GitAgent: Facilitating Autonomous Agent with Tool Extension from GitHub

Anonymous ACL submission

Abstract

While Large Language Models (LLMs) have demonstrated exceptional proficiency in natural language processing, their efficacy in addressing complex, multifaceted tasks remains limited. A growing area of research focuses on LLM-based agents equipped with external tools capable of performing diverse tasks. However, existing LLM-based agents only support a limited set of tools which is unable to cover a diverse range of user queries, especially for those involving expertise domains. It remains a challenge for LLM-based agents to extend their tools autonomously when confronted with various user queries. As GitHub has hosted a multitude of repositories which can be seen as a good resource for tools, a promising solution is that LLM-based agents can autonomously integrate the repositories in GitHub according to the user queries to extend their tool set. In this paper, we introduce **GitAgent**, an agent capable of achieving the autonomous tool extension from GitHub. GitAgent follows a four-phase procedure to incorporate repositories and it can learn human experience by resorting to GitHub Issues/PRs to solve problems encountered during the procedure. Experimental evaluation on our constructed GitBench involving over 21 repositories with 339 user queries demonstrates GitAgent's effectiveness against baselines.

1 Introduction

Significant advancements in Large Language Models (LLMs) (OpenAI, 2022, 2023) have showcased remarkable proficiency in natural language processing (NLP). Recently, a notable trend emerged to develop LLMs equipped with external tools (e.g., search engines (Nakano et al., 2021; Qin et al., 2023a), calculators (Schick et al., 2023), and knowledge bases (Parisi et al., 2022)) to function as agents capable of performing complex tasks, extending the capability boundary of LLMs beyond NLP tasks. Consequently, a variety of LLM-based agents have surfaced, e.g., ChatGPT Plugins (OpenAI), AutoGPT (AutoGPT), AutoGen (Wu et al., 2023b), XAgent (XAgent, 2023).

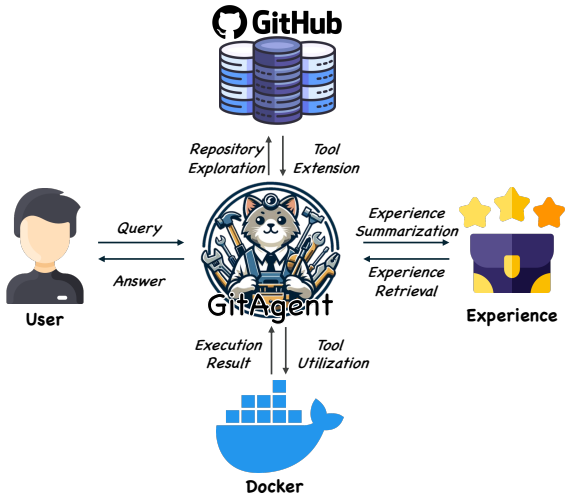

Figure 1: Illustration of GitAgent which can autonomous extend tools from GitHub to accomplish diverse user queries.

However, the range of tools that existing LLM-based agents can utilize remains notably limited (Parisi et al., 2022; Schick et al., 2023; Patil et al., 2023). It is particularly evident in real-world scenarios where user queries are highly diverse and often necessitate specific, domain-specialized tools. Even agents with a broad range of tools still struggle to cover all proper tools to address these specialized queries (Qin et al., 2023c). Hence, when confronted with various queries, it is necessary for agents to autonomously extend their supported toolset to meet the user demands. We observe that there already exist numerous repositories in host platforms like GitHub which is a good tool resource especially including many professional tools across diverse domains (e.g., `Qlib` for the quantitative investment). If LLM-based agents could search and deploy relevant repositories from GitHub, thereby extending their tools on their own, they can au-

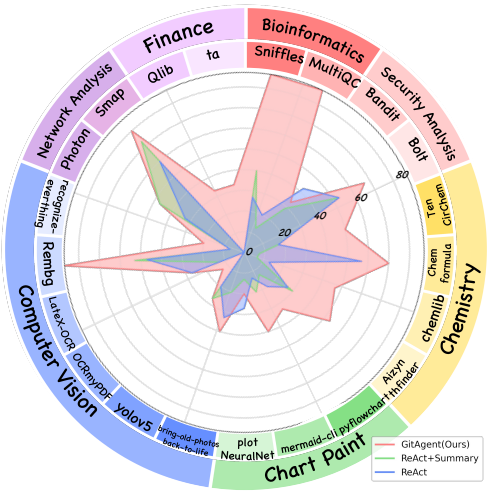

Figure 2: Illustration of the performance of GitAgent against baselines on 21 repositories from 7 domains in GitBench.

tonomously extend their capability boundary to address diverse user queries.

Tool extension based on GitHub has some challenges due to the lack of standardization of available repositories. (1) Presence of Flaws: Repositories on GitHub cannot guarantee absolute perfection, devoid of any flaws. Even those developed by notable organizations (e.g., Microsoft) may still suffer from inadequate maintenance, resulting in the presence of potential bugs in their source codes. (2) Incomplete Documentation. Documentations (e.g., README), serving as the primary guide, may not comprehensively detail all the necessary information for effective utilization. In particular, developers might not invest the required effort to create detailed, user-friendly documentation. Hence, agents will struggle to integrate repositories and further utilize them to accomplish user queries.

In this paper, we introduce **GitAgent**, an Autonomous Agent enabled to Extend Tools from GitHub. GitAgent decomposes the tool extension procedure into four phases: *Search*, *Setup*, *Apply*, and *Store*. It begins by searching suitable repositories from GitHub according to user queries and then configuring its environment. Subsequently, GitAgent utilizes the configured repository to fulfill the user query and ends by storing the repository for efficient handling of subsequent queries. To address the non-standardization challenge, as GitHub provides Issues and Pull Requests (PRs) which contains **human practice experience**, GitAgent can resort to Issues and PRs to learn how human practitioners solve problems (e.g., bugs) during the tool extension. Specifically, upon encountering a problem, GitAgent first summarizes the problem as a query and utilizes GitHub Issues/PRs API to search for pertinent Issues/PRs. It then judges the relevance of these Issues/PRs to determine their relevance and applicability to the current problem sequentially. If identifying a suitable one, it reads the contents of Issues/PRs to learn how to solve the problem. Finally, GitAgent summarizes the practical experience gained during the procedure to guide future use. When other queries come, GitAgent will utilize its summarized experience to avoid the same problem to improve the performance.

To evaluate the tool extension ability of our GitAgent, we collect a high-quality dataset **GitBench** which contains 21 repositories across 7 domains chosen manually from GitHub and designs 339 queries in total accordingly. Our results demonstrate that GitAgent achieves a higher pass rate against several baselines in addressing the user queries. Furthermore, we discuss the reasons behind failures, shedding light on potential future research directions.

In summary, our contributions are threefold: (1) We propose GitAgent, an autonomous LLM-based agent. It can autonomously integrate repositories in GitHub to extend tools to meet the diverse demands of user queries. (2) To address the non-standardization challenge, GitAgent can learn human practice experience based on Issues/PRs. It can further summarize the practical experience for utilization in the future to avoid the same problem. (3) We collect a high-quality dataset GitBench with diverse expertise domains and repositories to evaluate the tool extension ability. Experimental results demonstrate the effectiveness of GitAgent.

2 Methodology

2.1 Task Formulation

Given the user query Q, the agent aims to accomplish it and give the response R to the user, involving autonomously tool extension from external resources (i.e., GitHub).

$$Q \xrightarrow{\mathcal{M}_P(Q)} R \tag{1}$$

where $\mathcal{M}$ refers to the agent, P refers to the system prompt of the agent. In fact, to answer the query, the agent usually should take multi-step actions to

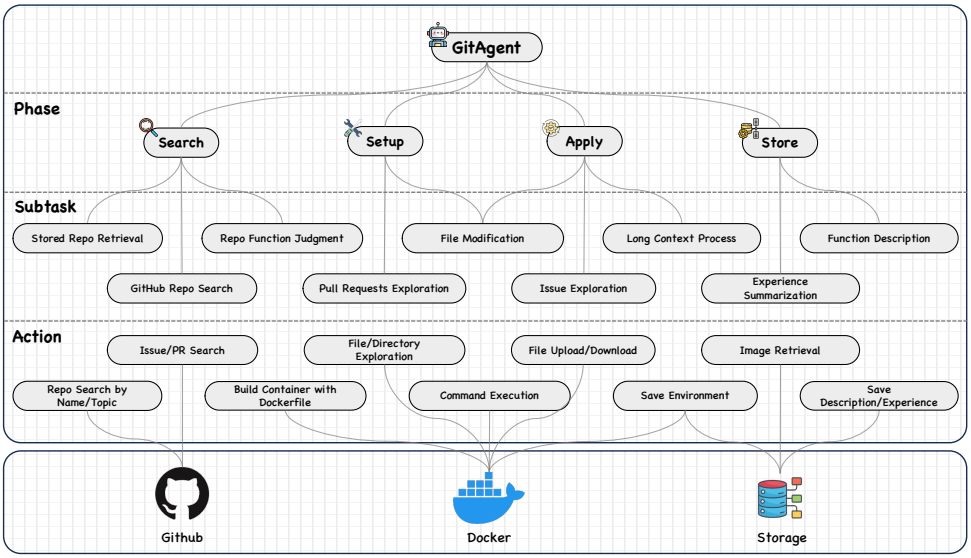

Figure 3: Illustration of the hierarchical task decomposition strategy of GitAgent.

solve it step-by-step.

$$\begin{aligned} A_i &= \mathcal{M}_P(Q, A_1, O_1, \cdots, A_{i-1}, O_{i-1}) \\ O_i &= \mathcal{E}(A_i), \quad i = 1, 2, \cdots, n \end{aligned} \tag{2}$$

where A_i refers to actions the agent takes (e.g., GitHub API calling, shell command execution, file I/O operation, etc), n refers to the number of the actions to accomplish the query Q, $\mathcal{E}$ is the environment, O_i is the observation from the environment after taking actions. Such an action-observation sequence forms a trajectory of the agent to accomplish the query.

$$Q \xrightarrow{\mathcal{M}_P(Q)} A_1 \xrightarrow{\mathcal{E}(A_1)} O_1 \cdots \xrightarrow{\mathcal{M}_P(Q, A_1, O_1 \cdots, A_n, O_n)} R \tag{3}$$

Within the trajectory, it would accumulate historical information $H = \{A_1, O_1, \cdots, A_n, O_n\}$ for each action-taking.

2.2 Overall Framework

As Figure 3 demonstrates, GitAgent employs a hierarchical task decomposition strategy to achieve tool extension from GitHub. GitAgent first contains 4 critical phases: *Search*, *Setup*, *Apply*, and *Store*. Each phase will continue to be decomposed into several subtasks with a series of actions.

Formally, GitAgent decomposes the whole process into four phases. The trajectory of GitAgent can be represented as follows (we ignore the action parts for clear representation):

$$Q \rightarrow O_{\text{Search}} \rightarrow O_{\text{Setup}} \rightarrow O_{\text{Apply}} \rightarrow O_{\text{Store}} \rightarrow R \tag{4}$$

where $O_{\text{Search}}, O_{\text{Setup}}, O_{\text{Apply}}, O_{\text{Store}}$ are the intermediate observations (i.e., results) of four phases respectively. Each phase is decomposed into several subtasks, which can be represented as:

$$O_{\text{phase}_i} \rightarrow O_{\text{Subtask}_1} \rightarrow O_{\text{Subtask}_2} \cdots \rightarrow O_{\text{phase}_{i+1}} \tag{5}$$

Similarly, subtasks are achieved by several actions:

$$O_{\text{Subtask}_i} \rightarrow O_{\text{Action}_1} \rightarrow O_{\text{Action}_2} \cdots \rightarrow O_{\text{Subtask}_{i+1}} \tag{6}$$

Search During the Search phase, the agent finds suitable repositories that can be used to accomplish user queries. The repositories come from two resources: repositories stored in the past and repositories hosted in GitHub. Hence, this phase contains three subtasks: (1) Stored Repository Retrieval: The agent retrieves from existing stored repositories by judging their suitability with the user query. If a repository is deemed suitable, its environment is loaded, bypassing the subsequent Setup phase, and directly enters the Apply phase. (2) GitHub Repository Search: If the stored repositories cannot be used to accomplish user queries, the agent will resort to GitHub to search for suitable ones. There are two ways to search for repositories. If the user queries specify the particular repositories, the agent will take action to call GitHub "*search_by_name*" API directly. If not, GitAgent should search for the proper repositories according to the repository function. As GitHub lacks the semantic search API, we resort to the topic search API. The agent would extract a list of potential GitHub topics from the query and subsequently call GitHub *search_by_topic* API to search repositories. (3) Repository Function Judgment: Upon

obtaining repository candidates, the agent judges each repository's suitability in resolving the user query. The agent will read the README of each repository to understand its function and then deliver a judgment on the repository's suitability.

Setup Upon identifying the suitable repositories, the agent would initiate the *Setup* phase aimed at configuring their execution environment. The agent commences by cloning repositories from GitHub and executing commands (including the installation of dependencies and download of requisite data) according to the README. Due to the non-standardization problem, there may exist flaws or bugs in the repositories so the agent will initiate a *Pull_Requests_Exploration* or *Issues_Exploration* subtask to leverage human practice experience to resolve the problems (details see in Section 2.3). If necessary, the agent will initiate a *File_Modification* subtask to modify the source files of the repository to fix the bugs.

Apply Given the configured environment, the agent proceeds to apply the repository to address the user query. This application process varies based on the complexity and design of individual repositories. Well-developed repositories provide clear entry for allowing straightforward applications (e.g., Command-Line Interface). Nevertheless, for those non-standardized repositories that do not provide clear entry, especially lacking detailed documentation, the agent needs to resort to human experience again (see in Section 2.3). If extensive output (e.g., lengthy execution logs) ensues, the agent needs to go to the *Long_Context_Process* subtask which writes a Python program (e.g., regular expressions) to extract critical information from the lengthy file. Thus, the *File_Modification* subtask is also involved.

Store After accomplishing the user query, the agent proceeds to store the repository together with its execution environment to facilitate future usage, especially for recurring or similar user queries. Specifically, the execution environment will be stored in a docker image so if a similar query comes, the agent can retrieve this repository and restore it to apply directly. To enhance the stored repository retrieval, the agent should abstract its understanding of the repository's functionalities (i.e., the *Function_Description* subtask) and summarize the experience in the apply phase (i.e., the *Experience_Summarization* subtask).

Note that although we design this hierarchical strategy, which phase, subtask, or action to be achieved is decided by GitAgent itself dynamically. We do not limit the agent's behavior strictly.

2.3 Human Experience Learning

Due to the non-standardization of repositories in GitHub, some repositories lack perfect README that may not contain critical information that is necessary to accomplish the Setup and Apply phases. Evenly, there may exist flaws in the source code. Under such circumstances, it is challenging for the agent to overcome by itself. Learning human experience becomes an efficient approach to resolving them. In practice, the agent can leverage GitHub Issues and Pull Requests to learn insightful human practice experience including bug-fixing strategies or application solutions.

Specifically, the agent first summarizes the encountered bugs or the required critical information into a problem statement Q_S according to the accumulated history H.

$$A_{\text{search}}(Q_S) = \mathcal{M}_{P_{\text{abs}}}(Q, H) \tag{7}$$

where P_{abs} refers to the prompt designed for problem statement summarization, $A_{\text{search}}(\cdot)$ refers to the action of calling GitHub Issues/PRs search API. And then the Issues/PRs API is called to search solution candidates according to the problem statement Q_S.

$$O_{\text{search}} = \mathcal{E}(A_{\text{search}}(Q_S)) \tag{8}$$

where $O_{\text{search}} = \{C_1, C_2, \cdots, C_n\}$ is the list of Issues/PRs returned by the GitHub API, C_i refers to the content of each Issue/PR. Getting the list of Issues/PRs, subsequently, the agent meticulously reviews and evaluates them sequentially to judge their applicability in resolving problems.

$$(J, S) = \mathcal{M}_{P_{\text{judge}}}(Q_S, C_i), \quad i = 1, 2, \cdots, n \tag{9}$$

where P_{judge} is the prompt for Issues/PRs applicability judgment, (J, S) is the judgment result containing the judgment J of the applicability and the solution S summarized by the agent to report how to solve the problem comprehensively.

After accomplishing the user queries, the agent will summarize the practice experience gained from the entire procedure.

$$E = \mathcal{M}_{P_{\text{exp}}}(Q, H) \tag{10}$$

where P_{exp} refers to the prompt designed for the practical experience summarization, and E is the

summarized practice experience. Then, the summarized practice experience will be stored together with the repositories, consolidating pertinent information alongside the corresponding docker image metadata. It constitutes the knowledge of the repository that can be retrieved during the Search phases. Once the stored repositories are retrieved, the stored experience will be concatenated into the context to learn the successful experience.

$$R = \mathcal{M}_P(Q, E) \tag{11}$$

Hence, the summarized experience can facilitate efficient utilization for addressing similar user queries in the future.

3 Experiment

3.1 Experiment Settings

Dataset To test the necessity of tool extension, we construct the high-quality dataset GitBench. We manually select 21 repositories for 7 different expertise domains from GitHub which obviously surpass the capability boundary of existing LLMs to evaluate our method. For each repository, we prompt `GPT-4` to generate 5 user queries according to README and filter the low-quality queries manually. Then, we manually design three types of prompts for each repository, which would be concatenated before user queries: (1) *Explicit Repo Prompt*: The prompt explicitly specifies which repository should be used together with its GitHub address. The agent can clone the provided repository directly from GitHub, simplifying the search procedure. (2) *Implicit Repo Hint*: Instead of specifying the target repository, this prompt only provides some keywords like domains or careers that the repository should involve. The agent should summarize the relevant topics to search proper repositories in GitHub. (3) *No Repo Prompt*: No prompt are concatenated before user queries. The agent should infer the repositories by itself. Concatenating these 3 prompts, we finally get 339 instructions in total. We manually constructed golden answers for each query for evaluation. The statistics of GitBench are listed in Table 1.

Domain	Num. of Repo	Num. of Query
Finance	2	45
Chemistry	4	66
Bioinformatics	2	30
Computer Vision	6	90
Network Analysis	2	30
Security Analysis	2	30
Chart Plot	3	48
Total	21	339

Table 1: Statistics of GitBench.

Metrics We employ **Pass Rate** as the metric for evaluation, utilizing GPT-4 to calculate this rate. The evaluation framework involves a comprehensive assessment of the entire execution process of a query, whereby GPT-4 analyzes the process and assigns a score ranging from 0 to 10. The tool extension process is segmented into four distinct phases, each of which is critically important to the overall evaluation. GPT-4 meticulously evaluates the degree of completion for Search, Setup and Apply phases (we ignore the Store phase as it does not influence the final answer generation), awarding a score within the interval of $[0, 3]$ based on the level of accomplishment observed. Additionally, GPT-4 undertakes an evaluation of the final answer provided by the agent, comparing it against a pre-established "golden answer" to determine its accuracy. This comparison results in a score allocation within the range of $[0, 1]$ for the correctness of the answer. Then, the scores are aggregated. An execution process that achieves a final score exceeding 9 is considered successful. This outcome is classified as a "pass" for the given query. To derive the Pass Rate, we compute the ratio of queries that have successfully passed to the total number of queries assessed. To validate the reliability of GPT-4 evaluation in pass rate, we sample 40 queries for each method. Then we engage humans to annotate the pass rate. Our GPT-4 evaluation demonstrates a high agreement of 87.5% in pass rate with human annotators. This result shows that our evaluation generates highly similar evaluation results to humans and can be viewed as a credible evaluation. For a more elaborate evaluation critique, please refer to Appendix A.

Baseline To validate the effectiveness of our GitAgent, we design the following baselines: (1) ReAct (Yao et al., 2022): ReAct is widely-used LLM-based agent task-solving technique (XAgent, 2023; AutoGPT; Wu et al., 2023b). It can accomplish intricate tasks on the fly by decomposing them into explicit intermediate steps. In our settings, ReAct is equipped with the same actions as our GitAgent to extend tools from GitHub for fair comparison. (2) ReAct+Summary: Due to the complexity of the tool extension, the whole process tends

Method	GPT-3.5	GPT-4
ReAct	2.4	24.6
ReAct+Summary	0.0	24.4
GitAgent	17.1	**47.3**

Table 2: Pass Rate of GitAgent against baselines.

Prompt	Search Success Rate
Explicit Repo Prompt	96.0
Implicit Repo Prompt	66.0
No Repo Prompt	32.0

Table 3: Analysis for the search difficulty.

to involve lengthy context, surpassing the context window of LLMs. Hence, we design this ReAct variant which will summarize the context when the length of context reaches the threshold.

Implementation Details We implement our GitAgent based on the `gpt-4-0125-preview` version and `gpt-3.5-turbo` with 0.7 temperature. All actions including API callings, command executions, and agent operations (e.g., summarization) are implemented based on the Function Calling feature[1] of GPT-4. To denote the finish of each phase (Search, Setup, Apply, Store), we additionally add *Submit* action in the function list. If the agent thinks it has accomplished each phase, it should call the *Submit* function to finish the phase.

3.2 Overall Evaluation

Table 2 reports the success rate of each method and we get several observations. (1) ReAct achieves a lower Pass Rate than GitAgent in both settings, which demonstrates that even if providing all actions, simply adapting the ReAct framework but not designing solutions for the tool extension cannot achieve good results. (2) ReAct+Summary achieves lower performance than ReAct because the summarization will lose critical information (especially in GPT-3.5 version which achieves 0.0%) Thus, it is infeasible to avoid the over-length problem by simply summarizing the long context. (3) All GPT-4-based methods outperform their GPT-3.5 counterparts significantly, showing that tool extension is a challenging task needing powerful LLMs to achieve. (4) GitAgent significantly outperforms all baselines in both settings, demonstrating the effectiveness of the tool extension.

3.3 Impact of Search Difficulty

As searching from GitHub to derive proper repositories is a challenging task, we conduct detailed experiments to show the impact of different repository search difficulties. As we introduced in § 3.1, we design three types of prompts to denote the target repositories. We calculate the Seach Success Rate of each type of prompt. If GitAgent can get the correct repositories of a query, it is denoted as search success. Then, we calculate the proportion of the search success queries over all queries of each type of prompt. The experimental results are listed in Table 3. From it, we can observe that Explicit Repo Prompt achieves the highest Search Success Rate (near 100%) as the prompt has specified repositories. Implicit Repo Prompt achieves 66.0% Search Success Rate, showing that GitAgent can infer the relevant GitHub Topics based on the domains or careers. As people will assign roles before using LLMs (Wang et al., 2023c), GitAgent can utilize this role prompts to search repositories to accomplish queries. Finally, if no repository prompt is provided, the search success rate decreases significantly. It demonstrates that GitAgent falls short of inferring GitHub Topics based on the query only. It needs further research in the future to improve the performance in this situation.

3.4 Impact of Setup&Apply Phase

Setup/Apply Difficulty	Easy	Medium	Hard	Total
Easy	72.3	69.0	56.2	64.4
Medium	60.7	70.0	41.5	57.7
Hard	50.0	67.0	51.5	57.4
Total	64.1	68.7	51.4	60.7

Table 4: Analysis for the setup & apply difficulty.

To study the impact of the Setup difficulty of different repositories, we manually divide the collected repositories into three classes: *Setup-Easy*: The README file provides a detailed and correct setup tutorial. *Setup-Medium*: The README file misses some details or has some slight flaws. *Setup-Hard*: The README file provides an incorrect tutorial (e.g., lacking maintenance). Similarly, we recalculate Pass Rate for these repositories. Table 4 gives the results and we find that both Medium and Hard repositories achieve similar Pass Rate (about 57%). We attribute it to the human experience learning capability which can learn human experience to overcome the imperfect README.

[1] https://openai.com/blog/function-calling-and-other-api-updates

Method	Pass Rate
GitAgent w/ PRs&Issues	47.3
GitAgent w/o PRs&Issues	40.3

Table 5: Results of human experience analysis.

Method	w/o SelfExp	w/ SelfExp
GPT-3.5	17.6	58.8
GPT-4	47.0	82.3

Table 6: Results of summarized experience analysis.

Similarly, we also study the Apply difficulty and we divide repositories into three classes: *Apply-Easy*: Simply running some commands in the command. *Apply-Medium*: Needing to write configure files or download some extra resources (e.g., data). *Apply-Hard*: Needing to modify the source code of the repositories. Pass Rate is also reported in Table 4 and we observe that Medium even achieves a higher Pass Rate than Easy. Nevertheless, Pass Rate of Hard decreases over 12% compared with Easy and Medium, which demonstrates that GitAgent can solve easy and medium repositories but needs further study to conquer hard repositories in the future.

3.5 Impact of Human Experience Learning

To evaluate the effectiveness of the human experience learning, we conduct the ablation study. We remove the PRs/Issues actions to re-run the main experiments. Experimental results are shown in Table 5 and we observe that without PRs/Issues, the pass rate decreases to 47.3%. It not only verifies the non-standardization problem of GitHub repositories but also proves that learning from PRs/Issues can overcome this challenge.

Furthermore, to validate the effectiveness of the summarized experience, We select 2 repositories: `Qlib` and `AiZynthFinder`, which both belong to the Hard Apply category. We run their 51 queries and utilize GPT-4-based GitAgent to store the repositories with summarized practice experience. We then re-run these queries but GitAgent would retrieve the stored repositories and utilize the summarized experience to accomplish the queries. Table 6 gives the results. We find that leveraging the experience summarized by GPT-4-based GitAgent, GPT-3.5-based can achieve a higher pass rate than GPT-4-based GitAgent without summarized experience. It further proves the effectiveness and necessity of human experience learning. Interestingly, we find that GPT-4-based GitAgent can achieve a higher Pass Rate even though it leverages the summarized experience by itself. It illustrates that GitAgent can learn experience from previous practical experience to boost its performance. Such an advantage is beneficial to real-world applications as it means that GitAgent will "evolve" along with its practice.

3.6 Error Analysis

Although our method can autonomously extend tools from GitHub, we still observe some failures.

Repository Select Failure A key challenge for GitAgent is selecting the appropriate repository from GitHub to address user queries. We noticed instances where GitAgent selected repositories that were not capable of resolving the given queries. This issue was particularly prevalent when the user query did not specify a particular repository, requiring GitAgent to autonomously determine the most suitable one. The agent's decision-making process relies heavily on the README files of repositories. However, these files sometimes lack clear and explicit descriptions of the repository's functionality, leading to misjudgments by the agent. For example, in the `Qlib` scenario, GitAgent erroneously selected the `vnpy` repository, suited for quantitative trading but not for research applications like setting specific models (e.g., LSTM). Although the README of `vnpy` contains the quantitative analysis description, it lacks explicit information on quantitative analysis model configuration, resulting in a mismatch between the user query and the chosen repository.

Environment Configuration Failure In cases like `Bringing-Old-Photos-Back-to-Life`, we observed failures in the environment setup. While an official dockerfile was present, it was outdated and non-functional without enough maintenance. The correct dockerfile was located within a Pull Request, which the agent should ideally access to find the accurate setup instructions. However, GitAgent sometimes opted to modify the existing, incorrect dockerfile rather than seeking the correct version in PRs. Due to unresolved bugs in the dockerfile, the agent was unable to correctly set up the environment, leading to failure.

Execution Configuration Failure The repository `Qlib` presented unique challenges, as it re-

quires writing a specific configuration file for execution. This file encompasses a range of parameters including dataset settings, model hyperparameters, and backtesting parameters. Incorrect settings in any of these parameters can lead to results that do not meet the user query's requirements. In practice, we observed that GitAgent may incorrectly set the time range for data or specified erroneous file paths, resulting in execution failure.

These observed failures primarily stem from the quality and complexity of the repositories. These insights underscore the need for further improvements in GitAgent's robustness and decision-making algorithms to enhance its success rate in diverse and complex scenarios.

We also conduct Case Study in Appendix C to further show the detailed process of GitAgent.

4 Related Work

LLM-based Agents Recent advancements in artificial intelligence have been significantly marked by the development of Large Language Models (LLMs) such as ChatGPT (OpenAI, 2022), GPT-4 (OpenAI, 2023), LLaMA (Touvron et al., 2023a,b), which have demonstrated remarkable proficiency across a diverse range of tasks. Benefiting from LLMs, LLM-based agents (AutoGPT; Wu et al., 2023b; Li et al., 2023; XAgent, 2023) have attracted research attention, which aims to extend the capabilities of LLMs to interact with external tools for accomplishing real-world tasks. Recently, the agent research can be divided into two lines. (1) **Single-Agent** researches focused on reasoning (Yao et al., 2022; Wei et al., 2023; Shinn et al., 2023), planning (Yao et al., 2023; Hao et al., 2023a; Besta et al., 2023; Sel et al., 2023; Ye et al., 2023), tool learning (Nakano et al., 2021; Huang et al., 2022; Ahn et al., 2022; Patil et al., 2023; Schick et al., 2023; Qin et al., 2023b,c; Qian et al., 2023b; Liu et al., 2023). (2) **Multi-Agent** work focused on collaboration(Chen et al., 2023; Wu et al., 2023b; Zhou et al., 2023), communication (Park et al., 2023; Li et al., 2023; Qian et al., 2023a; Wang et al., 2023b), evaluation (Chan et al., 2023; Zhang et al., 2023). Among these research topics, tool learning is a critical capability that empowers agents to accomplish beyond traditional NLP tasks.

Tool Learning Tool learning, as a critical capability in LLM-based agents, has garnered increasing attention, with studies demonstrating their growing adeptness in using tools and making decisions for complex tasks (Qin et al., 2023b; Vemprala et al., 2023; Nakano et al., 2021; Qin et al., 2023a; Shen et al., 2023; Wu et al., 2023a; Schick et al., 2023; Hao et al., 2023b; Qian et al., 2023b; Song et al., 2023; Qin et al., 2023c). Integrating external tools enhances the ability of agents to access up-to-date information (Yang et al., 2023; Nakano et al., 2021; Qin et al., 2023a), equips them with multimodal capabilities (Gupta and Kembhavi, 2023; Shen et al., 2023), and specializes their skills for specific domains (Jin et al., 2023). Despite these advancements, a notable limitation in existing research on tool learning for LLM-based agents is that the agent only supports a limited set of tools, which cannot meet the diverse demands of humans. Recently, some researchers are devoted to studying tool creation (Cai et al., 2023; Qian et al., 2023b; Wang et al., 2023a; Qian et al., 2023a) for agents to dynamically create tools while accomplishing tasks. However, the functionalities of the created tools are still simple and limited, and they cannot meet real-world complex user queries. Addressing this gap, we introduce a novel approach GitAgent that can achieve autonomous tool extension from GitHub repositories. As GitHub has a large number of repositories covering diverse domains (e.g., finance, chemistry, biology, etc) with versatile functionalities and GitHub contains valuable human experience (e.g., Issues and Pull Requests) additionally, we believe that extending tools from the host platform like GitHub is a promising way for agents.

5 Conclusion

In this paper, we present GitAgent, an autonomous agent that can extend tools from GitHub to address various user queries. Through a meticulously designed four-phase procedure and the implementation of learning human experience from GitHub Issues/PRs, GitAgent demonstrates its effectiveness in navigating the challenges posed by non-standardized repositories. Experimental results validate GitAgent's ability to achieve high pass rates in addressing user queries. However, while GitAgent showcases promising outcomes, challenges such as varying repository quality and unexpected issues during integration persist. This work prompts future research avenues, including refining GitAgent to adapt to repository variations, enhancing error handling mechanisms, and exploring methodologies for broader repository utilization.

6 Limitation

Our study has explored the tool extension capability of LLM-based agents, yet both researchers and practitioners must be mindful of certain limitations and risks when using the approach to develop new techniques or applications.

Firstly, our method relies on the utilization of Pull Requests (PRs) and Issues from GitHub as primary sources of human experience. However, it is important to recognize that similar functionalities may not be universally available across other repository hosting platforms. Consequently, the generalizability of our findings is confined to the GitHub ecosystem, posing a limitation to the applicability of our approach beyond this specific context.

Secondly, the dynamic nature of GitHub repositories, characterized by frequent updates, bug fixes, and the evolution of repository functionalities, introduces a layer of volatility. This fluidity can significantly impact the reproducibility of our experimental results over time, as the state of the repositories at the time of study may not reflect their future states.

Thirdly, our method necessitates the use of Docker for the execution of repository content. This dependency on Docker implicates substantial consumption of server resources, including CPU, memory, and storage. Such resource-intensive requirements may pose practical limitations on the scalability and feasibility of deploying GitAgent, particularly in environments with constrained computational resources.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, et al. 2022. Do as i can, not as i say: Grounding language in robotic affordances. *ArXiv preprint*, abs/2204.01691.

AutoGPT. Autogpt.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. 2023. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*.

Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2023. Chateval: Towards better llm-based evaluators through multi-agent debate. *arXiv preprint arXiv:2308.07201*.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14953–14962.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023a. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023b. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *arXiv preprint arXiv:2305.11554*.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 9118–9147. PMLR.

Qiao Jin, Yifan Yang, Qingyu Chen, and Zhiyong Lu. 2023. Genegpt: Augmenting large language models with domain tools for improved access to biomedical information. *ArXiv*.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large scale language model society.

Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Zengxian Yang, Kaikai An, et al. 2023. Mlbench: Large language models leverage open-source libraries for machine learning tasks. *arXiv preprint arXiv:2311.09835*.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *ArXiv preprint*, abs/2112.09332.

OpenAI. Chatgpt plugins.

OpenAI. 2022. OpenAI: Introducing ChatGPT.

OpenAI. 2023. Gpt-4 technical report.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*.

Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023a. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Cheng Qian, Chi Han, Yi R Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023b. Creator: Disentangling abstract and concrete reasonings of large language models through tool creation. *arXiv preprint arXiv:2305.14318*.

Yujia Qin, Zihan Cai, Dian Jin, Lan Yan, Shihao Liang, Kunlun Zhu, Yankai Lin, Xu Han, Ning Ding, Huadong Wang, et al. 2023a. Webcpm: Interactive web search for chinese long-form question answering. *arXiv preprint arXiv:2305.06849*.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, et al. 2023b. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023c. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *ArXiv preprint*, abs/2302.04761.

Bilgehan Sel, Ahmad Al-Tawaha, Vanshaj Khattar, Lu Wang, Ruoxi Jia, and Ming Jin. 2023. Algorithm of thoughts: Enhancing exploration of ideas in large language models. *arXiv preprint arXiv:2308.10379*.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

Yifan Song, Weimin Xiong, Dawei Zhu, Cheng Li, Ke Wang, Ye Tian, and Sujian Li. 2023. Restgpt: Connecting large language models with real-world applications via restful apis. *arXiv preprint arXiv:2306.06624*.

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

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. 2023. Chatgpt for robotics: Design principles and model abilities. Technical Report MSR-TR-2023-8, Microsoft.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023a. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Lei Wang, Jingsen Zhang, Xu Chen, Yankai Lin, Ruihua Song, Wayne Xin Zhao, and Ji-Rong Wen. 2023b. Recagent: A novel simulation paradigm for recommender systems. *arXiv preprint arXiv:2306.02552*.

Zekun Moore Wang, Zhongyuan Peng, Haoran Que, Jiaheng Liu, Wangchunshu Zhou, Yuhan Wu, Hongcheng Guo, Ruitong Gan, Zehao Ni, Man Zhang, Zhaoxiang Zhang, Wanli Ouyang, Ke Xu, Wenhu Chen, Jie Fu, and Junran Peng. 2023c. Rolellm: Benchmarking, eliciting, and enhancing role-playing abilities of large language models. *CoRR*, abs/2310.00746.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. 2023a. Visual chatgpt: Talking, drawing and editing with visual foundation models. *ArXiv preprint*, abs/2303.04671.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023b. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

XAgent. 2023. Xagent: An autonomous agent for complex task solving.

Linyao Yang, Hongyang Chen, Zhao Li, Xiao Ding, and Xindong Wu. 2023. Chatgpt is not enough: Enhancing large language models with knowledge graphs for fact-aware language modeling. *arXiv preprint arXiv:2306.11489*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *ArXiv preprint*, abs/2210.03629.

Yining Ye, Xin Cong, Yujia Qin, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2023. Large language model as autonomous decision maker. *arXiv preprint arXiv:2308.12519*.

Xinghua Zhang, Bowen Yu, Haiyang Yu, Yangyu Lv, Tingwen Liu, Fei Huang, Hongbo Xu, and Yongbin Li. 2023. Wider and deeper llm networks are fairer llm evaluators. *arXiv preprint arXiv:2308.01862*.

Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2023. D-bot: Database diagnosis system using large language models. *arXiv preprint arXiv:2312.01454*.

A Prompts

A.1 Main Agent

```
You are a professional programmer. Given
 a query, your task is to search for a
github repository and use it to solve
the query.

You should make sure the result of `
apply` function well completed the query
. If it is lack of required elements,
you can call `apply` again if you think
the result is close to what you want and
 you think this repository can be used
to solve your query. You can also call `
search_by_query` function to find
another repository if you think this
repository is not suitable for your
query.
```

A.2 Search Agent

```
You are a professional programmer. Given
 a task, you want to find a github
repository to solve the task.
```

```
You are a professional programmer. Given
 a task, you want to find a github
repository to solve the task. Now, your
colleagues have explored some
repositories. If you think any of the
repository(s) might can solve your task,
 call `use_existing_repository` function
 to use it. Otherwise, call `
find_a_new_repository` function to find
another repository.

You will be given the query of the task
and name(s) and description(s) of
existed repositories.

Repository's name: {{name of repository
1}}
Description: {{description  of
repository 1}}

Repository's name: {{name of repository
2}}
Description: {{description  of
repository 2}}

......
```

A.3 Setup Agent

```
You are a professional programmer. Your
task is to set up the environment of the
 repository and prepare necessary data.

You will be provided with the readme
file of the repository. You can also use
 `check_file_or_directory` function to
check the `/<==repo_name==>` directory
whether there is a existed Dockerfile.
If setting up the environment is complex
 and there is an existing dockerfile,
you can use `
set_container_with_existed_dockerfile`
function to directly use that dockerfile
. If there is any problem with the
dockerfile, you can try to use `
read_pulls_to_solve_problem` function to
 see the pulls of this repository to
solve the problem. However, `
read_pulls_to_solve_problem` should not
be used for reasons other than
troubleshooting issues with the
Dockerfile. If the existed dockerfile is
 built successfully, you can call `
submit` function directly with property
"work_directory" marked because the
required docker container has already
been built.

Usually the dockerfile is close to `/<==
repo_name==>`, so if you don't find it
in one or two try, it means there isn't
a dockerfile in this repository. You don
't need to try more times.

If there is no existing dockerfile, you
should analyze the readme file and
derive the necessary commands and
execute them to set up the environment
of the repository and prepare necessary
data in a given container, whose base
image is 'continuumio/miniconda3'. If
error happens due to inappropriate base
image, you can use `echo` to create a
dockerfile yourself, with proper base
image and necessary packages, and build
```

```
it.

While operating, please note the
following points:
- The commands will be run in a docker
container. You don't need to use virtual
 environments, use the base environment
only. Use pip or conda to install
packages. In special cases, you can use
apt-get to install necessary packages.
If you use apt-get, do not forget to use
 apt-get update and --fix-missing.
- Any command requiring execution in a
specific directory should be
reformulated as: `/bin/sh -c "cd <
specific directory> && <commands to be
executed in this directory>"`. Every
command must start with '/bin/sh -c " cd
 ' to locate a specific directory.
- The repository have been clone to the
root directory at `/<==repo_name==>`.
- Follow the sequence of the commands,
install all necessary packages first.
- Never create or activate any conda
environment even if the readme requires
or does so. You should install the
packages in the base environment.
- If you have problem with the version
of python, please reinstall python of
the appropriate version with `conda
install python=<version>`.
- If a function you called return you
with a file path, you should pass the
file path to the next function you call
if need.
- If there are different choices to do
the same task and you failed using one
of them, you can try another alternative
.

Your commands should be the parameter of
 the `execute_command` function. Each
time you should send one or many
commands. The `execute_command` function
 will run the commands and return the
output of the commands.

In this step, you should just set up the
 environment and prepare the data. You
don't need to run other programs or
train the model.
```

A.4 Apply Agent

```
You are a professional programmer. Your
task is to utilize a github repository
to solve a given query. You will operate
 in a docker container.

Note that it has been ensured that the
repository's environment has been set up
 and all the data required by the readme
 has been fully prepared, so you mustn't
 execute any command to set up the
environment or prepare the data or check
 relevant files about the environment or
 data anymore, unless the user provide
you with a link to download necessary
data. <==data_path==>

Also, all the dependencies have been
installed in the base environment,
please don't switch to any other conda
environment. If you find you lack of any
 packages or tools while operating, use
pip, conda or apt-get to install it. If
you use apt-get, do not forget to use `
apt-get update` and `--fix-missing`.

Your goal is to study the readme file
especially the command lines in it and
call appropriate functions to utilize
the repository to solve the query. Do
not execute any command to get result
that you can't perceive yourself, like
starting a server.

Note that the default configuration of
the final executable file may not meet
the demand of the query. If there is any
 special demands in the query, you
should check the final executable file
to check whether it meets the demand of
the query. If not, you should make
proper modification(s).

If you run a command and find the result
 lack of required element(s), which may
because the repository itself doesn't
support relevant function, you can check
 the issues to try to solve the problem.

If you need to deal with files provided
by the user, you should firstly use `
upload_directory_to_container` to upload
 it from local to the docker container.
By default, the path claimed in the
query is local path, you need to upload
it. If required message can be retrieved
 from the output of execution of the
program, summarize it to natural
language and submit it. If any file is
generated to answer the query, you
should use `
download_directory_from_container` to
download the file from the docker
container to local before you submit if
necessary. You should also ensure
required directories all exist before
running a program.

We only have CPU. If the repository
doesn't ask for configuration of device,
 ignore it.

Readme:{{readme}}
```

A.5 Modify Agent

```
You are a professional programmer. Your
task is to make modification(s) to code
files to meet the given requirement. You
 will be given the query of modification
, the content of a file and the path to
the file. If you think you can meet the
query through modifying this file, you
can modify this file.
```

```
If the query contains path that contains
 information for modification, transmit
that path at "query_file_path" in "
modify_entire_file". You don't need to
check the query file yourself, because
you may neglect important message by
checking and summarizing, just pass the
query path and let "modify_entire_file"
function to decide.

Code relevant to the query may not
always reside in the currently provided
file. In such cases, you should analyze
the `from...import...` or `<module name
>...` sections to suggest potential
target file paths.

If the target path in the current file
is relative path, you should decide the
target file based on the current
 f i l e s  path.

If it starts from a module's name, which
 suggests the file is a python package,
the file is in `/opt/conda/lib/python3
.11/site-packages/<package name>`
directory (python version should be
decided by using `which pip`). Don't
forget the suffix of the file.

You might need to locate the target file
 by checking the content of the files
recursively. After the target file is
located, you should use proper functions
 to modify the code.
```

A.6 Judge Agent

```
You are a professional programmer. Your
task is to judge how good a programmer
use a github repository to handle a
query. You will be given query and the
actions the programmer took to handle
the query. If the task includes input or
 output file, you will be given path to
programmer's output. Path to input is in
 the query and path to the ground truth
outcome will be given if there is ground
 truth. You can check the content in
these paths and use proper ways to judge
 the relevance of different files. If
the files are readable you can directly
check them. If not, you can use the
provided functions to check the md5 hash
 value of the files or compare the
similarities of different images. Note
that you can only check directory or
file saved in local. If no input path,
output path, truth path is given, do not
 check file or directory, just score
based on the log.

// For ReAct & ReAct + Summary
The rule of scoring is as follows. The
initial score is 0. You will be given
the log of user calling functions to use
 the repository. For correctly setting
up the environment and preparing the
data, 2 point should be added for
environment and 1 point should be added
for data. If no data is required, point
for data should be added.\nIn the given
application phase, 0~4 scores should be
added based on the performance. You
should judge the performance based on
whether it follows the instruction in
the readme. If right actions(including
commands and function calling) are taken
 and get a result, you should add 4. If
asked configuration is not applied or
wrong actions are taken, minus 1 point
for each fault based on 4. If ground
truth is provided, if the result of the
application is not correct, minus 1
point.\nIn conclusion, the final score
is the sum of the scores of the setup
(0~3) and application phase (0~4).

// For GitAgent
The rule of scoring is as follows. The
initial score is 0. You will be given
the log of user calling functions to use
 the repository, without the steps the
environment is setup.\nIn the given
application phase, 0~4 scores should be
added based on the performance. You
should judge the performance based on
whether it follows the instruction in
the readme. If right actions(including
commands and function calling) are taken
 and get a result, you should add 4. If
asked configuration is not applied or
wrong actions are taken, minus 1 point
for each fault based on 4. If ground
truth is provided, if the result of the
application is not correct, minus 1
point.\nGenerally, if valid output is
given, the score should be 4.

Query:{{query}}

Action:{{action_log}}

Input path:{{input_path}}

Output path:{{output_path}}

Ground Truth path:{{truth_path}}
```

B Repository Urls

- https://github.com/danielgatis/rembg
- https://github.com/ocrmypdf/OCRmyPDF
- https://github.com/cdfmlr/pyflowchart
- https://github.com/HarisIqbal88/PlotNeuralNet
- https://github.com/lukas-blecher/LaTeX-OCR
- https://github.com/s0md3v/Photon
- https://github.com/s0md3v/Bolt

- https://github.com/s0md3v/Smap
- https://github.com/MultiQC/MultiQC
- https://github.com/xinyu1205/recognize-anything
- https://github.com/bukosabino/ta
- https://github.com/molshape/ChemFormula
- https://github.com/tencent-quantum-lab/TenCirChem
- https://github.com/harirakul/chemlib
- https://github.com/ultralytics/yolov5
- https://github.com/mermaid-js/mermaid-cli
- https://github.com/microsoft/qlib
- https://github.com/fritzsedlazeck/Sniffles
- https://github.com/MolecularAI/aizynthfinder
- https://github.com/microsoft/Bringing-Old-Photos-Back-to-Life

C Case Study

To detail how GitAgent works during the whole tool extension process, we conduct the case study to demonstrate the behavior of GitAgent.

Adaptive Repository Search Strategies. GitAgent demonstrates a remarkable ability to autonomously select and implement varied search strategies for repository retrieval (see in Figure 4). This adaptability is evident from its high search success rate across different repositories. GitAgent tailors its search approach based on the specificity of the user query. For instance, in the case of `Sniffles`, where the repository name is provided (Figure 5), GitAgent directly searches for the repository using the given name. In contrast, for queries of Qlib, where no specific repository is mentioned (Figure 6), the agent summarizes relevant GitHub repository topics from the query and sequentially searches these topics to identify the most suitable repository.

Dynamic Handling of Setup Challenges. The agent is proficient in managing setup processes, even in the presence of bugs or incomplete information in the official repository documentation. For repositories like `AiZynthFinder`, with comprehensive setup instructions in the README (Figure 7), GitAgent efficiently follows the guidelines to set up the environment. Conversely, for repositories such as `Bringing-Old-Photos-Back-to-Life`, although it provides an official dockerfile to build the execution environment, there exist some bugs in it. In this situation, GitAgent proactively seeks solutions by reviewing pull requests (PRs) and modifies the dockerfile accordingly to rectify the bugs (Figure 8).

Efficient Execution of Complex User Queries. GitAgent capably handles the execution phase, addressing various challenges that arise with complex user queries. An example includes a scenario where the agent needed to install additional tools (e.g., "curl") mid-process to fulfill a data download request (Figure 9). In another instance, for a query requiring transaction details in Qlib (Figure 10), GitAgent resolved a missing directory issue by consulting GitHub Issues and adapting the configuration file to generate the required output. For example, a query asks GitAgent to download specific data for analysis (see in Figure 9). During the apply process, the agent found that the "curl" had not been installed and then it used "apt-get" to install "curl" before going on. For another example, a user query asks GitAgent to give the transaction details (see in Figure 10). GitAgent first ran the repo but when it tried to get the transaction details in the "record" directory, it found that there exists no "record" directory in the repository. Next, GitAgent resorted to the Issue from GitHub and found an issue that can solve the record output problem. Then, it modified the configuration file according to the issue content and re-run the command.

All the above phenomena demonstrate the robustness and flexibility of our method which can handle various non-standardized GitHub repositories to extend them as tools to accomplish user queries. The agent effectively navigates and utilizes non-standardized GitHub repositories, extending their functionalities to meet diverse user queries. This success can be attributed to our designed human experience learning, which enables GitAgent to focus on resolving subtasks by referring to human practice experience.

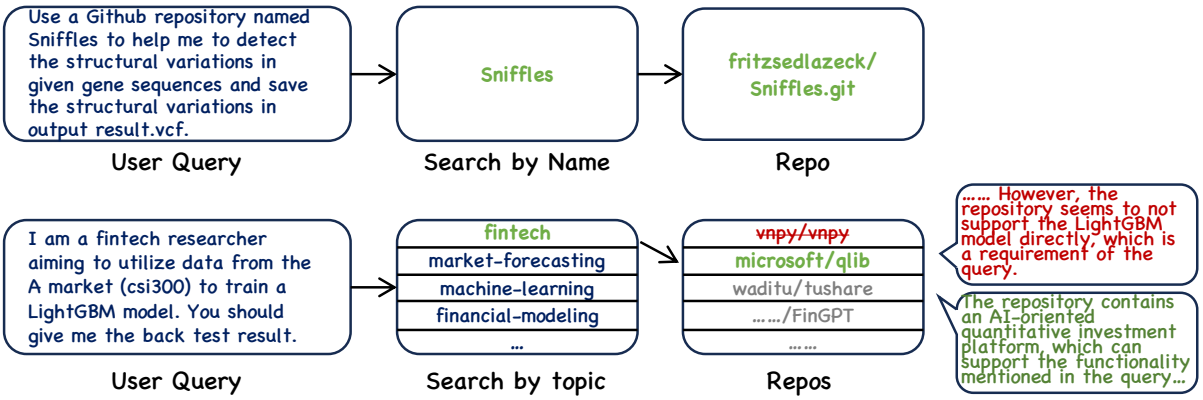

Figure 4: Comparison of two different repository search methods.

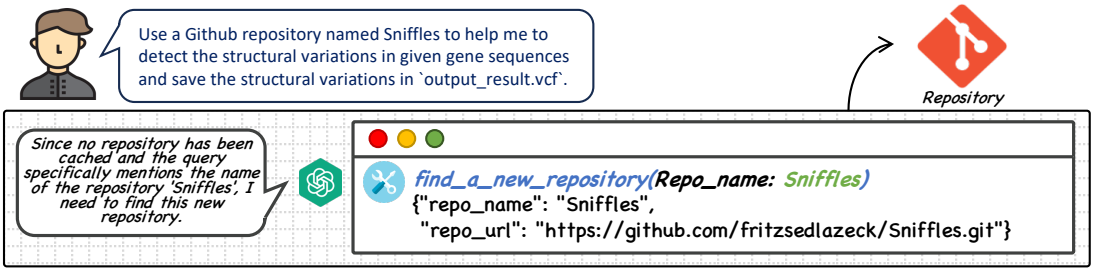

Figure 5: An example about the repository search of `Sniffles`.

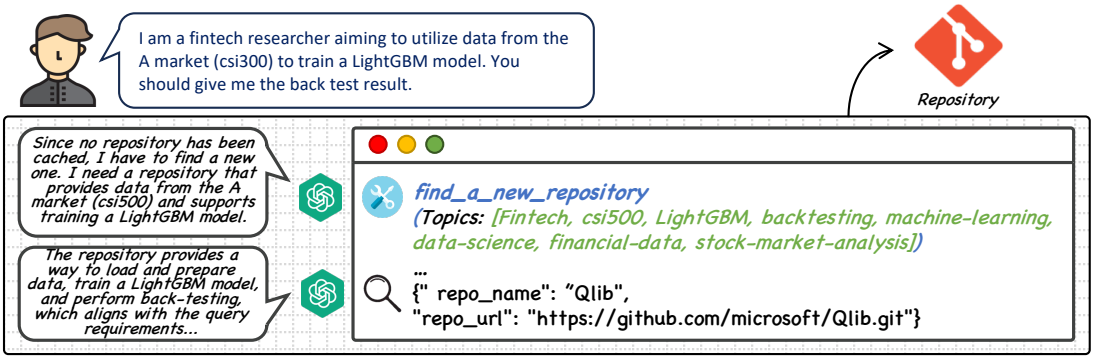

Figure 6: An example about the repository search of `Qlib`.

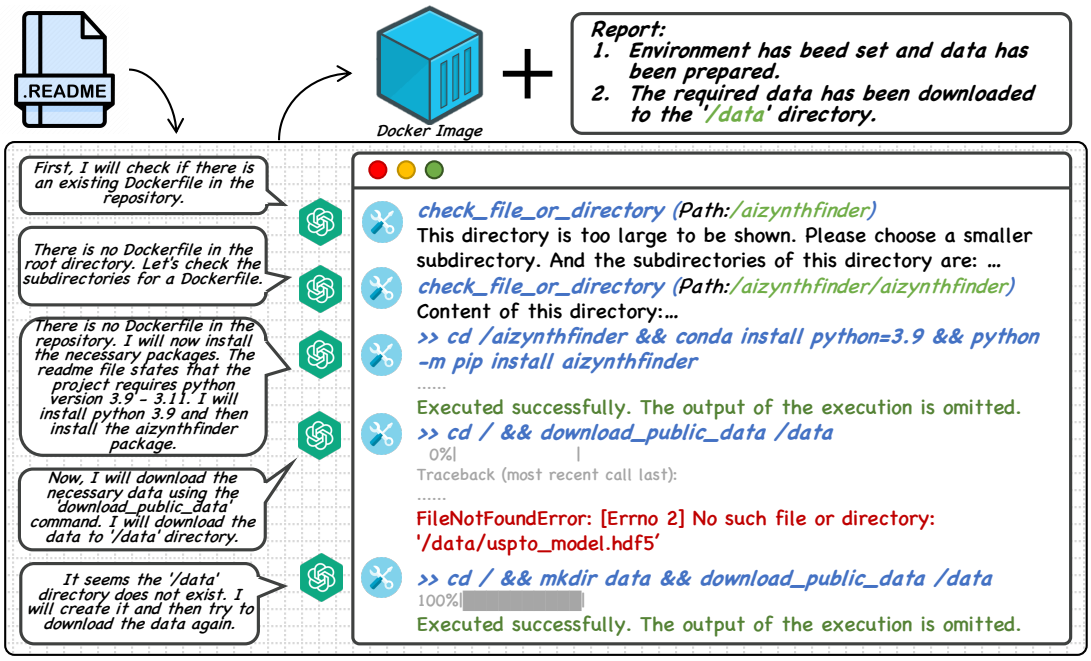

Figure 7: An example about the repository setup of `AiZynthFinder`.

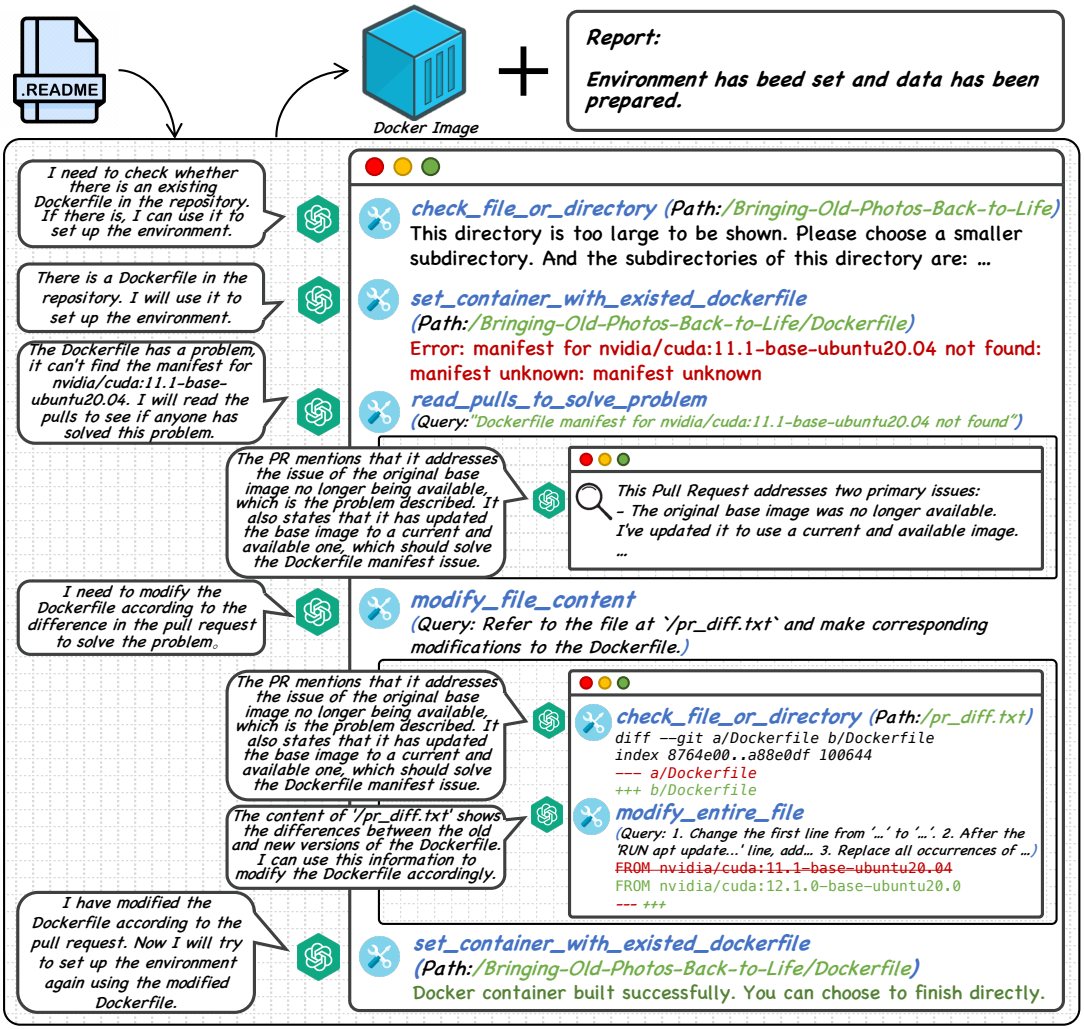

Figure 8: An example about the repository setup of `Bringing-Old-Photos-Back-to-Life`.

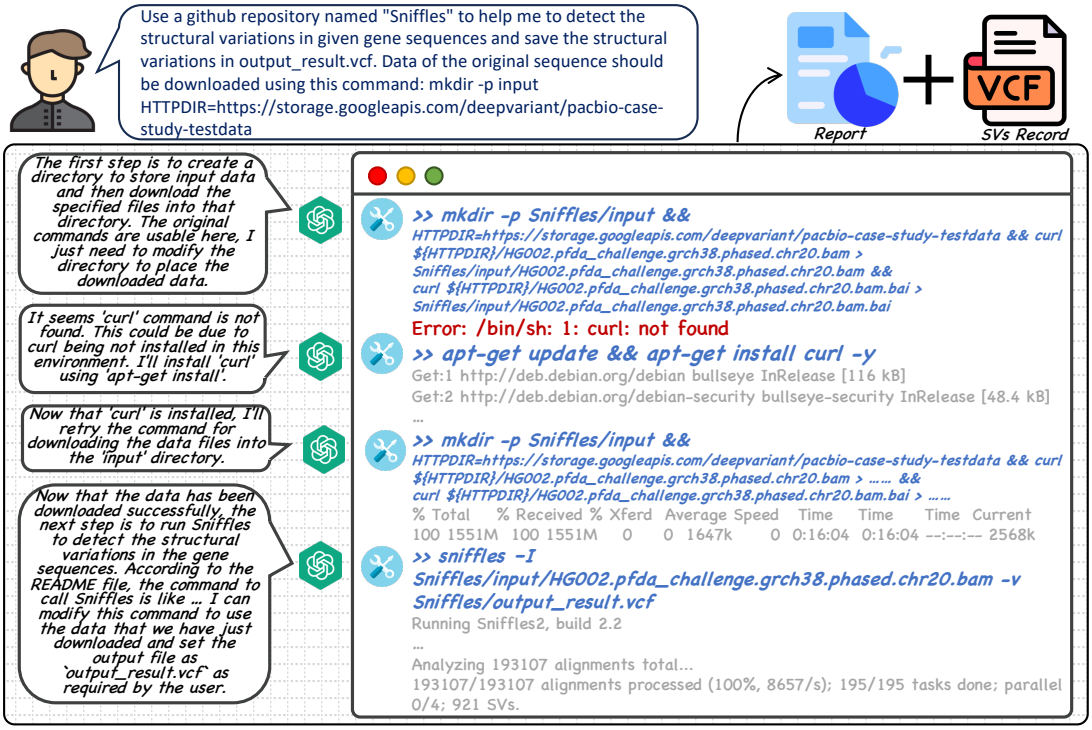

Figure 9: An example about the repository apply of `Sniffles`.

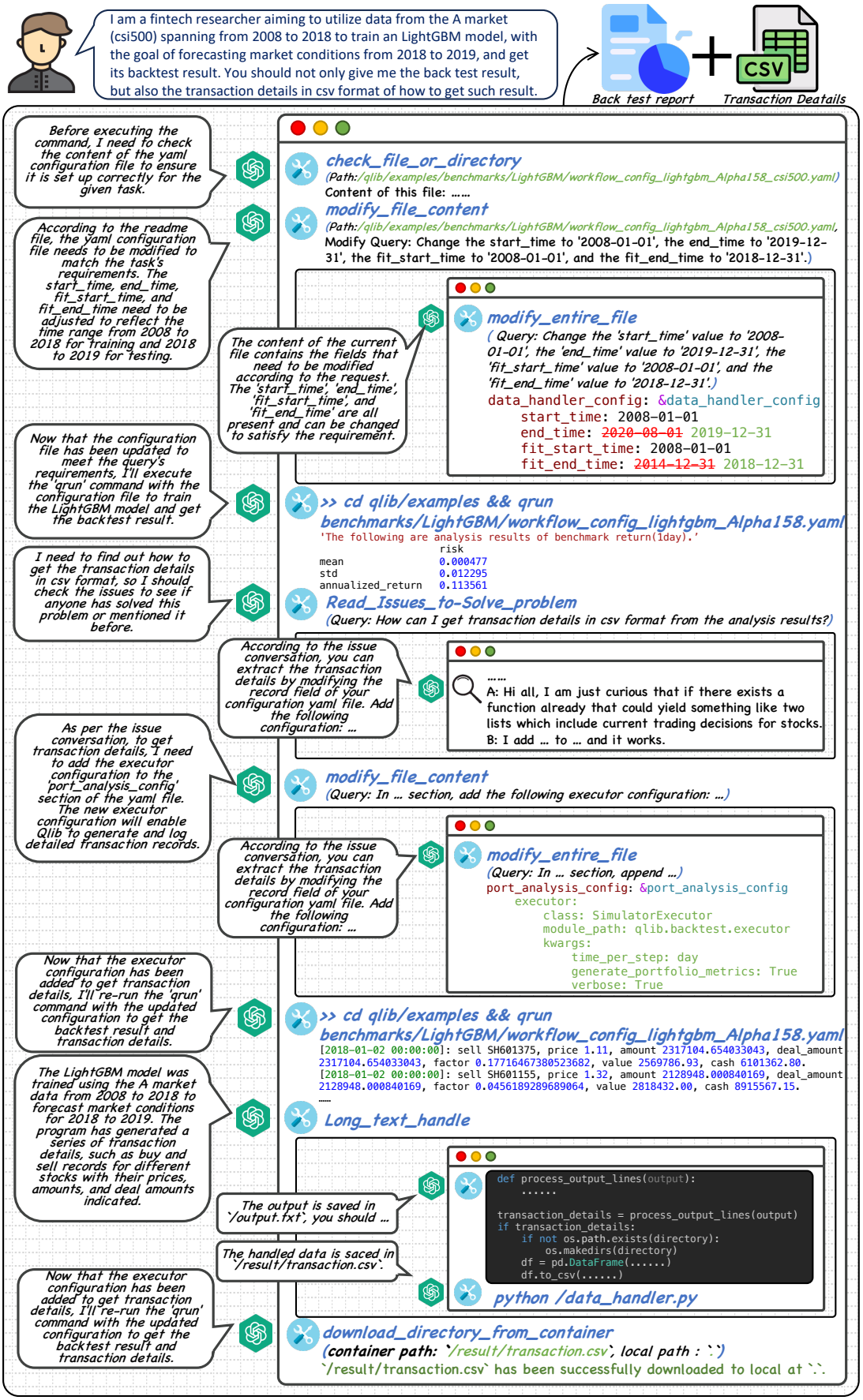

Figure 10: An example about the repository apply of `Qlib`.