Assignment: Build **Codebase Genius** – an Agentic Code-Documentation System

1 Background and inspiration

In this assignment you will build **Codebase Genius**: an AI-powered, multi-agent system that automatically generates high-quality documentation for any software repository. To help you understand how an agentic application is structured, you will first explore the `byLLM Task Manager` example from Jaseci Labs. This public project provides a fully-working back-end built with the **byLLM** and Streamlit front-end. You will learn how to set up and run the sample project, then apply similar patterns and best-practices when implementing your own code-documentation pipeline.

2 Reference implementation to study

Repository: `Agentic-AI/task_manager/byllm`

2.1 Backend (byLLM)

1. **Clone and navigate to the code.** Run the following commands to clone the repository and move into the task manager directory:

```
git clone https://github.com/jaseci-labs/Agentic-AI.git
cd Agentic-AI/task_manager/byllm
```

1. **Create and activate a virtual environment.** Use `python3 -m venv` to create a `venv` directory and activate it, then set your LLM provider key in a `.env` file. The example project uses OpenAI, so your `.env` file must define an `OPENAI_API_KEY`[1]. Change the code to use Gemini models with gemini API key, if needed.

2. **Install dependencies.** Install required packages from `requirements.txt`[2].

3. **Choose an implementation.** Navigate into one of the versioned folders (`v1`, `v2`, etc.) to select the implementation you wish to run[3]. Both versions have the same functionality but different way of building, use v1 first.

4. **Run the Jac server.** Start the backend by running:

```
jac serve main.jac
```

This command launches a local Jac server exposing the walkers defined in `main.jac`[4]. You can interact with the API using HTTP POST requests.

2.2 Frontend (Streamlit)

1. **Install front-end dependencies.** In the `FE` folder, install requirements with `pip install -r requirements.txt`[6].
2. **Run the Streamlit app.**

3. **Interact with the agent.** The Streamlit interface allows you to chat with the byLLM agent, view tasks, manage sessions and observe how the multi-tool prompt orchestrates different capabilities. Spend time exploring how the front-end communicates with the Jac backend.

3 Learning resources: Jac as your first language

Your assignment must be implemented in **Jac** (JacLang). Before writing any code, study the following resources thoroughly:

Resource	Description
Beginner's guide to Jac	Introduces Jac's core concepts such as nodes, edges, walkers and graphs.
Jac Language Reference	The official specification covering syntax, built-in modules, class definitions, events, APIs, etc. Use this as your authoritative reference when implementing your agents.

The quality of your solution will depend on how well you leverage Jac's unique constructs (walkers, abilities, node types, ability composition, etc.), so make this your primary programming language for the assignment.

4 Your task: build **Codebase Genius**

Now that you understand how a multi-agent Jac application works, you will design and implement **Codebase Genius**, following the high-level specification provided in the attached project description. Codebase Genius is an autonomous system that, given a public GitHub repository URL, produces high-quality markdown documentation of that codebase with clear prose and explanatory diagrams. Your implementation should be optimised for repositories written in Python and Jac but generalisable to other languages. The system will comprise several cooperating agents; each agent handles a specialised part of the workflow.

4.1 System architecture

Implement your solution as a **multi-agent pipeline**, similar to the byLLM example. Your architecture should include at least the following agents (these are suggested roles – feel free to adapt or extend them as long as the functionality is covered):

1. **Code Genius (Supervisor)** – orchestrates the workflow. It receives a GitHub URL, delegates work to subordinate agents, aggregates intermediate results and assembles the final documentation. The Supervisor should decide the order of operations based on repository structure and progress, ensuring that high-impact files are documented first.

2. **Repo Mapper** – clones the repository and produces a high-level map:

3. Build a **file-tree generator** to traverse the repository and return a structured representation of files and folders while ignoring irrelevant directories such as `.git` or `node_modules`.
4. Implement a **readme summariser** that reads `README.md` (or similar entry points) and summarises it into a concise overview. The summary will help the Supervisor plan subsequent analysis.

5. **Code Analyzer** – performs deeper analysis:

6. Parse source files using a parser such as Tree-sitter.
7. Construct a **Code Context Graph (CCG)** capturing relationships between functions, classes and modules. This graph should show which functions call others, inheritance relationships, composition, etc.
8. Provide APIs for the Supervisor or DocGenie to query relationships (e.g. "Which functions call `train_model`?").

9. **DocGenie** – synthesises the final documentation:

10. Convert structured data from the Repo Mapper and Code Analyzer into a well-organised markdown document. Include sections for project overview, installation, usage, and API reference.
11. Ensure the markdown is clear, logically ordered and human-readable. Use headings, bullet points, tables and diagrams judiciously.

You may implement these agents as separate Jac files or as walkers within one file, but they must communicate effectively. Consider using message-passing patterns or shared state for collaboration.

4.2 Workflow

1. **Accept a GitHub URL.** Validate that the repository is reachable and clone it to a temporary directory.
2. **Repository mapping.** The Repo Mapper produces a file-tree representation and README summary. Feed this information to the Supervisor so it can plan which parts of the codebase to analyse first.
3. **High-level planning.** The Supervisor prioritises entry-point files (`main.py`, `app.py`, etc.) and instructs the Code Analyzer to build the CCG for those components.
4. **Iterative analysis.** Analyse code modules iteratively: first high-impact modules, then utility modules (backfill coverage). Use the CCG to traverse relationships and decide next targets.
5. **Documentation generation.** The DocGenie assembles partial results into a final markdown report. It should integrate diagrams and automatically cite relevant functions or classes from the CCG. Save the documentation locally (e.g. under `./outputs/<repo_name>/docs.md`).

6. **Expose an API.** Provide an HTTP interface similar to the byLLM example (Jac server with walkers) that allows a user to supply a repository URL and download the generated documentation.

4.3 Implementation guidelines

- **Leverage Jac features.** Use Jac's node/edge graphs for representing repositories and code relationships. Walkers should traverse the graph to perform analysis and generation.
- **Use external tools responsibly.** For parsing, you may call Python code via Jac's `py_module` or integrate third-party libraries; ensure you document any external dependencies and include installation instructions.
- **Error handling and robustness.** Handle invalid URLs, private repositories, unsupported languages or parsing errors gracefully. Return informative error messages via your API.

4.4 More Info on the CodeBase Genius Refer to this

Click here -> CodeBase Genius

Slide deck -> CodeBase Genius

5 Deliverables

1. **Source code** for all Jac files and any supporting Python modules. Organise your code in a clean, logical directory structure (`agentic_codebase_genius/` or similar).
2. **Setup and run instructions** for both your backend and any front-end UI you create. The instructions must be self-contained and reproducible, similar to the byLLM example (virtual environment setup, dependency installation, environment variables, running the server, etc.).
3. **Markdown documentation generator**: a final script or API that produces the documentation for a repository of your choice (choose a moderately sized Python/Jac repository to demonstrate your system).
4. **Sample output**: provide an example of the generated documentation for the chosen repository. This should include at least one diagram illustrating function or class relationships.
5. **Report** (optional but recommended) summarising your design decisions, challenges encountered, and suggestions for future improvements.

6 Evaluation criteria

Your assignment will be evaluated on the following criteria:

Criterion	Details
Correctness and completeness	Does your system implement all required functionalities (cloning, file-tree mapping, CCG construction, documentation generation,

Criterion	Details
	diagrams, API)? Does it handle both Python and Jac codebases?
Code quality	Clear structure, appropriate use of Jac constructs, well-documented walkers and modules, readability and maintainability.
Documentation quality	Generated markdown should be organised, accurate and easy to follow. Diagrams must aid comprehension.
Instructions	Setup/run instructions must be comprehensive and reproducible. Ensure others can run your solution without guesswork, as in the byLLM example[1][9].
Creativity and extensibility	Bonus points for supporting additional programming languages, adding advanced analyses (e.g. cyclomatic complexity), or building a polished front-end.

7 Submission guidelines

- Host your code in the public Git repository that you submitted for the google form. Include a README describing how to run the system and where to find the generated documentation.
- Provide a video or written walkthrough demonstrating your system on a sample repository.
- Clearly mention any external libraries or APIs used, along with licensing considerations.

8 Tips for success

- Start by thoroughly running the byLLM Task Manager to understand the interplay between Jac code, multi-tool prompting and the Streamlit UI.
- Incrementally build your agents. Begin with the Repo Mapper, then the Code Analyzer, and finally the DocGenie. Test each component on small repositories before integrating them.
- Use version control and commit frequently. Comment your Jac code; other developers will appreciate clear documentation of your walkers and graphs.
- If you get stuck on Jac syntax or features, refer back to the beginner's guide and language reference. Jac has unique concepts; embracing them will make your solution more elegant.

Good luck, and may your agents generate world-class documentation!

---

[1] [2] [3] [4] [5] raw.githubusercontent.com

https://raw.githubusercontent.com/jaseci-labs/Agentic-AI/main/task_manager/byllm/BE/README.md

[6] [7] [8] [9] raw.githubusercontent.com

https://raw.githubusercontent.com/jaseci-labs/Agentic-AI/main/task_manager/byllm/FE/README.md