Exploring Autonomous Agents: A Closer Look at Why They Fail When Completing Tasks

Ruofan Lu†, Yichen Li†, Yintong Huo‡*
†The Chinese University of Hong Kong, Hong Kong. Email: {rflu, ycli21}@cse.cuhk.edu.hk
‡Singapore Management University, Singapore. Email: ythuo@smu.edu.sg

***Abstract*—Autonomous agent systems powered by Large Language Models (LLMs) have demonstrated promising capabilities in automating complex tasks. However, current evaluations largely rely on success rates without systematically analyzing the interactions, communication mechanisms, and failure causes within these systems. To bridge this gap, we present a benchmark of 34 representative programmable tasks designed to rigorously assess autonomous agents. Using this benchmark, we evaluate three popular open-source agent frameworks combined with two LLM backbones, observing a task completion rate of approximately 50%. Through in-depth failure analysis, we develop a three-tier taxonomy of failure causes aligned with task phases, highlighting planning errors, task execution issues, and incorrect response generation. Based on these insights, we propose actionable improvements to enhance agent planning and self-diagnosis capabilities. Our failure taxonomy, together with mitigation advice, provides an empirical foundation for developing more robust and effective autonomous agent systems in the future.**

***Index Terms*—LLM agents, autonomous agents, failure analysis.**

I. Introduction

The advancement of LLMs has enabled a new trend in task automation through autonomous agents [1]–[4]. These agents are designed to work together to interpret human commands, autonomously produce and execute code, and return the answer directly to the user. This synergistic workflow is capable of resolving more complicated problems, and provides an "end-to-end" answer without user involvement in the technical coding processes.

Current agent systems are implemented as a collaborative team of specialized LLMs abstracted into three core components (Figure 1): (1) Planner, who decomposes complex user requests into a sequential plan of tasks, (2) Code generator, which converts each sub-task into executable and functional code with the use of various tools or plugins, and (3) Executor, which runs the code and integrates with development environments. The executor collects outputs and errors, forming a feedback loop to the planner for refining or returning an answer. Some studies refer to the combined code generation and execution process as the code interpreter [5]–[8].

Despite the promising capabilities of agent systems, their performance remains merely understood beyond a basic success rate metric [9]. There is an absence of systematic analysis in exploring the intricate communication among agents,

* Yintong Huo is the corresponding author.

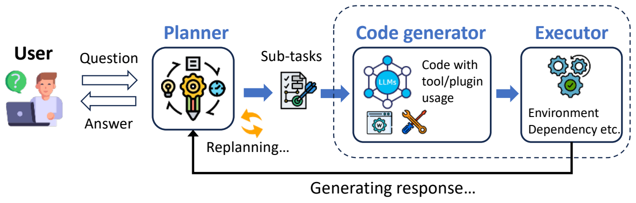

Fig. 1: The basic framework of an autonomous agent system.

their information passing mechanism, and the root causes of failures. For instance, whether a failure in web crawling task stems from incorrect planning or code generation. To advance these systems in the long run, it is essential to identify the fundamental bottlenecks by conducting a thorough investigation into failure origins.

To fill this gap, we built a benchmark containing 34 representative programmable tasks to evaluate current autonomous agents. Using this benchmark, we assessed three widely-used open-source agent frameworks paired with two LLM backbones. Our experiments show that approximately 50% of tasks are successfully completed by current agent collaborations, with failure causes including improper task planning, generation of nonfunctional code, and inadequate refinement strategies across iterations. Based on this analysis, we propose a three-level taxonomy of failure causes aligned with different task phases. Additionally, we offer several actionable recommendations aimed at enhancing planning capabilities and self-diagnosis mechanisms to advance autonomous agents.

We summarize the contribution of this paper as follows:

- *Benchmark*: We build a benchmark with programmable tasks to evaluate the capabilities of current autonomous agents.
- *Failure analysis*: Our evaluation of three popular agent frameworks reveals an approximately 50% task completion rate, categorizing failures into a three-level taxonomy based on task phases.
- *Actionable advice*: We provide suggestions to improve planning and self-diagnosis mechanisms, aimed at improving future autonomous agent systems.

II. Autonomous Agents in Software Engineering

Recent research has increasingly applied LLM-based agents to software engineering from two directions [10]: developing agents to handle specific SE tasks [11], [12] (e.g., debugging), and improving agent frameworks by enhancing role definitions and collaboration mechanisms [1], [13]. Within

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00330

TABLE I: A description of the evaluation of agent frameworks in terms of their design goals and collaborative strategies.

Framework	Design Goals	Collaboration Strategy
TaskWeaver	Translating user requests into executable code for task automation.	A stateful and linear workflow of plan generation, coding for each step, and an interpreter executes it.
MetaGPT	Generate projects simulating a software development company.	Encoding standard operating procedures into prompt sequences, following an assembly line to pass information to each other to finish complicated tasks.
AutoGen	A flexible framework for agents to solve tasks via conversation.	Adapted from flexible agent conversations, agents chat with each other, forming a dynamic and interactive collaboration to complete tasks.

these directions, agent-based approaches have demonstrated promising results across key SE domains measured by task success rates, such as requirement engineering [14], [15], code generation [16]–[18] and testing [19]–[21]. However, most prior work treats agent systems as monolithic entities and lacks in-depth analysis of their internal processes. In contrast, our study emphasizes a detailed analysis from the perspective of the agent framework itself, investigating the contributions and interactions of individual agents, exposing current limitations, and guiding the design of more effective, collaborative agent systems in the future.

III. Experiments

A. Benchmark construction

We selected three types of common coding tasks in daily life for our benchmark, with their sources as follows:

- **Web Crawling**: We search for the keyword "Web Crawling" on GitHub and Stack Overflow and construct tasks from the returned repositories and posts.
- **Data Analysis**: We incorporate a number of tasks from DABench [22], an end-to-end data analysis benchmark that requires agents to interact with an executable code environment to solve problems.
- **File Operations**: We curate tasks based on several Stack Overflow posts focusing on fundamental file operations using Python and Bash.

During task selection, we follow the criteria below to ensure benchmark quality. First, tasks have to be executable, with evaluation based on the outcomes of running the code rather than the code alone, differentiating our approach from typical code generation benchmarks. Second, tasks need to accommodate automated evaluation, leading us to exclude tasks like front-end interface generation that are challenging to assess programmatically. Finally, tasks are verified to be at least partially solvable by an agent, enabling meaningful exploration of design challenges; overly difficult or completely unsolvable tasks were left out. We carefully construct the benchmark consisting of 34 tasks, with human-verified ground-truth labels for automatic evaluation.

Metrics. Following prior work [23], [24], we measure success rate as the evaluation metric. A task is deemed successful only if its output exactly matches the ground-truth answer.

B. Studied subjects

Since our research focuses on the agent frameworks rather than the capabilities of the underlying LLMs, we select three popular open-source mainstream frameworks to examine their agent interconnectivity and collaboration mechanisms: TaskWeaver [1], MetaGPT [25], and AutoGen [2]. Table I presents their brief descriptions with design goals and collaboration strategies. For the LLM backbones embedded within each agent, we select GPT-4o [26] and GPT-4o mini [27] for evaluation.

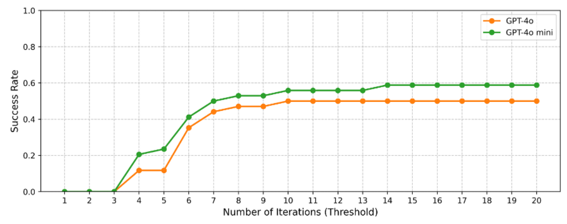

Fig. 2: The success rate concerning max iteration numbers.

TABLE II: Benchmark success rate (**GPT-4o**).

Agent	Web Crawling	Data Analysis	File Operations	All
TaskWeaver	16.67	66.67	75.00	50.00
MetaGPT	33.33	55.56	50.00	47.06
AutoGen	16.67	50.00	50.00	38.24

C. Implementation

After the task selection is completed, we designed a general prompt template to standardize requests across different categories of tasks, which contains the Task Description, Instruction, Constraints, and Environment Information.

We implemented the benchmark as a toolbox enabling automated execution and evaluation. The agent frameworks were deployed on a Linux server, each running within its respective containers and sandboxes, using Python 3.10.14. We used MetaGPT 0.8.1, AutoGen 0.2.36, and TaskWeaver (commit hash number cf76c3b70b29ef64185fd3c9af0510c9e2fcc51e). For the agent backbones, we employed two models: GPT-4o (gpt-4o-2024-08-06) and GPT-4o mini (gpt-4o-mini-2024-07-18). Results from task executions underwent post-processing and information extraction to facilitate efficient automated evaluation, while full logs were documented for later analysis.

IV. Result Analysis

This section presents evaluation results and summarizes the failure causes into a taxonomy according to experiments.

A. Quantitative analysis

Table II and III presents the evaluation results on GPT-4o and GPT-4o-mini with discussions as follows.

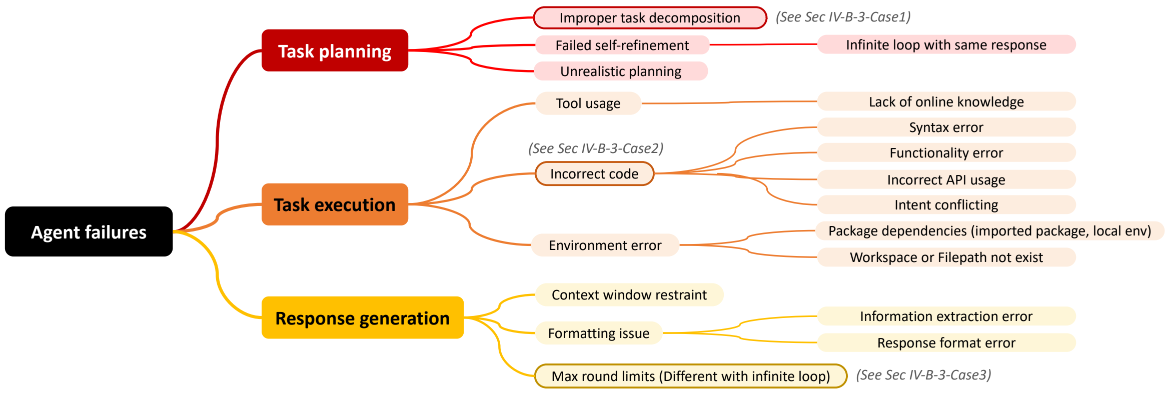

Fig. 3: The agent failure taxonomy, where the most frequent failure is highlighted and illustrated in Sec. IV-B3.

TABLE III: Benchmark success rate (**GPT-4o mini**).

Agent	Web Crawling	Data Analysis	File Operations	All
TaskWeaver	50.00	55.56	100.00	58.82
MetaGPT	25.00	66.67	50.00	50.00
AutoGen	41.67	44.44	100.00	50.00

Agent performance decreases on reasoning-intensive tasks. Agent performance varies by task. Using GPT-4o, agents perform well in Data analysis and File operations, with TaskWeaver scoring 67% and 75%, respectively. Web crawling is more challenging, with scores as low as 17%, due to its reasoning-intensive nature, requiring code generators to infer element paths from user intent and HTML data. Compared to unstructured web tasks, simpler, structured tasks like data analysis benefit more from autonomous agents.

A cross-agent comparison reveals distinct specializations. With GPT-4o, TaskWeaver leads in structured tasks such as Data analysis (67%) and File operations (75%), while MetaGPT excels in Web crawling (33%). Both TaskWeaver and AutoGen achieve perfect scores in File Operations, highlighting their architectures' strong compatibility with the lightweight model for executing precise, procedural tasks.

While stronger models have higher reasoning capabilities, they might run into the overthinking issue and compromise results. In our evaluation, both TaskWeaver and MetaGPT show strong results with GPT-4o, scoring 50.00% and 47.06% overall. Surprisingly, the smaller GPT-4o-mini outperforms, especially in web crawling tasks, indicating that simpler models can remain highly competitive. Analysis of execution logs reveals that GPT-4o's failures stem from a conflict between its task-planning processes (such as requesting additional confirmations) and built-in safety constraints (like denying web scraping), causing it to produce valid plans but then halt execution. Such "overthinking" ultimately results in task failure. The superior performance of GPT-4o-mini is consistent with prior research findings [28].

More iterations improve success, but with diminishing gains after a certain threshold. Figure 2 shows the success rate over the threshold of iterations in TaskWeaver. The success rate is zero for the first two iterations, indicating that a minimum number of attempts is necessary to solve the tasks. Between iterations 3 and 10, there is a rapid improvement in the success rate for both models, with the most significant gains occurring in this phase. After 10 iterations, increasing the maximum number of iterations yields only marginal gains.

B. *Failure study*

1) Manual investigation: The experiments were conducted across 34 tasks, 2 LLM backbones, and 3 agent frameworks, totaling 204 runs, with 104 task failures recorded along with detailed experimental logs. We recruit three authors, each with a minimum of two years of programming experience, to review the agent execution logs. These logs include comprehensive information such as prompt construction for each agent, individual agent outputs, and execution results for every iteration.

Our investigation began with the first-level failure taxonomy, where all annotators agreed to categorize failures according to the roles of key phases: task planning, task execution, and response generation. Next, each annotator independently reviewed the failure logs to summarize second-level failure reasons. Finally, they collaboratively discussed their categorizations and reached consensus on the final taxonomy.

2) Failure taxonomy: Figure 3 presents the failure taxonomy, which encompasses 19 distinct causes across three tiers.

Task planning. A planner is responsible for breaking down user instructions into a sequence of executable sub-tasks for the code generator. This role is critical since the planner's output directly guides subsequent agents and largely determines the success of the overall framework. We identified three common issues in planning: (1) improper task decomposition that generates steps that are logically incorrect or unsuitable for the assigned task; (2) failed self-refinement involves the model is unable to learn from its past errors, causing it to repeat the same failed sub-tasks in an infinite loop; (3) unrealistic planning refers to producing a sequence of plausible steps but exceeds the practical capabilities of downstream agents, making the sub-tasks impossible to execute.

Task execution. Task execution is the phase where the agent attempts to carry out the planned sub-tasks, involving the failures from the code generator and executor. Existing agent frameworks encounter three main failures: (1) the generator agent fails to exploit external tools (e.g., available functions),

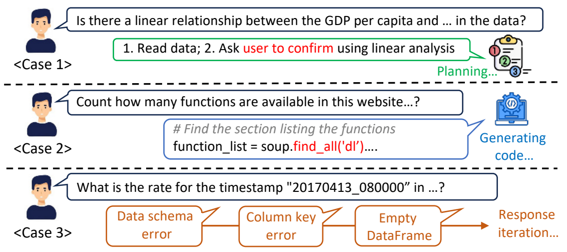
Is there a linear relationship between the GDP per capita and … in the data?
1. Read data; 2. Ask user to confirm using linear analysis
Planning…
<Case 1>
Count how many functions are available in this website…?
# Find the section listing the functions
function_list = soup.find_all('dl')….
Generating code…
<Case 2>
What is the rate for the timestamp "20170413_080000" in …?
Data schema error
Column key error
Empty DataFrame
Response iteration…
<Case 3>

Fig. 4: Three most common failure types.

often due to a lack of online or tool-use knowledge. (2) the generator agent produces flawed code with syntax errors, functionality errors (executable but deviating from the intended output), incorrect API usage with wrong parameters, or showing conflicts to its original goal, and (3) executions also fail with improper environmental setup, such as missing dependency package and accessing a file that does not exist.

Response generation. Response generation is the final stage where the agent produces output for the user or the planner to use in subsequent iterations. Failures at this stage relate to how results are perceived and presented, even after the code has been executed. Three main failures causes are: (1) context window restraint: the agent loses parts of the conversation, leading to responses that are disconnected from previous interactions (e.g., an overly large HTML file in a web crawling task), (2) formatting issue: the agent's output contains irrelevant information or does not comply with the required format (e.g., returning a sentence when a number is expected), (3) maximum rounds exceeded: The agent reaches a preset limit on the number of interaction turns without successfully completing the task, despite attempting various plans.

3) Common failure analysis: For the most common failure in phases (i.e., planning, execution, and response generation), we present one case for each in Figure 4 and analyse as follows.

Case 1: The user asks the agent to verify a linear relationship between data. However, instead of proceeding directly to generate the necessary code for the analysis, the planner adds a redundant step: asking the user for confirmation to use linear analysis, though such usage has been specified in task description. This unnecessary clarification introduces a bottleneck, halting the process until user feedback is provided. Such redundant planning not only delays the task but also degrades the user experience by creating unnecessary interaction.

Case 2: When tasked with counting the number of functions on a website, the agent generates code that operates on an incorrect assumption. The code `soup.find_all('dl')` presumes that all `<dl>` HTML tags on the page are used exclusively for listing functions. However, on complex webpages like technical documentation, these tags are often used for a variety of purposes, including navigation, definitions, or other structural elements. This flawed assumption leads to an incorrect count and demonstrates a failure to understand the contextual use of HTML structure, resulting in faulty code.

Case 3: The agent fails when trying to find a specific data point. It first gets a `KeyError` due to an additional space in a column name. The agent then switches to an alternative strategy of retrieving the entire row, which also fails in `Empty DataFrame`. Such an error implies that the agent faces challenges in self-correcting based on the output of its previous checks. It therefore leads to a loop of failures that ultimately exceeds the maximum attempts and causes the task to fail.

V. Actions on Mitigating Agent Failures

The failures analyzed highlight critical weaknesses in agent systems, particularly in planning and error correction. To address these, we propose two key strategies as follows.

Promoting planning ability with learning-from-feedback. Planner is the first and fundamental component of an autonomous agent, decomposing complex tasks into executable steps. We therefore advocate for a "learning-from-feedback" design, where agents learn to re-plan from their previous operational environment feedback. Recent work shows that agents can dynamically adjust plans based on tool feedback, deciding whether to refine or restart [29], [30] the pre-defined plan, avoiding rigid and illogical steps. Such a feedback-aware mechanism also shows promise in software engineering applications like program repair [31] and code generation [32], [33]. This allows the agent to adapt new strategies when faced with unexpected outcomes.

Developing early-stop and navigation mechanism. Failures like infinite loops and hitting round limits highlight the agent's inability to recover from repeated mistakes. To this end, future agent systems can develop a meta-controller that navigates to a certain agent upon root cause analysis, either re-planning to correct a strategic error or invoking a specialized tool to fix a local execution fault. Proper navigation can efficiently fix the problem, reducing task attempts and improving reliability. Moreover, if the system detects repetitive, unresolved errors, the mechanism should trigger an "early stop", halting the process before it hits the maximum round limit, thereby saving resources.

VI. Conclusion and Future Work

This study investigates autonomous agent systems, focusing on their collaboration mechanisms and the reasons behind their failures in completing end-to-end tasks. We evaluate three popular agent frameworks using a newly developed benchmark, analyze their outcomes, and classify the causes of failure. Additionally, we propose two practical design strategies for agent frameworks to address common failure modes. In the future, we aim to enrich the benchmark and implement these strategies. Our benchmark data and evaluation framework can be found at https://github.com/lurf21/Agent_Evaluation_Framework.

VII. Acknowledgment

We appreciate all the anonymous reviewers for their valuable and practical comments. The work was supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant.

References

[1] B. Qiao, L. Li, X. Zhang, S. He, Y. Kang, C. Zhang, F. Yang, H. Dong, J. Zhang, L. Wang *et al.*, "Taskweaver: A code-first agent framework," *arXiv preprint arXiv:2311.17541*, 2023.

[2] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen LLM applications via multi-agent conversations," in *First Conference on Language Modeling*, 2024.

[3] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin, "CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 13 643–13 658.

[4] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji, "Executable code actions elicit better llm agents," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[5] A. Zhou, K. Wang, Z. Lu, W. Shi, S. Luo, Z. Qin, S. Lu, A. Jia, L. Song, M. Zhan, and H. Li, "Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification," in *International Conference on Representation Learning*, vol. 2024, 2024, pp. 4468–4494.

[6] C. Zhang, S. Zhang, Y. Hu, H. Shen, K. Liu, Z. Ma, F. Zhou, W. Zhang, X. He, D. Lin *et al.*, "Cibench: Evaluating your llms with a code interpreter plugin," *arXiv preprint arXiv:2407.10499*, 2024.

[7] A. Low and Z. Y. Kalender, "Data dialogue with chatgpt: Using code interpreter to simulate and analyse experimental data," *arXiv preprint arXiv:2311.12415*, 2023.

[8] Y. Zhang, Z. Ma, Y. Ma, Z. Han, Y. Wu, and V. Tresp, "Webpilot: A versatile and autonomous multi-agent system for web task execution with strategic exploration," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 22, 2025, pp. 23 378–23 386.

[9] C. Guo, X. Liu, C. Xie, A. Zhou, Y. Zeng, Z. Lin, D. Song, and B. Li, "Redcode: Risky code execution and generation benchmark for code agents," *Advances in Neural Information Processing Systems*, vol. 37, pp. 106 190–106 236, 2024.

[10] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large language model-based agents for software engineering: A survey," *arXiv preprint arXiv:2409.02977*, 2024.

[11] J. He, C. Treude, and D. Lo, "Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–30, 2025.

[12] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Demystifying llm-based software engineering agents," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025.

[13] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[14] D. Jin, Z. Jin, X. Chen, and C. Wang, "Mare: Multi-agents collaboration framework for requirements engineering," *arXiv preprint arXiv:2405.03256*, 2024.

[15] B. Wei, "Requirements are all you need: From requirements to code with llms," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 2024, pp. 416–422.

[16] Y. Liu, P. Gao, X. Wang, J. Liu, Y. Shi, Z. Zhang, and C. Peng, "Marscode agent: Ai-native automated bug fixing," *arXiv preprint arXiv:2409.00899*, 2024.

[17] Y. Ding, M. J. Min, G. Kaiser, and B. Ray, "Cycle: Learning to self-refine the code generation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 392–418, 2024.

[18] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–30, 2024.

[19] N. Mündler, M. Müller, J. He, and M. Vechev, "Swt-bench: Testing and validating real-world bug-fixes with code agents," *Advances in Neural Information Processing Systems*, vol. 37, pp. 81 857–81 887, 2024.

[20] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[21] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[22] X. Hu, Z. Zhao, S. Wei, Z. Chai, Q. Ma, G. Wang, X. Wang, J. Su, J. Xu, M. Zhu, Y. Cheng, J. Yuan, J. Li, K. Kuang, Y. Yang, H. Yang, and F. Wu, "Infiagent-dabench: evaluating agents on data analysis tasks," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[23] B. Bogin, K. Yang, S. Gupta, K. Richardson, E. Bransom, P. Clark, A. Sabharwal, and T. Khot, "Super: Evaluating agents on setting up and executing tasks from research repositories," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 12 622–12 645.

[24] S. Yao, N. Shinn, P. Razavi, and K. Narasimhan, "τ-bench: A benchmark for tool-agent-user interaction in real-world domains," *arXiv preprint arXiv:2406.12045*, 2024.

[25] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, 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," in *The Twelfth International Conference on Learning Representations*, 2024.

[26] OpenAI, "Gpt-4o," 2024, https://openai.com/index/hello-gpt-4o/.

[27] ——, "Gpt-4o mini," 2024, https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

[28] J. A. Schnabel, J. R. Trippas, F. Scholer, and D. Hettiachchi, "Multi-stage large language model pipelines can outperform gpt-4o in relevance assessment," in *Companion Proceedings of the ACM on Web Conference 2025*, 2025, pp. 1288–1292.

[29] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR)*, 2023.

[30] S. Hao, Y. Gu, H. Ma, J. Hong, Z. Wang, D. Wang, and Z. Hu, "Reasoning with language model is planning with world model," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 8154–8173.

[31] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 819–831.

[32] Y. Peng, A. D. Gotmare, M. R. Lyu, C. Xiong, S. Savarese, and D. Sahoo, "Perfcodegen: Improving performance of llm generated code with execution feedback," in *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 2025, pp. 1–13.

[33] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, "Self-edit: Fault-aware code editor for code generation," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 769–787.