CMPE 492

Project Title

Ahmet Emre Şafak

Ege Ekşi

Advisor:

Atay Özgövde

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Broad Impact

Large Language Models (LLMs) heralded significant changes in artificial intelligence and software development. Before that, these Large Language Models were predominantly used for tasks on the natural language front: translation, summarization, and text generation. Now, the capability of LLMs has shot up notches higher in complexity and nuance with which the models can handle the task. In this regard, this term project deals with multiagent LLM-based software organizations that have immense social impact and that ought to be formed responsibly with the needful ethical considerations. The goal is to understand how multiagent LLMs can autonomously create complex applications, augment human capabilities, deal with complex challenges, and reshape various domains, adhering to principles of fairness, transparency, and accountability.

- **Enhanced Collaboration and Problem-Solving:** Multi-agent LLM systems can autonomously generate sophisticated software code, perform debugging, and engage in high-level software design. This partnership offers immense potential for accelerating software development, reducing errors, and optimizing decision-making processes.
- **Streamlined Workflow Automation:** By delegating programming tasks to multi-agent LLM systems, organizations can increase efficiency and liberate human resources for higher-order work. LLM-driven automation can find applications in software development, debugging, and high-level design, streamlining workflows and reducing operational costs.
- **Personalized and Adaptive Experiences:** Multi-agent LLMs can power highly tailored software solutions, providing personalized guidance and support in various domains. Such systems could revolutionize how software is developed and maintained.

- **Democratization of AI Capabilities:** Well-designed multi-agent LLM frameworks can lower barriers to entry for AI development. By providing pre-configured agents and modular architectures, these frameworks enable developers without extensive AI expertise to harness the power of LLMs for innovative applications, broadening the pool of those who can use AI as a tool.

1.2. Ethical Considerations

The far-reaching potential of multi-agent LLM applications necessitates careful ethical considerations to ensure their responsible development and deployment:

- **Bias and Fairness:** LLMs are trained on vast datasets that may contain inherent biases. It's crucial to proactively identify and mitigate these biases within multi-agent systems to prevent harmful outcomes. Strategies include dataset auditing, debiasing techniques, and continuous monitoring of system outputs.
- **Transparency and Explainability:** Users must have a reasonable degree of understanding of how multi-agent LLM systems work. This includes clarity around their decision-making processes and limitations. Explainable AI methods must be integrated into the design.
- **Accountability:** Clear lines of accountability are necessary for multi-agent LLM applications. This includes determining responsibility for potential harms, establishing redress mechanisms, and ensuring compliance with ethical standards.
- **Privacy and Data Security:** The collection, storage, and use of data by multi-agent LLM systems must adhere to rigorous privacy standards. Security measures are paramount to protect sensitive information.
- **Human-AI Partnership:** The relationship between humans and AI agents must be carefully considered to avoid over-reliance on AI outputs and to ensure that AI agents augment and empower human capabilities rather than replace human judgment.

2. PROJECT DEFINITION AND PLANNING

2.1. Project Definition

Project Title: LLM-Based Multi-Agent Software Organization

Project Overview:

This project leverages the advanced capabilities of LLMs, focusing on the concept of multi-agent software organizations. The core objective is to demonstrate how multiple LLMs can autonomously create complex software applications. The project involved:

- **Design and Development of Multi-Agent LLM Applications:** Utilizing frameworks like LangGraph, LangChain, and LangSmith, multiple LLM-based agents were developed to interact with the local file system and accomplish software tasks efficiently. The main architecture used was a Planner-Replanner-Programmer multi-agent system.
- **Benchmarking and Comparative Evaluation:** Different OpenAI models (GPT-4o, GPT-4-turbo, GPT-3.5) were tested for their performance in developing software applications, including games like 2048, Tetris, Snake, and a custom game with given requirements.
- **Analysis and Insights:** Interpreted the experimental results to identify strengths and weaknesses of different LLM models and drew conclusions about their performance in software development.

Project Goals:

- Gain a deeper understanding of how multi-agent LLM systems can autonomously create complex software applications.

- Provide insights to guide the selection of optimal LLM models for specific software development tasks.
- Contribute to the knowledge base on LLM-based multi-agent systems by offering empirical comparisons of different approaches.

Scope:

- The project focused on a set of multi-agent LLM design paradigms, primarily in the context of software development and game creation.

2.2. Project Planning

2.2.1. Project Time and Resource Estimation

Timeline:

February - March:

- Literature Review & Project Proposal
- Design of initial multi-agent LLM architectures

April - May:

- Implementation of multi-agent LLM systems
- Dataset selection and preparation
- Design and development of experimental benchmarks

June:

- Experimentation and comparison
- Data analysis and interpretation

- Final Report Writing

Estimated Effort Distribution:

The effort distribution is approximate and might change depending on the complexity of the project.

- **Literature Review:** 20%
- **Design of Multi-Agent Systems:** 30%
- **Implementation:** 25%
- **Experimentation & Evaluation:** 15%
- **Report Writing:** 10%

- **Hardware:**
  - No specific hardware tools were required.
- **Software:**
  - Programming languages: Python
  - LLM libraries/frameworks: HuggingFace , LangChain , LangSmith
  - Multi-agent libraries: LangGraph
- **Data:**
  - The experiments used for this project that included games like 2048, Tetris, Snake, and a custom-designed game with specific requirements.
- **Literature:**
  - MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework [1]

2.2.2. Success Criteria

Success for this project was measured by the following criteria:

- **Comparative Insights:** The project successfully generates clear comparisons

between different LLM models, highlighting their effectiveness in software development.

- **Empirical Evidence:** The experimental results provide robust evidence supporting the conclusions about LLM performance differences.
- **Analysis Depth:** The analysis demonstrates a nuanced understanding of the factors influencing multi-agent LLM performance, going beyond surface-level observations.
- **Practical Guidance:** The project findings offer actionable insights or recommendations for the selection or optimization of LLM models.
- **Clarity:** The final report is well-written and communicates the research process, results, and implications effectively.

2.2.3. Risk Analysis

This project carries several potential risks that need to be acknowledged and mitigated:

- **Dataset Limitations:** The quality and representativeness of the datasets used for experimentation can significantly influence the validity of the results. Biased or incomplete datasets may lead to misleading conclusions about multi-agent LLM performance.
- **Computational Resource Constraints:** Developing and experimenting with multiple multi-agent LLM systems can be computationally expensive.
- **LLM Black Box Problem:** The inherent complexity of LLM models can make it difficult to fully explain the decision-making processes within multi-agent systems. This might hinder the ability to pinpoint the root causes of performance differences.

3. RELATED WORK

Multiagent applications with LLMs generally narrate an emergent track of the field and focus on improving collaboration and decision-making by endowing agents with collective intelligence. The systems differ from a single-agent system in that the focus is on diverse agent profiles, inter-agent interactions, and the form of collective decision-making. This can help address more dynamic and complex tasks in this domain through the collaboration of multiple agents, each with distinctive rules of strategy and behavior and interacting in communication with one another.

A significant endeavor in the research of multi-agent LLM applications is consensus-seeking, where the agents negotiate to reach a joint decision or value. This is the technique by which the agents independently generate the strategies for reaching consensus without direct instructions regarding which strategies to follow, which fits perfectly with the ambitious goals of these systems. Strategies identified through experiments include the average strategy, in which all agents use the average state of all agents to adjust their state; the suggestible strategy, in which an agent takes the state of another agent, and this is a sign of cooperation; and the stubborn strategy, in which the agents do not adjust their state, considering the others will orient by themselves. In this manner, such systems effectively reduce the randomness or the hallucinations of LLMs.

The multiagent aspect of LLM systems architecture encompasses agents-environment interface, agent profiling, agent communication, and agent capability acquisition. The agents-environment interface specifies the distinctive operational environment in which an agent operates and through which agents interact with the operational environment—physical, sandboxed, or nonexistent—depending on the application. Agent profiling defines the agents' roles and actions or skills tailored in the light of specific goals. Communication among agents is essential to support the collective intelligence characterized by the paradigms of cooperation, debate, and competition. Finally, in agent

capability acquisition, agents learn and adapt to complex problems through feedback, which may be drawn from the environment, agent-agent interaction, human input, and, in some cases, no feedback at all.

More to the point, these systems are influenced by the network topology, which specifies how information is disseminated and the efficacy of a consensus-formation process. On the other hand, in negotiation dynamics, the topology can affect how rapidly a fully connected network can achieve a consensus relative to a sparsely connected network.

The concept of multi-agent systems based on this LLM approach holds great promise in developing collaborative problem-solving and decision-making systems across a broad spectrum of applications, from software and games development to financial markets and in modeling social behaviors. Their growth is channeled to bring about more adaptive, efficient, and intelligent systems capable of entering complicated interactions and solutions. [2]

4. METHODOLOGY

This section outlines the methodology employed to compare the performance of various multi-agent LLM application designs.

4.1. Multi-agent LLM Application Design and Development

We created a set of distinct multi-agent LLM applications. This involved defining the communication protocols, interaction strategies, and task allocation mechanisms for each application design. The specific design encompassed:

- **Planner-Replanner-Programmer Architecture:** This architecture involves three types of agents:
  - **Planner Agent:** Responsible for creating a detailed plan for completing the given task and delivering the generated plan to the programmer agent. This agent does not have access to any tools.
  - **Programmer Agent:** Responsible for deciding which steps will be completed during each iteration and executing those steps. This agent has access to tools such as ‘write_to_file‘, ‘read_from_file‘, and ‘execute_python_file‘.
  - **Re-planner Agent:** Responsible for assessing the previous plan and steps taken by the programmer agent, creating a new plan if necessary, and delivering it back to the programmer agent. This agent does not have access to any tools.

We also experimented with a hierarchical multi-agent architecture but ultimately settled on the Planner-Replanner-Programmer architecture due to its effectiveness.

4.2. Performance Evaluation

To assess the effectiveness of each multi-agent LLM application design, we manually tested and compared the success rates, error proportions, and tokens used. This evaluation focused on the following criteria:

- **Task Completion:** The success rate of each LLM model in completing the assigned tasks, which included the development of games like 2048, Tetris, Snake , and a custom game with given requirements.
- **Error Proportions:** The proportion of errors encountered during the task execution. These errors were categorized into incorrect implementation, loops between agents, laziness, and incorrect or missing tool usage.
- **Token Cost:** The average number of tokens spent by each LLM model on a given task. By calculating the tokens, we aimed to determine the average cost per execution per model.

The manual testing and comparative analysis provided a comprehensive assessment of the capabilities and limitations of different OpenAI models (GPT-4o, GPT-4-turbo, GPT-3.5) in a multi-agent setup. The evaluation process allowed us to identify the strengths and weaknesses of each model.

5. REQUIREMENTS SPECIFICATION

5.1. Project Goal

- To provide a comprehensive analysis of the Planner-Replanner-Programmer multi-agent architecture powered by large language models.
- Identify the strengths, weaknesses, and suitability of various LLM models (e.g., GPT-4o, GPT-4-turbo, GPT-3.5) for different tasks within this architecture.
- Establish best practices and guidelines for developing effective multi-agent LLM applications using this architecture.

5.2. Project Scope

- **Planner-Replanner-Programmer Architecture:** Focuses on systems where tasks are divided among Planner, Replanner, and Programmer agents.
  - Planner Agent: Creates a detailed plan for completing the given task.
  - Replanner Agent: Assesses the plan and results, providing feedback and adjustments.
  - Programmer Agent: Executes the steps and makes decisions about the necessity of re-planning or user feedback.
- **LLM Selection:** The project considered prominent LLMs suitable for multi-agent applications (e.g., GPT-4o, GPT-4-turbo, GPT-3.5).
- **Use Cases:** The analysis used the following use cases as benchmarks:
  - Collaborative Software Development: Multiple LLMs work together to generate code for applications (e.g., games like 2048, Tetris, Snake).
  - Task Execution: LLMs complete complex programming tasks with feedback loops between agents.

5.3. Requirements

- **Functional Requirements**
  - The system shall implement the Planner-Replanner-Programmer multi-agent architecture.
  - The system shall use standardized tasks and metrics to evaluate the performance of different LLM models within this architecture.
  - The system shall establish metrics to measure success rates, error proportions, and token usage.
  - The system shall provide in-depth qualitative analysis of communication patterns, emergent behaviors, and overall system effectiveness.
- **Non-Functional Requirements**
  - The system shall process and analyze outputs of different LLM models in a reasonable timeframe.
  - The system shall provide transparent reasoning and decision-making processes for interpretability.

5.4. Deliverables

- Comparative Analysis Report: Provide a detailed written report documenting the findings, comparing the strengths and weaknesses of different LLM models within the Planner-Replanner-Programmer architecture under various use cases.
- Code Repository: Maintain a well-organized code repository containing example implementations, testing suites, and benchmarking tools.

6. DESIGN

6.1. Information Structure

Our project consists of multiple LLM agents that are responsible for each other. Thus, it is not a conventional application which connects to database and does CRUD applications. Henceforth, we do not have any ER diagrams. However, we will add the finite-state machine of our application to inform you about our project. More about the information that is exchanged between the agents will be discussed in the next section.

6.2. Information Flow

The information flow within the project flows through a series of interactions between the planner agent, the software programmer agent and the replanner agent. Firstly, the planner agent receives a software task and processes it to create a detailed plan. This plan includes all steps and dependencies required to complete the task. Once the plan is generated it is passed on to the software programmer agent. This agent interprets the plan, creates and executes the necessary code files, ensuring that each step is followed.

As the software programmer agent progresses with the implementation, it may encounter scenarios that require additional clarification or modifications to the initial plan. In such cases, the programmer agent either requests feedback from the user or consults to the replanner agent. The replanner agent evaluates the current state of the project, taking into account any previous information or changes, and revises the plan accordingly. This revised plan is then sent back to the software programmer agent, ensuring a continuous and adaptive workflow. This iterative cycle of planning, execution, and re-planning enables our project to remain flexible and responsive to evolving requirements, thereby optimizing the overall development process.

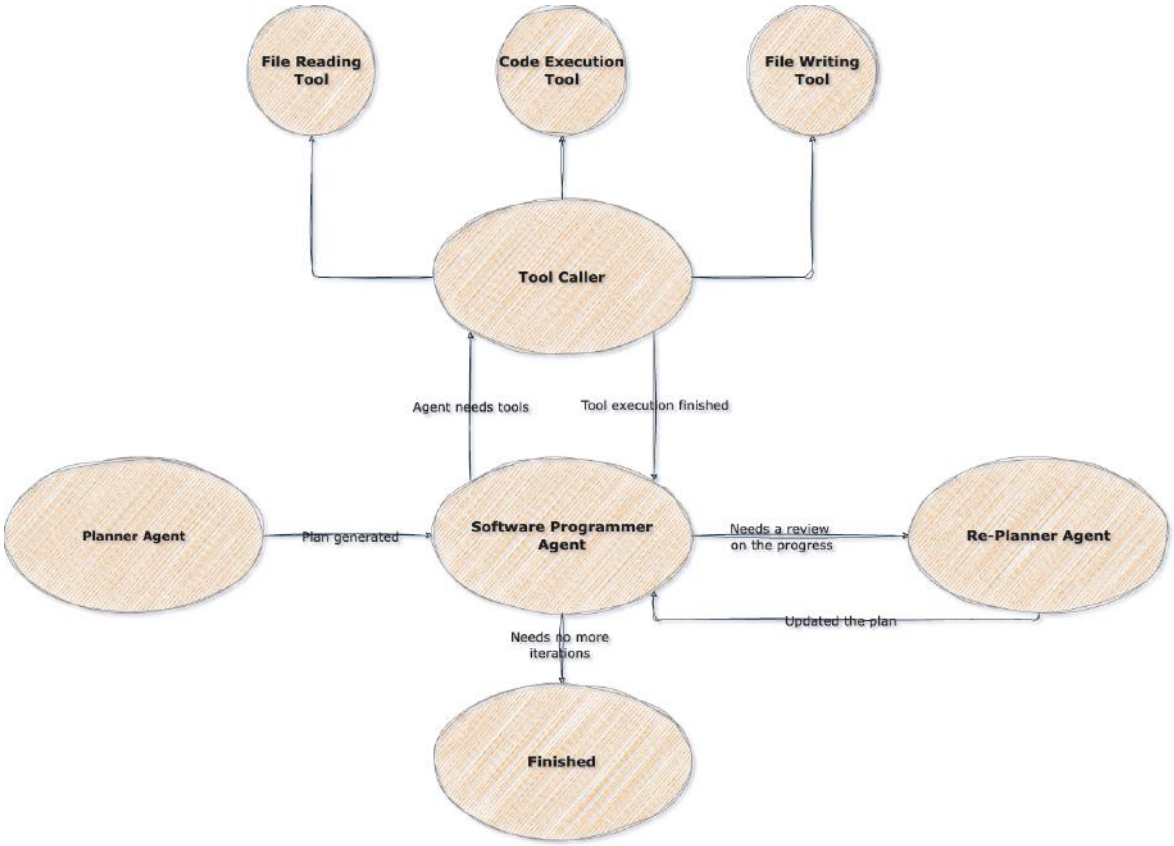

Figure 6.1. Finite State Machine of the Virtual Software Organization

6.3. System Design

The system design of the project revolves around efficient management of state by LLM agents. Each agents are responsible for specific tasks to ensure a smooth and dynamic workflow. The core of state management is encapsulated within structured class definition shown below. This class, PlanExecute, is a TypedDict that defines the essential elements of our project's state. It includes an input string, a plan which is a list of steps, past_steps that track the history of actions using annotated tuples combined with the operator.add, and a response string. The agents interact with this class to maintain and update the state throughout the project's lifecycle, enabling seamless transitions between planning, execution, and replanning phases.

```
class PlanExecute(TypedDict):
    input: str
    plan: List[str]
    past_steps: Annotated[List[Tuple], operator.add]
    response: str
```

This design ensures that all state-related information is centralized and managed effectively by the agents, allowing for a cohesive and adaptable development process.

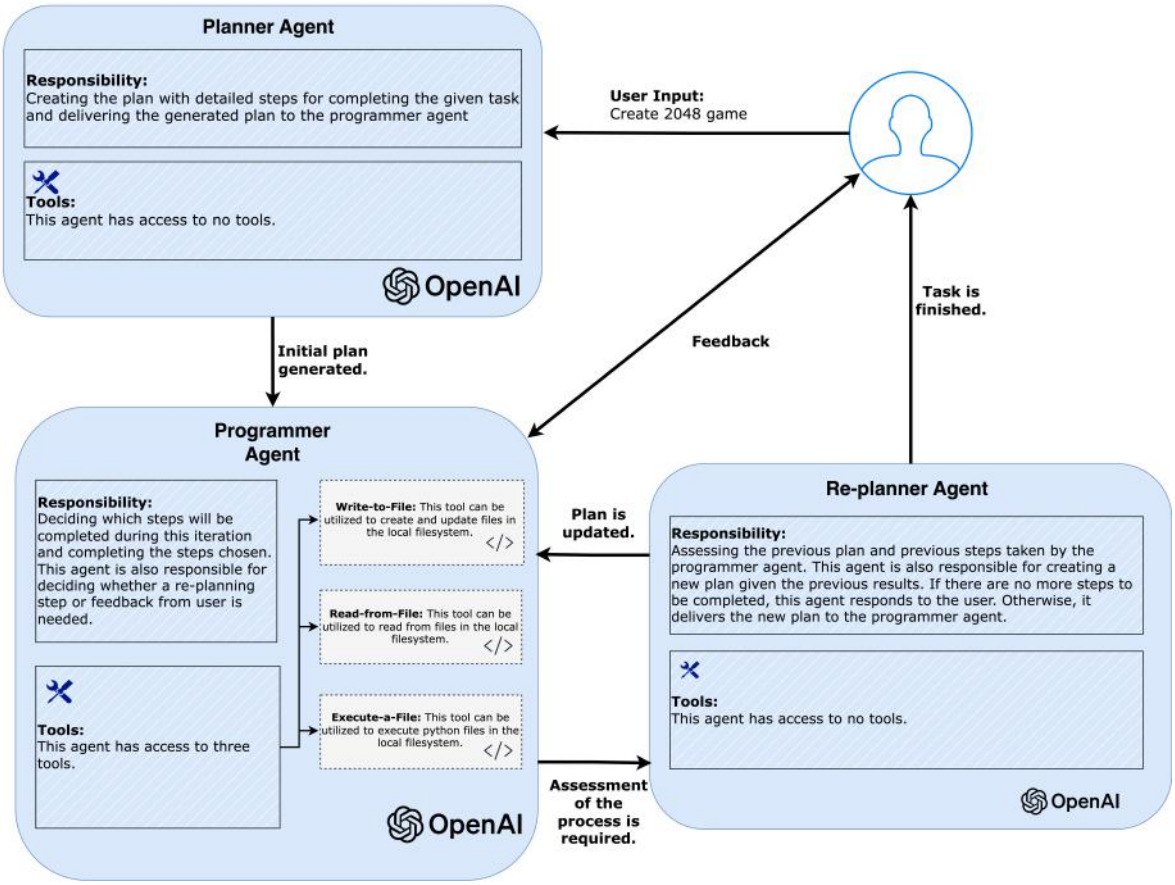

Figure 6.2. System Design

The schema above displays the full system design.

7. IMPLEMENTATION AND TESTING

7.1. Implementation

The work of our project is implemented upon libraries of LangGraph and LangChain, which are supposed to mediate between our code and the OpenAI LLM model. LangGraph and LangChain are state-of-the-art libraries built on top of language models. LangGraph itself creates a way of building and executing complex workflows for multi-agent interaction with language models. This allows developers to specify the interaction and dependencies of different agents in a way that is more structured and scalable. LangChain, on the other hand, focuses on enabling smooth communication of language models with external tools or APIs; it provides utilities to harness language models to work with external systems in fetching, processing, and executing some task or fact of interest. LangGraph, joined with LangChain, represents a comprehensive solution for enabling developers to build complex language model-driven applications, manage efficient development workflows, and integrate them with other tools.

Moreover, leveraging OpenAI's capabilities in making tool calls, we have developed three different agents: the planner agent, the software programmer agent, and the replanner agent. Each of these agents performs specialized tasks that a project workflow requires. We further developed several custom functions to enhance the functionality and performance of these agents. The functions 'write_to_file': writes the given content correctly to the given file names and take care of making all the relevant directories. The function read_from_file enables agents to read the contents of files, returning an empty string if the file is not readable. The execute_python_file functions actually run Python files, capture both the output and errors, and then prompt the user after receiving further input. Essentially, this will work in the effectiveness of the agents by enabling them to manage files and run code effectively.

Besides these, we have applied some advanced techniques of prompt engineering,

such as Chains of Thought, to make the agents more efficient in their performance and decision-making. These tool functionalities will be applied to let the agents implement an operational system that can manage the files correctly and execute the code accordingly. Advanced prompt engineering techniques, such as Chains of Thought, are used to optimize performance before implementation by the agents in an operational system that is robust and adaptive enough for any situation. Also, we have added in-context examples for improving the agents' performance. As communication between the agents is essential, we have ensured that the examples are in context and fine-tuned the prompts to increase performance.

7.2. Testing

In particular, language models are indeterministic, so it is almost impossible to apply traditional unit testing, as even two runs with the same input would give different outputs. Unit tests for our agents have not been implemented for this reason. Instead, we focused on a few other methods to ensure the robustness and reliability of our application. Parameters controlling language generation are temperature, seed, and top_p. The temperature parameter makes a model more deterministic the lower it goes and, on the contrary, higher to more unpredictable results. It can enforce the seed for reproducibility of results since it pins down the state of the random number generator. The parameter top_p, or nucleus sampling, bounds the choice of the model to a subset of the most likely outcomes increasing the diversification of the text being generated. As such, we test this with temperature 0, seed 423, and top_p value 1 in the middle position of this replica diversity spectrum.

Model name	Temperature	Top_p	Seed
GPT-4o	0	1	423
GPT-4-Turbo	0	1	423
GPT-3.5	0	1	423

Figure 7.1. Config of the LLMs

As it is possible to control different critical values of our application, like flow management, token counts, cost, and time, LangSmith allows us to have helpful analytics regarding the model's performance and thereby get an optimal design of our processes and resources control. We used LangSmith to be capable of observing and regulating the operational parameters of our agents so that the execution would be efficient and cost-effective. Experiments with our application were made to manually test its performance with various tasks, including 2048, Tetris, and Snake games. The same tests were completed in each one of the language models, namely GPT-4o, GPT-4-turbo, and GPT-3.5, to compare its performance. The tasks were executed ten times to obtain an exact estimation of the capabilities of the application. Thus, we tested the effectiveness and reliability of our agents in various manual testing scenarios and got a robust and adaptive system.

We used this tool, called LangSmith, to control aspects within our application: flow management, token counts, cost, and time. LangSmith showed beneficial details on how the model was performing, which helped us to provide our process optimizations and resource management with the highest quality. We used LangSmith to check and balance the operation parameters for every agent in action for effective and cost-related implementation.

As such, we also performed multiple deep manual tests with the application to check the performance in different tasks, like trying to generate the games 2048, Tetris, and Snake. The same tests were run on the other language models, including the GPT-4o, GPT-4-turbo, and GPT-3.5, for their comparison in performance. We did run the

application for each of the tasks ten times to compare the difference in performance. The manual testing approach has allowed us to ensure the effectiveness and reliability of our agents across a wide range of scenarios toward a robust and adaptive system.

8. Results

We evaluated the success rates of our application by testing it on four main tasks: Snake Game, 2048 game, Tetris game, and a custom game generation. Each task was executed 10 times with different LLMs to calculate the success rates. The Snake game proved to be the easiest to implement, followed by the 2048 game, the custom game, and finally the Tetris game. GPT-4-o consistently outperformed the other LLMs, demonstrating the highest success rates across all tasks. In contrast, GPT-3.5 performed poorly, often failing to generate any viable output for most repetitions. More detailed information can be derived from the chart below.

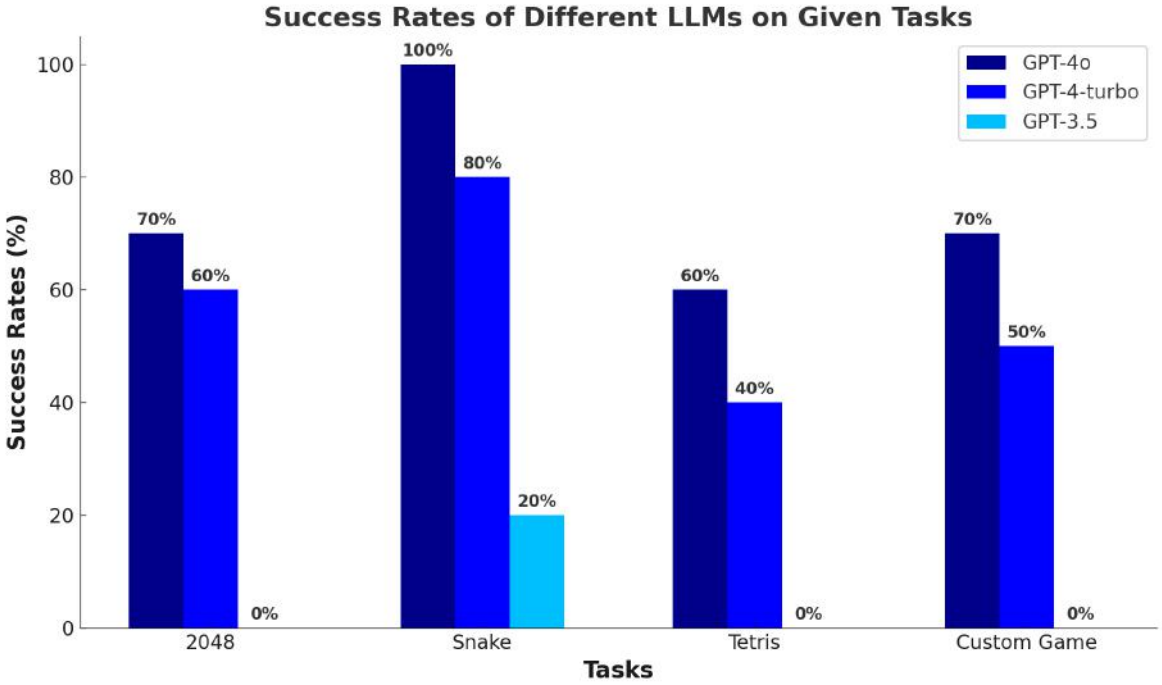

Figure 8.1. Success Rates of Different LLMs

We also analyzed the average number of tokens spent per execution by different LLMs. The number of tokens spent correlates with the difficulty of the tasks. GPT-4-o used the most tokens, followed by GPT-4-turbo and GPT-3.5. Despite GPT-4-o consuming more tokens, it remains more cost-effective compared to the GPT-4-turbo model, making it the most optimal choice. The average number of tokens was calculated by repeating each task 10 times and averaging the tokens spent. More details about the token usage can be found in the chart below.

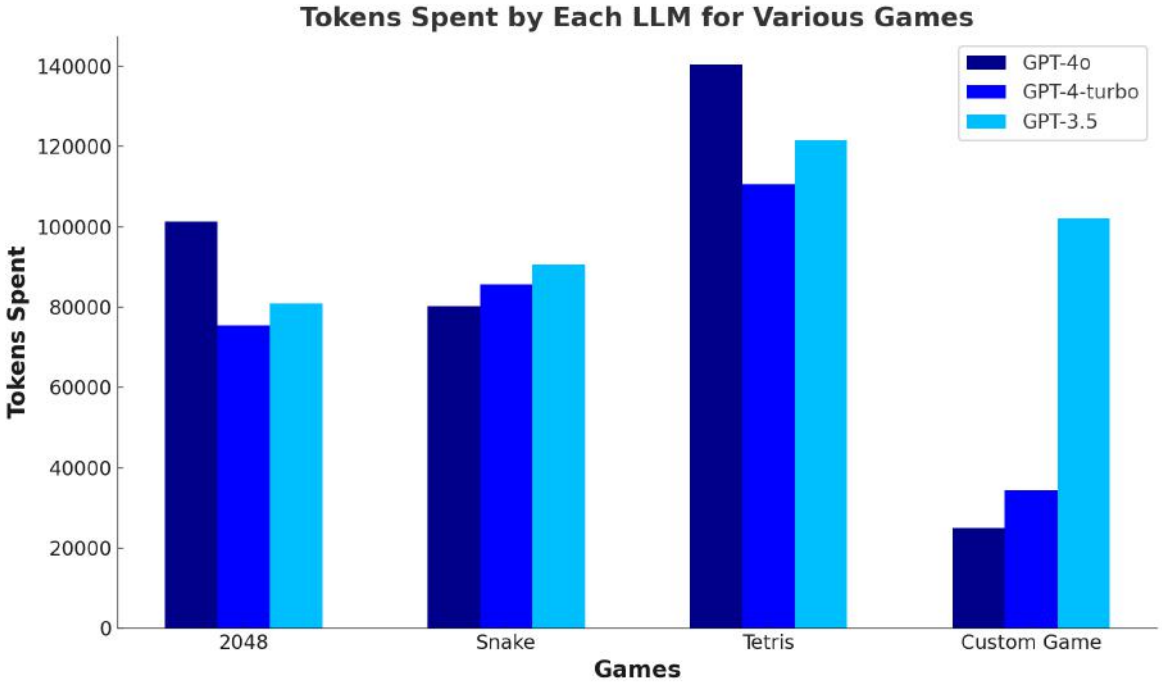

Figure 8.2. Average number of tokens spent per LLM

Additionally, we categorized the reasons for failure into several categories. Incorrect implementation refers to generating code that does not produce the correct results. Loop between agents indicates a failure in communication between agents, leading to infinite loops. Missing tool usage, observed only with GPT-3.5, refers to the agents' inability to use tools to generate files. Laziness is characterized by the inclusion of TODO statements instead of actual implementations. The proportions of these errors are illustrated in the following chart.

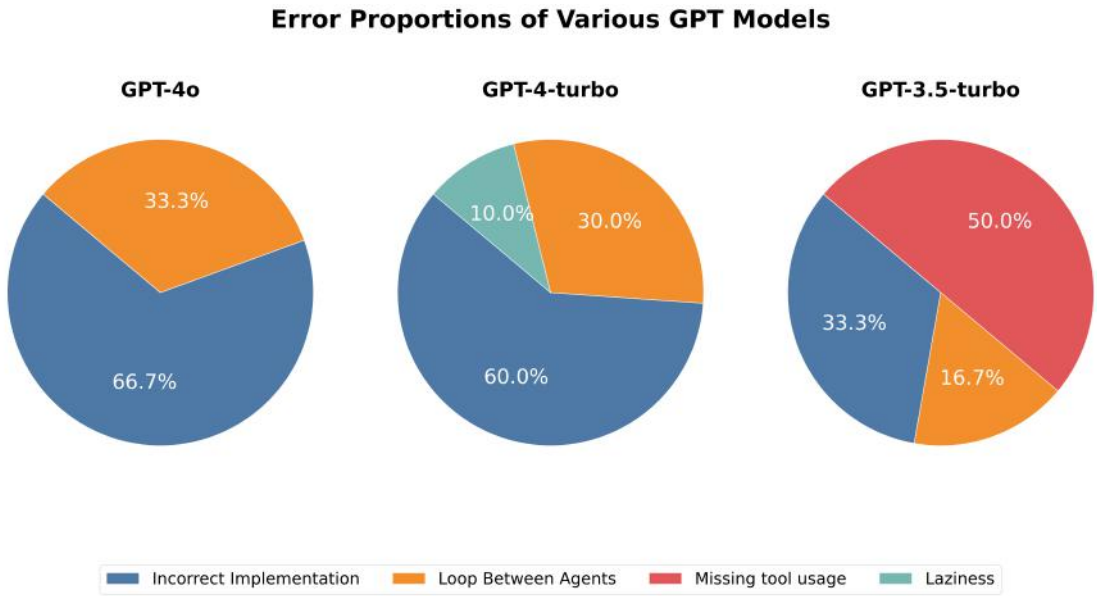

Figure 8.3. Error proportions

9. Conclusion

The results are pretty promising. It shows that the Large Language Models can potentially create virtual software organizations. Our LLM-based agent's application was successful at generating software programs that consist of multiple files. Thus, with the increasing capabilities of LLMs, this possibility becomes more real. Such models have associated low costs and are expected to increase even more with technological improvement.This area, hence, enables the usage of LLMs for many applications of software development. More to the point, it is possible to design more robust multiagent architectures for further competitive and complex tasks. The games the virtual software organization produces, such as 2048, Tetris, and Snake, are not even comparable. The findings of our study indicate that LLMs, which are increasing in capability toward being more sophisticated, will do more complex work. This development of capability will bring about innovation and efficiency in ways that one can only now imagine. Multi-agent systems powered by LLMs hold the potential of a considerable role in designing software in the future. We expect multi-agent LLM applications to become mainstream shortly. These will probably become less expensive, much more efficient, and highly successful systems that drive the future of systems development. In that sense, the auspicious results of our project echo some of the promise of these technologies. Potential applications are extended, paving the path for future innovation in virtual software organizations as continued research and implementation advances are made.

REFERENCES

1. Hong, S., M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu and J. Schmidhuber, "MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework", , 2023.

2. Guo, T., X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest and X. Zhang, "Large Language Model based Multi-Agents: A Survey of Progress and Challenges", , 2024.