CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have shown promise in automated code generation but typically excel only in simpler tasks such as generating standalone code units. However, real-world software development often involves complex code repositories with complex dependencies and extensive documentation. To enable LLMs to handle these real-world repo-level code generation, we present CodeAgent, a novel LLM-based agent framework that employs external tools for effective repo-level code generation. CodeAgent integrates five programming tools, enabling interaction with software artifacts for information retrieval, code implementation, and code testing. We implement four agent strategies to optimize these tools' usage. To the best of our knowledge, CodeAgent is the first agent framework specifically for repo-level code generation. In order to measure the effectiveness of our method at the repository level, we design a repo-level benchmark CodeAgentBench. The performance on this benchmark shows a significant improvement brought by our method, with improvements in pass rate ranging from 2.0 to 15.8. Further tests on the HumanEval benchmark confirm CodeAgent's adaptability and efficacy across various code generation tasks. Notably, CodeAgent outperforms commercial products like GitHub Copilot, showcasing superior accuracy and efficiency. These results demonstrate CodeAgent's robust capabilities in code generation, highlighting its potential for real-world repo-level coding challenges.

1 Introduction

Code generation automatically generates programs for the natural language (NL) requirement. Recent years have seen a trend in tackling code generation tasks with large language models (LLMs), such as Code Llama (Rozière et al., 2023), StarCoder (Li et al., 2023), and DeepSeekCoder (DeepSeek, 2023). Many efforts have been performed (Zhang et al., 2023b; Luo et al., 2023; Zheng et al., 2023) and shown impressive code generation abilities.

Despite achieving satisfactory performances, these studies mainly focus on simple generation scenarios including statement-level and function-level code generation. Statement-level code generation (Iyer et al., 2018; Athiwaratkun et al., 2022) aims to output statement-specific source codes. Function-level code generation (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021) predicts independent code that only invokes built-in functions and APIs from third-party libraries. For both scenarios, the length of the generated code is rather short, and they only generate standalone code units. However, more than 70% functions in the open-source projects are non-standalone (Yu et al., 2023). Developers typically write programs based on specific code environments, generally referring to code repositories. These repo-level code snippets usually have intricate contextual dependencies, which is too complex for existing LLMs to handle and generate (Li et al., 2024).

To enhance the efficacy of LLMs in repo-level code generation tasks, we draw inspiration from human programming practices. Developers typically employ a variety of tools to aid in complex programming. For instance, they might utilize search engines to explore key concepts or static analysis tools to identify pre-existing functions or classes. These tools are instrumental in the development of code projects. Embracing this idea, we propose a novel LLM-based agent framework CodeAgent that leverages external tools to help LLMs in repo-level code generation. With five programming tools, CodeAgent is capable of interacting with the software artifacts, including retrieving useful information, finding existing code symbols in the repository, and handling essential code testing. To guide LLMs to efficiently use tools, we draw on four agent strategies covering Re-

Act, Tool-Planning, OpenAIFunc, and Rule-based form. Based on agent strategies, LLMs can automatically select suitable tools for each repo-level task, finally providing a comprehensive response. ***To our knowledge, we are the first to adopt and optimize an agent-based method for the complex repo-level code generation task.***

In order to measure the effectiveness of our method at the code repository, we manually construct CODEAGENTBENCH, a benchmark specifically for repo-level code generation with a total of 101 functions and classes sourced from real code projects. It provides rich information about the repository, such as documentation and contextual dependency, to help LLMs better understand it. We further conduct extensive experiments for evaluation. We apply CODEAGENT to nine powerful open-source and closed-source LLMs with parameter sizes ranging from 13B to 175B to show the universality. Compared to directly generating from LLMs, experimental results on CODEAGENTBENCH reveal that CODEAGENT achieves significant improvements ranging from 2.0 to an extraordinary 15.8 across various LLMs. Further evaluations on well-known function-level benchmark HumanEval (Chen et al., 2021) confirm CODEAGENT's versatility in diverse code generation tasks. Remarkably, when compared to commercial products like GitHub Copilot (Dakhel et al., 2023), CODEAGENT stands out, demonstrating superior accuracy. These findings highlight the robust practical capabilities of CODEAGENT in the code generation community, underscoring its potential to evolve real-world repo-level coding challenges. We summarize our main contributions:

- We make an attempt to investigate repo-level code generation, which has crucial worth for understanding LLMs' performance in practical code generation scenarios.

- We propose CODEAGENT, an LLM-based agent framework for repo-level code generation. It develops five external programming tools to help LLMs complete the whole generation process and draw on four agent strategies to automatically optimize tools' usage.

- We construct CODEAGENTBENCH, a repo-level code generation benchmark, which has high-quality code repositories and covers diverse topics.

- Experimental results on nine LLMs show CODEAGENT's versatility and effectiveness in diverse code generation tasks, highlighting its potential for resolving real-world repo-level coding challenges.

2 Background

2.1 LLMs and Agents for Code Generation

LLMs have shown impressive capabilities in code generation since they have billions of parameters trained on a large amount of corpus with different training objectives. Recently, OpenAI [1] proposes GPT-3.5 and GPT-4 series models (e.g., ChatGPT (Chat, 2022)), which have shown strong generation abilities in coding. There are also various open-soured work, such as CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023), Code Llama (Rozière et al., 2023), WizardCoder (Luo et al., 2023) and DeepSeekCoder (DeepSeek, 2023).

Recent research has also increasingly shown that LLMs can be instrumental in developing AI agents (Palo et al., 2023; Wang et al., 2023a; Xi et al., 2023; Shen et al., 2023; Patil et al., 2023; Qin et al., 2023). Examples such as ToolFormer (Schick et al., 2023), Auto-GPT (AutoGPT, 2023), BabyAGI (BabyAGI, 2023), KwaiAgents (Pan et al., 2023) and ToolCoder (Zhang et al., 2023a) demonstrate LLMs' proficiency in tool utilization for complex tasks. **However, there is no relevant work targeting the complex coding capabilities of agent systems.** In this paper, we select GPT-4 (GPT-4, 2023), GPT-3.5 (GPT-3.5, 2023), and other powerful LLMs to design coding agent systems for real-world repo-level code generation.

2.2 Code Generation Tasks

Existing code generation tasks mainly focus on generating **standalone code units**, including statement-level (Yin et al., 2018) and function-level generation (Hendrycks et al., 2021; Chen et al., 2021). The generated programs are usually short and are independent of other codes. However, in software development, programmers mainly work within a code environment. They extend their functionalities based on the foundational code framework. Inspired by this, some studies (Yu et al., 2023; Liao et al., 2023) introduce intricate programming tasks that are based on particular code environments such as projects and code repositories. Nevertheless, these studies only provide lim-

[1] https://openai.com/

ited constraint information to LLMs, containing the requirements, signature information, and restricted code dependencies, leading to a difference in programming information needs from humans. To get closer to realistic programming scenarios, we formalize the repo-level code generation task and propose CodeAgent to help LLMs handle this complex task. We construct a repo-level code generation benchmark CodeAgentBench to evaluate our method and provide an analysis of benchmarks commonly used for these generation tasks in Table 7. Compared with existing code generation tasks, repo-level code generation is more consistent in real-world programming scenarios, fostering the evolvement of the code generation community.

3 Repo-level Code Generation Task

To fill the gap between existing code generation tasks and practical coding scenarios, we formalize the repo-level code generation task. Since a code repository generally contains intricate invocation relationships, only with a deep understanding of the code repository can LLMs generate satisfying programs that not only adhere to requirements but also seamlessly integrate with the current repository. Given a code repository, the repo-level code generation task aims to generate code based on all the software artifacts included in the repository, encompassing the **documentation**, **code dependency**, **runtime environment**, which form the task input. Here we give a detailed description of its composition format. Figure 1 shows an illustration of the repo-level code generation task.

Documentation It describes the generation targets and is the main input component of repo-level code generation. The documentation provides additional supporting information beyond the NL requirements. It contains class-level (class name, signature, and member function) and function-level (functional description, and params description) information of targets. Typically, the correctness of generated programs is verified with the test suite. The generated programs must conform to the interface (e.g., the input parameters). Thus, the documentation also provides the type and interpretation of input parameters and output values. In addition, considering that requirements usually contain domain-specific terminologies, the documentation explains these terms as well, such as mathematical theorems. As shown in Figure 1, documentation of the project contains rich information, where different elements are highlighted with diverse colors.

Contextual Dependency A key distinction of our new task from other independent code generation tasks is its inclusion of contextual dependencies. This aspect is crucial, as classes or functions typically interact with other code segments within the repository, such as import statements or other user-defined classes and functions. These interactions may occur within the same file or across multiple files. For instance, to implement the *RandomForest* class in Figure 1, it is necessary to utilize the *bootstrap_sample* function from *rf.py* and the *DecisionTree* class from *dt.py*, demonstrating the intricate code contextual dependencies involved.

Runtime Environment Different from natural language, program language is executable. Whether programs return target results after execution is a crucial manner to verify the correctness of generated programs. Developers typically depend on the execution feedback to correct errors in programs. The runtime environment provides all configurations needed to run the code repository and offers convenient interaction to ensure an all-sided evaluation of LLMs' performance on repo-level code generation.

4 CodeAgent Method

We introduce a novel LLM-based agent framework CodeAgent that leverages external tools to enhance the problem-solving abilities of LLMs in intricate repo-level code generation. CodeAgent seamlessly pauses generation whenever tools are called and resumes generation by integrating their outputs. These tools can assist LLMs with the entire code generation process, including information retrieval, code implementation, and code testing as shown in Table 1, thus interacting with the software artifacts (Section 4.1). Providing LLMs with access to tools, CodeAgent explores four agent strategies to optimize these tools' usage (Section 4.2). Figure 2 illustrates the overview of our CodeAgent.

4.1 Designed Programming Tools

Given a requirement, developers usually first gather relevant knowledge, then find and modify existing programs to meet the requirement, and finally verify programs with the assistance of tools. To mimic this process, we develop several programming tools that are specifically designed for LLMs.

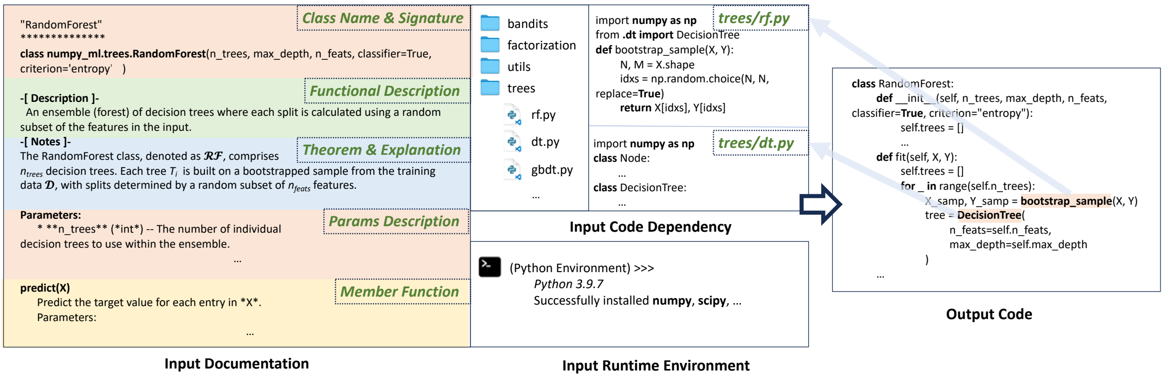

Figure 1: An illustrative example of the repo-level code generation. The task input contains complex descriptions, code dependencies, and runtime environment, which is more realistic than the existing benchmark.

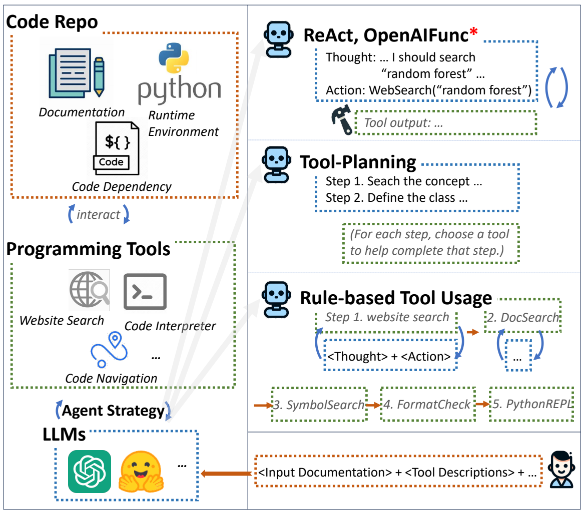

Figure 2: ***Left***: Overview of CODEAGENT. With our designed programming tools and agent strategies, LLMs interact with code repositories and generate repo-level code. ***Right***: Illustration of agent strategies in CODEAGENT. "*OpenAIFunc*" is similar to "*ReAct*" in the interaction mode, with some differences in the content generated by LLMs and the format of tool callings.

Tool Domain	Tool Name	Usage Pattern
Information Retrieval	Website Search	*WebSearch(input_query)*
	Documentation Reading	*DocSearch(input_name)*
Code Implementation	Code Symbol Navigation	*SymbolSearch(module_path or input_name)*
Code Testing	Format Checker	*FormatCheck()*
	Code Interpreter	*PythonREPL(input_code)*

Table 1: Programming tool statistics in CODEAGENT

CODEAGENT incorporates these external tools from three perspectives: information retrieval, code implementation, and code testing, which are commonly used by programmers in their daily work.

4.1.1 Information Retrieval Tools

Information retrieval tools are responsible for analyzing repositories and collecting resources, which is pivotal in understanding the problem domain. We develop popular website search and documentation reading as information retrieval tools.

Website Search Programmers often share solutions for various programming problems on websites where search engines consider them as knowledge resources. When encountering similar problems, developers only submit a question query to a search engine. The engine can provide useful programming suggestions. Inspired by this, CODEAGENT uses a popular search engine DuckDuckGo[2] to choose the most relevant websites, and then apply LLMs to summarize the website content as the final tool output [3]. In the process, we block websites that may lead to data leakage. The usage pattern of this tool is formatted as: *WebSearch(input_query)*, which will return the formatted content searched from websites.

Documentation Reading Besides gathering information from websites, we also retrieve relevant knowledge from the documentation of the repository. To achieve this, CODEAGENT leverages BM25 (Robertson et al., 2009) as the documentation reading tool. Given a class name or function name, it can retrieve correlative content from the documentation as its output. If the result is too long, the tool will use the LLM to summarize it and then provide it to LLMs for code generation. This tool is designed in the format: *DocSearch(input_name)*.

4.1.2 Code Implementation Tools

Code implementation tools aim to provide relevant code items (i.e., pre-defined symbol names and code snippets) in the code repository. LLMs modify and integrate these items into the generation

[2]https://duckduckgo.com/

[3]We choose *DuckDuckGo* because it provides a cheaper and more convenient API than other search engines such as *Google* and *Bing*.

process. It not only expedites the development process but also encourages code reuse. We build a code symbol navigation tool to help LLMs implement code snippets.

Code Symbol Navigation We use *tree-sitter* [4] to design the code symbol navigation tool. This tool explores code items from two types. The first type is oriented to the file or module-oriented parsing, where the tool performs static analysis of a file or module and provides symbol names defined in it, encompassing global variables, function names, and class names. The other type is the class or function symbol navigation. Given a class or function name, the tool finds its definition from the code repository. Combining the two types, this tool can traverse predefined source code within a repository, empowering LLMs to understand intricate dependencies and reuse codes. This tool is designed in the format: *SymbolSearch(module_path or input_name)*. The tool will detect what the input is and return the corresponding results (e.g., all defined symbols in the given file path or the implementation code corresponding to the given symbol name). When no parameters are provided, the default value is the path of the current file.

4.1.3 Code Testing Tools

After acquiring generated codes, we design code testing tools to format and test them, enhancing their correctness and readability.

Format Checker The tool is built to check the format correctness of generated codes. Specifically, we develop *Black* [5] as the format checker. It can check format errors such as indentation misalignment and missing keywords. Subsequently, it tries to rectify these errors and reorganizes code statements, enhancing the correctness and readability of generated codes. The usage pattern of this tool is: *FormatCheck()*, which will automatically format the most recently generated code and return the formatted version.

Code Interpreter The tool focuses on examining the syntax and function of programs. It furnishes a runtime environment so that LLMs can debug generated codes with execution feedback. The tool requires LLMs to provide a program to be executed, and then runs the code in the repository environment. Meanwhile, LLMs generate some test cases to verify whether the output of the generated code meets the expected results. When occurring errors, this tool will offer error information to facilitate LLMs to fix bugs until programs are error-free, which has been proven to be effective by many existing works (Chen et al., 2022; Zhang et al., 2023b) to correct output programs. The runtime environment is prepared for each task, as described in Section B.1.1. This tool is designed in the format: *PythonREPL(input_code)*, and the tool will return the executed result of the input code.

[4]https://tree-sitter.github.io/tree-sitter/

[5]https://github.com/psf/black

4.2 Agent Strategy

To guide LLMs to leverage these powerful tools properly, we develop four agent strategies for repo-level code generation, including ReAct, Tool-Planning, OpenAIFunc, and Rule-based Tool Usage. The interaction between LLMs and external tools is based on LangChain [6].

ReAct This strategy (Yao et al., 2022) prompts LLMs to generate reasoning traces and task-related actions in an interlaced fashion. Based on actions, ReAct selects the proper external tools and invokes them by providing input. The strategy then treats the output of tools as additional knowledge and decides whether to generate a final code or invoke other tools for further processing.

Tool-Planning We propose a variant, i.e., Tool-Planning, of Planning strategy (Wang et al., 2023b) that makes a plan before solving problems and has shown effectiveness in many studies (Zhang et al., 2022; Jiang et al., 2023). Different from Planning, our strategy can invoke proper tools based on the plan. Specifically, Tool-Planning first makes a plan to divide an entire task into several subtasks and then performs subtasks according to the plan. For complex subtasks, it will automatically choose an appropriate tool to assist LLMs in code generation.

OpenAIFunc Recently, some models (e.g., GPT-3.5 (GPT-3.5, 2023) and GPT-4 (GPT-4, 2023)) have the function-calling ability provided by OpenAI (OpenAIFunc, 2023). The interaction mode is similar to that of "ReAct", with some differences in the content generated by LLMs and the format of calling external tools.

Rule-based Tool Usage When faced with a complex problem, programmers often first learn related knowledge, then write programs, and check the

[6]https://python.langchain.com

Name	Domain	Samples	# Line	# DEP
numpyml-easy	Machine Learning	22	10.9	0.3
numpyml-hard	Machine Learning	35	85.4	2.6
container	Data Structure	4	130.3	8.0
micawber	Information Extraction	7	19.7	4.3
tinydb	Database	21	36.7	2.7
websockets	Networking	12	91.6	7.5
Total		101	57.0	3.1

Table 2: Statistics of CODEAGENTBENCH. # Line: average lines of code. # DEP: average number of code dependencies.

function of programs. Inspired by the workflow, we propose a rule-based strategy.

This strategy defines the order of tool usage and interlinks these tools by prompts. I) LLMs leverage website search to gather useful online information; II) LLMs then use documentation reading tool to search relevant classes and functions; III) Code symbol navigation is required to select and view the source codes of related classes and functions. Based on the above information, LLMs generate programs; IV) Subsequently, LLMs invoke the format checker to check the syntax and format of generated programs; V) Finally, LLMs use the code interpreter to evaluate the functional correctness of programs. Based on the feedback information, LLMs fix errors within programs. For each part, LLMs will autonomously cycle through the use of tools until it decides to move on to the next part or the cycle reaches its limit number (e.g., 3).

5 Experiment

We perform extensive experiments to answer three research questions: (1) How much can CODEAGENT improve the advanced code generation LLMs on repo-level code generation (Section 5.2); (2) What is the improvement of our CODEAGENT on classical code generation such as HumanEval (Section 5.3); (3) To what extent do our selected tools in the agent system help for repo-level coding (Section 5.4).

5.1 Experimental Setup

Benchmarks To evaluate our method on repo-level code generation, we follow the format described in Section 3 and construct a new benchmark **CODEAGENTBENCH**. We select five diverse topics and choose repositories with high stars from GitHub. The detailed construction process and statistics of our benchmark are shown in Table 2 and Section B.2. CODEAGENTBENCH contains 101 samples, and for each task, LLMs are provided with documentation containing the requirements needed to be implemented, along with a set of tools we designed, as well as full access permissions to code files in the repository. We use the self-contained test suite in each code repository to evaluate the correctness of generated programs.

In addition, to evaluate the generalization ability of CODEAGENT, we also perform experiments on function-level code generation. In this paper, we use a widely-used function-level benchmark **HumanEval** (Chen et al., 2021). It contains 164 programming problems with the function signature, docstring, body, and unit tests.

Base LLMs We apply CODEAGENT to nine most powerful LLMs, including GPT-3-davinci (GPT-3, 2022), GPT-3.5-turbo (GPT-3.5, 2023), GPT-4-turbo (GPT-4, 2023), Claude-2 (Claude, 2023), Llama2-70B-chat (Llama, 2023), Code Llama-34B (Rozière et al., 2023), WizardCoder-34B (Luo et al., 2023), DeepSeek-33B (DeepSeek, 2023) and Vicuna-13B (Chiang et al., 2023). Additional descriptions are provided as a part of Table 3.

Metrics Following previous works (Zan et al., 2022; Zheng et al., 2023), we use the pass rate as the metric, where we treat the generated program correctly only if its output is consistent with all ground truths of the test suite. Specifically, we are mainly concerned with **Pass@1** (Chen et al., 2021), which is a representative of the Pass@k family, because in real-world scenarios, we usually only consider the single generated code.

5.2 Repo-level Coding Performance

In our experiments, we utilized our specially designed repo-level benchmark, CODEAGENTBENCH, to assess the efficacy of CODEAGENT in enhancing the performance of nine prominent code LLMs. The results are presented in Table 3.

Our proposed CODEAGENTBENCH proves to be substantially more challenging than existing benchmarks, as evidenced by the relatively lower pass rates. On all base LLMs with various sizes, CODEAGENT consistently delivers significant performance improvements. Specifically, for GPT-4 model (GPT-4, 2023), we observe a maximum increase of 15.8, equating to a 72.7% relative enhancement over the baseline, i.e., NoAgent. The improvements of other LLMs range from 2.0 to an impressive 15.8, underscoring the effectiveness of our proposed approach. This demonstrates that the tools integrated within CODEAGENT provide

Models	Scales	NoAgent	Rule-based	ReAct	Tool-Planning	OpenAIFunc
Closed source LLM						
GPT-3-davinci (GPT-3, 2022)	175B	16.8	**24.8** ($\uparrow$ 7.9)	22.8 ($\uparrow$ 5.9)	18.8 ($\uparrow$ 2.1)	-
GPT-3.5-turbo (GPT-3.5, 2023)	-	19.8	**31.7** ($\uparrow$ 11.9)	30.7 ($\uparrow$ 10.8)	21.8 ($\uparrow$ 2.0)	28.7 ($\uparrow$ 8.9)
GPT-4-turbo (GPT-4, 2023)	-	21.8	**37.6** ($\uparrow$ 15.8)	34.7 ($\uparrow$ 12.9)	25.7 ($\uparrow$ 4.0)	34.7 ($\uparrow$ 12.9)
Claude-2 (Claude, 2023)	-	8.9	**10.9** ($\uparrow$ 2.0)	9.9 ($\uparrow$ 1.0)	9.9 ($\uparrow$ 1.0)	-
Open source LLM						
Llama2-70B-chat (Llama, 2023)	70B	10.9	**12.9** ($\uparrow$ 2.0)	11.9 ($\uparrow$ 1.1)	11.9 ($\uparrow$ 1.1)	-
Code Llama-34B (Rozière et al., 2023)	34B	2.0	**5.0** ($\uparrow$ 3.0)	4.0 ($\uparrow$ 2.0)	4.0 ($\uparrow$ 2.0)	-
WizardCoder-34B (Luo et al., 2023)	34B	2.0	**6.9** ($\uparrow$ 5.0)	5.0 ($\uparrow$ 2.7)	4.0 ($\uparrow$ 2.0)	-
DeepSeek-33B (DeepSeek, 2023)	33B	13.9	**24.8** ($\uparrow$ 10.9)	20.8 ($\uparrow$ 6.9)	15.8 ($\uparrow$ 2.0)	-
Vicuna-13B (Chiang et al., 2023)	13B	1.0	**1.0**	0.0	0.0	-

Table 3: The Pass@1 results of different agent strategies on CODEAGENTBENCH. "NoAgent" refers to the baseline where LLMs generate code solely based on the provided documentation.

useful information, aiding LLMs in producing accurate code solutions and effectively tackling complex repo-level coding challenges.

Across different LLMs, a notable trend is that more advanced LLMs exhibit greater improvements with the application of CODEAGENT. However, for Vicuna-13B model (Chiang et al., 2023), performance on CODEAGENTBENCH is notably poor, showing no appreciable enhancement with the agent strategy. In contrast, the improvement is quite pronounced for other high-capacity LLMs. Furthermore, we find that different agent strategies yield varying levels of enhancement. Among these strategies, Rule-based and ReAct strategies are more effective, whereas Tool-Plannig strategy appears less suited for the task.

5.3 Function-level Coding Performance

We further apply our CODEAGENT to function-level code generation with the well-known HumanEval benchmark (Chen et al., 2021). We adapt our approach to this scenario by omitting the documentation reading tool and code symbol navigation. The adjustment is necessitated as these tools are not applicable to the standalone code generation task. For this task, we strategically selected a range of representative LLMs for evaluation, constrained by our available resources and computational capacity. The pass rate results are detailed in Table 4.

The results once again highlight the efficacy of CODEAGENT in enhancing the performance of code LLMs across all metrics. Notably, the maximum improvements observed for each model span from 6.1 to 9.7 on Pass@1. These findings underscore the versatility and effectiveness of our CODEAGENT in augmenting the capabilities of LLMs across a variety of code generation tasks.

5.4 Ablation Study

To investigate the influence of tools incorporated in CODEAGENT, we conduct an ablation study focusing on tool utilization in repo-level code generation. We choose GPT-3.5-turbo with ReAct as the base model, named GPT-3.5-ReAct. We meticulously track the usage frequency of each tool during code generation processes, with the statistics presented in Table 5 under the column *# Usage*. Subsequently, we exclude one tool at a time from our approach, allowing us to isolate and understand the individual contribution of each tool. The performances of these ablation scenarios are shown in Table 5, categorized under the column *Ablation Result*.

Our findings reveal that the code symbol navigation tool is particularly pivotal in our agent system. On average, CODEAGENT utilizes this tool approximately 2.45 times per code generation, a frequency higher than the counterpart of other tools. Notably, the performance significantly declines when this tool is omitted, underscoring its critical role in enhancing the effectiveness of our approach. Furthermore, the ablation results confirm that each tool in our agent system contributes positively to the overall improvement. This evidence not only validates the effectiveness of our strategy design but also highlights the utility of programming tools in addressing the repo-level coding task.

6 Discussion

6.1 Compared with Commercial Products

Nowadays, a lot of mature commercial products are available to support complex code generation tasks. It is essential to compare CODEAGENT with these established products. We categorize them into two distinct groups: (1) *IDE Products* are AI-powered autocomplete-style suggestion tools integrated within IDE software. Notable examples

Models	NoAgent	Rule-based	ReAct	Plan	OpenAIFunc
GPT-3.5-turbo (GPT-3.5, 2023)	72.6	**82.3** (↑ 9.7)	79.3 (↑ 6.7)	73.8 (↑ 1.2)	81.1 (↑ 8.5)
CodeLLaMA-34B (Rozière et al., 2023)	51.8	**59.7** (↑ 7.9)	58.2 (↑ 6.4)	54.1 (↑ 2.3)	-
WizardCoder-34B (Luo et al., 2023)	73.2	**79.4** (↑ 6.2)	77.6 (↑ 4.4)	75.6 (↑ 2.4)	-
DeepSeek-33B (DeepSeek, 2023)	78.7	**84.8** (↑ 6.1)	83.5 (↑ 4.8)	81.1 (↑ 2.4)	-

Table 4: The Pass@1 results of different agent strategies on the HumanEval benchmark.

	# Usage	Ablation Result
GPT-3.5-ReAct	-	30.7
Websit Search	0.30	27.7 (↓ 3.0)
Documentation Reading	0.84	26.7 (↓ 4.0)
Code Symbol Navigation	2.45	22.8 (↓ 7.9)
Format Check	0.17	29.7 (↓ 1.0)
Code Interpreter	0.22	29.7 (↓ 1.0)
GPT-3.5-NoAgent	-	19.8

Table 5: Average tool usage number and ablation result on CODEAGENTBENCH for GPT-3.5-ReAct.

	NumpyML-easy	NumpyML-hard
Our Agent		
GPT-3.5	**14**	**3**
GPT-4	**17**	**5**
IDE Product		
GitHub Copilot	7	1
Amazon CodeWhisperer	5	0
Agent Product		
AutoGPT (with GPT-4)	2	0

Table 6: Performance compared with commercial programming products (the number of solved problems).

are *GitHub Copilot* (Copilot, 2023) and *Amazon CodeWhisperer* (CodeWhisperer, 2023). (2) *Agent Products* encompass autonomous agents driven by GPT-4 (GPT-4, 2023). They are capable of executing a variety of tasks, including coding, such as well-known *AutoGPT* (AutoGPT, 2023).

Considering that IDE products are primarily designed as completion systems, we limit human interactions to less than three times per task to ensure a fair comparison. The evaluation is conducted on the *numpyml* subset of CODEAGENTBENCH manually by an experienced Python developer. Table 6 shows the number of solved problems on different products and our CODEAGENT.

The results demonstrate that CODEAGENT works better than existing products on complex coding scenarios. In addition, despite both CODEAGENT and AutoGPT being agent-based approaches, CODEAGENT exhibits numerous optimizations tailored for repo-level coding tasks, thereby making it better than AutoGPT in the task. Compared to IDE products that can also analyze complex code dependencies, our method benefits from the flexibility inherent in the agent system, resulting in a substantial lead over IDE products.

6.2 Qualitative Analysis

We explore generated cases to assess CODEAGENT (e.g., GPT-3.5-ReAct) and the baseline model (e.g., GPT-3.5-NoAgent). The comparative analysis is shown in Figure 3 and Figure 4.

CODEAGENT typically begins with examining the code dependencies in the repository, subsequently refining its code generation strategy through a step-by-step process known as "chain-of-thought". As in Figure 3, the input documentation specifies the need for a class with member functions *set_params* and *summary*. CODEAGENT, assisting with the symbol navigation tool, finds the base class and identifies the member function *_kernel* as a key component for implementation. This is reflected in the generated thought process:

> *"The set_params and summary methods can be inherited from the base class without modifications ... The '_kernel' method needs to be overridden ..."*
>
> *(Generated by* CODEAGENT*-GPT-3.5-ReAct)*

On the contrary, GPT-3.5-NoAgent lacks access to detailed information on code structures, resulting in incorrect code solutions, as depicted in Figure 4.

7 Conclusion

We formalize the repo-level code generation task to evolve real-world coding challenges. To enhance LLMs to handle repo-level code generation, we propose CODEAGENT, a novel LLM-based agent framework. CODEAGENT develops five programming tools, enabling LLMs to interact with software artifacts, and designs four agent strategies to optimize tools' usage. To evaluate the effectiveness of our CODEAGENT, we construct CODEAGENTBENCH, a new benchmark for repo-level code generation that includes rich information about the code repository. Experiments on nine LLMs show that CODEAGENT achieves a significant improvement on diverse programming tasks, highlighting its potential in real-world coding challenges.

Limitation

Although our work is a very early exploration of this area, there are several limitations on our work that we aim to address as quickly as possible:

Firstly, we propose a new task format for the repo-level code generation task and release CODEAGENTBENCH. Our preliminary experiments prove that the impact of LLMs' memorization on pre-training data is slight for fair evaluation. However, it still needs further experiments to eliminate this hidden danger. We will follow the relevant research to further understand its influence on our proposed benchmark.

Secondly, we only incorporate simple tools to CODEAGENT. Some advanced programming tools are not explored. The limitation may restrict the agent's ability in some challenging scenarios.

Thirdly, in Section 6.1, the comparison with commercial products is not rigorous since experiments are done manually. We will study how to evaluate IDE products more standardly.

Finally, since LLMs are very sensitive to input prompts, it is very important to optimize prompts in the agent system. We will continue to explore better agent strategies based on the current approach.

References

Rie Kubota Ando and Tong Zhang. 2005. A framework for learning predictive structures from multiple tasks and unlabeled data. *Journal of Machine Learning Research*, 6:1817–1853.

Galen Andrew and Jianfeng Gao. 2007. Scalable training of L1-regularized log-linear models. In *Proceedings of the 24th International Conference on Machine Learning*, pages 33–40.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

AutoGPT. 2023. https://agpt.co.

BabyAGI. 2023. https://github.com/yoheinakajima/babyagi.

Chat. 2022. https://chat.openai.com/.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)*.

Claude. 2023. https://www.anthropic.com/index/claude-2.

CodeWhisperer. 2023. https://aws.amazon.com/codewhisperer/.

Copilot. 2023. https://github.com/features/copilot.

Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734.

DeepSeek. 2023. https://huggingface.co/deepseek-ai.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*.

GPT-3. 2022. https://platform.openai.com/docs/models/gpt-base.

GPT-3.5. 2023. https://platform.openai.com/docs/models/gpt-3-5.

GPT-4. 2023. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*.

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Zhi Jin, Hao Zhu, Huanyu Liu, Kaibo Liu, Lecheng Wang, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yihong Dong, Yuqi Zhu, Bin Gu, and Mengfei Yang. 2024. Deveval: Evaluating code generation in practical software projects. *CoRR*, abs/2401.06401.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Dianshu Liao, Shidong Pan, Qing Huang, Xiaoxue Ren, Zhenchang Xing, Huan Jin, and Qinying Li. 2023. Context-aware code generation framework for code repositories: Local, global, and third-party library awareness. *arXiv preprint arXiv:2312.05772*.

Llama. 2023. https://huggingface.co/meta-llama/Llama-2-70b-chat.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAIFunc. 2023. https://openai.com/blog/function-calling-and-other-api-updates.

Norman Di Palo, Arunkumar Byravan, Leonard Hasenclever, Markus Wulfmeier, Nicolas Heess, and Martin A. Riedmiller. 2023. Towards A unified agent with foundation models. *CoRR*, abs/2307.09668.

Haojie Pan, Zepeng Zhai, Hao Yuan, Yaojia Lv, Ruiji Fu, Ming Liu, Zhongyuan Wang, and Bing Qin. 2023. Kwaiagents: Generalized information-seeking agent system with large language models. *CoRR*, abs/2312.04889.

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

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *CoRR*, abs/2307.16789.

Mohammad Sadegh Rasooli and Joel R. Tetreault. 2015. Yara parser: A fast and accurate dependency parser. *Computing Research Repository*, arXiv:1503.06733. Version 2.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

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 arXiv:2302.04761*.

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. *CoRR*, abs/2303.17580.

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 Ji-Rong Wen. 2023a. A survey on large language model based autonomous agents. *CoRR*, abs/2308.11432.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023b. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*.

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 Huan, and Tao Gui. 2023. The rise and potential of large language model based agents: A survey. *CoRR*, abs/2309.07864.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *arXiv preprint arXiv:2306.14898*.

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 arXiv:2210.03629*.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, pages 476–486.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. *arXiv preprint arXiv:2302.00288*.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. Cert: Continual pre-training on sketches for library-oriented code generation. *arXiv preprint arXiv:2206.06888*.

Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023a. Toolcoder: Teach code generation models to use apis with search tools. *arXiv preprint arXiv:2305.04032*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023b. 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), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 769–787.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

A Details of Case Study

Here we show the illustration of the case study for CODEAGENT (GPT-3.5-ReAct) and GPT-3.5-NoAgent in Figures 3 and 4.

We can find a distinct operational pattern in CODEAGENT in Figure 3. Through meticulous analysis, CODEAGENT leverages code symbol navigation tool to scrutinize information within the 'utils.kernels' module, where the target class for implementation resides. Our custom-designed tool proficiently navigates to the module, offering insights into its contents, including package details, defined functions and classes, through a static analysis process. Importantly, CODEAGENT discovers a crucial class named 'KernelBase' and obtains detailed information about it with another use of the tool. Within 'KernelBase', there is an abstract method named '*_kernel*' that needs to be implemented. CODEAGENT recognizes this method as essential for the development process, highlighting its importance. Compared with the NoAgent in Figure 4, our approach accurately captures this content hidden in the complex information in the code repository, and precisely implements the final code.

We also notice that during the third tool invocation, CODEAGENT calls the code interpreter tool and execute a piece of code that appears insignificant. We have observed similar situations in other cases as well. We attribute this to LLMs still lacking proficient mastery of some complex programming tools. This insight directs our future research towards enhancing LLMs' ability to more effectively use complex programming tools.

B Details of CODEAGENTBENCH

In this section, we introduce the details of our CODEAGENTBENCH benchmark. We describe its composition format (Section B.1), the construction process (Section B.2), and provide a detailed comparison with existing benchmarks (Section B.3).

B.1 Benchmark Composition

Code repository contains intricate invocation relationships. Only with a deep understanding of code repository can LLMs generate satisfying programs that not only adhere to requirements but also seamlessly integrate with the current repository. Inspired by this, each task of our benchmark provides rich information, encompassing the documentation, code dependency, runtime environment, self-contained test suite, and canonical solution, which form the input and output.

B.1.1 Benchmark Input

Documentation Documentations are the main input component of our benchmark and describe the generation targets. We follow the code documentation format used in a popular documentation creation tool Sphinx [7]. Figure 1 illustrates an example of documentation in CODEAGENTBENCH, where different elements are highlighted with diverse colors. When accomplishing a new task, our prepared documentation can provide LLMs with all-sided details that need to be considered to ensure that the generation target has been well-defined and constrained.

Contextual Dependency Contextual dependency is an important role in our benchmark. To accurately identify these dependencies, we developed a static analysis tool using *tree-sitter* [8]. Our designed tool allows us to extract all user-defined elements (such as class names, function names, constants, and global variables) and public library names from each file. These elements are then stored in a knowledge base. For any given function, we use this knowledge base to locate its source file, parse the file to identify all user-defined symbols and public libraries, and finally determine its contextual dependencies by exact matching of symbol names and scopes. On average, each sample in CODEAGENTBENCH involves around 3.1 code dependencies, thereby closely simulating real-world programming conditions. Detailed information is shown in Table 2.

Runtime Environment Developers often use feedback from running programs to find and fix mistakes. In CODEAGENTBENCH, we build a sandbox environment for each task. The sandbox environment provides all configurations needed to run the repository and offers convenient interaction to ensure an all-sided evaluation of LLMs' performance on repo-level code generation.

B.1.2 Benchmark Ground-truth Output

Canonical Solution We use the answers included in the repository as the initial solutions and invite three participants to manually refine them. The first participant checks surface errors of solutions based on the repository information. The second

[7] https://www.sphinx-doc.org/

[8] https://tree-sitter.github.io/tree-sitter/

Input

"PolynomialKernel"
****

class numpy_ml.utils.kernels.PolynomialKernel(d=3, gamma=None, c0=1)
The degree-*d* polynomial kernel.

-[Notes]-
For input vectors \mathbf{x} and \mathbf{y}, the polynomial kernel is:
k(\mathbf{x}, \mathbf{y}) = (\gamma \mathbf{x}^\top \mathbf{y} + c_0)^d
In contrast to the linear kernel, the polynomial kernel also computes similarities *across* dimensions of the **x** and **y** vectors, allowing it to account for interactions between features.
As an instance of the dot product family of kernels, the polynomial kernel is invariant to a rotation of the coordinates about the origin, but *not* to translations.

Parameters:
*** **d** (*int*)** -- Degree of the polynomial kernel. Default is 3.
*** **gamma** (*float** or **None*)** -- A scaling parameter for the dot product between *x* and *y*, determining the amount of smoothing/resonlution of the kernel. Larger values result in greater smoothing. If None, defaults to 1 / *C*. Sometimes referred to as the kernel bandwidth. Default is None.
*** **c0** (*float*)** -- Parameter trading off the influence of higher-order versus lower-order terms in the polynomial. If *c0* = 0, the kernel is said to be homogenous. Default is 1.

set_params(summary_dict)
Set the model parameters and hyperparameters using the settings in *summary_dict*.
Parameters:
summary_dict (*dict*) -- A dictionary with keys 'parameters' and 'hyperparameters', structured as would be returned by the "summary()" method. If a particular (hyper)parameter is not included in this dict, the current value will be used.
Returns:
new_kernel (Kernel instance) -- A kernel with parameters and hyperparameters adjusted to those specified in *summary_dict*.

summary()
Return the dictionary of model parameters, hyperparameters, and ID

Tool-1 (Code Symbol Navigation)

SymbolSearch(numpy_ml.utils.kernels)
=>
import_package_or_function: **re**, **np**, **ABC**, **abc**, **abstractmethod**
mannual_defined_function: **pairwise_l2_distances**, **kernel_checks**
mannual_defined_class: **LinearKernel**, **KernelInitializer**, **KernelBase**, **RBFKernel**

Tool-2 (Code Symbol Navigation)

SymbolSearch(KernelBase)
=>
The code is:
```python
class KernelBase(ABC):
    def __init__(self):
        ...
    @abstractmethod
    def _kernel(self, X, Y):
        ...
    def __call__(self, X, Y=None):
        ...
    def __str__(self):
        ...
    def summary(self):
        ...
    def set_params(self, summary_dict):
        ...
```

Tool-3 (Code Interpreter)

PythonREPL(......)
=>
The output is empty.

The input code:
import numpy as np
from numpy_ml.utils.kernels import KernelBase

CodeAgent (GPT-3.5-ReAct) Output

Let's continue with the implementation of the PolynomialKernel class. The kernel function for this class involves a dot product of the input vectors, raised to a specified degree, and optionally scaled and shifted. This can be implemented using numpy's dot product function and basic arithmetic operations.

The set_params and summary methods can be inherited from the base class without modifications, as they should work the same for any kernel. The `_kernel` method needs to be overridden to implement the polynomial kernel function.

Here is the completed implementation:

```python
class PolynomialKernel(KernelBase):
    def __init__(self, d=3, gamma=None, c0=1):
        super().__init__()
        self.parameters = {"d": d, "c0": c0, "gamma": gamma if gamma is not None else 1.0}
        self.hyperparameters = {"id": "PolynomialKernel"}

    def _kernel(self, X, Y):
        gamma = self.parameters["gamma"]
        c0 = self.parameters["c0"]
        d = self.parameters["d"]
        result = (gamma * np.dot(X, Y.T) + c0) ** d
        return result
```

In this class, the `__init__` method sets up the degree, gamma, and c0 parameters, with defaults as specified. If gamma is not provided (None), it's default value is set to 1.

The `_kernel` method implements the polynomial kernel function. It first retrieves the parameters from the class' attribute `parameters`. It then computes the polynomial kernel function as `(gamma * dot_product(X, Y) + c0) ** d` and returns the result. Note that `np.dot(X, Y.T)` is used to compute the dot product of X and Y. The transpose of Y (Y.T) is used since in numpy, the dot function computes dot product along the last axis of the input arrays, so we need to make sure the vectors in X and Y align correctly.

Figure 3: Case study of our proposed CodeAgent (GPT-3.5-ReAct) on CodeAgentBench. We omit some input prompts and LLM's thoughts for viewing convenience.

person runs the solutions to identify and fix execution bugs. The last participant is responsible for executing solutions with the test suite, aiming to ensure its functional correctness. Through the iterative process, we can ensure the robustness and reliability of solutions as much as possible.

B.1.3 Benchmark Evaluation

Self-Contained Test Suite To evaluate the correctness of generated programs, CodeAgentBench furnishes a self-contained test suite for each task. We first analyze and extract test cases contained in the repository. We then invite two participants to manually add test cases to enhance

GPT-3.5-NoAgent Output

```
Here is the implementation of the `PolynomialKernel` class in the `numpy_ml.utils.kernels` module:

```python
class PolynomialKernel:
    def __init__(self, d=3, gamma=None, c0=1):
        self.d = d
        self.gamma = gamma
        self.c0 = c0

    def set_params(self, summary_dict):
        if 'parameters' in summary_dict:
            self.d = summary_dict['parameters'].get('d', self.d)
            self.gamma = summary_dict['parameters'].get('gamma', self.gamma)
            self.c0 = summary_dict['parameters'].get('c0', self.c0)
        if 'hyperparameters' in summary_dict:
            self.d = summary_dict['hyperparameters'].get('d', self.d)
            self.gamma = summary_dict['hyperparameters'].get('gamma', self.gamma)
            self.c0 = summary_dict['hyperparameters'].get('c0', self.c0)

    def summary(self):
        return {
            'parameters': {'d': self.d, 'gamma': self.gamma, 'c0': self.c0},
            'hyperparameters': {'d': self.d, 'gamma': self.gamma, 'c0': self.c0},
            'ID': id(self)
        }
```

Note that the `PolynomialKernel` class has three parameters in its constructor: `d`, `gamma`, and `c0`. The `set_params` method allows updating the parameters of the kernel using a dictionary, and the `summary` method returns a dictionary containing the current parameters, hyperparameters, and the ID of the kernel instance. ...
```

Figure 4: Case study of the baseline GPT-3.5-NoAgent on CODEAGENTBENCH. Compared with Figure 3, GPT-3.5-NoAgent generates the incorrect programs.

its coverage as much as possible. In CODEAGENTBENCH, each task has at least one unit test case. Whereafter, another participant manually checks the correctness of the test suite. Given a new task, we run the corresponding unit test code to verify the generated programs based on our sandbox environment. We treat the generated program correctly only if its output aligns with all ground truths of the test suite. For fairness, LLMs can not access the test suite during code generation.

B.2 Benchmark Construction Process

To make CODEAGENTBENCH diverse, we select five prevalent topics judged by ten developers and choose repositories with high stars from GitHub. The selected topics contain machine learning, data structure, information extraction, database, and networking. To ensure the quality, we only select repositories that use *pytest* [9] and *unittest*[10] as the test framework and its documentation is generated by *Sphinx*[11] tool. We also filter out complex repositories that are hard to deploy and test. Then, we extract all functions and classes in code repositories and arrange two participants to sequentially execute them. Our construction costs approximately 600 person-hours. Each participant possesses 2-5 years of Python programming experience. Finally, we get 101 functions and classes collected from real code projects in Python. The statistics of CODEAGENTBENCH are shown in Table 2.

[9]https://docs.pytest.org/
[10]https://docs.python.org/3/library/unittest.html
[11]https://www.sphinx-doc.org/

B.3 Compared with Existing Benchmarks

We perform a detailed analysis of existing code generation benchmarks in Table 7. Compared to the previous benchmarks, our CODEAGENTBENCH has two main advantages. On the one hand, it is closer to real-world code generation scenarios. On the other hand, CODEAGENTBENCH provides pretty complex information that is related to the code repository, including documentation, contextual dependency, runtime environments, and test suites. In Figure 5, we give an illustrative example of HumanEval, a function-level code generation benchmark. Compared with ours in Figure 1, it is obvious that our constructed CODEAGENTBENCH contains complex descriptions and code dependencies, which is more realistic than the existing benchmark. This information can efficiently prompt LLMs for repo-level code generation.

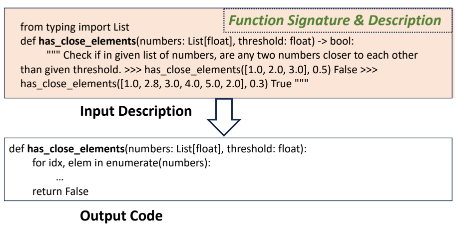

Figure 5: An illustrative example of existing benchmark HumanEval.

Benchmark	Language	Source	Task	Samples	# Tests	# Line	# Tokens	# Input
CoNaLA (Yin et al., 2018)	Python	Stack Overflow	Statement-level	500	✖	1	4.6	NL
Concode (Iyer et al., 2018)	Java	Github	Function-level	2000	✖	-	26.3	NL
APPS (Hendrycks et al., 2021)	Python	Contest Sites	Competitive	5000	✔	21.4	58	NL + IO
HumanEval (Chen et al., 2021)	Python	Manual	Function-level	164	✔	11.5	24.4	NL + SIG + IO
MBXP (Athiwaratkun et al., 2022)	Multilingual	Manual	Function-level	974	✔	6.8	24.2	NL
InterCode (Yang et al., 2023)	SQL, Bash	Manual	Function-level	200, 1034	✔	-	-	NL + ENV
CodeContests (Li et al., 2022)	Python, C++	Contest Sites	Competitive	165	✔	59.8	184.8	NL + IO
ClassEval (Du et al., 2023)	Python	Manual	Class-level	100	✔	45.7	123.7	NL + CLA
CoderEval (Yu et al., 2023)	Python, Java	Github	Project-level	230	✔	30.0	108.2	NL + SIG
RepoEval (Liao et al., 2023)	Python	Github	Repository-level	383	✖	-	-	NL + SIG
CODEAGENTBENCH	Python	Github	Repository-level	101	✔	57.0	477.6	Software Artifacts (NL + DOC + DEP + ENV)

Table 7: The statistics of existing widely-used code generation benchmarks. # Tests: whether a benchmark has the test suite. # Line: average lines of code. # Tokens: average number of tokens. # Input: Input information of LLMs. NL: Natural language requirement. IO: Input and output pairs. SIG: Function signature. CLA: Class skeleton as described in Section 2.2. ENV: Runtime environment. DOC: Code documentation. DEP: Code dependency.