JutulGPT

Elling Svee (elling.svee@gmail.com)

Summer internship at SINTEF Digital.

August 05, 2025

1. Introduction

1.1. AI agents

An *AI agent* is an autonomous system that perceives its environment, makes decisions, and performs tasks to achieve specific goals. These agents are usually powered by *large language models* (LLMs) that can understand and generate human language. What sets an AI agent apart for a standard LLM is their ability to call external functions, often called *tools*. Tools such as document search, code execution, or API calls extend the agent's abilities beyond simple text generation. This lets them handle more complex tasks, work autonomously, and adapt to changing needs. The recent surge in popularity of AI agents shows that they can be a powerfull utility for developers and researchers. However, most agents are used to generate code for popular, well-documented frameworks and libraries. There is less research on how agents can work with smaller or more specialized packages, and this is an area that needs further exploration.

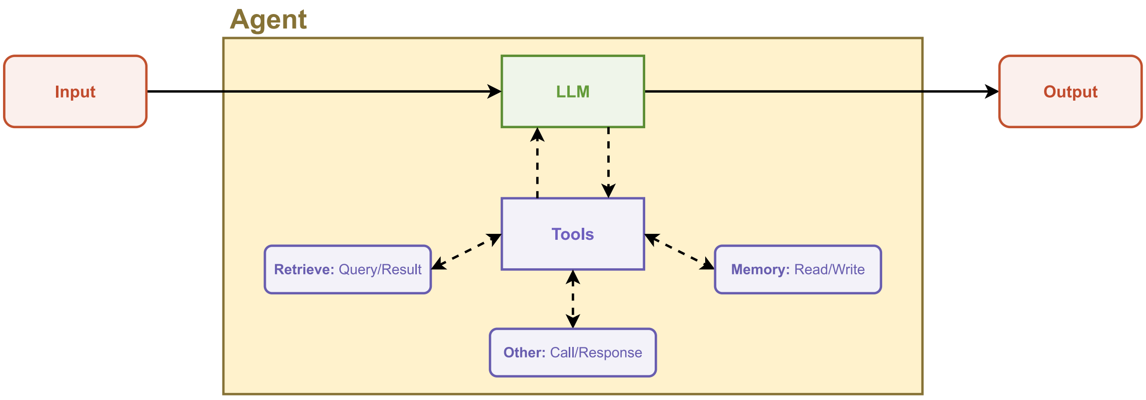

Figure 1: A simple AI agent that uses tools to perform actions and generate output.

1.2. Focus

This work explores and develops AI agents specialized for the JutulDarcy.jl° package. This is a high-performance porous media and reservoir simulator developed by the Applied Computational Science° group at SINTEF Digital. An agent for JutulDarcy can help experts with programming tasks and guide newcomers in using the package. A well-designed agent can reduce manual coding effort by streamlining tasks like setting up simulations and retrieving information from the documentation. Beyond code generation, an agent can also assist with data analysis, visualization, and experimental design.

SINTEF Digital is not the only organization exploring the potential of AI-driven development. Another aspect of my work is therefore to evaluate the benefits and disadvantages of developing custom AI agents for specific packages, instead of relying on pre-made general-purpose systems like GitHub Copilot° or Claude Code°. Custom agents can offer local deployment, better integration, and more control over their behavior. However, maintaining them requires ongoing effort in terms of updates, monitoring and integration with other systems.

Lastly, building an effective agentic system also means understanding how users interact with the AI. This involes investigating what they need, how they prefer to work, and what use-cases matter most. My work has also focused on seeing how the agent can be implemented and integrated into existing workflows.

The full implementation of the JutulGPT agents is available on GitHub (see Table 1). It is open source under the MIT license°, allowing free use and modification by SINTEF and the wider community.

Name	Link
JutulGPT	https://github.com/ellingsvee/JutulGPT°
JutulGPT-GUI	https://github.com/ellingsvee/JutulGPT-GUI/°

Table 1: JutulGPT code repositories

2. Framework

Although the agents are meant to write and evaluate Julia° code, they are implemented in Python° using the LangGraph° framework. LangGraph provides a structured way to define the agent's behavior, manage state and handle user interactions. It is a popular and well-documented library for creating production-ready° AI applications, and is open source under the MIT license. Another benefit is that is supports both local and remote LLMs, allowing for flexibility in deployment. We can therefore use Ollama° to run local models, while still being able to switch to a remote models.

3. Tools

One of the keys to building effective agents is providing a well thought-out set of tools. For the JutulDarcy agent, the main focus has been on tools for retrieving information from the JutulDarcy documentation. For some workflows, it is also necessary to include tools for code evaluation. Inspired by the Cursor and Copilot prompts°, the agent was also given extra tools to make it more autonomous.

3.1. Retrieval

Three retrieval tools are implemented. All are used to gather information based on a query, but each retrieves different types of content. None of the tools access the JutulDarcy source code, and instead focus on documentation and examples. Retrieving source code could be useful for tasks that lack documentation, but was not needed for the current use cases. Currently, the tools retrieve context from the JutulDarcy documentation. Similar methods could be used for other packages like Fimbul.jl°. Its documentation similarly structured, but less extensive. Yet, incorporating the package could require adjusting the retrieval approach.

`grep_search` is the simplest retrieval tool. It performs a keyword search in the JutulDarcy documentation. By default, it returns up to 20 results, but this can be adjusted. It also supports regex and filetype filters, though these are rarely used in practice. The tool is useful for both retrieving file content and getting an overview of which files contain relevant information. The agent often follows up a search by reading the full content of some of the files found.

`retrieve_function_documentation` fetches documentation for specific function names. This is useful for quickly accessing information about Jutul and JutulDarcy functions without searching the entire documentation. It uses the `@doc` macro, so it always provides up-to-date documentation as long as the latest Jutul and JutulDarcy releases are installed.

`retrieve_examples` performs a semantic search to find relevant JutulDarcy examples stored in a vector database. Full examples are embedded, which helps generate code for complete simulations. However, embedding large chunks of text can make results less relevant to specific queries. An alternative would be to embed smaller chunks or use a more targeted retrieval approach.

Currently, `grep_search` and `retrieve_examples` get their context from a documentation folder included in the JutulDarcy repository. This is not sustainable, since the folder must be updated whenever JutulDarcy changes. A better solution would be to retrieve documentation directly from the package, ensuring the agent always has the latest information and removing the need for manual updates.

3.2. Code evaluation

Code evaluation can be implemented either as tools that the agent calls dynamically or as functions that are built directly into the agent's workflow. In this subsection I describe the functions themselves, while referring to Section 5 for details on their use.

The `run_linter` function performs static code analysis to identify potential issues in the generated code. It checks for common programming mistakes, style violations, and other aspects that could affect code quality. This function is implemented using the StaticLint.jl° package, which is also used by the Julia extension° for Visual Studio Code. While static analysis can highlight potential problems, it may not detect all errors or suggest solutions. To address this limitation, the `run_julia_code` function can be used to execute Julia code and return any errors that occur during runtime. At present, this function requires the complete code to be provided, but it could be extended to run only a specific section. This would allow for more efficient testing and debugging, as the agent would not need to re-run the entire code after each change.

A more experimental option is the `execute_terminal_command` tool, which can run arbitrary terminal commands. This can be useful for installing packages, managing files, or running external programs. However, because it can potentially execute harmful commands, it should be used with caution. The current implementation does not include automated safety checks, but it always stops and prompts the user before executing a command. Additional tools could also be developed to improve code evaluation further. For example, running a predefined set of tests could help verify correctness and functionality, while dynamic analysis during runtime could provide information about performance and behavior.

3.3. Environment

For more dynamic and autonomous agents, tools for interacting with the environment can be valuable. I have implemented functions that read from files (`read_from_file`), write to files (`write_to_file`), list the names of files in a directory (`list_files_in_directory`), and return the current working directory (`get_working_directory`). Although not implemented, another useful capability would be access to selected `git` commands. For example, the `git diff` command could be used to compare the current code with a previous version, allowing the agent to track its own modifications. This could be especially helpful when generating large amounts of code in one session.

4. Human in the loop

Providing the model with more tools can lead to more complex behavior. This often results in the model acting more autonomously, which can be both an advantage and a challenge. Smaller language models in particular tend to struggle with complex tasks. In my experience, they sometimes choose the wrong tools, get stuck on minor errors, or generate code that does not meet the user's requirements. One way to address these issues is to include human feedback in the agent's decision-making process, a method commonly referred to as human-in-the-loop (HITL). This approach allows users to provide input and corrections, ensuring that the agent's actions align with their expertise.

I have implemented HITL-interactions in the workflow and in many of the available tools. For example, when the agent retrieves information from the JutulDarcy documentation or examples, the user can review and filter the retrieved content. When the agent generates code based on user requirements, it presents the code to the user for review before execution. This gives the user an opportunity to make manual adjustments or suggest improvements.

Because the need for HITL-interactions depends on both the user and the task, I have made it possible to disable them before initializing the agent. How to do this is explained in the appendix. Another possible enhancement would be to provide human interaction as a dedicated tool, allowing the agent to request user input dynamically at any point in the workflow.

5. Workflows

Having provided the LLM with tools and implemented HITL-interactions, the agent still rarely generates correct code on the first try. The best option is therefore to design a *workflow*, which is specified through a sequence of actions that the agent should take. This is commonly visualized through a directed graph, where the nodes represents actions the agent performs, and the edges represent the flow of information between these actions.

When implementing the agent, I tested several different workflows, each with its own strengths and weaknesses that i describe in this section. For most tasks I recommend using the *Evaluator-optimizer* workflow described in Section 5.3, or the *Augmented LLM* workflow described in Section 5.4. Yet, for more specific tasks the others should also be considered.

I recommend reading the Building Effective Agents article by Anthropic for a more in-depth discussion of the different workflows. It provides valuable insights into how to design and implement effective agents using LLMs, and gives concrete tips on when to use the different approaches. The appendix also details how to use tools effectively, and can be worth looking into.

5.1. Linear

The linear workflow is the simplest option, and works by having the agent follow a fixed sequence of steps. For setting up a simulation, this sequence typically involves retrieving documentation, generating the simulation code, evaluating the code, and finally analyzing the results of the evaluation. As illustrated in Figure 2, the process returns to the first step if the code fails, allowing the agent to correct the error.

This approach works reasonably well for most tasks and is a good choice for smaller models. However, it can be too rigid for tasks that require extensive debugging. It also requires the workflow designer to think carefully about how information is passed from one step to the next.

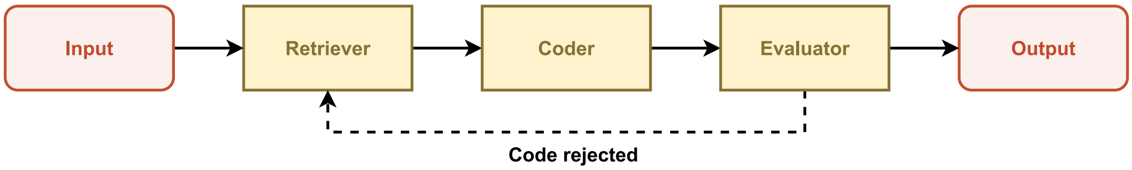

Figure 2: Linear workflow

5.2. Orchestrator-synthesizer

Setting up a simulation in JutulDarcy often involves tasks such as discretizing the domain, defining the system, and running the simulation. The orchestrator-synthesizer workflow, illustrated in Figure 3, can be useful for this type of work. In this approach, an orchestrator divides the overall task into smaller subtasks, separate agents handle each subtask, and a synthesizer combines the resulting code into a complete solution.

The main advantage of this approach is that assigning smaller subtasks to different agents prevents overwhelming the language model with too much information at once. This is especially beneficial for smaller models that may struggle with complex problems. The orchestrator can also manage dependencies between subtasks, ensuring they are completed in the correct order and that the output from one step is correctly incorporated into the next.

The main challenge with this workflow lies in the number of decisions the designer must make. It is necessary to determine whether the retrieved context should be shared between agents or kept separate, how to manage task dependencies, and how to merge the generated code into a coherent script. Furthermore, not all tasks can be easily divided into smaller subtasks, making the workflow unsuitable in some situations. Because of these limitations, I decided not to use the orchestrator-synthesizer workflow in the final JutulDarcy agent. However, I still consider it a promising approach that is worth exploring further in the future.

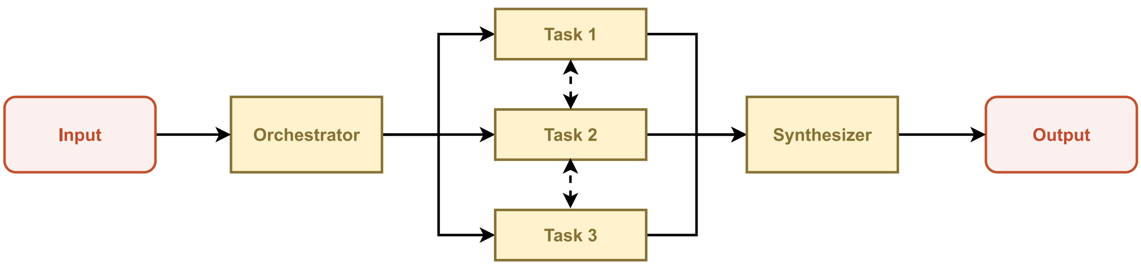

Figure 3: Orchestrator-synthesizer workflow

5.3. Evaluator-optimizer

The evaluator-optimizer workflow is simpler than the orchestrator-synthesizer, and uses an iterative process to generate and refine code. It is similar to the linear workflow, but combines the retrieval and generation steps into a single step. As shown in Figure 4, the agent first generates code based on the user's requirements while retrieving relevant information from the documentation. It then evaluates the generated code and, if errors are found, passes them back to the code generator for correction.

When used with sufficiently capable language models, this approach is very effective. Giving the agent greater flexibility in its use of tools makes it better at fixing errors and responding to user feedback. In the case of an error, the agent typically retrieves information directly related to that specific problem, which often leads to faster and more accurate fixes than when relying solely on context related to the broader task. For these reasons, the evaluator-optimizer workflow is my preferred choice for the JutulDarcy agent, and is used in one of the two implemented agents.

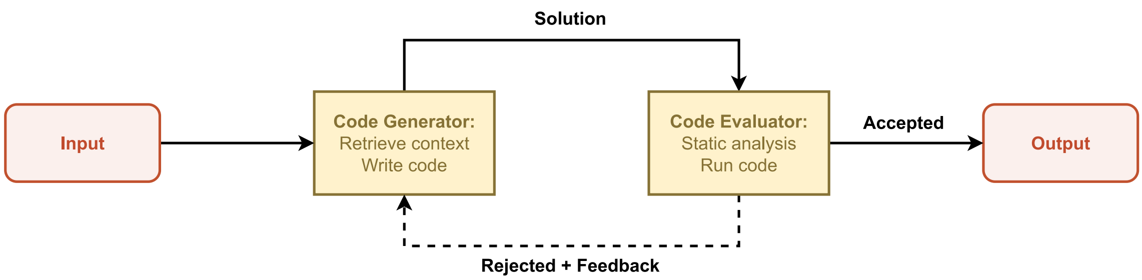

Figure 4: Evaluator-optimizer workflow

5.4. Augmented LLM

As the LLMs become more and more capable, we have seen a shift towards a simpler and more dynamic workflow. For the *augmented LLM*, the language model is provided with tools and a set of instructions but is not bound to a fixed sequence of steps. Instead, the model acts as both the code generator and evaluator, calling tools as needed. This approach, illustrated in Figure 5, allows the LLM to interact directly with the user, retrieve documentation, generate code according to user requirements, and evaluate the resulting code.

This workflow requires a carefully designed prompt to guide the model's behavior, along with access to a large set of tools. Smaller models often struggle with this approach, but it can be very effective when using large LLMs. It also provides the most *Copilot*-like experience, as it allows the agent to operate autonomously. This type of agent is implemented in JutulGPT, and can be run as an alternative to the evaluator-optimizer workflow.

For future work, I recommend reviewing the System prompts and models of AI tools° repository for examples of how the augmented LLM workflow is applied in systems such as Cursor and GitHub Copilot. The repository also contains a comprehensive list of tools that could serve as inspiration when designing agents.

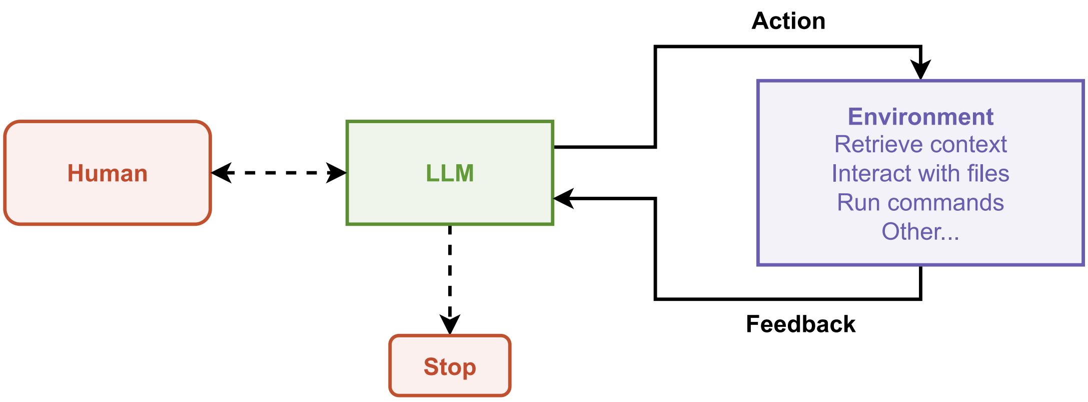

Figure 5: Augmented LLM workflow

6. Interfaces

I have implemented two interfaces for interacting with the agent. The first is a *command-line interface* (CLI) that allows interaction through a terminal, while the second is a web-based *graphical user interface* (GUI) that offers a more user-friendly experience. For most users, the most convenient option would likely be a Visual Studio Code extension, as this would allow them to work with the agent directly from within their code editor. Such an extension would provide a seamless experience but is not yet implemented and should be considered for future work.

6.1. CLI

The CLI is the fastest and most flexible way to interact with the agent. Inspired by Claude Code° and Gemini CLI°, the agent can be run directly from the terminal. It uses the Python-based rich° library to format and render Markdown in the terminal, making the responses easy to read and interact with. The CLI can also open a specified text editor for a more seamless editing experience. It supports streaming output so that users can see responses as they are generated.

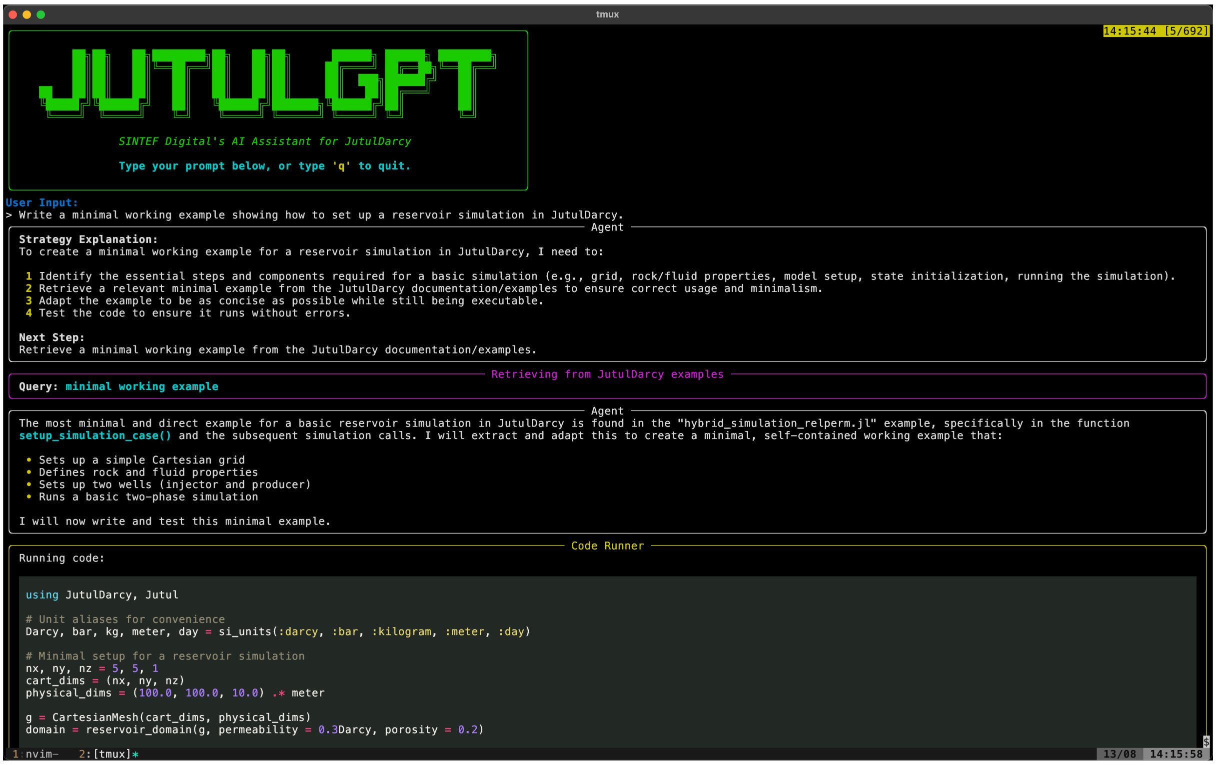

Figure 6: Example of the CLI for interacting with the JutulGPT agent.

6.2. GUI

The web-based GUI, shown in Figure 7, offers the same features as the CLI but with a ChatGPT-like interface. The backend is largely the same as the CLI, although separate functions are required for human interactions. The GUI was built on top of the open-source LangGraph Agent Chat UI repository, with modifications to support the configuration settings specific to JutulGPT.

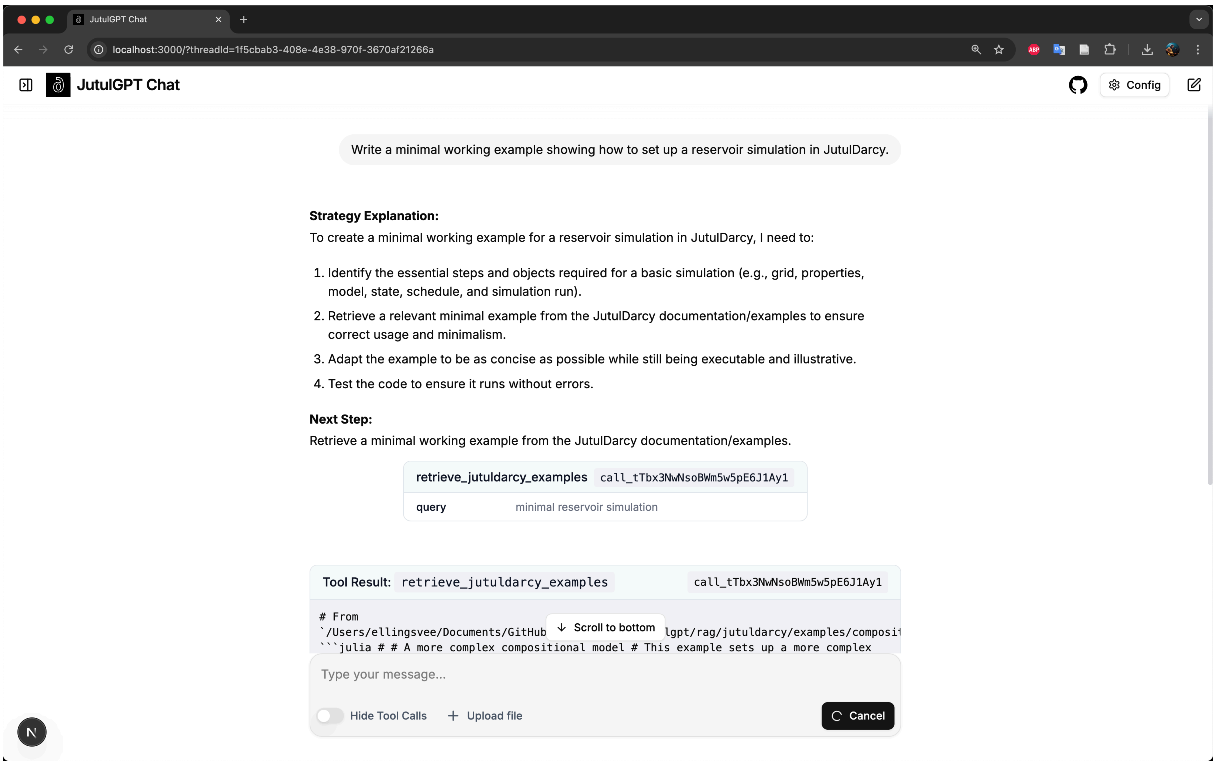

Figure 7: Example of the UI for interacting with the JutulGPT agent.

7. Future work

The current implementation of the JutulDarcy agent provides a solid foundation, but there are several opportunities for improvement and extension.

At present, the agent can set up simulations, but more complex scenarios may require additional user input or guidance. Future work could focus on improving the agent's ability to handle such scenarios more effectively. This could involve expanding the agent's knowledge base, enhancing the retrieval tools, and refining the feedback from the code evaluator. It would also be valuable to explore how the agent should approach code generation for tasks not covered by existing examples, as some of these cases remain difficult for the agents to handle.

Selecting the most appropriate workflow depends on both the complexity of the task and the capabilities of the language model. Simpler and more linear workflows tend to work better for smaller models, while larger models often benefit from a more dynamic setup. The agents could be improved by more carefully matching models to specific tasks, and designing specialized workflows for particular model sizes and use cases.

Another important area for improvement is the interaction between Python and Julia. It would be highly beneficial to keep Julia running continuously in the background, avoiding the need to restart it each time code is executed. This would also improve iterative coding workflows, such as those described in Section 5.2 and Section 5.3. For example, rather than rerunning the full code after each change, it would be possible to run only the modified sections. The JuliaCall package offers a way to enable more direct interaction between Python and Julia, although its current limitations in multithreading support make it harder to use alongside the GUI.

Finally, the CLI-interface could be improved by allowing users to change settings at runtime. For instance, users should be able to switch the model or adjust the available tools without restarting the agent. This would make the system more flexible and adaptable to different use cases. Currently, the CLI supports changing some settings before initialization, but this capability has not yet been added to the GUI. The LangGraph RunnableConfig° supports this feature, but it is not yet implemented.

I look forward to seeing how this work develops in the future and hope to continue contributing to it. For questions or suggestions, please feel free to create an issue on GitHub or contact me by email. I am also open to collaborating with others who share an interest in this area of research and development.

Appendix: A very brief overview of the code

The settings for the agent is set in the `src/jutuldarcy/configuration.py` file. This contains some static settings, and a `BaseConfiguration` class of settings that can be configured at runtime. See the `README.md` file for specifics. However, some important settings and variables to note are:

- `cli_mode` : If set to `True` , the agent will run in CLI mode, while `False` will run in GUI mode.
- `LLM_MODEL_NAME` and `EMBEDDING_MODEL_NAME` : The default model for the LLMs and embedding models. Note that this can be overridden in the `BaseConfiguration` .
- `HumanInteraction` : A class that defines how the agent interacts with the user. Here, the different human-in-the-loop interactions can individually be disabled or enabled.

I have implemented agents for the evaluator-optimizer and the augmented LLM workflows. The evaluator-optimizer agent is found in `src/jutulgpt/agents/agent.py` , while the augmented LLM is found in `src/jutulgpt/agents/autonomous_agent.py` . Both inherit from form the `BaseAgent` class in `src/jutulgpt/agents/agent_base.py` , which provides most of the core functionality. In the CLI-mode, the agents can be invoked by calling the associated `run()` function in `BaseAgent` , like what is done in `examples/agent.py` and `examples/autonomous_agent.py` . For running in the GUI, we need to specify the location of the complied workflow-graph in the `langgraph.json` file. How do to this should be self explanatory.

The rest of the project should be relatively easy to understand based on their names and locations. Yet, one thing to note is that now is the full JutulDarcy- and Fimbul-documentation placed in `src/jutulgpt/rag/` . As mentioned previously, this should be moved to a more appropriate location, and not be a part of the JutulGPT repository. Tests are set up to be implemented using `pytest` . However, due to the rapid changes to the project, I have not yet implemented any tests. They can be places in the `tests/` folder, and can be run using `uv run pytest` .