key and doc_version).
3. **Contextual Detail Augmentation:** Utilizing RAG transforms the standup from a robotic status change report to a **contextualized narrative** (e.g., detailing the requirements addressed from PRD v1.2, section 3.2), providing necessary depth.[19]
4. **Structured Narrative Generation:** The LLM, instructed as a "Team Lead" persona, synthesizes the raw status updates and the retrieved context into the structured standupSummarySchema, ready for display or posting.[20]

Part 8: New Core Feature: The AI Scrum Assistant Chatbot (Conversational RAG Agent)

The Chatbot requires a robust, low-latency conversational RAG architecture to address "scrum master doubts," utilizing the comprehensive RAG pipeline.

A. Chatbot RAG Chain Architecture Blueprint (LangChain.js)

The conversational agent operates via a multi-step RAG flow, orchestrated by LangChain.js.[21]

Step	Component/Service	Input	Output/Action

1. Query	React Frontend / Orchestrator	User Query + Session History	Initial Query Payload
2. Contextualization	LangChain.js (CQG Model)	Query + Raw History	Standalone Search Query (Query Rewriting)
3. Retrieval	Chroma DB / ai.service.js	Standalone Query + Metadata Filters	Top N relevant Context Chunks (with Metadata)
4. Generation	LangChain.js / Gemini 1.5 Pro	Query + Context + System Prompt	Final Response Text + Source Citations
5. Memory Update	Orchestrator	Query, Context, Response	Update Session History (Summary & Raw Log)

The **Contextualization Layer (CQG)** uses an LLM to perform Query Rewriting (or De-contextualization). If a user asks an ambiguous follow-up question (e.g., "Tell me more about that"), the CQG Model uses the chat history to rewrite the query into a fully specified, standalone question, preventing retrieval failure.[23]

B. Implementing Advanced Conversational Memory

To handle complex, extended conversations, a Hybrid Memory Strategy is necessary.[25]

- **Hybrid Memory Strategy: Short-Term Memory (STM)** retains the raw text of the last N turns for immediate context. **Long-Term Memory (LTM)** periodically generates a concise summary of older turns, which is stored as a structured entity, providing context pruning and reducing token consumption.[24]
- **Intent Routing and Tool Utilization:** The agent leverages Gemini's **Tool/Function Calling** capabilities to route the user's intent: RAG for informational requests (e.g., "What is the status of JIRA-123?") versus the transactional issue_service.js API tool for actions (e.g., "Block JIRA-123").[1]

C. "Cashing Doubts": Long-Term Semantic Memory

When a complex question ("scrum master doubt") is successfully resolved by the chatbot and manually validated via the HITL Dashboard, the query, context, and the validated response are indexed into a separate, prioritized collection: verified_answers. This acts as high-confidence organizational knowledge, prioritized during retrieval to maximize reliability and speed when similar doubts arise.[26]

Part 9: Building the Human-in-the-Loop (HITL) Dashboard

The React frontend is the critical Human-in-the-Loop component, implemented after all core AI features are complete.[1]

A. Dashboard Architecture: React Query + Shadcn/ui

The frontend uses React Query (TanStack) for robust Server State Management.[1]

- **Mutation Flow:** The "Approve" and "Reject" buttons are implemented with the useMutation hook.[1]
- **Atomic Updates:** Upon successful mutation (ticket creation), the onSuccess callback runs queryClient.invalidateQueries(['ai-suggestions']). This automatically re-fetches the list of suggestions, ensuring the UI is always in sync.[1]

B. Frontend Requirements and Trust Design (Chatbot)

The Chatbot must adhere to high standards of UI/UX, prioritizing clarity and trust for a professional B2B application.[27]

- **Mandatory Source Citation:** Every generated response that relies on retrieved context *must* display inline or immediately adjacent source citations. These citations should be

interactive links (e.g., a clickable Jira Key or a link to the relevant PRD section), allowing the user to instantly verify the factual grounding of the response. This is the primary mechanism for mitigating the risk of LLM hallucination.[29]

Part 10: Conclusions and Next Steps

The expanded architecture successfully provides comprehensive knowledge grounding across all Jira data types via a real-time, multi-collection RAG pipeline.

The next steps in the development flow are prescribed as follows:

- **Stage 1 (Data Foundation):** Focus on completing the Ingestion Service (webhook_service.js, indexing_service.js), finalizing the ETL logic for metrics, and executing the initial, full indexing into the dedicated Chroma collections.
- **Stage 2 (Intelligence Layer):** Implement the Query Rewriting/CQG model and the Hybrid Memory management within the new conversational_agent.js module.
- **Stage 3 (User Experience):** Integrate the final RAG chains into the React Frontend, focusing on developing the Chatbot UI and prioritizing trust design through mandatory, interactive source citation display.[30]

Works cited

1. A Pragmatic Technical Blueprint for the AI Scrum Assistant.pdf
2. Data Model - Chroma Docs, accessed November 12, 2025, https://docs.trychroma.com/docs/overview/data-model
3. How to Use Metadata in RAG for Better Contextual Results | Unstructured, accessed November 12, 2025, https://unstructured.io/insights/how-to-use-metadata-in-rag-for-better-contextual-results?modal=try-for-free
4. Build a custom RAG agent - Docs by LangChain, accessed November 12, 2025, https://docs.langchain.com/oss/python/langgraph/agentic-rag
5. Metadata for Jira - Atlassian Marketplace, accessed November 12, 2025, https://marketplace.atlassian.com/apps/42075/metadata-for-jira
6. Webhooks - Jira Software Cloud - Atlassian Developer, accessed November 12, 2025, https://developer.atlassian.com/cloud/jira/software/webhooks/
7. Best practices on working with WebHooks in Jira Data Center - Atlassian Support, accessed November 12, 2025, https://support.atlassian.com/jira/kb/best-practices-on-working-with-webhooks-in-jira-data-center/
8. Why Your Enterprise RAG System Needs Real-Time Vector Database Updates (And How to Build Them), accessed November 12, 2025,

https://ragaboutit.com/why-your-enterprise-rag-system-needs-real-time-vector-database-updates-and-how-to-build-them/
9. cronjob: How to reindex only what is needed - Magento Stack Exchange, accessed November 12, 2025, https://magento.stackexchange.com/questions/14828/cronjob-how-to-reindex-only-what-is-needed
10. Chunking Strategies to Improve Your RAG Performance - Weaviate, accessed November 12, 2025, https://weaviate.io/blog/chunking-strategies-for-rag
11. Mastering Document Chunking Strategies for Retrieval-Augmented Generation (RAG) | by Sahin Ahmed, Data Scientist | Medium, accessed November 12, 2025, https://medium.com/@sahin.samia/mastering-document-chunking-strategies-for-retrieval-augmented-generation-rag-c9c16785efc7
12. Build an unstructured data pipeline for RAG - Azure Databricks | Microsoft Learn, accessed November 12, 2025, https://learn.microsoft.com/en-us/azure/databricks/generative-ai/tutorials/ai-cookbook/quality-data-pipeline-rag
13. How to Use Metadata in RAG for Better Contextual Results | Unstructured, accessed November 12, 2025, https://unstructured.io/insights/how-to-use-metadata-in-rag-for-better-contextual-results?modal=contact-sales
14. Prompt Engineering Patterns for Successful RAG Implementations - Shittu Olumide Ayodeji, accessed November 12, 2025, https://iamholumeedey007.medium.com/prompt-engineering-patterns-for-successful-rag-implementations-b2707103ab56
15. View and understand the velocity chart | Jira Cloud - Atlassian Support, accessed November 12, 2025, https://support.atlassian.com/jira-software-cloud/docs/view-and-understand-the-velocity-chart/
16. How Burndown and Velocity compliment each other - Help Center, accessed November 12, 2025, https://help.zenhub.com/support/solutions/articles/43000483134-how-burndown-and-velocity-compliment-each-other
17. How Multi-Context Processing Could Make or Break An LLM Project - Galileo AI, accessed November 12, 2025, https://galileo.ai/blog/multi-context-processing-llms
18. Top 5 LLM Prompts for Retrieval-Augmented Generation (RAG) - Scout, accessed November 12, 2025, https://www.scoutos.com/blog/top-5-llm-prompts-for-retrieval-augmented-generation-rag
19. Daily and weekly stand up meeting template - Bluedot AI Note Taker, accessed November 12, 2025, https://www.bluedothq.com/blog/daily-and-weekly-stand-up-meeting-template
20. The Ultimate Fucking Guide to Prompt Engineering : r/PromptEngineering - Reddit, accessed November 12, 2025, https://www.reddit.com/r/PromptEngineering/comments/1j8m0rs/the_ultimate_fu

cking_guide_to_prompt_engineering/
21. Q&A with RAG - LangChain overview, accessed November 12, 2025, https://js.langchain.com/v0.1/docs/use_cases/question_answering/
22. Build a simple RAG Chatbot with LangChain | by Kong Nopwattanapong - Medium, accessed November 12, 2025, https://medium.com/credera-engineering/build-a-simple-rag-chatbot-with-langchain-b96b233e1b2a
23. RAG: Answer follow up questions : r/LLMDevs - Reddit, accessed November 12, 2025, https://www.reddit.com/r/LLMDevs/comments/1epk9c2/rag_answer_follow_up_questions/
24. Repeated Conversational Memory in RAG-based Chatbot - Gemini Apps Community, accessed November 12, 2025, https://support.google.com/gemini/thread/376054018/repeated-conversational-memory-in-rag-based-chatbot?hl=en-gb
25. Conversational Memory for LLMs with Langchain - Pinecone, accessed November 12, 2025, https://www.pinecone.io/learn/series/langchain/langchain-conversational-memory/
26. Building a RAG-Based Chatbot with Memory: A Guide to History-Aware Retrieval - Chitika, accessed November 12, 2025, https://www.chitika.com/rag-based-chatbot-with-memory/
27. UI UX Design for Chatbot: Best Practices and Examples | by Lollypop Design, accessed November 12, 2025, https://lollypop-studio.medium.com/ui-ux-design-for-chatbot-best-practices-and-examples-5d69ff2840f5
28. Browse thousands of Chatbot images for design inspiration - Dribbble, accessed November 12, 2025, https://dribbble.com/search/chatbot
29. What is RAG? - Retrieval-Augmented Generation AI Explained - Amazon AWS, accessed November 12, 2025, https://aws.amazon.com/what-is/retrieval-augmented-generation/
30. Citations in the Key of RAG - Mark Anthony Cianfrani, accessed November 12, 2025, https://cianfrani.dev/posts/citations-in-the-key-of-rag/