Advancing Agentic Systems: Dynamic Task Decomposition, Tool Integration and Evaluation using Novel Metrics and Dataset

Adrian Garrett Gabriel **Alaa Alameer Ahmad** **Shankar Kumar Jeyakumar**

CARIAD SE
Major-Hirst-Straße 7
38442 Wolfsburg
marek.mayer@cariad.technology

Abstract

The rapid advancements in Large Language Models (LLMs) and their enhanced reasoning capabilities are opening new avenues for dynamic, context-aware task decomposition, and automated tool selection. These developments lay the groundwork for sophisticated autonomous agentic systems powered by LLMs, which hold significant potential for process automation across various industries. These systems demonstrate remarkable abilities in performing complex tasks, interacting with external systems to augment LLMs' knowledge, and executing actions autonomously. To address the challenges and harness the opportunities presented by these advances, this paper makes three key contributions.

- We propose an advanced agentic framework designed to autonomously process multi-hop user queries by dynamically generating and executing task graphs, selecting appropriate tools, and adapting to real-time changes in task requirements or tool availability.
- We introduce novel evaluation metrics tailored for assessing agentic frameworks across diverse domains and tasks, namely Node F1 Score, Structural Similarity Index, and Tool F1 Score.
- We develop a specialized dataset based on the AsyncHow dataset to enable in-depth analysis of agentic behavior across varying task complexities.

Our findings demonstrate that asynchronous and dynamic task graph decomposition significantly improves system responsiveness and scalability, particularly in handling complex, multi-step tasks. Through detailed analysis, we observe that structural and node-level metrics are more critical in sequential tasks, whereas tool-related metrics dominate in parallel tasks. In particular, the Structural Similarity Index (SSI) emerged as the most significant predictor of performance in sequential tasks, while Tool F1 Score proved essential in parallel tasks. These findings highlight the need for balanced evaluation methods that capture both structural and operational aspects of agentic systems. Our specialized dataset enables comprehensive evaluation of these behaviors, providing valuable insights into improving overall system performance, with the importance of both structural and tool-related metrics validated through empirical analysis and statistical testing.

The evaluation of agentic systems presents unique challenges due to the intricate relationships between task execution, tool usage, and goal achievement. Our evaluation framework, validated through empirical analysis, offers valuable insights for improving the adaptability and reliability of agentic systems in dynamic environments.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

1 Introduction

Recent advances in LLMs have catalyzed the development of sophisticated agentic systems capable of automating multistep tasks, interacting with external systems, and adapting to changing contexts [1, 2]. These systems are promising in industries requiring autonomous workflow processing and tool integration. Despite their potential, LLM-based systems face limitations in industrial settings due to lack of training on proprietary data and the challenges of fine-tuning. Fine-tuning LLMs for each business use case requires costly, labor-intensive dataset collection and processing. In such contexts, LLMs are limited in their ability to manage real-time decision making in dynamic environments.

A scalable alternative to fine-tuning LLMs for each use case is the development of agentic systems that dynamically integrate external tools. Augmenting LLMs with tools allows these systems to handle complex queries and adapt without constant retraining. Agentic frameworks enable LLMs to decompose tasks into smaller sub-tasks, whose significance is highlighted in [3], select the appropriate tools for each task, and adjust to real-time changes in tool availability.

One of the first frameworks that allowed LLMs to interact with external tools is LangChain Project, which paved the way for more sophisticated agentic platforms such as BabyAGI Project and AutoGen [4]. These systems represent important steps toward the realization of autonomous AI agents, but are often constrained by high latency, limited adaptability, and insufficient support for dynamic tool integration. Moreover, current systems lack comprehensive evaluation methods that fully capture the complexity of task graph generation and tool selection, limiting their scalability and reliability in industrial applications.

To address these challenges, our work presents a framework[1] that advances the capabilities of traditional agentic systems. Our framework integrates real-time tool selection, dynamic task graph generation, and an evaluation mechanism to assess agentic behavior across diverse tasks and domains. The proposed architecture consists of five core components: the Orchestrator that generates task graphs based on user queries, the Delegator that manages task distribution, ensuring seamless communication between tasks, the agents that autonomously execute tasks using LLMs, the tools that provide predefined functions necessary for task completion, and the Executor that handles the execution sequence, optimizing for both parallel and sequential task execution.

A significant aspect of this work is the development of a comprehensive evaluation framework[2]. Existing agentic systems often lack domain-specific metrics to rigorously assess their performance in handling task decomposition and tool integration. To fill this gap, we propose these novel metrics:

Node and Tool F1 Scores: These metrics assess the system's precision and recall in matching task nodes to the expected task graph, ensuring accurate task decomposition, and in selecting the appropriate tools for each task within the graph.

Structural Similarity Index (SSI): A metric that assesses the overall fidelity of the task graph generated by the system compared to the expected graph, capturing both node and edge similarities to ensure the system preserves the logical structure of tasks.

Additionally, we introduce a specialized dataset[3] designed to evaluate agentic systems, which allows a detailed analysis of the interdependencies between task decomposition, tool selection, and system performance, providing a foundation for evaluating agentic.

In summary, this paper makes the following contributions:

- An agentic framework for dynamic task decomposition, tool integration, and autonomous task execution, designed for complex, multi-hop queries.
- Novel evaluation metrics, including the Node F1 score, the Structural Similarity Index, and the Tool F1 score, for detailed task-specific assessment of agentic systems.
- A specialized dataset to evaluate agentic behavior across, supporting analysis of task graph generation, tool selection, and system performance.

[1] A detailed guide to replicate our agentic system is in Section S1.1 of the Supplementary Material
[2] A detailed guide to replicate our evaluation framework is in Section S1.2 of the Supplementary Material
[3] The dataset is available at this link

The remainder of this paper is structured as follows - Section 2 discusses related work in agentic systems, task graph generation, and evaluation of agentic systems. Section 3 provides a detailed explanation of the architecture of our proposed framework. Section 4 presents our evaluation metrics and dataset, followed by empirical results in Section 5. Finally, Section 6 concludes with future directions and implications for agentic systems in real-world industrial settings.

2 Related Work

2.1 Agentic Frameworks for Task Graph Generation and Tool Selection

The field of AI agent systems powered by LLMs has seen substantial development, with various frameworks proposed to enhance collaboration, task automation, and system scalability. Recent surveys by Wang et al., Guo et al., Masterman et al., and Xi et al. have explored these advancements in detail, highlighting the evolving landscape of LLM-powered AI agents [5, 6, 7, 8, 9]. In [10], Crawford et al. thoroughly explored a flexible agent framework with a focus on the planning and execution components of an AI agent. [3] particularly explores the importance of planning and task graph generation.

Existing frameworks, such as LangGraph, AutoGen, and BabyAGI, provide various approaches to task generation, tool selection, and handling multi-hop user queries [11, 12, 13]. For instance, LangGraph enables stateful workflows with tasks managed cyclically or sequentially, but it does not dynamically adjust the task graph during execution as our proposed framework does [14]. AutoGen and BabyAGI offer dynamic task generation but are limited by predefined toolsets and scalability issues, respectively [15, 16]. Our framework improves upon these methods by introducing real-time adaptability with a Task-Aware Semantic Tool Filtering mechanism, allowing for on-the-fly integration of new tools and dynamic task graph adjustments to handle increased task complexity [17]. This capability ensures continuous and efficient task execution, enhancing existing approaches. Despite these advancements, several limitations and shortcomings exist in current frameworks:

Lack of testability: Existing frameworks often do not support rigorous unit testing of each component, making it challenging to ensure reliability and correctness in complex tasks [16]. Our framework architecture is highly modular, enabling each agent to be tested individually, addressing this gap.

High latency: The absence of efficient task parallelization mechanisms leads to increased latency, hindering real-time performance [18, 19]. Our system decomposes user queries into sub-tasks and executes them in parallel wherever possible, reducing latency.

Limited customizability: Many frameworks offer limited flexibility for customization, restricting their applicability across diverse domains and specific use cases [16, 15]. Our customizable design ensures applicability across various domains.

2.2 Evaluation Metrics and Datasets for Agentic Frameworks

Despite advancements in agentic frameworks, there remains a notable gap in comprehensive evaluation metrics and datasets that accurately assess the performance of these systems across diverse tasks and domains. Existing benchmarks, such as *AgentBench* and *VisualAgentBench*, focus on LLM performance across diverse environments and visual task automation but fall short in evaluating task graph structures in depth [20, 21]. Similarly, *TASKBENCH* introduces "Tool Graphs" to measure task automation processes but provides limited analysis of intermediate steps [22], while *AgentQuest* emphasizes multi-step reasoning with little focus on task graph fidelity [23]. Although *VillagerAgent* explores multi-agent task dependencies in simulated environments, its focus is more on coordination than on detailed task decomposition [24].

Our proposed evaluation framework fills these gaps by introducing detailed metrics—such as Node F1 Score, Structural Similarity Index, and Tool F1 Score—paired with specialized datasets to assess task decomposition, tool selection, and execution through task graph metrics. These metrics and datasets offer a granular analysis of agent performance, addressing the complexity of multi-step reasoning, task automation, and the interdependencies between evaluation metrics. This approach addresses gaps in current benchmarks by focusing on intermediate steps and structural fidelity, offering a more nuanced assessment applicable to real-world scenarios.

3 Agentic Framework Architecture

The framework consists of the following major components, as illustrated in Figure 1. These components include the Orchestrator, the Delegator, Tools, Agents, and the Executor. The diagram outlines the flow of a user query through these components, showing how each element interacts to produce a final response.

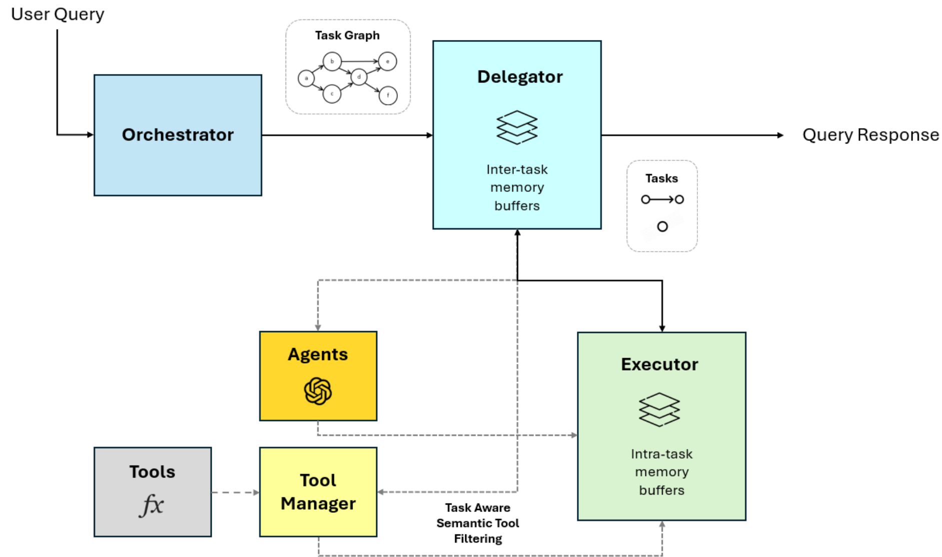

Figure 1: Overview of the framework architecture, showcasing the flow of a user query through the system.

The Orchestrator as shown in Figure 1 analyzes the user's input to produce a Directed Acyclic Graph (DAG) with task nodes and dependency edges. Asynchronous task graph decomposition, as explored by recent work on graph-enhanced LLMs, allows parallel task execution, enabling real-time adaptation to task changes and dependencies. This reduces execution time by shortening critical paths, which is crucial for handling complex, dynamic queries [3]. Hence, borrowing from this work and Graph Theory, the Orchestrator can be instructed to produce a task graph that is optimized for one or more of the following concepts:

- **Coarse Grained Task Decomposition**: A strategy in which the user query is decomposed into a relatively small number of large tasks, each representing a substantial amount of work. This approach minimizes the overhead associated with task management, such as scheduling, synchronization, and inter-task communication, by focusing on larger, more independent units of work [25], [26]. While this can reduce the potential for parallelism, it simplifies execution and can be more efficient in scenarios where the overhead of managing numerous small tasks is prohibitive.
- **Fine Grained Task Decomposition**: A strategy where a user query is broken down into a large number of small, granular tasks. Each task represents a minimal unit of work, allowing for a high degree of parallelism as many tasks can be executed simultaneously. However, this approach often requires more sophisticated management of task dependencies, communication, and synchronization, which can introduce significant overhead [25], [26].
- **Critical Path Optimization**: A strategy that focuses on identifying and shortening the critical path—the longest sequence of dependent tasks that determines the minimum time required to obtain a complete response to a User Query. By optimizing tasks in this path, the overall latency can be reduced [27].

As depicted in Figure 1, the Delegator receives the task graph from the Orchestrator and is responsible for assigning tasks to the appropriate agents or tools. It plays a critical role in managing intra-task and inter-task communication by utilizing inter-task memory buffers, ensuring that each task has the necessary context and data from its dependent predecessors.

The Delegator also consolidates the results from all completed tasks to form the final response to the user query. In scenarios where tasks have direct or indirect dependencies, the Delegator ensures that the inter-task memory buffers are populated with the necessary task descriptions and execution results, thereby enabling subsequent tasks to execute with full awareness of the context provided by their dependencies.

Within the framework, agents are dynamic components, as illustrated in the diagram (Figure 1). They are responsible for utilizing a LLM to execute specific tasks. For instance, the PythonAgent can generate Python functions on the fly, acting as tools that can be called with arguments. The PythonAgent is particularly useful when a task requires a function that is not yet available in the existing set of tools.

Tools are static components that consist of pre-defined Python functions, which can accept arguments and execute code as needed for various tasks. As shown in Figure 1, the tool manager is responsible for task-aware semantic tool filtering. This ensures that only the most relevant tools are passed to the LLM based on the specific task requirements, rather than overwhelming it with the entire tool cache. This semantic filtering enhances the efficiency and accuracy of task execution by reducing unnecessary processing overhead [10].

The Executor (see Figure 2) is responsible for carrying out the execution of the tasks specified in the Task Graph produced by the Orchestrator. The Executor interacts with various components such as the Delegator, Agents, and Tool Manager, managing both intra-task and inter-task memory buffers to ensure efficient task execution. It processes tasks while respecting both direct and indirect dependencies within the task graph, optimizing for parallel execution where possible while ensuring the correct task order when dependencies exist.

In terms of execution, tasks that are independent of each other can be run concurrently, enabling parallelism where possible to maximize efficiency, an example of which is shown in Appendix B. For tasks with direct dependencies, sequential execution is required to ensure that input data from predecessor tasks is available before execution proceeds. Indirect dependencies, where tasks depend on the results of other tasks via intermediates, are also managed by the system to enforce the correct execution order.

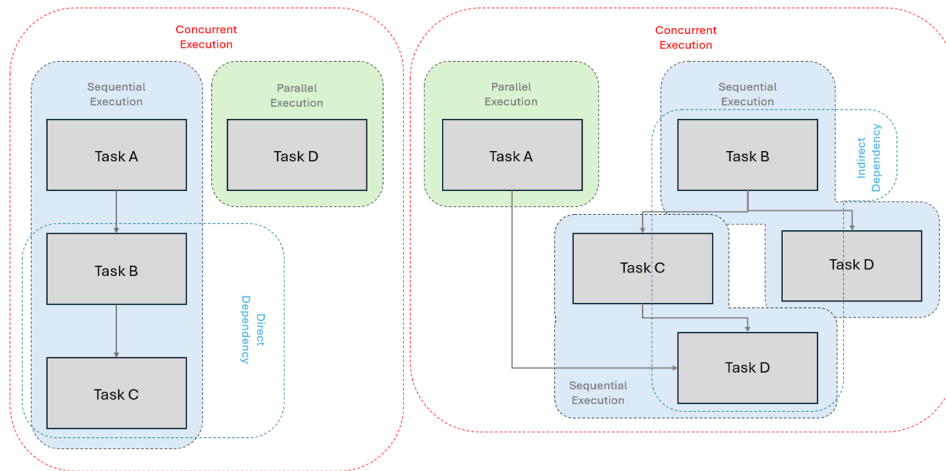

Figure 2: Execution flow managed by the Executor showcasing a simpler task flow on the left and a more complex flow on the right.

The execution flow in Figure 2 illustrates two typical scenarios. On the left side, a simpler case is presented where tasks in a task graph are grouped and executed sequentially (Task A $\rightarrow$ Task B $\rightarrow$ Task C) with direct dependencies, while Task D is executed in parallel. On the right side, a more complex scenario shows Task A being executed in parallel, while Tasks B, C, and D have both direct and indirect dependencies, requiring a combination of sequential and parallel execution.

4 Evaluation Framework

Evaluating agentic systems necessitates a multidimensional analysis of all the individual components as well as the final output. Our approach focuses on evaluating intermediate steps as well as final outcomes, using a robust set of metrics designed to assess the system's ability to decompose tasks, select appropriate tools, and execute tasks effectively. This evaluation framework is grounded in existing literature, including the gaps identified by Gioacchini et al.[23], Shen et al.[22], and Liu et al.[20], which highlight the need for a more granular evaluation of agentic behaviors in LLMs. To address these gaps, we developed a comprehensive evaluation dataset inspired by Lin et al.[28], integrating task graphs, tool usage and final outputs. This dataset supports a more nuanced analysis of agentic systems by capturing both intermediate processes and final results, ensuring a holistic evaluation of system performance.

4.1 Dataset Creation

Our dataset is specifically designed to evaluate the agentic behavior of LLM-driven systems across various domains and tasks. The dataset creation process was structured to ensure representativeness, and relevance to real-world scenarios based on the **AsyncHow** [28] dataset. The **AsyncHow** dataset was particularly suited for this study due to its unique structure that covers parallel, and sequential task graphs. This comprehensive coverage allows for a thorough evaluation of agentic systems, which need to handle different types of task relationships and dependencies. The dataset's design, which includes a mix of simple linear workflows and more intricate interdependent tasks, supports the comprehensive evaluation of the system's ability to manage these varying complexities effectively. The **AsyncHow** dataset's validation for each scenario within the parallel and sequential categories of task graphs ensures that it is a robust foundation for evaluating the proposed agentic systems. As this is one of the most important steps in an agentic system we found it ideal to use it as a foundational dataset. Below is a detailed breakdown of the steps involved.

Task Graph Construction: We randomly sampled 50 task graphs from the **AsyncHow** dataset, ensuring diversity while maintaining empirical manageability for the evaluation of agentic systems. By selecting 50 scenarios for the parallel and sequential task graph category, the study ensured that the dataset remains robust yet feasible for in-depth analysis. The empirical soundness of selecting 50 scenarios, resulting in more than 250 tools, was determined to be sufficient for evaluating the correctness of tool selection and the overall performance of the agentic systems.

Tool Function Generation: Tool descriptions were parsed and translated into synthetic Python functions. These functions were designed to replicate the behavior of real-world tools, ensuring that the agent's interactions with these tools are realistic and contextually appropriate. These functions contribute to creating a realistic and challenging environment for the agentic system as they span a variety of tasks, such as simulating the return of data as JSON from API calls or the data from a candidate for a potential job interview. The functions were then executed to generate final responses for each scenario, providing a benchmark for evaluating the system's performance.

Final Dataset Composition: The final dataset includes a comprehensive set of components: scenario names, task graphs, tool functions, expected tool call sequences, gold standard responses, and complexity categories for each scenario. This structure facilitates a detailed evaluation of both intermediate steps and final outcomes, enabling a more nuanced assessment of the agent's performance.

With this data, we can evaluate each component of our framework:

- **Task graph composition:** We can compare the task graph from the evaluation dataset to the task graph of our agentic system and validate it according to several similarity metrics (4.2).
- **Tool selection:** We can match the list of expected tools selected against the list of tools our agentic system chose.
- **Answer generation:** We can judge the gold standard answer against the answer the agentic system created.

4.2 Evaluation Metrics

To rigorously evaluate the agentic system's performance, we employ a set of metrics that measure the accuracy and effectiveness of task decomposition, tool selection, and task execution. These metrics provide a comprehensive assessment of the system's capabilities, ensuring a thorough evaluation of task graphs, tool identification, and answer generation accuracy.

Precision, Recall, and F1 Score are core metrics used to evaluate the accuracy and completeness of tool identification. The precision score reflects the system's ability to correctly identify relevant tools without including false positives, while recall measures its success in identifying all relevant elements. The F1 Score balances precision and recall, providing a comprehensive view of the system's performance in identifying tools.

$$\text{Precision}_{\text{tool}} = \frac{TP_{\text{node}}}{TP_{\text{tool}} + FP_{\text{tool}}}, \quad \text{Recall}_{\text{tool}} = \frac{TP_{\text{tool}}}{TP_{\text{tool}} + FN_{\text{tool}}}, \quad \text{F1 Score}_{\text{tool}} = \frac{2 \times \text{Precision}_{\text{tool}} \times \text{Recall}_{\text{tool}}}{\text{Precision}_{\text{tool}} + \text{Recall}_{\text{tool}}} \tag{1}$$

$$\text{Precision}_{\text{node}} = \frac{TP_{\text{node}}}{TP_{\text{node}} + FP_{\text{node}}}, \quad \text{Recall}_{\text{node}} = \frac{TP_{\text{node}}}{TP_{\text{node}} + FN_{\text{node}}}, \quad \text{F1 Score}_{\text{node}} = \frac{2 \times \text{Precision}_{\text{node}} \times \text{Recall}_{\text{node}}}{\text{Precision}_{\text{node}} + \text{Recall}_{\text{node}}} \tag{2}$$

$$\text{Precision}_{\text{edge}} = \frac{TP_{\text{edge}}}{TP_{\text{edge}} + FP_{\text{edge}}}, \quad \text{Recall}_{\text{edge}} = \frac{TP_{\text{edge}}}{TP_{\text{edge}} + FN_{\text{edge}}}, \quad \text{F1 Score}_{\text{edge}} = \frac{2 \times \text{Precision}_{\text{edge}} \times \text{Recall}_{\text{edge}}}{\text{Precision}_{\text{edge}} + \text{Recall}_{\text{edge}}} \tag{3}$$

These formulas are applicable for tool identification, node and edge matching, as accurate identification is crucial for the successful decomposition and execution of tasks by the agent.

Node and Edge Matching: We also assess how well the system matches nodes (tasks) and edges (dependencies) in the task graph to the expected structure. These metrics are key to evaluating the structural integrity and correctness of task decomposition.

Node Label Similarity: To assess how semantically similar the nodes in the actual task graph are to those in the expected graph, we use a cosine similarity measure based on node labels:

$$\text{Node Label Similarity} = \frac{1}{|N_{\text{expected}}|} \sum_{i=1}^{|N_{\text{expected}}|} \max_{j \in N_{\text{actual}}} \text{CosineSim}(L_i, L_j) \tag{4}$$

Graph Evaluation: Beyond node and edge matching, we employ metrics that evaluate the entire structure of the task graph, providing a more holistic assessment of its accuracy.

- **Graph Edit Distance (GED):** GED quantifies the dissimilarity between the actual task graph and the expected graph by calculating the minimum number of edit operations (insertions, deletions, substitutions) needed to transform one graph into the other. It provides a more granular view of graph differences than simple node or edge matching.
- **Structural Similarity Index (SSI):** This metric combines node and edge similarities into a single score, offering a comprehensive evaluation of the task graph's structural fidelity:

$$\text{SSI} = \frac{\text{Node Label Similarity} + \text{Edge F1 Score}}{2} \tag{5}$$

- **Path Length Similarity:** This metric compares the lengths of paths between corresponding nodes in both the actual and expected task graphs, providing insight into how well the system captures the broader structure of task dependencies:

$$S_{PL} = \frac{1}{|V|^2} \sum_{(u,v) \in V_1 \times V_2} \exp\left(-\alpha \left| d_{G_1}(u,v) - d_{G_2}(u,v) \right|\right) \tag{6}$$

Complexity Score: The complexity score as defined by [28] as the total number of vertices and edges within a task graph: $|V| + |E|$, where V is the number of nodes and E is the number of edges. This metric provides a measure of the graph's structural complexity and helps compare task graphs of varying sizes.

5 Results and Discussion

Upon investigating our agentic framework we noticed that explicitly mentioning decomposition strategies (coarse grained vs fine grained) allowed the system to adapt based on the complexity and interdependence of tasks extracted from a user query and saw a significant improvement in task accuracy and reduction in inefficiency - lesser amount of redundant tasks were produced. This is in agreement with [29, 30], which highlight the importance of multi-granularity approaches in AI frameworks, particularly in complex domains like multi-hop question answering.

We conducted an in-depth analysis of the proposed metrics on our agentic system to understand their impact on the system's performance for both sequential and parallel tasks. The analysis revealed that structural and node-level metrics are more critical in sequential tasks, while tool-related metrics are prominent in parallel tasks. Our choice of evaluation metrics is validated by prior research on LLMs and their performance in graph-based tasks. Studies have shown that combining precision and recall, such as in the Node F1 Score, is essential for accurately evaluating systems that interact with complex graph structures, as it minimizes false positives and negatives [31]. The Structural Similarity Index (SSI) has also been validated as a reliable metric for assessing the preservation of both the functional and structural aspects of task graphs [32].

The **Structural Similarity Index** (SSI) emerged as the most significant predictor of Answer Score, with a strong positive correlation ($r = 0.470$, $p < 0.001$). Plot can be seen in the Appendix in Figure 3. Node Label Similarity also showed a substantial positive correlation ($r = 0.447$, $p < 0.01$). Interestingly, Expected Task Complexity exhibited a moderate negative correlation ($r = -0.293$, $p < 0.05$), suggesting that as tasks become more complex, performance tends to decrease. These correlations were further reflected in real-world applications of the agentic system, where scenarios requiring fine-grained task decomposition revealed high Node Precision and Recall, but also highlighted challenges with managing complex dependencies, as indicated by lower Edge F1 Scores. When considering the absolute coefficients from our linear regression model, SSI again emerged as the most important feature, followed by Edge F1 Score and Node Label Similarity. This aligns with our correlation analysis, emphasizing the importance of structural accuracy in sequential tasks. The model achieved an R-squared value of 0.3631, indicating that these features explain approximately 36.31% of the variance in Answer Score for **sequential** tasks.

For parallel tasks, we observed a shift in the importance of metrics, with tool-related features gaining prominence. Plots can be seen in Appendix A in Figure 4. Tool F1 Score showed the strongest correlation with Answer Score ($r = 0.476$, $p < 0.001$), closely followed by Tool Recall ($r = 0.474$, $p < 0.01$) and Tool Precision ($r = 0.414$, $p < 0.01$). SSI and Node Label Similarity both demonstrated moderate positive correlations ($r = 0.380$, $p < 0.05$ for both). Interestingly, our regression analysis revealed that SSI and Node Label Similarity had the highest importance scores, despite not having the strongest correlations. This suggests a complex interplay between these structural features and tool-related metrics in parallel tasks. The model for parallel tasks achieved an R-squared value of 0.3933, explaining 39.33% of the variance in Answer Score.

Our findings highlight differences in the factors influencing agentic system performance between sequential and parallel tasks. **Structural metrics** (SSI and Node Label Similarity) are important across both task types, but their relative importance is higher in sequential tasks. Practical application of our evaluation framework also supports these findings, where task decomposition in real-world scenarios displayed strong structural performance but revealed weaknesses in dependency management (as seen in lower Edge F1 Scores). **Tool-related metrics** (Tool F1 Score, Recall, and Precision) are more strongly correlated with performance in parallel tasks, suggesting that effective tool selection and usage become critical in more complex, non-linear task structures. **Edge-related metrics** (Edge Precision and Edge F1 Score) show some importance in sequential tasks but are not significant in parallel tasks, possibly due to the more complex relationships between nodes in parallel structures.

6 Limitations

While the proposed agentic framework offers significant advancements in evaluating agentic systems, it faces several limitations. A primary issue is the lack of support for multi-agent communication, which limits the framework's effectiveness in scenarios requiring task coordination among multiple agents. The system is only suited for single-agent environments or cases where agents operate

independently. Expanding this capability would significantly enhance its versatility in complex settings.

Precision, Recall, and F1 Score—are also sensitive to dataset imbalances. When there are very few or many expected tools, these metrics may not capture performance nuances effectively, leading to skewed results. Another limitation is that these metrics do not account for the context in which the tools are applied. A tool may be identified correctly, but if used in an inappropriate context, this error is not reflected in the metrics, potentially leading to inaccurate performance assessments. Calculating Graph Edit Distance (GED) is computationally intensive, especially for large graphs, and this sensitivity of GED to cost, along with the ambiguity in its interpretation, reduces its effectiveness in evaluating complex task graphs. The matching of nodes and edges can suffer when multiple nodes or edges possess similar labels, which can require manual intervention or the implementation of additional rules to resolve conflicts. This increases the complexity of the task and reduces the reliability of fully automated methods. Moreover, edge matching is heavily dependent on the accuracy of node matching, which means that any errors in node similarity evaluations can propagate and negatively affect the assessment of task dependencies. The Structural Similarity Index (SSI) also presents certain drawbacks. First, it is highly sensitive to node similarity scores, which can be problematic when nodes - representing tasks - are described in significantly different ways. This can distort the overall similarity measure. Additionally, SSI assigns equal weight to node and edge similarities, which may not be appropriate in all cases. In some situations, task dependencies, represented by edges, may be more important than the individual tasks themselves, and equal weighting could overlook this context. Finally, SSI computation, particularly for large graphs can be computationally expensive, further complicating its application in large-scale evaluations.

7 Future Work

Building on the limitations identified, future research should focus on the following. First, the integration of multi-agent communication protocols would allow for dynamic collaboration, enabling the system to handle more complex, real-time scenarios that require the collaboration of multiple agents. Additionally, introducing causal inference methods to the evaluation framework would offer deeper insights into system performance by moving beyond correlation-based metrics. Optimizing scalability remains a critical challenge, especially for large-scale real-time environments, where reducing latency and computational overhead is essential. Finally, developing more sophisticated datasets that reflect real-world randomness and multi-agent interactions is crucial for testing and improving the scalability of agentic systems. Refer to Appendix C for detailed concrete future research directions to extend this work.

8 Conclusion

Analysis of our agentic framework revealed that task graph based decomposition [28] and explicitly integrating coarse-grained and fine-grained decomposition strategies led to improved task accuracy and a reduction in inefficiencies by minimizing redundant tasks. This adaptability, driven by the complexity and interdependence of user query tasks, highlights the critical role of multi-granularity approaches in agentic systems. Moreover, while these strategies enhanced system performance, challenges in managing complex dependencies, especially in fine-grained tasks, became evident.

Our work also presents a comprehensive framework for evaluating agentic systems driven by LLMs. We introduce proper metrics, including the **Node F1 Score**, **Structural Similarity Index**, and **Tool F1 Score**, to assess the performance of these systems in task decomposition and tool integration scenarios. Additionally, our specialized dataset created for this study enables an in-depth analysis of agentic behavior across various task complexities. Our findings emphasize the importance of both **structural and tool-related metrics** in determining overall system performance, with notable differences between sequential and parallel tasks. While structural metrics are more critical in sequential tasks, tool usage becomes more significant in parallel tasks, highlighting the need for balanced evaluation methods that capture both the structural and operational aspects of agentic systems. The proposed framework and evaluation methodology provide a strong foundation for further research. Addressing the outlined limitations and pursuing the proposed future work will be critical in advancing agentic systems capable of handling more complex, real-time, and collaborative environments.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[2] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozire, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[3] Fan Lin et al. Graph-enhanced large language models in asynchronous plan reasoning. *arXiv preprint arXiv:2402.02805*, 2024.

[4] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023.

[5] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), 2024.

[6] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges, 2024.

[7] Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. The landscape of emerging ai agent architectures for reasoning, planning, and tool calling: A survey, 2024.

[8] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. The rise and potential of large language model based agents: A survey, 2023.

[9] Hung Du, Srikanth Thudumu, Rajesh Vasa, and Kon Mouzakis. A survey on context-aware multi-agent systems: Techniques, challenges and future directions, 2024.

[10] Noel Crawford, Edward B Duffy, Iman Evazzade, Torsten Foehr, Gregory Robbins, Debbrata Kumar Saha, Jiya Varma, and Marcin Ziolkowski. Bmw agents–a framework for task automation through multi-agent collaboration. *arXiv preprint arXiv:2406.20041*, 2024.

[11] LangGraph. Langgraph: Build resilient language agents as graphs. `https://github.com/langchain-ai/langgraph`, 2024.

[12] Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. Autoagents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288*, 2023.

[13] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2023.

[14] N. Crawford et al. Bmw agents: A framework for task automation through multi-agent collaboration. *arXiv preprint arXiv:2406.20041*, 2024.

[15] Q. Wu et al. Autogen: Enabling next-generation large language model applications. *Microsoft Research*, 2024.

[16] Microsoft Research. Autogen: A no-code developer tool for building and debugging multi-agent systems. *arXiv preprint arXiv:2408.15247*, 2023.

[17] Xiaoming Jin et al. A comprehensive survey of dynamic graph neural networks: Models, frameworks, benchmarks, experiments and challenges. *arXiv preprint arXiv:2405.00476*, 2024.

[18] A. Gupta et al. Autoagents: A framework for task generation and tool selection in multi-agent systems. *arXiv preprint arXiv:2309.17288*, 2024.

[19] L. Zhang et al. Tdag: A multi-agent framework based on dynamic task decomposition and agent generation. *arXiv preprint arXiv:2402.10178*, 2024.

[20] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

[21] Xiao Liu, Tianjie Zhang, Yu Gu, Iat Long Iong, Yifan Xu, Xixuan Song, Shudan Zhang, Hanyu Lai, Xinyi Liu, Hanlin Zhao, et al. Visualagentbench: Towards large multimodal models as visual foundation agents. *arXiv preprint arXiv:2408.06327*, 2024.

[22] Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. Taskbench: Benchmarking large language models for task automation. *arXiv preprint arXiv:2311.18760*, 2023.

[23] Luca Gioacchini, Giuseppe Siracusano, Davide Sanvito, Kiril Gashteovski, David Friede, Roberto Bifulco, and Carolin Lawrence. Agentquest: A modular benchmark framework to measure progress and improve llm agents. *arXiv preprint arXiv:2404.06411*, 2024.

[24] Yubo Dong, Xukun Zhu, Zhengzhe Pan, Linchao Zhu, and Yi Yang. Villageragent: A graph-based multi-agent framework for coordinating complex task dependencies in minecraft. *arXiv preprint arXiv:2406.05720*, 2024.

[25] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[26] Michael J Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[27] Harold Kerzner. *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*. Wiley, 2017.

[28] Fangru Lin, Emanuele La Malfa, Valentin Hofmann, Elle Michelle Yang, Anthony Cohn, and Janet B Pierrehumbert. Graph-enhanced large language models in asynchronous plan reasoning. *arXiv preprint arXiv:2402.02805*, 2024.

[29] Xing Cao and Yun Liu. Coarse-grained decomposition and fine-grained interaction for multi-hop question answering. *arXiv preprint arXiv:2101.05988*, 2021.

[30] Xiang Victor Lin, Richard Socher, and Caiming Xiong. Multi-hop knowledge graph reasoning with reward shaping. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 3243–3253, 2018.

[31] Yanbang Wang, Hejie Cui, and Jon Kleinberg. Microstructures and accuracy of graph recall by large language models. *arXiv preprint arXiv:2402.11821*, 2024.

[32] Peter Wills and François G Meyer. Metrics for graph comparison: a practitioner's guide. *Plos one*, 15(2):e0228728, 2020.

Supplementary Material

S1.1 Replicating the Agentic Task Decomposition Framework

This section provides a detailed guide for replicating the agentic task decomposition framework, including Python function shells with brief descriptions. This guide assumes familiarity with Python, machine learning, and related libraries.

Our framework decomposes user queries into a Directed Acyclic Graph (DAG) of tasks and uses an LLM for tool selection and execution. The system components include the **Orchestrator**, **Delegator**, **ToolManager**, and **GraphExecutor**. These work together to generate tasks, select tools, and ensure parallel or sequential execution.

S1.1.1 Prerequisites

The following Python libraries are required (from `requirements.txt`):

- openai = 1.35.10
- graphviz = 0.20.3
- requests = 2.32.3
- wikipedia = 1.4.0
- tqdm = 4.66.4
- termcolor = 2.4.0
- fastapi = 0.111.1
- matplotlib = 3.9.1
- faiss-cpu = 1.8.0.post1
- loguru = 0.7.2
- pytest-asyncio = 0.24.0
- websockets = 13.0.1
- transformers = 4.44.2
- torch = 2.4.1

A system with at least 32GB RAM and a GPU for faster LLM inference is recommended.

The system relies on the `AsyncHow-Based Agentic Systems Evaluation Dataset` available at link.

S1.1.2 Core Components and Function Shells

The **Orchestrator** is responsible for generating a Directed Acyclic Graph (DAG) from user queries, breaking them into manageable tasks.

```
class Orchestrator:
    def __init__(self, gpt_client_manager):
        """
        Initializes the Orchestrator with the GPT client manager for
        processing user queries into a task graph (DAG).
        """
        self.client = gpt_client_manager

    def produce_task_graph(self, user_query):
        """
        Calls the LLM to convert a user query into a task graph.
        The task graph is a DAG representing tasks
        and their dependencies.
        """
        return task_graph
```

The **ToolManager** dynamically loads Python functions from files and selects relevant tools for each task.

```
class ToolManager:
    def __init__(self):
        """
        Initializes the ToolManager, which is
        responsible for loading, embedding, and
        selecting tools for execution
        based on task descriptions.
        """

    def parse_all_function_files(self,
                    function_files_path="python_tools"):
        """
        Parses Python files containing
        tool functions and embeds their descriptions
        to make them available for
        selection during task execution.
        """

    def filter_tools_by_tasks(self, task_list):
        """
        Filters available tools using
        FAISS based on the semantic similarity
        between the task description and tool embeddings.
        """
```

The `ToolManager` links directly to the `GraphExecutor`, which uses filtered tools for task execution.

The **GraphExecutor** handles task execution, using tools selected by the ToolManager. It can execute tasks in parallel or sequentially, depending on dependencies.

```
class GraphExecutor:
    def __init__(self, my_gpt_client_manager,
                 include_indirect_dependencies=False,
                 timing_profiler=None):
        """
        Initializes the GraphExecutor,
        which executes tasks either sequentially or
        in parallel while handling dependencies.
        """
        self.client = my_gpt_client_manager.openai_client
        self.tool_manager = None
        self.timing_profiler = timing_profiler

    def initialize_tool_manager(self, tool_manager):
        """
        Sets the ToolManager instance, which
        will be used to filter and select tools
        for task execution.
        """

    def execute_task_graph_sequentially(self):
        """
        Executes tasks in the task graph sequentially,
        respecting the dependencies between tasks.
        """
        return execution_results
```

```
    def execute_task_graph_parallely(self):
        """
        Executes tasks in the task graph in
        parallel where possible,
        optimizing for concurrency.
        """
        return execution_results

    def _execute_task_func(self, task, inter_task_buffer):
        """
        Executes a single task, interacting
        with the GPT model for task results,
        and managing the inter-task message buffer
        to handle dependencies.
        """
        return final_response, execution_timing, tool_calls
```

The `GraphExecutor` interacts with the `ToolManager` to call relevant tools and with the `Delegator` for managing task execution order.

The **Delegator** manages the execution of tasks by ensuring proper sequencing and handling inter-task communication.

```
class Delegator:
    def __init__(self, pipeline):
        """
        Initializes the Delegator, which is
        responsible for managing the flow of
        tasks, including communication between them.
        """
        self.pipeline = pipeline

    def execute_task_graph(self):
        """
        Orchestrates the execution of the entire task graph,
        ensuring each task is executed in the
        correct order and managing inter-task buffers.
        """
        return task_results
```

The `Delegator` consolidates results from the `GraphExecutor` and ensures that tasks dependent on others receive the necessary input.

The **Feedback System** generates real-time feedback for users during task execution, keeping them informed about task progress.

```
class FeedbackSystem:
    def add_to_queue(self, task_id, task_label):
        """
        Adds a task to the feedback queue
        to provide real-time updates to users
        about the progress of that specific task.
        """
```

The `FeedbackSystem` is integrated with the `GraphExecutor` to deliver updates to the user as tasks are executed.

The **Timing Profiler** tracks execution times for tasks and tools, enabling detailed performance analysis.

```
class TimingProfiler:
```

```
    def start_task_timing(self, task_id, task_label):
        """
        Starts tracking the execution time
        for a specific task, storing the start time.
        """

    def stop_task_timing(self, task_id):
        """
        Stops tracking the execution time
        for a specific task, storing the
        end time and calculating the duration.
        """
```

The `TimingProfiler` links to both the `GraphExecutor` and the `FeedbackSystem` for detailed time tracking of task execution and feedback generation.

The **Pipeline** class orchestrates the entire framework, ensuring proper initialization, task decomposition, and execution. It interacts with the Orchestrator, ToolManager, GraphExecutor, and FeedbackSystem to process user queries.

```
class Pipeline:
    def __init__(self, semantic_tool_filtering=True,
                 include_indirect_dependencies=True,
                 generate_feedback=True,
                 profile_execution_timings=True):
        """
        Initializes the Pipeline with various options, including:
        - Semantic tool filtering:
            Filters tools based on task relevance.
        - Include indirect dependencies:
            Includes results from all ancestor tasks.
        - Generate feedback:
            Provides real-time feedback during task execution.
        - Profile execution timings:
            Tracks the execution time for tasks and tools.
        """
        self.semantic_tool_filtering = semantic_tool_filtering
        self.include_indirect_dependencies = include_indirect_dependencies
        self.generate_feedback = generate_feedback
        self.profile_execution_timings = profile_execution_timings
        self.tool_manager = ToolManager()
        self.graph_executor = None
        self.orchestrator = Orchestrator(gpt_client_manager)
        self.feedback_system = None

    def configure_for_query(self, user_query):
        """
        Configures the pipeline for a new user query.
        This includes initializing the task graph,
        selecting tools, and setting up
        feedback and profiling options.
        """
        task_graph = self.orchestrator.produce_task_graph(user_query)
        self.graph_executor = GraphExecutor(self.orchestrator.client,
                                            self.include_indirect_dependencies,
                                            self.profile_execution_timings)
        self.graph_executor.initialize_tool_manager(self.tool_manager)
        self.graph_executor.initialize_task_graph(task_graph)

    def run_with_pretty_print(self, user_query):
```

```
        """
        Executes the user query by
        first configuring the pipeline and then running
        the task graph execution either
        sequentially or in parallel.
        Outputs are displayed with real-time feedback
        and execution timings.
        """
        self.configure_for_query(user_query)
        if self.generate_feedback:
            self.graph_executor.initialize_feedback_system(
                                        self.feedback_system)

        # Executes the task graph and provides pretty-printed results
        results = self.graph_executor.execute_task_graph_sequentially()
        # or parallely
        print(results)
```

The `Pipeline` class links directly to all the major components (`Orchestrator`, `ToolManager`, `GraphExecutor`, and `FeedbackSystem`) and acts as the main interface for the entire framework. It ensures that tasks are processed, tools are selected, and feedback is generated in a streamlined way.

S1.1.3 Prompt Skeletons

The framework uses specific prompts to interact with the LLM (GPT-4). These prompts handle task graph generation, task execution, tool selection, and user feedback. Below is a skeleton of the main prompts from the system.

```
PROMPT_TASK_GRAPH = """
You are responsible for generating a task graph
from the following user query. Decompose the query
into individual tasks and create a Directed Acyclic Graph (DAG)
with nodes as tasks and edges as dependencies.
Ensure there are no cyclic dependencies.

User Query: {user_query}

Respond with the task graph in the following JSON format:
{
  "nodes": [
    {"id": 1, "label": "Task description 1"},
    {"id": 2, "label": "Task description 2"}
  ],
  "edges": [
    {"from": 1, "to": 2}
  ]
}
"""

PROMPT_CONSOLIDATE_TASKS = """
You are an assistant operating within an LLM-based Agentic Architecture.
Your task is to generate a final response to the user's query
by considering the results of multiple tasks.
These tasks were generated from the user's query
using a task graph. Ensure the final response
addresses all aspects of the user's query.

User Query: {user_query}

Task Results: {task_results}
```

```
Generate a concise final response in 50 words or less.
"""

PROMPT_FEEDBACK = """
You are responsible for generating a short feedback phrase
for a task that is being processed.
The feedback should be friendly and let the user
know their task is in progress.

Task: {task_description}

Generate a new feedback phrase:
"""
```

`PROMPT_TASK_GRAPH`: This prompt is used to decompose a user query into a Directed Acyclic Graph (DAG). Each node represents a task and edges represent dependencies.

`PROMPT_CONSOLIDATE_TASKS`: This prompt consolidates the results of multiple tasks into a single final response to the user's query.

`PROMPT_FEEDBACK`: This prompt generates feedback to inform the user of task progress during execution.

S1.1.4 Replication Guide

Step 1: Environment Setup Install dependencies using the provided `requirements.txt` file:

```
pip install -r requirements.txt
```

Step 2: Configure GPT-4 Set up GPT-4 through an Azure OpenAI deployment and configure credentials in `gpt_client.py`.

Step 3: Prepare Tools Add your custom Python tools in the `python_tools` folder. The `ToolManager` will automatically detect and embed these tools.

Step 4: Execute Pipeline Run the pipeline with a sample user query:

```
query = "What time does the sun rise today?"
pipeline.run_with_pretty_print(query)
```

Step 5: Analyze Results The execution results and timing profile will be stored in a JSON file for performance analysis.

This guide, with function shells, descriptions, step-by-step instructions and prompt skeletons, provides a detailed replication process for the agentic framework. The modular nature of the components allows for easy customization and adaptation to other domains.

S1.2 Evaluation and Dataset Creation

To evaluate the performance of our agentic task decomposition framework, we generated a custom evaluation dataset. This section describes the process of creating task graphs, tools, and the evaluation metrics used for analyzing the framework.

S1.2.1 Dataset Creation

The evaluation dataset is built by creating task graphs for sequential, parallel, and asynchronous task execution scenarios. Each scenario is associated with a list of tools, and the task graph is generated based on tool descriptions.

```
def create_seq_task_graph(edges, node_descriptions):
    """
    Creates a sequential task graph
```

```
    based on a list of edges and node descriptions.
    This function removes 'Start' and 'End' nodes
    and maps tasks to unique IDs.

    Args:
        edges (list of tuples): List of edges
        where each edge is a tuple (from_node, to_node).
        node_descriptions (list of str): List of
        task descriptions.

    Returns:
        dict: A dictionary representing the task graph
        with nodes and edges.
    """
    task_map = {}
    node_list = []
    current_task_id = 1

    for i, description in enumerate(node_descriptions):
        if description not in ["Start", "End"]:
            task_id = f"task_{current_task_id}"
            task_map[str(i+1)] = task_id
            node_list.append({'id': task_id,
                              'label': description})
            current_task_id += 1

    edge_list = []
    for from_node, to_node in edges:
        if from_node in task_map and to_node in task_map:
            edge_list.append({'from': task_map[from_node],
                              'to': task_map[to_node]})

    return {'task_graph': {'nodes': node_list,
                           'edges': edge_list}}
```

This function is responsible for creating task graphs from a list of task descriptions and edges. A similar function is used for generating task graphs for parallel tasks.

```
def create_parallel_graph(edges, tools):
    """
    Creates a parallel task graph based on a list of tools. No edges are added.

    Args:
        edges (list of tuples): List of edges
        representing the dependencies (if any).
        tools (list of str): List of tool names.

    Returns:
        dict: A dictionary representing the task graph with nodes and edges.
    """
    task_graph = {
        "task_graph": {
            "nodes": [],
            "edges": []
        }
    }

    for i, tool in enumerate(tools):
        task_id = i + 1
        task_graph["task_graph"]["nodes"].append({"id": task_id,
```

```
                                                    "label": tool})

    return task_graph
```

S1.2.2 Evaluation Process

We evaluated the task decomposition framework using synthetic tasks designed for both sequential and parallel task execution. The following function helps create a structured evaluation dataset:

```
def create_async_graph(edges, node_descriptions):
    """
    Creates an asynchronous task graph by mapping nodes to unique integer IDs.

    Args:
        edges (list of tuples): List of edges
        where each edge is a tuple (from_node, to_node).
        node_descriptions (list of str): List of
        task descriptions.

    Returns:
        dict: A dictionary representing the task graph
        with nodes and edges.
    """
    task_graph = {
        "task_graph": {
            "nodes": [],
            "edges": []
        }
    }

    for i, desc in enumerate(node_descriptions):
        task_graph["task_graph"]["nodes"].append({"id": i+1,
                                                    "label": desc})

    for from_node, to_node in edges:
        task_graph["task_graph"]["edges"].append({"from": from_node,
                                                    "to": to_node})

    return task_graph
```

During the dataset creation, semantic duplicates are removed to ensure a high-quality dataset.

```
def remove_semantic_duplicates(folder_path,
                               similarity_threshold=0.8):
    """
    Identifies and removes semantically similar
    duplicates from the folder names.

    Args:
        folder_path (str): Path to the folder
        containing subfolders.
        similarity_threshold (float): Threshold above which
        two folders are considered duplicates.

    Returns:
        list: A list of unique subfolder names.
    """
    model = SentenceTransformer('all-MiniLM-L6-v2')
    subfolder_names = [name for name in os.listdir(folder_path)
    if os.path.isdir(os.path.join(folder_path, name))]
```

```
    embeddings = model.encode(subfolder_names,
                              convert_to_tensor=True)

    unique_subfolders = []
    for i, name in enumerate(subfolder_names):
        if not any(util.cos_sim(embeddings[i], embeddings[j]) >
        similarity_threshold for j in range(i)):
            unique_subfolders.append(name)

    return unique_subfolders
```

Tools for the evaluation were synthetically generated to mimic real-world APIs or data retrieval functions. Below is the prompt-based function for creating these tools:

```
def generate_functions(data, folder):
    """
    Generates Python functions based on
    the tool descriptions and stores
    them in the specified folder.

    Args:
        data (dict): Contains tool names and
        corresponding function code.
        folder (str): Path to the folder where the
        functions will be saved.
    """
    base_directory = os.path.join("../..", folder)

    for function_name, function_code in zip(data["function_names"],
                                            data["functions"]):
        function_directory = os.path.join(base_directory,
                                          function_name)
        os.makedirs(function_directory, exist_ok=True)

        file_path = os.path.join(function_directory,
                                 f"{function_name}.py")
        with open(file_path, "w") as file:
            file.write(function_code)
```

S1.2.3 Prompt Skeletons

Below are the prompt skeletons used during the evaluation and dataset creation process. These prompts guide the LLM in generating Python functions, removing duplicates, and consolidating tool outputs.

```
PROMPT_GENERATE_FUNCTIONS = """
This prompt instructs the LLM to generate Python functions
based on a list of tool descriptions.
It includes instructions for creating synthetic data
and formatting the function signatures and bodies.

### Input:
- Tool Descriptions: {tools}
- User Query: {query}

### Output:
- JSON object containing the functions,
their names, and a gold standard response.
"""
```

```
PROMPT_REMOVE_DUPLICATES = """
This prompt guides the system in identifying and
removing semantically similar duplicates
from a dataset folder. It ensures that no
two tools or tasks are too similar.

### Input:
- Folder Path: {folder_path}
- Similarity Threshold: {similarity_threshold}

### Output:
- List of unique tool/task names.
"""

PROMPT_CONSOLIDATE_TASKS = """
This prompt consolidates the results of several tool
executions into a final response.
It ensures that the final output is concise, accurate,
and addresses the user's query.

### Input:
- Task Results: {task_results}

### Output:
- Consolidated response based on tool outputs.
"""

PROMPT_COMPLEXITY_SCORE = """
This prompt categorizes user queries based on their complexity,
assigning them into predefined categories.
It ensures that each query is analyzed for its difficulty and
aligned with the appropriate evaluation metric.

### Input:
- User Query: {query}

### Output:
- A complexity score/category for the query.
"""
```

`PROMPT_GENERATE_FUNCTIONS`: Generates Python functions based on tool descriptions, ensuring realistic outputs and proper formatting.

`PROMPT_REMOVE_DUPLICATES`: Helps identify and remove semantically similar duplicates from a dataset.

`PROMPT_CONSOLIDATE_TASKS`: Consolidates tool execution results into a single, accurate response.

`PROMPT_COMPLEXITY_SCORE`: Categorizes user queries into complexity levels for evaluation purposes.

S1.2.4 Replication Guide

To replicate the dataset creation and evaluation process, follow these steps:

Step 1: Setup the Environment Install the necessary dependencies and tools listed in the `requirements.txt` file. Ensure that the system has access to an LLM such as GPT-4 through the OpenAI API or Azure OpenAI.

```
pip install -r requirements.txt
```

Step 2: Generate Task Graphs Run the provided Python scripts to generate task graphs for sequential, parallel, and asynchronous scenarios. These graphs will form the backbone of the evaluation dataset, mapping tasks to tools and establishing dependencies.

```
# Generate sequential task graphs
create_seq_graphs(df_name="asynchow_seq_df.csv")

# Generate parallel task graphs
create_parallel_graphs(df_name="asynchow_para_df.csv")

# Generate asynchronous task graphs
create_async_graphs(df_name="asynchow_async_df.csv")
```

Step 3: Generate Python Functions for Tools For each tool in the dataset, generate a Python function that mimics its behavior. Use the function generation script to create these tools and save them in the designated folder. This step involves feeding tool descriptions to the LLM, which generates the corresponding Python code.

```
# Generate functions based on tool descriptions
generate_functions_from_tool_list()
```

Step 4: Remove Semantic Duplicates Ensure that no redundant or semantically similar tools are included in the dataset by running the duplicate removal script.

```
remove_semantic_duplicates(folder_path="tools", similarity_threshold=0.8)
```

Step 5: Execute and Evaluate Finally, execute the tasks using the generated tools and evaluate the system's performance by analyzing the task results and feedback.

```
# Run the task execution and feedback generation
pipeline.run_with_pretty_print(query)
```

After running the evaluation, analyze the complexity scores and tool performance to measure the framework's ability to handle complex task decomposition and execution workflows.

This process generates a complete evaluation dataset, tests the system across multiple task execution scenarios, and provides detailed insights into the system's performance. By following the outlined steps, researchers and developers can replicate the evaluation process and extend the framework to additional scenarios or datasets.

Appendix

A Evaluation Plots

The plots show the statistically significant correlations in regards to the Answer Score.

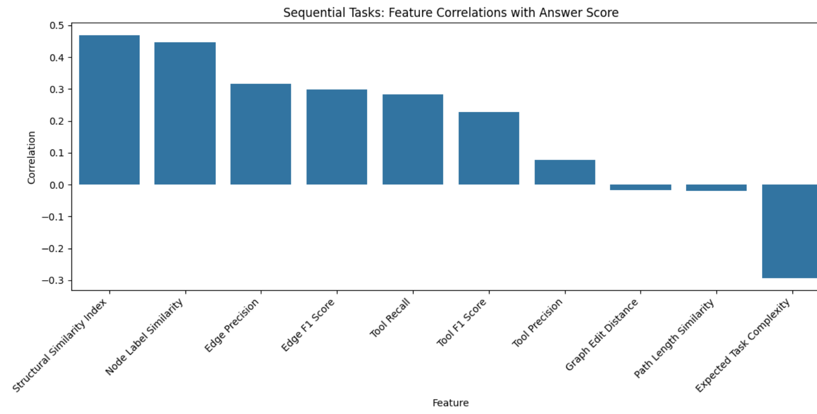

Figure 3: Feature Correlations with Answer Score for Sequential Tasks

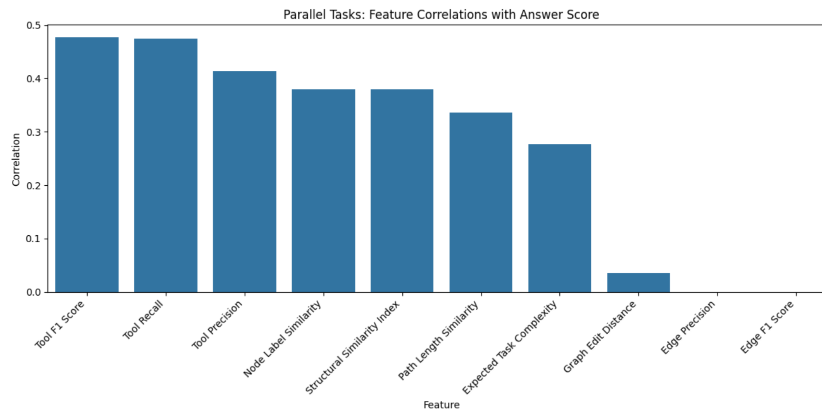

Figure 4: Feature Correlations with Answer Score for Parallel Tasks

B Additional plots for timing profiling of parallel execution of task graphs

Figure 5 shows the specific timing of tasks during a parallel execution scenario. This timing diagram demonstrates how independent tasks (e.g., Task 1 and Task 4) are executed in parallel, while tasks that have dependencies (e.g., Task 2 and Task 3) are scheduled sequentially, respecting their dependencies. The timing diagram demonstrates the framework's ability to efficiently manage parallel execution while ensuring dependent tasks are executed in the correct order.

C Additional Future Work

Based on the results and implications of our current study, we plan to explore several avenues for future work that could further enhance the capabilities and evaluation of agentic systems:

- **Multi-Agent Communication:** Developing and integrating multi-agent communication capabilities to enhance the collaborative aspect of our framework. This would allow for

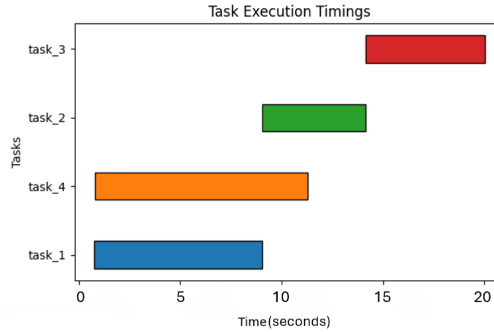

Figure 5: Task execution timings for parallel execution. The diagram shows the start and end times for four tasks (Task 1, Task 2, Task 3, and Task 4). Tasks 1 and 4, which are independent, are executed in parallel, while Task 2 starts after Task 4, and Task 3 follows Task 2, showing sequential execution due to dependencies.

more complex task execution scenarios where multiple agents must coordinate and share information.

- **Causal Influence Score (CIS):** Developing the CIS metric to evaluate the causal relationships between the tools used and the final outcomes or answers generated by the agent. Unlike traditional precision and recall metrics, CIS would focus on understanding how each tool's use causally impacts overall task success.
- **Dynamic Adaptation Metric (DAM):** Introducing the DAM metric to measure an agent's ability to dynamically adapt its tool selection and task execution strategies in response to changing conditions or feedback during task execution.
- **Inter-tool Dependency Metric (ITDM):** Creating the ITDM metric to evaluate the degree of dependency between tools used by the agent within a task. This metric would capture whether the agent effectively coordinates between multiple tools or relies too heavily on certain tools.
- **Task Graph Robustness Index (TGRI):** Implementing the TGRI metric to measure the robustness of the task graph generated by the agent against errors or changes. This metric would evaluate how small modifications to the input or environment affect the task graph and its resulting accuracy.
- **Contextual Consistency Score (CCS):** Developing the CCS metric to measure how consistently an agent applies tools and constructs task graphs in contexts similar to previously encountered scenarios. This would assess the agent's ability to generalize across similar tasks.
- **Explainability Score (ES):** Introducing the ES metric to measure how well the agent can explain its decisions, particularly its choice of tools and task graph construction. This metric would be crucial for ensuring transparency and building trust in agentic systems.
- **Cumulative Learning Rate (CLR):** Developing the CLR metric to measure how quickly and effectively the agent improves its performance over time as it encounters more tasks. This would capture the agent's learning efficiency and its ability to adapt and improve.
- **Interaction Cost Metric (ICM):** Creating the ICM metric to evaluate the efficiency of an agent's interactions in terms of resource usage, such as time, computational power, or energy, in achieving a task. This metric would be vital for optimizing the resource efficiency of agentic systems.

These future directions aim to build on our current findings, enhancing both the theoretical and practical aspects of agentic system design and evaluation. By developing these additional metrics and capabilities, we hope to provide a more comprehensive framework for assessing and improving the performance of agentic systems in increasingly complex environments.