Multi-Agent System Supporting Software Development Processes

Jakub Dulas

June 2025

1 Introduction

This report presents an analysis and the results of a research and development project focused on creating a **multi-agent system capable of autonomously generating code based on a given project scope**. The motivation behind this project stems from the increasing demand for automated solutions in software development to enhance efficiency, reduce manual effort, and accelerate the delivery of functional software. By leveraging the power of large language models (LLMs) and advanced agent architectures, I aim to demonstrate the feasibility and effectiveness of an AI-driven approach to code generation.

The system was designed using a **ReAct agent architecture** and implemented in **Python** using the **LangGraph** library for agent orchestration and **Streamlit** for the graphical user interface (GUI). A key element of the system was the application of various reasoning strategies within the code generation agent (Coder) to evaluate their impact on the quality and efficiency of the generated software.

2 System Architecture

The system comprises four main types of agents, collaborating to achieve project goals:

- **Project Manager:** Receives the project scope from the user, plans tasks, divides them into sprints, and assigns them to the appropriate agents. It coordinates the workflow, iterating through tasks and ensuring their chronological execution.
- **Researcher:** Responsible for gathering necessary information, exploring technologies, and identifying solutions required for task completion.
- **Coder:** The primary agent responsible for code generation. It features three reasoning versions, allowing for experimentation with different approaches to breaking down larger tasks into smaller subtasks and executing them sequentially:
  - **ReAct (No Reasoning):** A basic approach where the agent operates on observation, thought, and action.
  - **ReAct + CoT (Chain-of-Thought Reasoning):** The agent generates internal "thoughts" that help it in reasoning and planning subsequent steps.
  - **ReAct + ToT (Tree-of-Thought Reasoning):** The agent explores multiple reasoning paths, evaluating them and selecting the best one.
- **Documenter:** Creates documentation for the generated code, including function descriptions, usage instructions, and other essential materials.

2.1 Operational Flow

The system's operational flow is as follows:

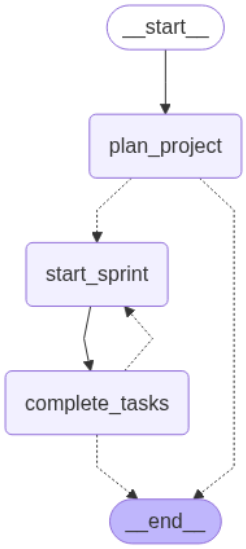

Figure 1: Overall System Flow Diagram

1. **User defines project scope:** The user provides a general description of the project to be completed.
2. **Project Manager plans (plan_project):** The Project Manager analyzes the scope, breaks it down into smaller, manageable tasks, and organizes them into sprints. Each task is assigned to the appropriate agent. This planning phase ensures a structured approach to complex projects, moving from a high-level goal to actionable steps.
3. **Task iteration and execution (start_sprint, complete_tasks):** The Project Manager iterates through the planned tasks within a sprint. Depending on the task type, the relevant agent (Researcher, Coder, Documenter) is activated and executes its task chronologically. This iterative process allows for continuous progress and adjustment.
4. **Agent-specific execution:**
    - **ReAct Agent Architecture:** All agents (Project Manager, Researcher, Coder, Documenter) generally follow a ReAct architecture, where an `agent` node performs actions, observes results, and then decides on the next step, potentially utilizing `tools`. This loop continues until a task is completed or the agent decides to `__end__`.

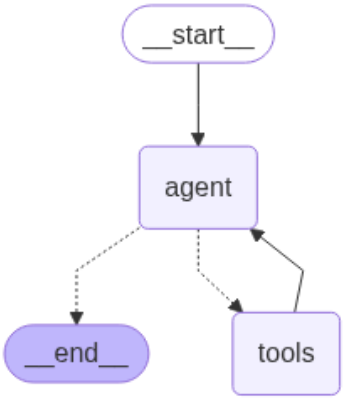

Figure 2: ReAct Agent Architecture

- **Coder Agent Flow:** The Coder agent's core loop involves a `reason` phase, followed by `next_step`, and `llm_node`. It can utilize `tools` to assist in code generation or fall back to a

`zero_state` which leads to termination if no further action is possible. The decision within the `reason` node determines which enhanced reasoning strategy (CoT or ToT) is employed.

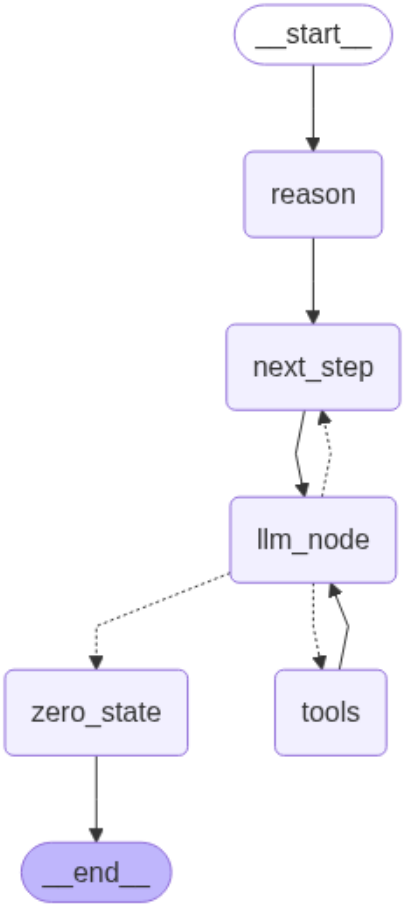

Figure 3: Coder Agent Flow

- **Chain-of-Thought (CoT) Reasoning:** When CoT reasoning is active, the `reasoning_node` generates internal thoughts that guide its actions. This allows the agent to break down a problem into a sequence of intermediate steps, improving the coherence and correctness of its outputs.

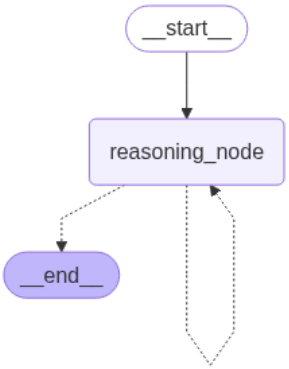

Figure 4: Chain-of-Thought (CoT) Reasoning Flow

- **Tree-of-Thought (ToT) Reasoning:** The ToT approach involves a more complex reasoning process. After creating a goal (`create_goal`), the `reasoning_node` actively builds a tree of possible thoughts and actions (`build_tree`), scores and prunes less promising paths

(`score_and_prune`), and then cycles back to the reasoning node to continue exploration or to `format_response` and `__end__`. This allows for a more exhaustive and robust exploration of solutions.

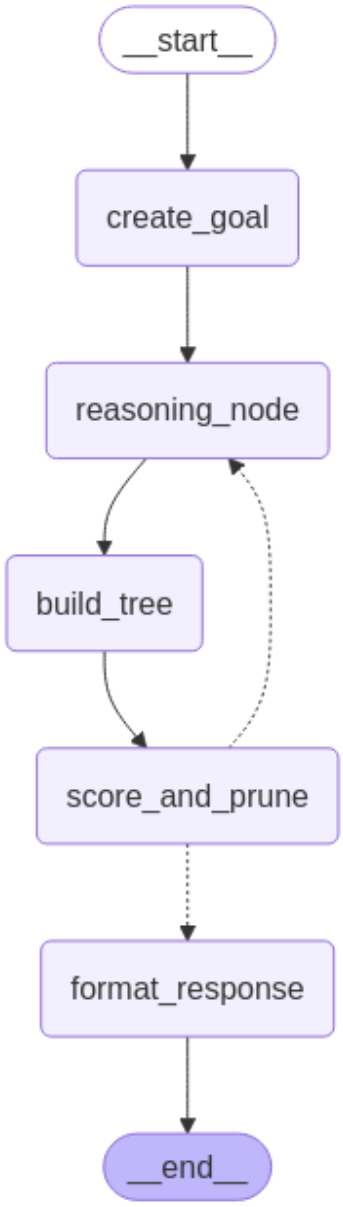

Figure 5: Tree-of-Thought (ToT) Reasoning Flow

5. **Code generation and verification:** The Coder generates code according to requirements. Depending on the chosen Coder version, different reasoning strategies are employed to optimize the process.
6. **Documentation:** The Documenter creates comprehensive documentation for the generated code.

3 Test Projects

Six defined projects, categorized into three types, were used to test the system, allowing for a comprehensive evaluation of its capabilities. These projects were chosen to cover a range of complexities and common software development paradigms, from simple command-line tools to more involved web applications and games, ensuring a thorough assessment of the agent's code generation abilities across different scenarios.

3.1 Simple Console Applications

- **Project 1: Temperature Converter (Single-file):** A simple temperature unit converter (°C, °F, K) with a CLI interface and error handling. This project primarily tests basic logical operations, input/output handling, and error management within a single file, serving as a baseline for fundamental code generation.
- **Project 2: CLI Note Manager (Multi-file):** A command-line note manager allowing note creation, deletion, and viewing. It requires a modular structure with multiple files (`main.py`, `note.py`, `storage.py`, `utils.py`). This project assesses the agent's ability to design and implement a multi-file structure, manage data persistence (even if simple), and handle user interactions in a CLI environment.

3.2 Games

- **Project 3: Tic Tac Toe:** A classic 3x3 game for the terminal (2 players or player vs. AI) with ASCII board rendering and win/draw detection. This project tests algorithmic thinking for game logic, state management, and basic AI implementation (if chosen).
- **Project 4: Snake Game with GUI:** A classic Snake game with a Pygame GUI, keyboard controls, collision detection, and visual board/score display. This is a significantly more complex project, evaluating the agent's proficiency in GUI programming, event handling, game loop implementation, and rendering.

3.3 Web Applications

- **Project 5: TODO List (Backend + simple frontend):** A web application for task management with a REST API (Flask, Django, or FastAPI) and a simple frontend (HTML+JS). This project assesses full-stack development capabilities, including API design, database interaction (even if simulated or in-memory for simplicity), and basic frontend integration.
- **Project 6: Weather App (frontend querying external API):** A web application displaying weather data for a selected city, fetching data from a public API (e.g., OpenWeatherMap) and dynamically rendering the information. This project focuses on API integration, asynchronous operations, and dynamic UI updates, reflecting common modern web development tasks.

4 Measurement Metrics

To assess the performance and quality of the generated code, the following metrics were used. These metrics were chosen to provide a comprehensive view, covering aspects from development efficiency to code quality and functional correctness, thereby allowing for a thorough comparison between the different reasoning strategies.

- **Time of generation (s):** The time taken for the agent to generate the code. This is a critical efficiency metric, indicating how quickly the system can produce a solution.
- **Number of lines of code:** The total count of code lines. While not a direct quality metric, it can indicate verbosity or conciseness.
- **Number of used tokens:** The count of tokens consumed by the language model. This directly correlates with computational cost and can indicate the verbosity of the model's internal reasoning or output.
- **Number of lines of code after changes:** The count of code lines after any manual corrections. This helps in understanding the degree of post-generation effort required.
- **Levenshtein distance:** The Levenshtein distance from the "ideal" solution (closer to 0 is better). This metric quantifies the edit distance between the generated code and a pre-defined correct solution, serving as a direct measure of functional correctness and similarity.

- **Cyclomatic complexity (avg):** The average cyclomatic complexity of the code, measuring control flow complexity. Lower values generally indicate simpler, more maintainable code. **Code duplication (%):** The percentage of duplicated code. High duplication suggests poor code design and maintainability issues.
- **Lint errors (Python):** The number of errors reported by a Python linter (PEP 8). This indicates adherence to coding standards and best practices.
- **Notes:** Additional remarks concerning issues or specific characteristics of the generated code, providing qualitative insights not captured by quantitative metrics.

5 Results and Analysis

The results for the three Coder versions—**No Reasoning (ReAct)**, **CoT Reasoning (ReAct + CoT)**, and **ToT Reasoning (ReAct + ToT)**—are presented below. All tests were conducted using the `gpt-4.1` model.

5.1 v0.2 - CoT Reasoning

Table 1: Results for CoT Reasoning

Metric	Proj 1	Proj 2	Proj 3	Proj 4	Proj 5	Proj 6	Notes
Time of generation (s)	171.98	612.75	459.72	625.31	860.55	385.27	
Number of lines of code	128	377	400	292	657	252	
Number of used tokens	102095	314406	237500	324657	445496	207828	
LOC after changes	128	377	400	292	678	252	
Levenshtein distance	0	0	0	0	293	146	
Cyclomatic complexity (avg)	2.2	2.6	2.9	2.2	1.4	2	
Code duplication (%)	0	0	0	0	0	0	
Lint errors (Python)	0	4	1	0	1	N/A	
Additional Notes for Projects							
P5: Added scripts to html, removed exports P6: Added API key, removed exports							

CoT Reasoning Analysis: The model with CoT reasoning demonstrated **relatively long generation times**, especially for more complex projects (P2, P4, P5). Nevertheless, for simple tasks (P1, P2, P3, P4), the code was **functionally correct** (Levenshtein distance = 0). For P5 and P6, the system generated code requiring **manual corrections**, primarily related to frontend configuration (adding scripts to HTML, removing exports) and API keys. Low average cyclomatic complexity indicates **well-structured code**, and the number of linting errors was minimal. This approach effectively balanced increased processing with higher reliability for most tasks, although complex integrations sometimes required human oversight.

5.2 v1.1 - No Reasoning

Table 2: Results for No Reasoning

Metric	Proj 1	Proj 2	Proj 3	Proj 4	Proj 5	Proj 6	Notes
Time of generation (s)	92.25	167.72	450.09	200.06	287.23	196.37	
Number of lines of code	77	188	372	227	313	243	
Number of used tokens	45590	85116	234158	95994	171589	112006	
LOC after changes	N/A	191	N/A	227	322	238	
Levenshtein distance	0	61	0	31	140	142	
Cyclomatic complexity (avg)	5.25	3.2	3.65	2.44	1.81	N/A	
Code duplication (%)	0%	0%	0%	0%	0%	0%	
Lint errors (Python)	1	0	0	0	0	N/A	
Additional Notes for Projects							
P2: Function to validate input was badly used							
P3: Agent added ability to select O or X for Human vs AI game							
P4: Created dir 'snake_game' which is useless							
P5: CORS Errors							

No Reasoning Analysis: The version without additional reasoning was **significantly faster** in code generation and consumed **fewer tokens**. However, this resulted in **lower code correctness** for more complex projects (P2, P4, P5, P6), reflected by a higher Levenshtein distance. For example, in P2, the input validation function was improperly used, and P5 encountered **CORS errors**. In some cases (P4), **unnecessary directories were created**. Cyclomatic complexity varied, and the number of linting errors was generally low. This approach proved most effective for **very simple and precisely defined tasks**, where the overhead of complex reasoning is not beneficial.

5.3 v1.2 - ToT Reasoning (Updated)

Table 3: Results for ToT Reasoning (Updated)

Metric	Proj 1	Proj 2	Proj 3	Proj 4	Proj 5	Proj 6	Notes
Time of generation (s)	300.82	562.42	743.00	707.95	699.62	552.51	
Number of lines of code	91	255	809	801	690	443	
Number of used tokens	208949	441787	711946	661526	545036	393723	
LOC after changes	N/A	255	N/A	N/A	690	436	
Levenshtein distance	0	4	0	0	23	575	
Cyclomatic complexity (avg)	3.8	2.23	3.07	2.34	1.5	N/A	
Code duplication (%)	0%	0%	0%	0%	0%	0%	
Lint errors (Python)	1	5	5	0	1	N/A	
Additional Notes for Projects							
P5: No ability to remove tasks on frontend. Needed to fix CORS (set to all hosts)							

ToT Reasoning Analysis: The ToT Reasoning strategy continued to show the **longest generation times** and **highest token consumption** among the methods, aligning with its iterative and tree-like reasoning. While it performed well for Projects 1, 3, and 4 with a Levenshtein distance of 0, indicating **perfect functional correctness**, Projects 2 and 5 still had minor issues, such as the lack of task deletion functionality on the frontend in P5 and the necessity for manual CORS configuration. Notably, Project 6 showed a **very high Levenshtein distance (575)**, suggesting significant functional discrepancies or incompleteness, which highlights a key area for improvement. The larger number of code lines in Projects 3, 4, and 5 might imply more verbose or comprehensive solutions. Lint errors were present but manageable. This inconsistency for highly complex and integrated projects suggests that while ToT improves reasoning, it may still be sensitive to specific problem domains or the complexity of external integrations.

6 Comparison of Reasoning Strategies

Table 4: Comparison of Reasoning Strategies

Metric	No Reasoning (ReAct)	CoT Reasoning (ReAct + CoT)	ToT Reasoning (ReAct + ToT)
Generation Time	Fastest	Medium	Slowest
Code Correctness	Lowest (often requires fixes)	Medium (good for simple, needs fixes for complex)	Highest (good for *some* complex, but inconsistent for others)
Code Complexity	Varied	Low/Medium	Low/Medium
Token Usage	Lowest	Medium	Highest
Stability/Consistency	Lowest	Medium	Mixed (High for some, low for others)

Conclusions from Comparison:

- **No Reasoning (ReAct):** This approach is the **fastest** and consumes the **fewest tokens**. However, it tends to generate **lower-quality code** that often requires manual intervention, especially for complex projects. It's best suited for very simple, well-defined tasks where speed is paramount, and correctness can be easily verified manually.
- **CoT Reasoning (ReAct + CoT):** Offers **better code quality** than "No Reasoning," striking a good balance between generation time and token consumption. It's a solid choice for **moderately complex projects**, providing a good trade-off between speed and reliability.
- **ToT Reasoning (ReAct + ToT):** Despite having the **longest generation times** and **highest token consumption**, ToT Reasoning often generates **highly correct solutions for complex problems** (e.g., Project 3 and 4). However, the results for Project 6 show that its performance can be **inconsistent across different complex projects**, sometimes leading to significant functional errors. This highlights that while ToT has the potential for robust solutions through its iterative reasoning, it might still struggle with specific task complexities or nuances, indicating areas for further refinement. Its strength lies in deep, structured problem-solving, but this comes at a higher resource cost and is not a guaranteed panacea for all complex scenarios.

7 Final Conclusions

The project to create a multi-agent autonomous code generation system has demonstrated that a multi-agent approach with ReAct agents is **promising**. The different reasoning strategies within the Coder agent have a **significant impact on the efficiency and quality of the generated code**. The choice of the appropriate strategy should depend on the **complexity of the project and the priorities** (e.g., generation speed vs. code quality). For simpler projects, the "No Reasoning" approach might suffice, while more complex projects benefit from CoT or ToT, albeit with increased resource consumption and potential inconsistencies in very challenging domains.

Future work could involve further optimization of reasoning strategies, the introduction of automated code verification and testing mechanisms, and the expansion of other agents' roles (e.g., a debugging agent capable of identifying and fixing errors). Developing these areas could significantly contribute to increasing the autonomy and effectiveness of such systems, moving closer to truly self-improving and self-correcting software development agents.

8 References

The multi-agent system and its orchestration heavily leverage concepts and libraries from the LangChain ecosystem, particularly `LangGraph` for defining and executing agentic workflows as state machines. The agent architectures, including ReAct, Chain-of-Thought (CoT), and Tree-of-Thought (ToT), are based on established research in large language model reasoning and related implementations:

- **ReAct:** "ReAct: Synergizing Reasoning and Acting in Language Models" by Shunyu Yao et al. (2022). This paradigm combines reasoning traces with task-specific actions to enable powerful problem-solving.
- **Chain-of-Thought (CoT) Overview:** IBM Think: Chain-of-Thoughts. This article provides an overview of the Chain-of-Thought reasoning technique.
- **Tree-of-Thought (ToT) Overview:** IBM Think: Tree-of-Thoughts. This article introduces the Tree-of-Thought reasoning method.
- **CoT and ToT in AI Agents:** MonsterAPI Blog: Chain-of-Thought (CoT) and Tree-of-Thought (ToT) Reasoning Models in AI Agents. Further discussion on these reasoning models in the context of AI agents.
- **MetaGPT:** GitHub: FoundationAgents/MetaGPT. A multi-agent framework that inspires collaborative agent design.