Quality Assessment of ChatGPT Generated Code and their Use by Developers

Mohammed Latif Siddiq
msiddiq3@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Lindsay Roney
lroney@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Jiahao Zhang∗
jzhang38@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

ABSTRACT
The release of large language models (LLMs) like ChatGPT has revolutionized software development. Numerous studies are exploring the generated response quality of ChatGPT, the effectiveness of different prompting techniques, and its performance in programming contests, among other aspects. However, there is limited information regarding the practical usage of ChatGPT by software developers. This data mining challenge focuses on DevGPT, a curated dataset of developer-ChatGPT conversations encompassing prompts with ChatGPT’s responses, including code snippets. Our paper leverages this dataset to investigate (RQ1) whether ChatGPT generates Python & Java code with quality issues; (RQ2) whether ChatGPT-generated code is merged into a repository, and, if it does, to what extent developers change them; and (RQ3) what are the main use cases for ChatGPT besides code generation. We found that ChatGPT-generated code suffers from using undefined/unused variables and improper documentation. They are also suffering from improper resources and exception management-related security issues. Our results show that ChatGPT-generated codes are hardly merged, and they are significantly modified before merging. Instead, based on an analysis of developers’ discussions and the developer-ChatGPT chats, we found that developers use this model for every stage of software development and leverage it to learn about new frameworks and development kits.

∗J. Zhang was a visiting student at ND from SUSTech (Southern University of Science and Technology) when this research was performed.

CCS CONCEPTS
- Software and its engineering → Software performance; Software usability; Empirical software validation.

KEYWORDS
datasets, chatgpt, security, quality, pull-request, open-coding

ACM Reference Format:

1 INTRODUCTION
The release of GitHub Copilot [16] and ChatGPT [2], two code generation assistants based on Large Language Models (LLMs), is reshaping software development [27]. A recent survey [30] showed that 92% of 500 US-based developers are using LLM-based code generation tools for both work and personal use. Code generation models help developers automate repetitive tasks, focusing on higher-level challenging tasks [43].

Although these code generation assistants are increasingly popular among developers [43], prior studies have shown that they can generate code that contains quality issues, such as code smells, security smells, and vulnerabilities [25, 33]. A recent study [32] also showed that code generation models are fine-tuned with samples containing code/security smells that leak to the generated code. With the increasing use of LLM-based code assistants, these problematic code snippets can get deployed into production, negatively affecting the software system’s security and reliability.

Although there are studies about code generation models [25, 32], there is no study about chat-style multimodel, i.e., ChatGPT [2]. As developers can make conversations about software development that are not limited to traditional code generation, ChatGPT can provide answers to developers in every step of software development (e.g., software management, testing, deployment etc.). Moreover,
there is no study about the ChatGPT-generated code from the perspective of code quality issues mined from real software developers’ prompts.

Thus, in this data mining challenge, our work investigates the quality issues of ChatGPT-generated code and their influence on developers’ discussions. We used the DevGPT dataset [42] to investigate the code and security smells in the generated Python and Java code using three static analyzers, Pylint [4], Bandit [1] and CodeQL [17]. We also investigated if the ChatGPT-generated Code merged into the code base. Furthermore, we did an open coding of the discussion and conversations in the pull requests where developers posted ChatGPT share links.

Our work provides the first view of quality issues in the code generated by ChatGPT and how the code is used among software developers. Our result shows that ChatGPT-generated codes have issues using undefined and unused variables and no proper documentation. They also suffer from improper resources, exceptions, and cryptography security management and use hard-coded credentials. Our result also shows that most ChatGPT-generated codes are not merged from the pull requests. Developers use the conversation to learn about libraries, frameworks, refactoring, and debugging code. The replication package can be found here: https://anonymous.4open.science/r/DevGPT-Study/.

2 BACKGROUND AND RELATED WORK

This section provides a high-level overview of code generation and code quality issues and related papers.

2.1 LLM-based Code Generation

LLM-based code generation techniques produce source code from av given prompt. A prompt can be a combination of natural language and source code. The code generation problem can be treated as a sequence-to-sequence (seq2seq) learning problem [35] and prior works used Recurrent Neural Networks (RNN) and neural networks based on Long Short-Term Memory (LSTM) [31, 35] to generate source code. More recently, the attention-based transformer architecture revolutionized the field of language learning [40]. There are several transformer-based deep learning models that are fine-tuned with code-related datasets for source code generation [18, 20, 36], search [11], and summarization [14]. Examples of this type of model include CodeBERT [11], CodeT5 [41], and Codex [8]. Codex models are powering GitHub Copilot [16], which are descendants of GPT-3 model [6]. ChatGPT is optimized for conversation and also can produce source code.

2.2 Code Smell

A code smell (“bad code smell” or “smell”) is an indicator of an improper choice of system design and implementation strategy [12, 13]. These flaws can stifle software development or increase the chance of future errors or failures [26]. An example of a smell taken from Siddiq et al. [32] is using the wrong exception catching order, as shown in Figure 1. The TypeError block is never reached, as the Exception block will catch all exceptions.

Security code smells (or simply “security smells”) are a subset of code smells. They are common code patterns that may lead to security flaws [28, 29, 32]. Although security smells may not be a vulnerability per se, they are symptoms that signal the prospect of a vulnerability [15]. For example, the code snippet in Figure 1 has a security smell related to the use of hard-coded credentials (CWE-798) [32]. This code snippet checks the password to a hard-coded string (i.e., "passw0rd!"), which can be exploited using a list of commonly used passwords.

Code and security smells introduce quality issues and negatively affect a project’s maintainability, readability, and security [13]. As manually identifying smells is time-consuming and may not scale [21], several works [7, 19, 24] developed techniques to detect code smells. Lanza and Marinescu [21] proposed a metric-based detection strategy to detect code smells, while Chen et al. [9] implemented a code smell detection tool, Pysmell, which can detect 11 code smells in a Python project. Di Nucci et al. [10] described machine learning-based experiments with a new dataset configuration that includes instances of multiple types of smells.

2.3 Empirical Studies on Code LLMs

Prior works investigated the usability of code generation models [38], whether they can generate vulnerable code [25] or whether it is as bad as humans in generating insecure code [5]. Siddiq et al. [32] focused on quality issues in the code generation datasets and their leakage from the closed and open source models. In other projects, they created a dataset and framework to systematically evaluate generated code from the security perspective [33, 34]. Liu et al. [22] focused on the quality issues of ChatGPT-generated code, but they are limited to programming problems from LeetCode [3]. Unlike these prior studies, we study real prompts made by software developers who used ChatGPT to generate code.

3 METHODOLOGY

In this study, we answer three research questions:

RQ1 Does ChatGPT generate code with smells?

A recent study [32] showed that datasets used to train open-source LLMs contain code and security smells that leak to the output generated by these models. In this RQ, we investigate whether ChatGPT, a closed-source LLM, also generates code with quality issues, i.e., containing code smells and security smells.

RQ2 Are ChatGPT-generated source codes merged into open source projects?

In this RQ, we investigate whether developers reuse code generated by ChatGPT in merged pull requests. In doing so, we also inspect what type of changes (if any) developers make prior to merging the generated code into production.

RQ3 How are developers using ChatGPT?
In this question, we aim to understand the use cases for ChatGPT beyond code generation. We investigate how developers use ChatGPT by analyzing the chat conversations.

To answer these RQs, we use the DevGPT dataset [42]. This dataset contains ChatGPT’s sharing links found in posts made on HackerNews as well as in code files, commits, issues, pull requests, and discussions posted on GitHub. This dataset has 2,345 share links. Each shared link corresponds to a chat between a developer and ChatGPT. This chat conversation can include not only natural language but also code written in different programming languages. In this study, we focus on the generation of code written in Python and Java as these are two popular languages among developers [23].

The next sections describe how we answer each of these RQs.

### 3.1 RQ1: Smells in Generated Code

To answer RQ1, we retrieved all share links mined from GitHub, i.e., links found in files, commits, issues, pull requests (PRs), or PRs’ discussions. Next, we excluded conversations that did not include code written in Python or Java. This way, we obtained 696 Python files from 357 ChatGPT conversations. Since 52 of these Python files (7.47%) had syntax errors, in the end, we had 644 ChatGPT-generated Python codes. In the case of Java, we initially obtained 147 Java files from 69 ChatGPT conversations. We converted them into Maven projects, named the files according to the public class’ name, and added dependencies as needed in order to make the code compilable. Yet, not all the generated Java codes could be compiled as some used classes that were not available in the conversation. Once we discarded un compilable samples, in the end, we had a total of 67 Java source codes from 35 ChatGPT conversations. To identify quality issues, we ran **Pylint** (v3.0.2) [4] and **Bandit** (v1.7.5) [1] on the Python files and **CodeQL** (v2.14.6) [17] for the Java files.

**Code Smell Analyzers.**

- **Pylint** [4] is a static analyzer for Python that can display five message types: *fatal* (i.e., unable to process the file), *error* (i.e., code smells that may lead to runtime errors), *warning* (i.e., Python-specific smells), *convention* (i.e., coding standard violations), and *refactor* (i.e., code smells that can be fixed through refactoring). Similar to a prior study [32], we ignored messages related to style issues about whitespaces, newlines, and invalid names as well as import-related messages as Pylint is not reliable in checking for import statements’ usage [39]. We also omitted *fatal* messages.

- **Bandit** [1] is a static analyzer that detects security smells in Python code. Each detected security smell maps to an ID from the Common Weakness Enumeration (CWE ID) [37].

- **CodeQL** [17] is a static analyzer that checks for style issues and vulnerabilities in a project by executing QL queries against a database generated from the source code. We used CodeQL to check for issues related to the following categories: *Advisory, Architecture, Compatibility, Complexity, Dead Code, Language Abuse, Performance, Violations of Best Practices and Security.*

### 3.2 RQ2: Generated Code in Merged PRs

To answer RQ2, we leveraged the metadata data from pull requests (PRs) where ChatGPT share links were mentioned. We only included PRs whose status was “merged” and that had at least one modified-added Python or Java file, obtaining a total of 50 pull requests. We then manually inspect each of these PRs and compare them with the source file(s) in the pull request in order to identify the files that contain verbatim copies or similar code to the ones generated by ChatGPT. For each PR, we also note what changes (if any) were made by developers to the generated code before making the pull request.

### 3.3 RQ3: ChatGPT Use Cases

In RQ3, we analyzed the developers’ discussion in 39 out of the 50 pull requests retrieved in RQ2, along with the chat conversation within the ChatGPT share link. We could not analyze 11 PR discussions as they were not in English or had no discussion texts. During this open coding, we analyzed the pull request’s title, body, and each comment made by the developers where the ChatGPT conversation is mentioned (if it is not mentioned in the PR body). We analyzed the context in order to annotate them with concepts (codes). This coding was performed by one of the authors of this paper, who has a software development experience of 2 years. The open code was then vetted by two other authors with 3 and 10 years of experience each. After reviewing the pull request information, we collaboratively highlighted the key points (which are presented in Section 4.3).

### 4 RESULTS

This section describes our findings and answers our RQs.

#### 4.1 RQ1: Generated Code Quality

Table 1 shows the quality issues found by Pylint, Bandit, and CodeQL to the Python/Java code generated by ChatGPT. We found that most messages for Python files are *error* and *convention* type. The top messages for each category are *undefined variables, missing docstrings, too few public methods in a generated class, and redefining a variable declared in the outer scope*. ChatGPT-generated code also suffers from security issues. Results from Bandit [1] show that there are 84 security issues and the top three of them are *requests without a timeout (CWE-400: Uncontrolled Resource Consumption), using pseudo-random generators (CWE-330: Use of Insufficiently Random Values) and using asserts (CWE-703: Improper Check or Handling of Exceptional Conditions).*

For Java code, the issues are about *not using proper JavaDoc in public methods, constructors, or variables, dead code, unused maven dependencies, and unused and unread local variables.* In terms of security smells, ChatGPT mainly provides hard-coded credentials in the code, which is related to *CWE-798: Use of Hard-coded Credentials.*
4.2 RQ2: Generated Code in Merged PRs

Out of the 50 analyzed PRs containing ChatGPT share links, only 6 of them (12%) contained generated code that was merged into the repository. Out of these 6 merged codes, 3 of them were verbatim copies of the generated code, whereas the other 3 contained modifications. To make the modification, developers changed the code to make it compatible with their code base. For example, the following code has been generated by ChatGPT, which uses Jira rest API to check permission to create an issue.

```python
if response.status_code == 200:
    permissions = response.json().get('permissions', {})
    can_create_issue = permissions.get('CREATE_ISSUES', {}).get('can', False)
    return self.can_create_issue(can_create_issue)
```

The following code, taken from a merged pull request3, uses the example above and is modified for the same purpose.

```python
def can_create_issue(issue) -> bool:
    self_jira_permissions = self._permissions
    project_key = self.project_key
    project_permissions = project_key.permissions.get('CREATE_ISSUES')
    return self.can_create_issue(issue, project_permissions)
```

4.3 RQ3: ChatGPT Use Cases

After doing an open coding of PRs’ discussions and the corresponding conversations with ChatGPT, we observed the following use cases for ChatGPT:

- **Libraries & Frameworks**: We found that ChatGPT was used in 36% of cases to get help with the APIs for (1) standard (built-in) Python modules/ Java packages and (2) external Python modules

3https://github.com/app-sre/qontract-reconcile/pull/3630/files

/ Java APIs. For example, we observed conversations in which developers sought help with Python’s string manipulation methods, datetime module, and deepcopy of objects. ChatGPT’s conversations were also used for understanding public libraries, especially for machine and deep learning frameworks, e.g., Pytorch, TensorFlow, pandas, skrub, pydantic, SparkAI etc.

- **Code Formatting and Refactoring**: We found 6 (15%) chats in which developers used ChatGPT for code formatting and standardization. For example, ChatGPT provided suggestions to a developer on how to re-structure a developer’s Python script into multiple files and folders. In another use case, we found that developers used ChatGPT to obtain advice on how to automatically check type hints in a Python project, to which case ChatGPT suggested using the library mypy.

- **Testing and Deploying**: We found 9 chats (23%) in which ChatGPT provided suggestions to developers related to version controlling, unit & integration testing, dependency management and documentation. Developers also use it to obtain help for deployment frameworks like Jira.

- **SDK Help**: In 4 of the analyzed conversations, ChatGPT was used to understand and troubleshoot software development kits (SDKs). The most commonly mentioned SDKs were Nylas-SDK, AWS SDK for Python (Boto3), and konfuzio-SDK.

- **Debugging**: Although ChatGPT prompts are limited to 4,000 tokens, we found 3 conversations in which developers used ChatGPT to debug file uploading issues, audio conversation problems, and arbitrary code execution.

- **Miscellaneous**: In 8 cases, ChatGPT was also used in miscellaneous use cases such as networking, bioinformatics-related data manipulation, message exchange, and audio streaming.

5 DISCUSSION

**Quality Issues in the Generated Code.** Our findings in RQ1 demonstrated that ChatGPT-generated codes have quality issues like not following standard coding practice, need refactoring, and security smells. The generated codes use undefined variables and return data outside of the functions. They are also poorly documented and keep undefined variables and arguments in the method. ChatGPT-generated code also faced security issues like not using timeout in resource-sensitive API calling, providing `assert` without proper exception management, and using hardcoded credentials.

**Implication for the Developers.** Developers are using ChatGPT for various purposes, as our findings in RQ3. They can be used to learn about unfamiliar libraries and development kits. Developers make conversations about code formatting, testing, and deployment. They also can be used for code debugging. However, our results in RQ1 show they are not free from code and security smells. They should be appropriately vetted before using the code base.

5.1 Threats of Validity

In our work, the dataset is used for a mining challenge, DevGPT [42], which can introduce external threats to validity. However, the dataset is vetted by the organizers. Our work focused on Python and Java, two of the top-used programming languages among developers, according to a survey from Stack Overflow [23].
Another validity threat to this work is that we used three static analyzers (Pylint [4], Bandit [1], and CodeQL [17]) to detect code quality issues. They can introduce false positives, but their precision is significantly high according to the literature [25, 32]. Another threat to this work concerns the manual evaluation of the PRs’ discussions and chats. The author’s professional software development experience and the peer review by the other senior authors mitigated this internal validity threat.

### 6 CONCLUSION AND FUTURE WORK

ChatGPT is revitalizing the software development process. Our work shows developers use them for code formatting, debugging, testing, and deployment. They are using it for learning new frameworks. However, the generated code from ChatGPT can be sub-standard and have security smells. In addition to that, they can be merged into the software code repositories. In the future, we are going to extend our work to other programming languages and investigate how the developers are making conversation to reach the solution.

### REFERENCES


