It's a multi-agent system designed to automate nearly the entire development lifecycle for a Spring Boot microservice, from understanding requirements to creating a pull request.

Here is a step-by-step breakdown of how it works:

**Step 1: Initialization and User Input**

The process begins when you run the main method.

1. Get SRS Path: The application first prompts you to enter the full path to your Software Requirements Specification (SRS) document. This is the primary input that drives the entire workflow.

2. Parse SRS Metadata: It reads the SRS file and looks for three specific keys:

• GitHub-URL: The URL of the target repository.

• checkout\_branch: The main development branch (e.g., main or master) from which to start.

•Repository-Name: The local directory name for the project. If any of these are missing, the program stops.

**Step 2: Change Detection (The "Should I Work?" Step)**

This is a crucial efficiency step. Before generating any code, the agent first determines if there's actually any work to do.

1. **Prepare for Analysis**: It clones the repository (if it doesn't exist locally) or checks out the checkout\_branch and pulls the latest changes to ensure it's up-to-date.
2. **Run Change Analysis:** The performChangeAnalysis method is called. It finds the last known version of the SRS (stored in .ai-state/srs.txt from a previous run) and gives both the old and new SRS documents to the ChangeAnalysisAgent.
3. **Make a Decision**: The ChangeAnalysisAgent (an LLM) compares the two documents.
   * **If NO changes are found,** it responds with the exact text "No changes detected.". The application prints a success message and exits. No code is generated, and no branches are created.
   * **If changes ARE found**, it generates a human-readable changelog, and the workflow proceeds to the next step.

**Step 3: Setting Up the Workspace**

Now that the agent knows work is required, it prepares the development environment.

1. **Create Feature Branch:** It creates a new, unique Git branch with a timestamp (e.g., feature\_20231027123456). All subsequent work will happen on this isolated branch.
2. **Clean the Workspace:** To ensure a fresh start, it deletes the core source code folders (src, pom.xml, .github) from the new feature branch. This prevents conflicts with stale, old files.
3. **Record Changelog:**  The changelog generated in Step 2 is written to a new file, AI\_CHANGELOG.md, on the feature branch.

**Step 4: The Core AI Code Generation Workflow**

This is where the magic happens. A SequentialAgent orchestrates a series of specialized LLM agents, each passing its output to the next. This entire sequence is wrapped in a retryWithBackoff function to make it resilient to temporary network or model overload errors.

1. **RequirementsAgent**: Takes the raw SRS text and refines it into a structured format that other agents can easily understand (e.g., Feature, Input, Output, Logic). It also generates a one-line commit message.
2. **DependencyAgent**: Analyzes the structured requirements and determines the necessary Maven dependencies (like spring-boot-starter-web or spring-boot-starter-data-jpa).
3. **CodeGenAgent**: This is the main code writer. It takes the structured requirements and generates all the necessary Java source code for a complete Spring Boot application, including controllers, services, repositories, and entities. It adds a // File: ... comment at the top of each code snippet to mark its intended file path.
4. **TestGenAgent**: Takes the Java code from the CodeGenAgent and generates corresponding JUnit 5 test cases, also marking them with // File: ... comments.

**Step 5: Assembling the Project on the File System**

The AI has generated all the necessary text; now, the application builds the actual project structure.

1. **Write Source Code:** The writeClassesToFileSystem method takes the combined output from the CodeGenAgent and TestGenAgent. It intelligently parses this output, using the // File: comments as a guide to create the correct directory structure and write each class to its own .java file. It also cleans up any markdown formatting (` ```
2. **Create pom.xml:** It generates a complete pom.xml file, populating it with the dependencies identified by the DependencyAgent.
3. **Add Configuration:** It adds a standard application.yml for database and server configuration and a ci.yml file to set up a GitHub Actions workflow for continuous integration.
4. **Update README.md**: It intelligently updates the README.md file, inserting a summary of the project, its requirements, and its dependencies into a special <!-- AI-SUMMARY --> block. This preserves any human-written content in the file.

**Step 6: Finalizing and Creating the Pull Request**

The code is now fully generated and sitting on a local feature branch. The final step is to deliver it.

1. **Save State:** The new SRS document is saved to .ai-state/srs.txt, so it can be used as the "old" version for the next run.
2. **Commit and Push:** The application stages all the new and modified files, commits them using the message generated by the RequirementsAgent, and pushes the new feature branch to the remote GitHub repository.
3. **Create Pull Request**: It attempts to use the GitHub CLI (gh) to automatically create a pull request from the new feature branch to the original checkout\_branch.
   * **If gh is installed**, a PR is created, and the application opens the PR link in your browser
   * **If gh is NOT installed**, it gracefully fails, prints a manual link to create the PR, and attempts to open that link in your browser instead.

End

Thank you.