ProgAI: Enhancing Code Generation with LLMs For Real World Challenges

Afsal Ahamad A
Department of Artificial Intelligence and Machine Learning
Sri Shakthi Institute of Engineering and Technology
Coimbatore, India

Harshad D
Department of Artificial Intelligence and Machine Learning
Sri Shakthi Institute of Engineering and Technology
Coimbatore, India

Mrs. S. Nivedha
Department of Artificial Intelligence and Machine Learning
Sri Shakthi Institute of Engineering and Technology
Coimbatore, India

Abstract
Large Language Models (LLMs) have shown promise in automated code generation but generate code units with errors because of reasons like hallucinations. Real-world soft- ware development, however, often involves complex requirements with complex dependencies and extensive documentation. To fill this gap, our research pivots towards evaluating LLMs in a more realistic setting real-world repo-level code generation. We introduce ProgAI, a manually curated LLM for proficient code generation. This LLM supports
Code generation 4 coding languages – namely C++, Java, Python and C. We assess nine leading LLMs on code generation tasks and observe a decline in their performance. To tackle this, we present ProgAI, a novel LLM-based agent framework that employs external tools for effective code generation. ProgAI integrates four programming tools, enabling interaction with software artifacts for information retrieval, code symbol navigation, and code testing. We implement four agent strategies to optimize these tools' usage. Our experiments on ProgAI show that ProgAI
enhances LLM performance significantly, with improvements ranging from 18.1% to 25%. Further tests on the HumanEval benchmark confirm ProgAI's adaptability and efficacy across various code generation tasks. Notably, ProgAI outperforms commercial products like

Github Copilot, showcasing superior accuracy and efficiency. These results demonstrate ProgAI'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 CodeLlama (Rozière et al., 2023), StarCoder (Li et al., 2023), and DeepSeekCoder (DeepSeek, 2023). To evaluate the effectiveness of LLMs on code generation, many efforts have been performed (Zhang et al., 2023b; Luo et al., 2023; Zheng et al., 2023). They demonstrate LLMs have impressive code generation abilities, *e.g.*, CodeLlama (Roz- ière et al., 2023) achieves 55% pass rate on the MBPP benchmark (Austin et al., 2021). Despite achieving satisfactory performances, they 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 gen- eration (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, on the one hand, the length of the generated code is rather short. For instance, the average line of each target code is 11.5 on the most popular benchmark Hu manEval (Chen et al., 2021). On the other hand, 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 that usually have intricate con- textual dependencies.Python code repositories with a total of 101 func- tions sourced from Github. It provides rich in- formation about the repository, such as documen- tation and contextual dependency, to help LLMs better understand it. Besides, each repository has a self-contained test suite to efficiently verify the correctness of generated codes. Given a requirement, LLMs generate desired programs that not only meet the requirement but also ensure seam- less integration with the current project repository. When tested with ProgAI, we observe that even the most powerful model, GPT-4 (GPT-4, 2023), only achieves 21.8% pass rate, indicating a decline in performance compared to standalone code generation. These models struggle with handling complex code tasks, often leading to inaccurate code outputs. Further details and examples are provided in Section 6.2 and Figure 5.

To alleviate this problem, we follow the research line of LLM-based agents (Palo et al., 2023; Wang et al., 2023a; Xi et al., 2023) and introduce a novel LLM-based agent framework ProgAI that leverages external tools to help LLMs in repo-level code generation.

We conduct extensive experiments for evalu- ation. We apply ProgAI to nine powerful open-source and closed-source LLMs with parameter sizes ranging from 7B to 175B to show the universality. Compared to directly generating from LLMs, experimental results on ProgAI reveal that PROGAI achieves performance improvements ranging from 18.1% to an extraordinary 25% across various LLMs. Further evaluations on well-known function-level bench- mark HumanEval (Chen et al., 2021) confirm ProgAI's versatility and effectiveness in di- verse code generation tasks. Remarkably, when

[2]https://github.com/

compared with commercial products like Github Copilot, ProgAI stands out, demonstrating superior accuracy. These findings highlight the robust practical capabilities of PROGAI in code generation community, underscoring its potential to evolve real-world repo-level coding challenges. In summary, we make the following main contri

-butions:

- We make an attempt to investigate production-level code generation, which has crucial worth for understanding LLMs' performance in practical scenarios.
- We construct PROGAI, a repo-level code generation LLM. It has high- quality code repositories and diverse topics.
- We propose ProgAI that leverages tools to tackle repo-level code generation.
- Experimental results on nine LLMs demonstrate our ProgAI's versatility and effectiveness in diverse code generation tasks, high- lighting its potential for real-world repository- level coding challenges.

1 Background

1.1 LLMs and Agents for Code Generation

Code generation can automatically generate source codes given an NL requirement, which attracts more and more attention in industry and academia. Nowadays, 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. Early, Lu et al. (2021) adapt GPT-2 (Radford et al., 2019) model on the source code, resulting in CodeGPT. Chen et al. (2021) fine-tune GPT-3 (Brown et al., 2020) models on code corpus to produce CodeX (Chen et al., 2021) and GitHub Copilot (Dakhel et al., 2023). Recently, OpenAI [3] proposes GPT-3.5 and GPT-4 series models (*e.g.*, ChatGPT (Chat, 2022)), which have shown strong generation capabilities in natu- ral language and programming languages. Since neither CodeX (Chen et al., 2021) nor GPT-3.5 (GPT-3.5, 2023) is open-sourced, some researchers

[3]https://openai.com/

attempt to replicate them, bringing in CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023), CodeLLaMA (Rozière et al., 2023), WizardCoder (Luo et al., 2023) and DeepSeekCoder (DeepSeek, 2023). Recent research has increasingly shown that LLMs can be instrumental in developing AI agents (Palo et al., 2023; Wang et al., 2023a; Xi et al., 2023). Examples such as ToolFormer (Schick et al., 2023), HuggingGPT (Shen et al., 2023), Gorilla (Patil et al., 2023), ToolLlama (Qin et al., 2023) and ToolCoder (Zhang et al., 2023a) demonstrate LLMs' proficiency in tool utilization. Projects like Auto-GPT (AutoGPT, 2023), BabyAGI (BabyAGI, 2023), and KwaiAgents (Pan et al., 2023) high- light the autonomous capabilities of LLM-powered agents in fulfilling human 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.

1.2 Benchmarks for Code Generation

The code generation benchmarks contain various coding tasks and commonly verify the correctness of generated code by Pass@k (Chen et al., 2021). To date, many code generation benchmarks (Yin et al., 2018; Athiwaratkun et al., 2022; Hendrycks et al., 2021) focus on generating independent pro- grams. Based on the granularity of examples, these benchmarks mainly be categorized into statement-level and function-level generation. Statement-level generation only predicts statement-specific source codes. CoNaLA (Yin et al., 2018) is a representative statement-level benchmark where the target of each example contains one statement. Function-level generation (Hendrycks et al., 2021; Chen et al., 2021) requires LLMs to predict stan- dalone function-oriented programs given an NL requirement. The generated programs only invoke built-in functions or APIs from third-party libraries. This is by far the most common code generation task. Besides, some benchmarks (Li et al., 2022) contain algorithm questions. LLMs aim to gener- ate an algorithm solution like a human programmer. These benchmarks play a significant role in the advancement of the code generation field.

However, in software development, program- mers mainly work within a specific code envi- ronment, commonly referring to a code reposi- tory. The code repository includes documenta- tion, predefined programs, historical issues, and submitted commits, serving as the foundational framework for developers to build and extend their functionalities. In this case, developers must understand and navigate the code repository environment thoroughly to ensure seamless integra- tion of newly added code with existing code. Re- cently, some project-level (Yu et al., 2023) and repository-level (Liao et al., 2023) benchmarks have been proposed to investigate the challenging generation scenario. Nevertheless, they only pro- vide limited constraint information to LLMs, such as the requirements, signature information, and restricted code dependencies. When it comes to LLM-based code generation techniques, solely re- lying on inputting simplified information to LLMs fails to align with the existing environment. In this paper, we construct a repository-level code generation benchmark ProgAI. Com- pared with existing repository-level benchmarks, ProgAI provides rich input information (*i.e.*, requirement, documentation, code dependency, and runtime environment), and self-contained test suites. It is closer to real-world programming scenarios, fostering the evolvement of code generation community.

2 ProgAI Benchmark

In this section, we introduce our ProgAI benchmark. We illustrate its composition format (Section 3.1) and construction process (Section 3.2) in detail.

2.1 Benchmark Composition

Code repository contains intricate invocation relationships. Only with a deep understanding of code repository can LLMs generate satisfying pro- grams 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 docu- mentation, code dependency, runtime environment, self-contained test suite, and canonical solution, which form the input and output.

2.1.1 Benchmark Input

Documentation Documentations are the main input component of our benchmark and describe the generation targets. The documentation pro- vides additional supporting information beyond the NL requirements. It contains target class-level (class name, signature, and member function) and

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
PROGAIBENCH	Python	Github	Repository-level	101	✓	57.0	477.6	Software Artifacts (NL + DOC + DEP + ENV)

Table 1: 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.

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 PROGAIBENCH. # Line: average lines of code. # DEP: average number of code dependencies.

function-level (functional description, and params description) information. 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 in- terpretation of input parameters and output val- ues. Considering that requirements usually contain domain-specific terminologies, the documentation also explains these terms such as the mathematical theorem. We follow the code documentation for- mat used in a popular documentation creation tool Sphinx [4]. Figure 1 illustrates an example of docu- mentation in PROGAI, where different elements are highlighted with diverse colors. When accomplishing a new task, our prepared documen- tation 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 A key distinction of our PROGAI from other benchmarks

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

is its inclusion of contextual dependencies. This aspect is crucial, as classes or functions typically interact with other code segments within the same 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.

To accurately identify these dependencies, we developed a static analysis tool using *tree-sitter* [5]. 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 sam- ple in PROGAI involves around 3.1 code dependencies, thereby closely simulating real- world programming conditions. Detailed informa- tion is shown in Table 2.

Runtime Environment Different from natu- ral language, program language is executable. Whether programs return target results after execu- tion is a crucial manner to verify the correctness of generated programs. Developers typically de-

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

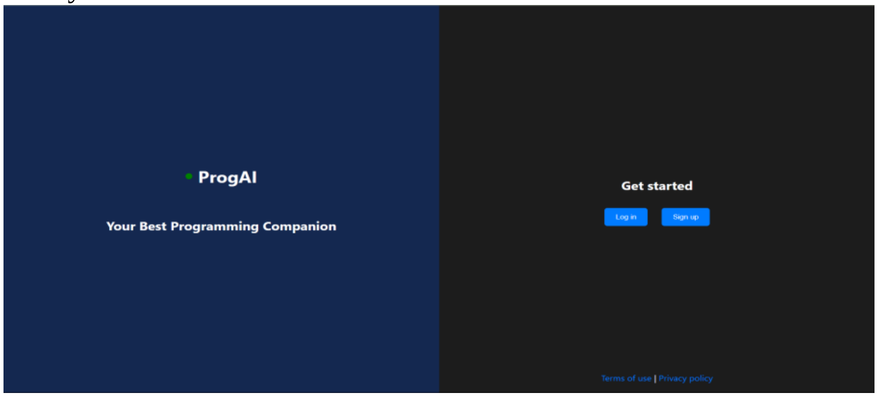
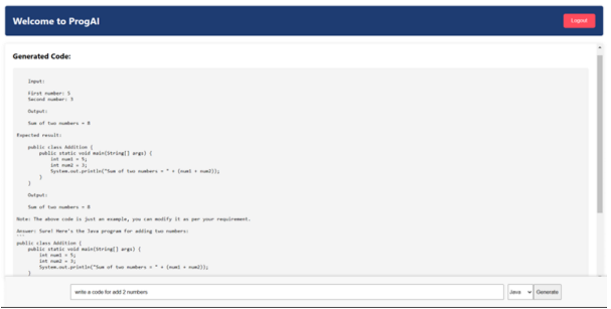

ProgAI
Your Best Programming Companion
Get started
Log in
Sign up
Terms of use | Privacy policy

Figure 1: ProgAI – Our Proposed Model



```
    Input:

    First number: 5
    Second number: 3

    Output:

    Sum of two numbers = 8

Expected result:

    public class Addition {
        public static void main(String[] args) {
            int num1 = 5;
            int num2 = 3;
            System.out.println("Sum of two numbers = " + (num1 + num2));
        }
    }

    Output:

    Sum of two numbers = 8

Note: The above code is just an example, you can modify it as per your requirement.

Answer: Sure! Here's the Java program for adding two numbers:
```
public class Addition {
    public static void main(String[] args) {
        int num1 = 5;
        int num2 = 3;
        System.out.println("Sum of two numbers = " + (num1 + num2));
    }
```

write a code for add 2 numbers

Figure 2: Input and Output

pend on the execution feedback to correct errors in programs. In PROGAI, 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' perfor- mance on repo-level code generation.

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

### 2.1.3 Benchmark Evaluation

**Self-Contained Test Suite** To evaluate the cor- rectness of generated programs, PROGAI 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
```

its coverage as much as possible. In PROGAI, 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 envi- ronment. 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.

2.2 Benchmark Construction Process

To make PROGAI 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 net- working. To ensure the quality, we only select repositories that use *pytest* [6] and *unittest*[7] as the test framework and its documentation is generated by *Sphinx*[8] tool. We also filter out complex repos- itories that are hard to deploy and test. Then, we extract all functions and classes in 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 with 5 repositories in Python. The statistics of PROGAI are shown in Table 2. For the potential data leakage issue, we conduct a preliminary experiment and discussion in Seciton 6.1.

Compared to the previous benchmarks, our PROGAI has two main advantages.

[6]https://docs.pytest.org/
[7]https://docs.python.org/3/library/unittest.html
[8]https://www.sphinx-doc.org/

Documentation Code Dependency
Runtime Environment

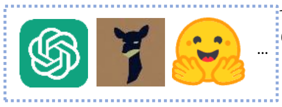
interact

Tools

Website Search
Code Navigation
Code Interpreter
...

interac
t

New Requireme nt Documenta tion



New Code

LLMs

Tool Domain	Tool Name	Format
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 3: Tool statistics in PROGAI

programs to meet the requirement, and finally ver- ify generated programs with the assistance of tools. To mimic this process, PROGAI incorporates external tools from three perspectives: information retrieval, code implementation, and code testing, which are commonly used by programmers in their daily work.

Figure 3: Overview of our PROGAI.

On the one hand, it is closer to real-world code generation scenarios. On the other hand, PROGAI provides self-contained information, in- cluding documentation, contextual dependency, runtime environments, and test suites. The infor- mation can efficiently prompt LLMs for repo-level code generation.

3 ProgAI Method

We perform a preliminary exploration to assess and improve LLMs' ability on repo-level code generation. Considering that developers often use various tools to assist with programming, we follow the research line of LLM-based agents (Palo et al., 2023; Wang et al., 2023a; Xi et al., 2023) and introduce a novel LLM-based agent framework ProgAI that leverages external tools to en- hance the problem-solving abilities of LLMs in intricate repo-level code generation. ProgAI seamlessly pauses generation whenever tools are called and resumes generation by integrating their outputs. The tools of ProgAI can assist LLMs with the entire code generation process, including information gathering, code implementation, and code testing (Section 4.1), thus interact with the software artifacts. Providing LLMs with access to tools, ProgAI explores four agent strategies to guide LLMs to interact with these tools properly for generation (Section 4.2). Figure 3 illustrates the overview of our ProgAI.

3.1 Tools

Given a requirement, developers usually first gather relevant knowledge, then find and modify existingInformation Retrieval Tools

Information retrieval tools are responsible for analyzing repositories and collecting resources, which is pivotal in understanding the problem domain. In this paper, we use popular website search and documentation reading tools to gather information.

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 use- ful programming suggestions. Inspired by this, ProgAI uses a popular search engine Duck- DuckGo[9] to choose the most relevant websites, and then apply LLMs to summarize the website content as the final tool output. [10] In the process, we block websites that may lead to data leak- age. This tool is designed in the format: *Web-Search(input_query)*, and the tool will return the formatted content searched from websites. If the result is too long, the tool will use the LLM to summarize it.

Documentation Reading Besides gathering information from websites, we also retrieve relevant knowledge from the documentation of the repos- itory. To achieve this, ProgAI 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 and then provide them to LLMs for code generation. This tool is designed

[9]https://duckduckgo.com/

[10]We choose *DuckDuckGo* because it provides a cheaper and more convenient API than other search engines such as *Google* and *Bing*. in the format: *DocSearch(input_name)*, and the tool will return the content searched from project documentations. Similar to the website search tool, the tool will summarize the results as necessary.

3.1.1 Code Implementation Tools

Code implementation tools aim to provide relevant code items (*i.e.*, pre-defined symbol names and code snippets) in the repository. LLMs modify and integrate these items into their generation process. It not only expedites the development process but also encourages code reuse. We leverage code sym- bol navigation as the code implementation tool.

Code Symbol Navigation Based on *tree-sitter*, we build a code symbol navigation tool for Python. This tool explores code items from two types. The first type is file or module-oriented parsing, where code symbol navigation performs static analysis of a file or module and provides symbol names de- fined 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 repos- itory, empowering LLMs to understand intricate dependencies and reuse codes. This tool is de- signed in the format: *SymbolSearch(module_path or input_name)*. The tool will detect what the in- put 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.

3.1.2 Code Testing Tools

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

Format Checker It is employed to check the format correctness of generated codes. Specifically, we use *Black* as the format checker, which is a Python code formatter. It executes generated codes and checks 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. This tool is designed in the format: *FormatCheck()*, and the tool will automatically formats the model's 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 improve the accuracy of output programs. The runtime environment is prepared for each task, as described in Section 3.1.1. This tool is designed in the format: *PythonREPL(input_code)*, and the tool will return the executed result of the input code.

Agent Strategy

To guide LLMs to leverage these powerful tools properly, we explore three existing agent strate- gies for repo-level code generation, including ReAct (Yao et al., 2022), Tool-Planning (Wang et al., 2023b), and OpenAIFunc (OpenAI-Function, 2023). These agent strategies are put forward to augment LLMs' reasoning and problem-solving abilities. In addition, inspired by the human pro- gramming process, we also propose a Rule-based strategy to access with tools. We perform our ex- periments based on LangChain [13].

ReAct The strategy is a general paradigm that synergizes reasoning and acting with LLMs, while also interacts with the external environments to in- corporate additional information for solving prob- lems (Yao et al., 2022). It allows LLMs to perform dynamic reasoning and thus can adjust and main- tain reasoning plans in time. In repo-level code generation, ReAct prompts LLMs to generate rea- soning traces and task-related actions in an interlaced fashion. Based on actions, ReAct selects the proper external tools and invokes them by provid- ing 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 ap- propriate tools to assist LLMs in code generation.

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

OpenAIFunc Recently, some models (*e.g.*, GPT-3.5 (GPT-3.5, 2023) and GPT-4 (GPT-4, 2023)) have the function-calling ability. Given a task description, they can detect when a function should be called, thus rendering them ideal for integration with external tools. In this strategy, we use the function-calling interface (OpenAI-Function, 2023) provided by OpenAI, which can accept human- created tools. For repo-level code generation, we provide the model with external tools described in Section 4.1. It then determines the time and type of invoking tools, and offers the input to the called tools. LLMs combine with the returned results of tools to generate programs.

Rule-based Tool Usage When faced with a complex problem, programmers often first learn external knowledge (*e.g.*, website search) and background knowledge within the repository, then write programs, and finally check the functionality of programs. Inspired by the workflow, we propose a rule-based strategy for generating codes.

The strategy defines the order of tool usage and interlinks these tools through prompts. ❶ LLMs leverage website search to gather useful online information; ❷ LLMs then use documentation reading tool to search relevant classes and functions; ❸ code symbol navigation is required to select and view the source code of related classes and functions, meanwhile learn the pre-defined symbols in the same file as the target code. Based on the above information, LLMs generate programs. ❹ Subsequentially, LLMs invoke the format checker to check the syntax and format of generated programs. ❺ Finally, LLMs generate some test cases and use the code interpreter to evaluate the functional correctness of programs. LLMs fix the errors until the feedback information is error-free. For each part, the large model will autonomously cycle through the use of tools until it decides to move on to the next part or the cycle reaches its limit (we set the limit to be less than 3).

4 Experiment

Based on the constructed benchmark and agent strategies, we conduct the first study to evaluate existing LLMs on repo-level coding challenges. We perform extensive experiments to answer four re- search questions: (1) How much can PROGAI improve the advanced code generation LLMs on PROGAI (*e.g.*, repo-level code genera- tion) (Section 5.2); (2) What is the improvement of our PROGAI 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); (4) How helpful is PROGAI for human programmers compared with the commercial code generation products (Section 5.5).

4.1 Experiment Setup

Dataset We evaluate our approach on our new benchmark **PROGAI** as described in Section 3. For each task in the benchmark, LLMs are provided with documentation containing the re- quirements needed to be implemented, along with a set of tools we designed, as well as full access permissions to code files in the repository. In ad- dition, to evaluate the generalization ability of our PROGAI, we also perform experiments on function-level code generation. In this paper, we use **HumanEval** (Chen et al., 2021) since it is a widely used benchmark. It contains 164 program- ming problems with the function signature, doc- string, body, and unit tests. We give an illustration of these benchmarks in Figure 1 and Figure 2.

Base LLMs We investigate the effectiveness of several popular code LLMs. Specifically, we ap- ply PROGAI 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), CodeLlama-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 4.

Metrics Following previous works (Zan et al., 2022; Zheng et al., 2023), we use the pass rate as the metric to evaluate the performance of LLMs, 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.

4.2 Repo-level Coding Task

In our experiments, we utilized HumanEval to assess the efficacy of PROGAI in enhancing the performance of nine prominent code LLMs. The specially designed benchmark, is repository-level code generation that encompasses intricate documentation and code dependencies, accurately mirroring real-world scenarios. The results of comparing ProgAI to these nine models are presented in Table 4.

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 18.1% to an impressive 25%, underscoring the effective- ness of our proposed approach. This demonstrates that the tools integrated within PROGAI provide crucial repo-level 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 ProgAI. How- ever, for *Vicuna-13B* model (Chiang et al., 2023), performance on ProgAI 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-Planning strategy appears less suited for the task.

4.3 Function-level Coding Task

We further apply our PROGAI to function- level code generation with the well-known Hu- manEval 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 5.

The results once again highlight the efficacy of ProgAI in enhancing the performance of code LLMs across all metrics. Notably, the maxi- mum improvements observed for each model span from 7.8% to 15.3%. These findings underscore the versatility and effectiveness of our proposed PROGAI in augmenting the capabilities of LLMs across a variety of code generation tasks.

4.4 Ablation Study

To investigate the influence of tools incorporated in PROGAI, 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 6 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 6, categorized under the column *Ablation Result*.

Our findings reveal that the code symbol navigation tool is particularly pivotal in our agent system. On average, PROGAI 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. Further- more, 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.

4.5 Compared with Commercial Products

Nowadays, a lot of mature commercial products are available to support complex code generation tasks. It is essential to compare PROGAI with these established products. We categorize them

Models	Scales	NoAgent	Rule-based	ReAct	Tool-Planning	OpenAIFunc
Closed source LLM						
GPT-3-davinci (GPT-3, 2022)	175B	16.8	**24.8** (↑ 7.9)	22.8 (↑ 5.9)	18.8 (↑ 2.1)	-
GPT-3.5-turbo (GPT-3.5, 2023)	-	19.8	**31.7** (↑[illegible]	30.7 (↑ 10.8)	21.8 (↑ 2.0)	28.7 (↑ 8.9)
GPT-4-turbo (GPT-4, 2023)	-	21.8	**37.6** (↑[illegible]	34.7 (↑ 12.9)	25.7 (↑ 4.0)	34.7 (↑ 12.9)
Claude-2 (Claude, 2023)	-	8.9	**10.9** (↑ 2.0)	9.9 (↑ 1.0)	9.9 (↑ 1.0)	-
Open source LLM						
LLaMA2-70B-chat (Llama, 2023)	70B	10.9	**12.9** (↑ 2.0)	11.9 (↑ 1.1)	11.9 (↑ 1.1)	-
CodeLLaMA-34B (Rozière et al., [illegible]	34B	2.0	**5.0** (↑ 3.0)	4.0 (↑ 2.0)	4.0 (↑ 2.0)	-
WizardCoder-34B (Luo et al., 2023)	34B	2.0	**6.9** (↑ 5.0)	5.0 (↑ 2.7)	4.0 (↑ 2.0)	-
DeepSeek-33B (DeepSeek, 2023)	33B	13.9	**24.8** (↑[illegible]	20.8 (↑ 6.9)	15.8 (↑ 2.0)	-
Vicuna-13B (Chiang et al., 2023)	13B	1.0	**1.0**	0.0	0.0	-

Table 4: The Pass@1 results of different agent strategies on PROGAIBENCH. "NoAgent" refers to the baseline where LLMs generate code solely based on the provided documentation. Other columns indicate our agent strategies as described in Section 4.2.

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

	# Usage	**Ablation Result**
GPT-3.5-ReAct	-	30.7

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).

Websit Search	0.30	27.7 ($\downarrow$ 3.0)
Documentation Reading	0.84	26.7 ($\downarrow$ 4.0)
Code Symbol Navigation	2.45	22.8 ($\downarrow$ 7.9)
Format Check	0.17	29.7 ($\downarrow$ 1.0)
Code Interpreter	0.22	29.7 ($\downarrow$ 1.0)
GPT-3.5-NoAgent	-	19.8

Table 6: Average Tool Usage Number and Ablation Result on PROGAIBENCH 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 7: Performance compared with commercial programming products (the number of solved problems).

into two distinct groups: (1) *IDE Products* are AI-powered autocomplete-style suggestion tools integrated within IDE software. Notable examples are *Github Copilot*[14] and *Amazon CodeWhisperer*[15]. (2) *Agent Products* encompass autonomous agents

[14]https://github.com/features/copilot
[15]https://aws.amazon.com/codewhisperer/

We employ the versions of these products with their default settings. Considering that IDE prod- ucts 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 ProgAI. All tasks are manually exe- cuted by an experienced Python developer. Table 7 shows the number of solved problems on different products and our ProgAI.

The results demonstrate that PROGAI works better than existing products on complex coding scenarios. In addition, despite both PROGAI and AutoGPT being agent-based approaches, PROGAI 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 approach benefits from the flexibility inherent in the agent system, which can use different tools whenever needed, resulting in a substantial lead over IDE products. These findings show the practical capabilities of PROGAI in the code generation community.

5 Discussion

5.1 Impact of LLMs' Memorization on Pre-Training Data

To examine the potential impact of LLMs memorization on pre-training data, we conduct a simplified obfuscation study by extending the methodology (Tang et al., 2023). Specifically, we modify the input documentation with obfuscated prompts. These prompts are altered by replacing function or parameter names with placeholder terms, which are concise GPT-generated summaries in their original form. We also reorganize the file structure of the code repository. The adjustment aims to challenge LLMs' memorization on pre-training data.
The results of the obfuscation study, while slightly lower, are comparable to the counterpart obtained with non-obfuscated versions. The obfus- cated prompts do not hinder LLMs' code genera- tion ability. **The findings suggest that the per- formance of LLMs on repo-level code genera- tion cannot be solely attributed to memorization. On the contrary, our propose ProgAI is the key to performance improvement.** We plan to conduct additional experiments on our PROGAI to further investigate the ex- tent of LLMs' memorization on pre-training data.

5.2 Case Study

We further observe the generated cases to assess PROGAI (*e.g.*, GPT-3.5-ReAct) and the base- line model (*e.g.*, GPT-

3.5-NoAgent). The comparative analysis is shown in Figure 4 and Figure 5.
We can find a distinct operational pattern in PROGAI. It typically begins with examining the code dependencies in the repository, sub- sequently refining its code generation strategy through a logical, step-by-step process known as "chain-of-thought". As in Figure 4, the in- put documentation specifies the need for a class with member functions *set_params* and *summary*. PROGAI, 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 PROGAI-*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 5.

6 Conclusion

We explore how LLMs perform in the real- world repo-level code generation task. We create PROGAI, a new LLM for repo- level code generation that includes rich information about the code repository, such as documentation and contextual dependency, to help LLMs better understand the code repository. To enhance the repo-level code generation ability, we propose ProgAI, an agent-based framework that uses external tools to assist with LLMs. Evaluation on nine LLMs shows that ProgAI works well for diverse programming tasks, highlighting its po- tential in real-world repo-level 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 ProgAI. 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 PROGAI. Some advanced programming tools are not explored. The limitation may restrict the agent's ability in some challenging scenarios.
Thirdly, in Section 5.5, 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 train- ing of L1-regularized log-linear models. In *Proceed- ings 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. Tom

Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

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 Ka- plan, 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.

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. Classe- val: 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, Man- tas 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 gen- eration with large language model. *arXiv preprint arXiv:2303.06689*.

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.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

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*.

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

Norman Di Palo, Arunkumar Byravan, Leonard Hasen- clever, Markus Wulfmeier, Nicolas Heess, and Mar- tin 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.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

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. Hugging- gpt: Solving AI tasks with chatgpt and its friends in huggingface. *CoRR*, abs/2303.17580.

Xiangru Tang, Bill Qian, Rick Gao, Jiakang Chen, Xinyun Chen, and Mark Gerstein. 2023. Biocoder: A benchmark for bioinformatics code generation with contextual pragmatic knowledge. *CoRR*, abs/2308.16458.