Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation

Li Zhong, Zilong Wang

University of California, San Diego
lizhong@ucsd.edu, zlwang@ucsd.edu

Abstract

Recently, large language models (LLMs) have shown an extraordinary ability to understand natural language and generate programming code. It has been a common practice for software engineers to consult LLMs when encountering coding questions. Although efforts have been made to avoid syntax errors and align the code with the intended semantics, the reliability, and robustness of the code generation from LLMs have not yet been thoroughly studied. The executable code is not equivalent to reliable and robust code, especially in the context of real-world software development. For example, the misuse of APIs in the generated code could lead to severe problems, such as resource leaks, program crashes, etc. Existing code evaluation benchmarks and datasets focus on crafting small tasks such as programming questions in coding interviews. However, this deviates from the problems developers typically consult LLMs about. To fill the missing piece, we propose a dataset ROBUSTAPI for evaluating the reliability and robustness of code generated by LLMs. We collect 1208 coding questions from Stack Overflow on 18 representative Java APIs. We summarize the common misuse patterns of these APIs and evaluate them on current popular LLMs. The evaluation results show that even GPT-4 has 62% of the generated code that contains API misuses. It would cause unexpected consequences if the code is introduced into real-world software.

Introduction

The new era of language modeling arrives when large language models (LLMs) are capable of generating customized code according to the user's needs (Ye et al. 2023; OpenAI 2023a; Anil et al. 2023). It is not surprising that more and more software engineers choose to query large language models for the answer to the coding questions, such as generating a code snippet using certain APIs or detecting bugs in a few lines of code. Large language models are able to respond more suitable and customized answers for the question compared with searching in the online programming forums, such as Stack Overflow.

Such a fast pace conceals potential risks in the code generation of large language models. From the perspective of software engineering, the robustness and reliability of generated code have not yet been thoroughly studied even if numerous works have been made to avoid syntax errors and improve semantic understanding in the generated code (Xu et al. 2022; Chen et al. 2021; Shen et al. 2023a; Luo et al. 2023). Unlike the online programming forums, the generated code snippets are not reviewed by the community peers and thus suffer from API misuse, such as missing boundary checking in file reading and variable indexing, missing file stream closing, failure in transaction completion, etc. Even if the code samples are executable or functionally correct, misuse can trigger serious potential risks in production, such as memory leaks, program crashes, garbage collection failures, etc, as shown in Figure 1. To make things worse, the programmers asking these questions could be vulnerable to the risk if they are novices to the APIs and cannot tell the violations in the generated code snippets. Therefore, it is essential to contemplate the code reliability while evaluating the code generation by large language models.

To evaluate the code generation of large language models, most of the existing benchmarks focus on the functional correctness of the execution result from the generated code, which means the code is acceptable as long as it is functional for the user's purpose (Chen et al. 2021; Yin et al. 2018; Lu et al. 2021). We argue that the correct execution result is important but it is not only the case in the software development scenario. What the engineers really need is a reliable code sample without potential risks in the long run. Moreover, the domain of most current programming datasets is far from software engineering. The data source is mostly online coding challenge websites, such as Codeforces, Kattis, Leetcode, etc (Hendrycks et al. 2021; Austin et al. 2021). Although remarkable progress has been made, we argue that they fail to substantially help the software development in practical scenarios.

To this end, we propose ROBUSTAPI, a comprehensive benchmark to evaluate the reliability and robustness of code generated by large language models, including a dataset of coding questions and an evaluator using the abstract syntax tree (AST) (Fischer, Lusiardi, and Von Gudenberg 2007). In the dataset, we target creating an evaluation setting that is close to real software development. Thus we collect representative questions about Java from Stack Overflow. Java is one of the most popular programming languages and is widely used in software development because of its *write*

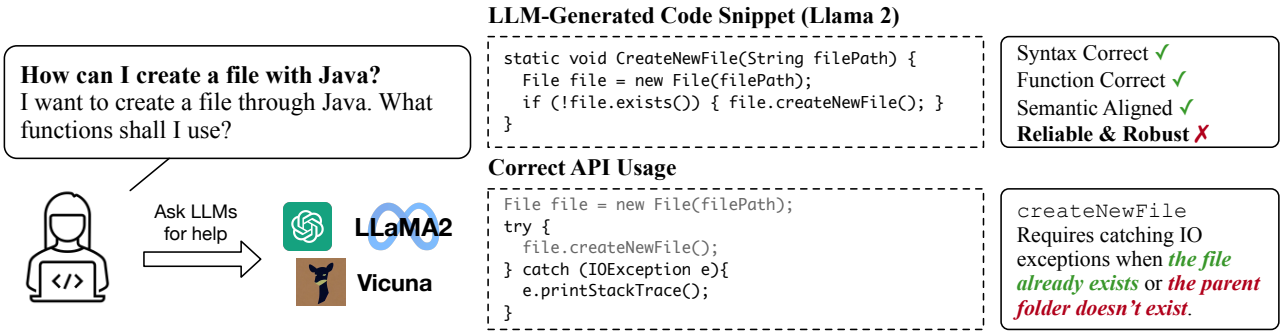

Figure 1: The scenario where software engineers consult large language models for the answer to the programming questions. The generated code snippet is not reliable and has potential risks in the software development.

once, run anywhere (WORA) feature[1]. For each question, we provide a detailed description and the related Java API. We design templates to trigger large language models to generate the code snippet and the corresponding explanation. We also provide an evaluator that analyzes the generated code snippets using the abstract syntax tree (AST) and compares them with the expected API usage patterns. Following Zhang et al. (2018), we formalize the API usage patterns into structured call sequences, as shown in Figure 2. The structured call sequences present how these APIs can be properly used to eliminate the potential system risks. Any violations of such structured call sequences would be considered as API misuse from the perspective of software engineering.

We collect 1208 real questions from Stack Overflow which involves 18 representative Java APIs. We run experiments on the close-sourced language models (GPT-3.5 and GPT-4 (OpenAI 2023a)) as well as the open-sourced language models (Llama-2 (Touvron et al. 2023), Vicuna-1.5 (Chiang et al. 2023). We use the default hyper-parameter settings of the models without extensive hyper-parameter tuning. We further design two experiment settings, zero-shot and one-shot, where none or one demonstration sample is provided in the prompt. We conduct a comprehensive analysis of the generated code and study the common API misuse cases of current large language models. We would like to bring up the important issues of API misuse in the code generation by large language models, and provide a new dimension to evaluate large language models other than the commonly-used functional correctness. The main purpose of this benchmark is not to evaluate the functional correctness of the generated code, but instead, we focus on reliability and robustness. We hope this work could facilitate future research on this topic and help create a more robust coding helper out of large language models to step further into real artificial general intelligence. We open-source our dataset and evaluator on GitHub[2]. We summarize our contribution as follows.

- We propose a new benchmark, RobustAPI, to evaluate the reliability and robustness of code generation by large language models. This is an important but not yet well-studied perspective to evaluate the code quality apart from functional correctness.
- We provide a well-formalized evaluation framework including a dataset of Stack Overflow questions and an API usage checker using AST. We report the performance of popular large language models, including GPT-3.5, GPT-4, Llama-2, and Vicuna-1.5.
- We conduct a comprehensive analysis of the code generation performance of current large language models. We summarize the common API misuse for each model and point out the promising improvement direction for the future research.

Related Work

Code Quality of LLM-Sythesized Code With the release of Copilot (Chen et al. 2021) and other commercial code assistant tools based on LLMs, the security and code quality of these tools gradually get the attention of the research community. Yetistiren, Ozsoy, and Tuzun (2022) assess the quality of LLM-generated code from the aspects of compilation correctness, functional correctness, and code efficiency. Siddiq et al. (2022) studied code smells in code generated by LLMs, which is the poor design in code like unusually long method, or duplicated code. Poesia et al. (2022) shows that LLMs can make implementation errors in the code like syntax errors or semantic errors deviating from users' intention. Jesse et al. (2023) studied simple, stupid bugs in Codex and other LLMs, which shows that AI code assistants can help avoid some of such simple bugs but have a higher chance of introducing bugs that are hard to detect. As for security impact, Pearce et al. (2022) designed 89 security-sensitive scenarios for Copilot to complete the code for users, which shows approximately 40% of the code is vulnerable. Perry et al. (2022) conducted the first large-scale user study to examine whether users interacting with AI Code assistants write secure code. They find that those users wrote significantly less secure code while they believe their code was secure. Sandoval et al. (2023) conducts a user study to assess the security of low-level code with pointer and array manipulations generated by AI-based coding assistants. They find under this specific scenario, the assistants

[1]https://en.wikipedia.org/wiki/Java_(programming_language)
[2]https://github.com/FloridSleeves/RobustAPI

do not introduce more security bugs than humans. Liu et al. (2023) enlarges HumanEval (Chen et al. 2021) by generating test cases with higher coverage which serve as an add-on to the existing programming benchmarks but the evaluation still focuses on functional correctness and simple programming questions far from software development. Shen et al. (2023b) evaluates the reliability of ChatGPT by testing on adversarial examples, which however has a different meaning of 'reliability' in their context. In this paper, we refer to reliability as the ability of code to resist failure, high workload, and unexpected input.

Quality Assessment of Code in Online Forum Existing literature in the software engineering field has investigated the quality of code from online forums and warned developers of the potential issues. Yang, Hussain, and Lopes (2016) finds that the majority of code examples given in Stack Overflow answers cannot be compiled. Zhou and Walker (2016) pointed out that 43% of the posts investigated by them contained deprecated APIs, while Fischer et al. (2017) found that 29% of the code contains security risks. In Zhang et al. (2018), the authors analyze the code by call sequence extraction and slicing, and compare it to the manually validated API usage rules, which concludes that 31% of the code examples in Stack Overflow answers contain API misuse and could produce unexpected behaviors.

Methodology

In this section, we describe RobustAPI, a comprehensive benchmark to thoroughly evaluate the reliability and robustness of LLM-generated code. We describe the process of data collection and prompt generation when constructing the dataset. Then we present the API misuse patterns evaluated in RobustAPI and discuss the potential consequence of violations. Finally, we introduce the static analysis method in RobustAPI for detecting the API usage violations which leverages the abstract syntax tree and achieves higher evaluation accuracy in evaluating the API misuse in code generated by LLMs compared to rule-based method such as keywords matching.

Data Collection

To take advantage of the existing research efforts in the software engineering field, we build RobustAPI based on the dataset from ExampleCheck (Zhang et al. 2018) as our starting point. ExampleCheck is proposed to study the frequent Java API misuse in online Q&A forums. We select 18 popular Java APIs from the dataset as shown in Table 1. These 18 APIs cover 6 domains including string processing, data structure, mobile development, crypto, I/O and database operation. Then we crawl questions relevant to these APIs from Stack Overflow. We only select the questions with online answers and we keep the questions whose provided answer contains API misuse. In this way, we guarantee that the questions in RobustAPI are answerable and non-trivial so we can use them to effectively evaluate the LLMs' ability in answering coding questions that *humans are prone to make mistakes*. After filtering, we get 1208 questions in total. The distribution of questions for each domain is shown in Table 1.

API	Domain	Conseq*	Github*
StringTokenizer.nextToken	String Process (307)	(iii)	13.3K
String.getBytes		(iii)	88.1K
JsonElement.getAsString		(iii)	4.4K
List.get	Data Structure (404)	(iii)	2.7M
Map.get		(iii)	2.4M
Iterator.next		(iii)	918K
ProgressDialog.dismiss	Mobile Develop (75)	(iii)	54K
TypedArray.getString		(iv)	6.8K
ApplicationInfo.loadIcon		(v)	3.6K
Activity.setContentView		(v)	4.6K
Cipher.init	Crypto (10)	(iii)	66.3K
RandomAccessFile.write	I/O (390)	(i)	129K
BufferedReader.readLine		(iii)	74.8K
PrintWriter.write		(i)	1.1M
File.mkdirs		(ii)	73.2K
File.createNewFile		(i)	176K
FileChannel.write		(i)	5.2K
SQLiteDatabase.query	Database (22)	(iv)	4K
Total	1208		7.8M

Table 1: 18 popular Java APIs in RobustAPI. They are easily misused by developers according to the existing literature of software engineering (Zhang et al. 2018). *Consequences: (i) data loss; (ii) file system corruption; (iii) program crash; (iv) resource leak; (v) user interface bug. *Github: occurrences of this API on Github.

After collecting the questions, we convert them into the JSON format with the following fields: {`id`, `api`, `question`, `origin`}. `id` field contains the unique id we assign for each sample. `api` field contains the API that we specifically instruct the large language models to use as a question hint. `question` field contains the title and description of the Stack Overflow questions. `origin` field contains the original URL of this sample.

Prompt Generation

In the prompt, we start with the task introduction and the required response format. Then we append the few-shot demonstrations on this API when conducting experiments in the few-shot settings. The demonstration examples satisfy our provided response format. Next, we append the question and the corresponding API hint for this question. This prompt simulates a user asking coding questions without providing any additional hints from the API documentation which is a typical scenario when novice developers seek help from large language models. Due to the chat completion nature of state-of-the-art LLMs, we wrap the question and answer with special tags to instruct LLMs to generate answers to the questions. The prompt template is adapted from (Patil et al. 2023), which can help LLMs follow a specific generation template so that we can extract more compilable code snippets from the response.

Demonstration Samples

Demonstration samples have been proven helpful to LLMs in understanding natural language. To thoroughly analyze LLMs' ability in code generation, we design two few-shot settings, One-shot-irrelevant and One-shot-relevant.

In the one-shot-irrelevant setting, we provide LLMs with an example using an irrelevant API (e.g. `Arrays.stream`). We assume this demonstration example would eliminate the syntax errors in the generated code.

In the one-shot-relevant setting, we provide LLMs with an example using the same API as the given question. The provided example contains a pair of question and answer. The question in the demo example is not present in the testing dataset and we manually revise the answer to ensure that there is no API misuse in it and that the semantics well align with the questions.

Java API Misuse

When using the APIs provided by language libraries, developers need to follow the API usage rules so that they can take full advantage of the ideal API effect. Violating these rules and misusing the APIs could result in unexpected behaviors in production. A typical example is the file operation. When opening and writing a file through `RandomAccessFile`, two usage rules need to be enforced: (1) Reading the file could throw exceptions. If the buffer limit is reached before the expected bytes are read, the API would throw `IndexOutOfBoundsException`. Also, if the file is concurrently closed by other processes, the API would throw `ClosedChannelException`. To deal with these exceptions, the correct implementation should enclose the API inside `try-catch` blocks. (2) The file channel should be closed after usage. Otherwise, if this code snippet is inside a long-lasting program that is concurrently running in multiple instances, the file resources could be run out. Therefore, the code needs to invoke `close` API after all file operations. The correct usage are shown as following:

Correct API Usage:

```
try {
   RandomAccessFile raf =
     new RandomAccessFile("/tmp/file.json", "r");
   byte[] buffer = new byte[1024 * 1024];
   int bytesRead = raf.read(buffer, 0, buffer.length);
   raf.close();
} catch(Exception e) {...}
```

In ROBUSTAPI, we summarized 41 API usage rules from the 18 APIs, which are validated in the documentation of these APIs (Zhang et al. 2018). These rules include: (1) The guard condition of an API, which should be checked before API calls. For example, check the result of `File.exists()` before `File.createNewFile()` (2) Required call sequence of an API, which should be called in a specific order. For example, call `close()` after `File.write()`. (3) Control structures of an API. For example, enclose `SimpleDateFormat.parse()` with `try-catch` structure.

Detecting API Misuse

Existing research in evaluating the code generated by LLMs usually uses test cases, which falls short when testing the reliability and robustness of code. To deal with this challenging problem, we use static analysis for ROBUSTAPI, which has relatively mature solutions in detecting API misuse (Zhang et al. 2018; Nguyen et al. 2014; Wang et al. 2013; Huang et al. 2023). To evaluate the API usage correctness in code, ROBUSTAPI detects the API misuses against the API usage rules by extracting call consequences and control structures from the source code, as shown in Figure 2. The code checker first checks the code snippets to see whether it is a snippet of a method or a method of a class so that it can enclose this code snippet and construct an abstract syntax tree (AST) from the code snippet. Then the checker traverses the AST to record all the method calls and control structures in order, which generates a call sequence. Next, the checker compares the call sequence against the API usage rules. It infers the instance type of each method call and uses the type and method as keys to retrieve corresponding API usage rules. Finally, the checker computes the longest common sequence between the call sequence and the API usage rules. If the call sequence does not match the expected API usage rules, the checker will report API misuse.

Experimenet

Experiment Setup

In the experiments, we evaluate ROBUSTAPI on four LLMs: GPT-3.5 (OpenAI 2023a), GPT-4 (OpenAI 2023a), Llama-2 (Touvron et al. 2023), Vicuna-1.5 (Chiang et al. 2023). We use the default hyper-parameter settings of each model without further extensive hyper-parameter tuning. All experiment results are Pass@1 unless specified. For all models, we evaluate three experiment settings:

- **Zero-shot:** No example is provided in the prompt. The prompt only contains the instruction, question.
- **One-shot-irrelevant:** ROBUSTAPI provides one example of an irrelevant task in the prompt.
- **One-shot-relevant:** ROBUSTAPI provides one example of the same API with the correct usage in the prompt.

The examples for shot generations are manually written and double-checked by the authors. Then they are evaluated against the API usage checkers to make sure they are aligned with the API usage rules.

Evaluation Metrics

To quantitatively evaluate the reliability of the generated code, we define the following values and our metrics are computed based on them. Supposing that we have N questions in our dataset, we divide them into three groups.

- N_{misuse}: The number of cases where our API usage checker detects the API usage violations.
- N_{pass}: The number of cases where our API usage checker does not detect the API usage violations.
- $N_{\text{non-comp}}$: The number of cases where the LLM fails to generate code or the generated code is not compilable.

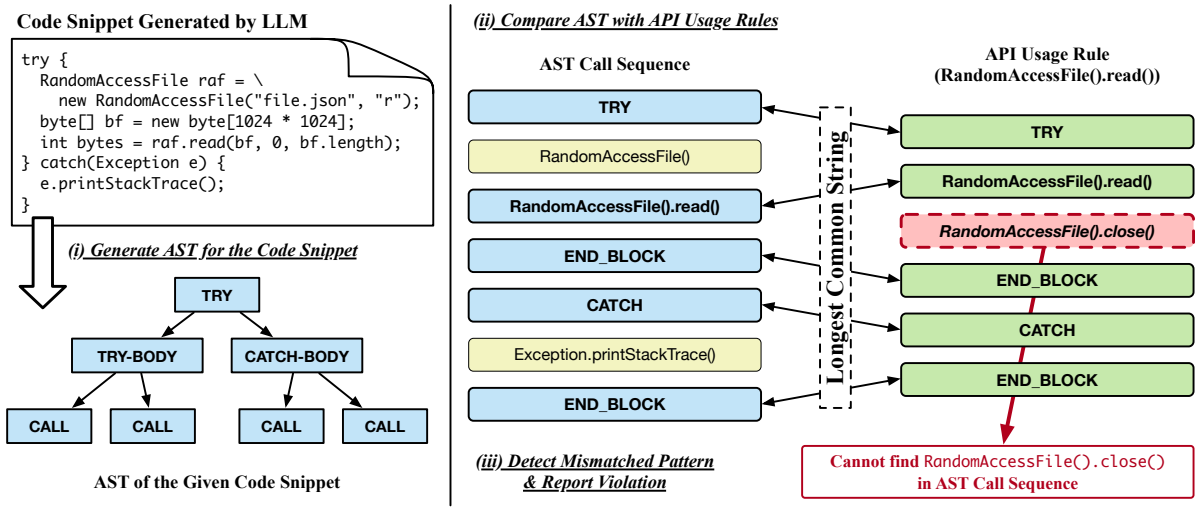

Figure 2: The workflow of Our API Checker. The API checker uses the static analysis method and analyzes the generated code with the abstract syntax tree (AST). The API misuse is detected when the AST call sequence and the API usage mismatches.

Based on the values, we define our metrics.

- **API Misuse Rate** $= N_{\textbf{misuse}}/(N_{\textbf{misuse}} + N_{\textbf{pass}})$**:** To analyze the proportion of misuse cases among the compilable code snippets. It reveals how reliable the generated code is after the users filter out the non-compilable cases.
- **Compilation Rate** $= (N_{\textbf{misuse}} + N_{\textbf{pass}})/N$**:** To analyze the proportion of compilable cases among all questions. It is necessary to consider the percentage of compilable cases in order to eliminate the influence from the extreme situations, such as when only a few compilable code snippets are generated.
- **Overall API Misuse Percentage** $= N_{\textbf{misuse}}/N$**:** To analyze the proportion of misuse cases among all questions.

Research Questions

We conduct a series of experiments on state-of-the-art LLMs based on RobustAPI, which demonstrate the usability and effectiveness of RobustAPI. The experiments provide insights on the ability to answer real-world coding questions and the robustness and reliability of these answers regarding API misuse problems. In the experiment, we try to answer the following questions:

- **Q1:** What are the API misuse rates in answering real-world coding questions by these LLMs?
- **Q2:** How do irrelevant shots affect the results?
- **Q3:** Can correct API usage examples reduce the misuse?
- **Q4:** Why does generated code fail the API usage check?

API Misuse Rate

Firstly, we present the API misuse rate of each model based on RobustAPI on the left of Figure 3. In this figure, the higher the API misuse rate is, the worse the code reliability and robustness for this large language model. The API misuse rate is calculated by dividing answers that can be compiled and contains API misuses by all the answers that

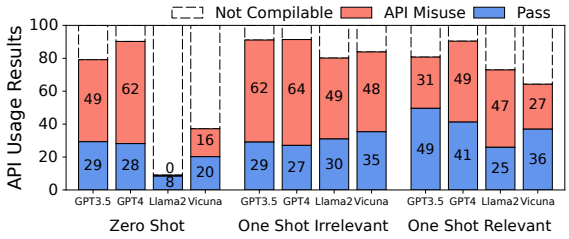

Figure 3: Result of Checking API Usage from LLMs. Red bars are the percentage of answers that contain API misuse, which is the *lower*, the *better*. The white bars in dot lines are the percentage of code answers that are not compilable.

can be compiled. From the evaluation results, all the evaluated models suffer from API misuse problems, even the state-of-the-art commercial models like GPT-3.5 and GPT-4. In zero-shot settings, Llama has the lowest API misuse rate. However, this is partially due to that most of Llama's answers do not include any code. A counter-intuition finding is that GPT-4 actually has a higher API misuse rate than GPT-3.5, though the coding ability of GPT-4 is proved to be *"40% more advanced than its predecessor, GPT-3.5"* (OpenAI 2023b). We also evaluate a code-specialized large language model, DeekSeekCoder(Piplani and Bamman 2018), which is trained on a variety of programming languages including Java, and surpasses many existing Code LLMs. We report the results of `deepseek-coder-6.7b-base` and `deepseek-coder-6.7b-instruct`. We observe that the code-specialized large language model can generate more compilable samples. However, the API misuse rate is not significantly better than other models. This indicates that with the code generation ability of large language models is largely improved nowadays, the reliability and robustness of code in real-world production rises as an unnoticed issue. And the space for improvement is huge for this problem.

The execution time for static analysis is shown in Table 3.

Model	Zero-shot			One-shot-irrelevant			One-shot-relevant		
	Misuse Rate ↓	Compilable Rate ↑	Overall Misuse ↓	Misuse Rate ↓	Compilable Rate ↑	Overall Misuse ↓	Misuse Rate ↓	Compilable Rate ↑	Overall Misuse ↓
GPT 3.5	62.97%	79.14%	49.83%	68.09%	91.06%	62.00%	38.56%	80.71%	31.13%
GPT 4	68.81%	90.23%	62.09%	70.38%	91.39%	64.32%	54.40%	90.40%	49.17%
Llama 2*	7.34%*	9.02%*	0.66%*	61.36%	80.13%	49.17%	64.47%	72.93%	47.02%
Vicuna 1.5	45.66%	37.17%	16.97%	57.85%	83.86%	48.51%	42.53%	64.24%	27.32%
ds-coder-6.7b-base	41.55%	40.65%	16.89%	75.60%	95.90%	72.43%	64.12%	67.14%	43.05%
ds-coder-6.7b-instruct	47.52%	50.00%	23.76%	59.04%	96.61%	57.04%	38.40%	86.01%	33.03%

Table 2: Performance of Each LLM on ROBUSTAPI. ↓: the *lower* the *better*. ↑: the *higher* the *better*. Misuse Rate is the proportion of misuse cases among the compilable cases; Compilation Rate is the proportion of compilable cases among all questions; Overall Misuse is the proportion of misuse cases among all questions. *Though Llama2 has a low misuse rate, its compilation rate is significantly lower than other models.

The time difference is due to the different coding styles of each LLM, all of which are within 7 minutes.

GPT 3.5	GPT 4	Llama 2	Vicuna 1.5	DeepSeek-Coder
6m 31s	6m 56s	6m 36s	6m 19s	6m 36s

Table 3: Execution Time of Static Analysis in ROBUSTAPI.

***Finding* 1.** *Answers to real-world coding questions from the state-of-the-art large language models widely have API misuse problems.*

One-Shot-Irrelevant Results

In this experiment, ROBUSTAPI gives a pair of question and answer as an example to show the model how to follow the template required by the instructions. The example contains no information about the API usage checked by ROBUSTAPI. The result is shown in the middle of Figure 3. However, for most models, the irrelevant shot does not significantly reduce the API misuse rate but on the contrary, slightly increases the misuse rate. One possible reason for this is the irrelevant shot provided to the large language models actually encourages the models to give a lengthy code solution, which increases the chance of API misuse. API misuse rate of Llama increases significantly after adding the irrelevant shot because it has more valid answers that contain code snippets. Overall, adding an irrelevant shot triggers the large language models to generate more valid answers, which enables a better evaluation of the code reliability and robustness.

***Finding* 2.** *Among all the answers containing compilable code, 57-70% of the LLM answers contain API misuse, which could lead to severe consequence in production.*

***Finding* 3.** *Irrelevant shot examples does not help decrease the API misuse rate but triggers more valid answers, which show to be effective for benchmarking the model performance.*

One-Shot-Relevant Results

In this experiment, ROBUSTAPI adds a manually-written shot in the prompt, which performs a different task but uses the same API. This gives hints to LLMs on how to use these APIs correctly. From the results, after adding the correct usage shot, the API misuse rates of GPT-3.5, GPT-4, and Vicuna significantly drop. This indicates an effective improvement under this experiment setting. As for Llama, the relevant shot does not improve the performance. This experiment shows that some LLMs can effectively 'learn' the correct API usage and follow the usage. However, since existing language models are trained with data from code repositories if the training datasets contain a large number of API violations, the language models are prone to generate code with API misuses, which explains the high API misuse rate in zero-shot and one-shot-irrelevant evaluation. We show Pass@k results of one-shot-relevant in Table 4.

Pass@k	Misuse Rate	Compilation Rate	Overall Misuse
Pass@1	39.06%	76.08%	29.72%
Pass@5	21.98%	93.79%	20.61%
Pass@10	16.51%	96.27%	15.89%

Table 4: Pass@k results of GPT 3.5 (T=1, one-relevant-shot).

***Finding* 4.** *Some LLMs can learn from the correct usage example, which reduce the API misuse rate.*

Robustness Analysis

We evaluate the benchmark on GPT 3.5 under different temperatures (Table 5). From the result, changing temperature does not significantly change the misuse rate and compilation rate. To study the effect of different prompting methods, we study how the API misuse rate changes when we replace the one-shot examples with the API usage rules. We feed the symbolized rules to ChatGPT and get the rules in natural language. We add the usage rules as part of the prompts and evaluate GPT-3.5 with ROBUSTAPI. The results are shown in Table 6, which indicates that the API usage rules might not help reduce the API misuse rate compared to one-shot relevant examples.

Temp.	Misuse Rate	Compilation Rate	Overall Misuse
T = 0	38.56%	80.71%	31.13%
T = 0.5	39.77%	80.13%	31.87%
T = 1.0	39.06%	76.08%	29.72%

Table 5: Results of GPT 3.5 with different temperature (Pass@1, one-relevant-shot).

Prompt	Misuse Rate	Compilation Rate	Overall Misuse
Usage Rule	65.01%	79.78%	51.86%
Relevant Shot	38.56%	80.71%	31.13%

Table 6: Results of GPT 3.5 with API usage rules (T=0, Pass@1).

***Finding* 5.** *Increasing temperature or replacing one shot examples with API rules does not affect the API misuse rate significantly.*

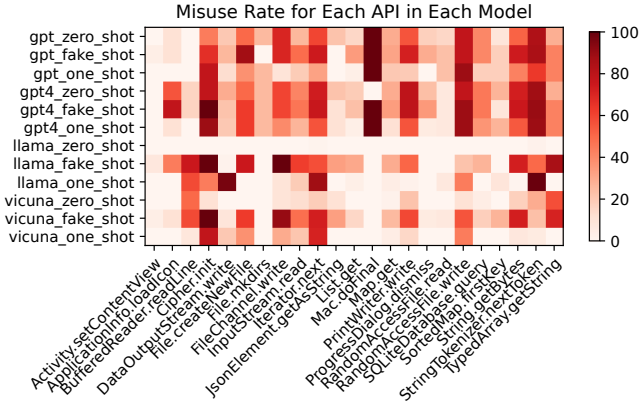

Figure 4: Misuse rate of each API by each LLM. The *deeper* the color, the *higher* the misuse rate. G3.5, G4, LMA, Vic are short for GPT3.5, GPT4, Llama2, Vicuna1.5.

In this section, we discuss the answers from LLMs that cannot pass the API usage check in ROBUSTAPI evaluation. There are two categories for failure cases: cases that are not compilable, and cases that are compilable but contain API misuses as shown in Figure 3. We refer to the ability to be compiled successfully as *compilability*. The compilation failure rate is calculated by dividing the number of cases that can be compiled to the total number of cases in the benchmarks. GPT-4 performs the best among all the models regarding compilability, which has less than 10% of answers that cannot be compiled across all experiment settings. Adding a few shots to prompts helps reduce the compilation failure rate in the evaluation results for all models. As for the API misuse rate, we dive deeper into the APIs that LLMs are prone to misuse. Figure 4 details the misuse rate of each API for each LLM. Among all APIs, the Android development API `Activity.setContentView` has the lowest misuse rate across all the models.

Case Study: API Misuse in GPT-3.5

Taking GPT-3.5 as an example, we show a typical sample that GPT-3.5 reacts differently under different experiment settings. This question asks the model to help write a string to the file using API `PrintWriter.write`. Under zero-shot and one-irrelevant-shot settings, the answers differ slightly but both misuse the API by not catching exceptions. After giving the model the correct API usage example, the model learns how to use the API and responds correctly.

Zero Shot:
```
PrintWriter writer = new PrintWriter("f.txt", true);
writer.write("text to append");
writer.close();
```
One Irrelevant Shot:
```
String text = "Hello, World!";
PrintWriter writer = new PrintWriter("f.txt", true);
writer.write(text);
writer.close();
```
One Relevant Shot:
```
try {String text = "Hello, World!";
PrintWriter writer = new PrintWriter("f.txt", true);
writer.write(text);
} catch (IOException e) {e.printStackTrace();}
```

Discussion

Extend to Other Language ROBUSTAPI focuses on Java API usage since Java is one of the most widely used languages in software development and has a special niche in web and Android ecosystems so that its API misuses may cause more serious problems in real applications. Theoretically, the method proposed in this paper can also be applied to other languages like Python.

Future Work The API misuse problem proposed in our research can motivate many further research directions. First, how to improve the quality of generated code aside from functionality alignment. To achieve this goal, in-context learning, fine-tuning, and pre-training would be critical to improving existing models. Besides, other online code community like Github could also be a useful resource to evaluate code models, as proposed in a recent work (Jimenez et al. 2023). As we believe, evaluating and improving LLMs on the perspective of real-world software development is a demanding and important problem.

Conclusion

In this paper, we propose a benchmark ROBUSTAPI to study the API misuse behaviors in code generated by LLMs. From the benchmark results on state-of-the-art models, we find that API misuse widely exists in large language models even when the code is executable and aligned with users' intention. Under different experiment settings, we explore effective methods of benchmarking and improving the API misuse rate of LLMs. To inspire and accelerate future research on this problem, we open source the dataset and benchmark in https://github.com/FloridSleeves/RobustAPI.

Acknowledgments

The authors sincerely appreciate the reviewers and chairs of the AAAI for their constructive and insightful comments. Their expertise and thorough reviews have significantly contributed to the enhancement of this paper.

References

Anil, R.; Dai, A. M.; Firat, O.; Johnson, M.; Lepikhin, D.; Passos, A.; Shakeri, S.; Taropa, E.; Bailey, P.; Chen, Z.; et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chiang, W.-L.; Li, Z.; Lin, Z.; Sheng, Y.; Wu, Z.; Zhang, H.; Zheng, L.; Zhuang, S.; Zhuang, Y.; Gonzalez, J. E.; Stoica, I.; and Xing, E. P. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality.

Fischer, F.; Böttinger, K.; Xiao, H.; Stransky, C.; Acar, Y.; Backes, M.; and Fahl, S. 2017. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, 121–136. IEEE.

Fischer, G.; Lusiardi, J.; and Von Gudenberg, J. W. 2007. Abstract syntax trees-and their role in model driven software development. In *International Conference on Software Engineering Advances (ICSEA 2007)*, 38–38. IEEE.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Huang, H.; Shen, B.; Zhong, L.; and Zhou, Y. 2023. Protecting data integrity of web applications with database constraints inferred from application code. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 632–645.

Jesse, K.; Ahmed, T.; Devanbu, P. T.; and Morgan, E. 2023. Large Language Models and Simple, Stupid Bugs. *arXiv preprint arXiv:2303.11455*.

Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; and Narasimhan, K. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770*.

Liu, J.; Xia, C. S.; Wang, Y.; and Zhang, L. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*.

Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C.; Drain, D.; Jiang, D.; Tang, D.; et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568*.

Nguyen, H. A.; Dyer, R.; Nguyen, T. N.; and Rajan, H. 2014. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 166–177.

OpenAI. 2023a. GPT-4 Technical Report. *ArXiv*, abs/2303.08774.

OpenAI. 2023b. GPT-4 Technical Report. arXiv:2303.08774.

Patil, S. G.; Zhang, T.; Wang, X.; and Gonzalez, J. E. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; and Karri, R. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, 754–768. IEEE.

Perry, N.; Srivastava, M.; Kumar, D.; and Boneh, D. 2022. Do users write more insecure code with AI assistants? *arXiv preprint arXiv:2211.03622*.

Piplani, T.; and Bamman, D. 2018. DeepSeek: Content based image search & retrieval. *arXiv preprint arXiv:1801.03406*.

Poesia, G.; Polozov, O.; Le, V.; Tiwari, A.; Soares, G.; Meek, C.; and Gulwani, S. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*.

Sandoval, G.; Pearce, H.; Nys, T.; Karri, R.; Garg, S.; and Dolan-Gavitt, B. 2023. Lost at c: A user study on the security implications of large language model code assistants. *arXiv preprint arXiv:2208.09727*.

Shen, B.; Zhang, J.; Chen, T.; Zan, D.; Geng, B.; Fu, A.; Zeng, M.; Yu, A.; Ji, J.; Zhao, J.; et al. 2023a. PanGu-Coder2: Boosting Large Language Models for Code with Ranking Feedback. *arXiv preprint arXiv:2307.14936*.

Shen, X.; Chen, Z.; Backes, M.; and Zhang, Y. 2023b. In chatgpt we trust? measuring and characterizing the reliability of chatgpt. *arXiv preprint arXiv:2304.08979*.

Siddiq, M. L.; Majumder, S. H.; Mim, M. R.; Jajodia, S.; and Santos, J. C. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 71–82. IEEE.

Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Wang, J.; Dang, Y.; Zhang, H.; Chen, K.; Xie, T.; and Zhang, D. 2013. Mining succinct and high-coverage API usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, 319–328. IEEE.

Xu, F. F.; Alon, U.; Neubig, G.; and Hellendoorn, V. J. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10.

Yang, D.; Hussain, A.; and Lopes, C. V. 2016. From query to usable code: an analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*, 391–402.

Ye, J.; Chen, X.; Xu, N.; Zu, C.; Shao, Z.; Liu, S.; Cui, Y.; Zhou, Z.; Gong, C.; Shen, Y.; et al. 2023. A comprehensive capability analysis of gpt-3 and gpt-3.5 series models. *arXiv preprint arXiv:2303.10420*.

Yetistiren, B.; Ozsoy, I.; and Tuzun, E. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 62–71.

Yin, P.; Deng, B.; Chen, E.; Vasilescu, B.; and Neubig, G. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, 476–486.

Zhang, T.; Upadhyaya, G.; Reinhardt, A.; Rajan, H.; and Kim, M. 2018. Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). *IEEE, New York, United States*, 886–896.

Zhou, J.; and Walker, R. J. 2016. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 266–277.