How It Works: The RAG Architecture

The project uses a powerful AI technique called **Retrieval-Augmented Generation (RAG)**. Instead of just sending the one-liner to the AI, it follows a smarter process:

1. **Retrieval:** It first **retrieves** relevant context about your project. In this case, it clones the public GitHub repositories you specify in the .env file. This gives it access to your actual source code.
2. **Augmentation:** It then **augments** (enhances) the original one-liner by combining it with the most relevant code snippets it found.
3. **Generation:** Finally, it sends this rich, context-filled prompt to the **Google Gemini** AI model to **generate** the final, detailed story.

This ensures the generated story isn't generic; it's tailored to your project's specific technologies, coding patterns, and existing components.

Component-by-Component Breakdown

The project is broken down into several key components, each with a specific job:

1. **main.py (The Front Door):**
    a. This file uses **FastAPI** to create a simple web server.
    b. It serves the index.html file, which is the user interface you see in your browser.
    c. It provides the /create-story/ API endpoint that listens for the user's prompt. When it receives a prompt, it kicks off the entire process by calling the orchestration_service.
2. **knowledge_base_manager.py (The Librarian):**
    a. This is a setup script you run once. Its job is to build the project's "memory."
    b. It reads the GITHUB_REPO_URLS from your .env file.
    c. It clones each repository, loads the source code (specifically .py files in this configuration), and splits it into logical chunks.
    d. It uses a special AI model (SentenceTransformer) to convert these code chunks into numerical representations called **embeddings**.
    e. It stores all these embeddings in a local vector database created with **ChromaDB**. This database is now a searchable knowledge base of your entire codebase.
3. **context_retrieval.py (The Search Engine):**

    a. When you submit a prompt, this component takes it and converts it into an embedding.
    b. It then searches the ChromaDB database to find the code chunks with the most similar embeddings.
    c. It returns the text of these top-matching code snippets, which serve as the context for the AI.
4. **orchestration_service.py (The Conductor):**
    a. This is the central controller that manages the entire workflow.
    b. It takes the user's prompt, gets the relevant context from the context_retrieval engine, and then constructs a detailed "master prompt."
    c. This master prompt is a carefully engineered set of instructions that tells the AI its role, gives it the context it needs, provides an example of a good story, and includes the user's original request.
5. **llm_interface.py (The AI Communicator):**
    a. This module's only job is to take the master prompt and send it to the Google Gemini API.
    b. It's configured to ask Gemini to respond in a structured **JSON** format, which makes the output predictable and easy to work with.
6. **jira_integration.py (The Scribe):**
    a. This component receives the structured JSON output from the AI.
    b. Instead of connecting to Jira, it formats the title, user story, and acceptance criteria into a clean Markdown layout.
    c. It saves this content to the output.md file in the project's root directory.

End-to-End Workflow

1. **Setup:** You run python knowledge_base_manager.py. It clones the specified GitHub repos and builds the local chroma_db knowledge base.
2. **Run:** You start the server with uvicorn main:app --reload.
3. **Interact:** You open your browser to http://127.0.0.1:8000, type in a prompt (e.g., "add a logout endpoint"), and click "Generate Story".
4. **Process:** The system finds relevant code about your existing API endpoints, builds a master prompt, and sends it to Gemini.
5. **Result:** Gemini returns a structured story, which the system saves to output.md. Your web browser shows a success message confirming the file has been created.