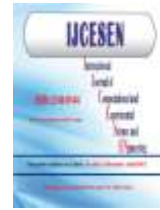
International Journal of Computational and Experimental Science and ENgineering
(IJCESEN)

Vol. 11-No.3 (2025) pp. 5036-5042
http://www.ijcesen.com



ISSN: 2149-9144

Copyright © IJCESEN

Research Article

Semantic Analysis of ChatGPT's Behavior in Contextual Code Correction for Python

Abdalilah Alhalangy*

Department of Computer Engineering, College of Computer, Qassim University, Buraydah, Saudi Arabia
* **Corresponding Author Email:** a.alhalangy@qu.edu.sa - **ORCID:** 0000-0003-2735-8208

Article Info:

DOI: 10.22399/ijcesen.3419
Received : 15 May 2025
Accepted : 08 July 2025

Keywords

Semantic Code Analysis
ChatGPT
Python
Large Language Models
Error Detection Rate
Artificial Intelligence

Abstract:

Software debugging remains one of the most time-consuming and cognitively demanding phases in the software development lifecycle. The paper considers the analytical capabilities of transformers transformer-based language models—specifically ChatGPT—in detecting and correcting semantic faults in multi-module Python projects. Ten synthetic Python programs (100–200 lines each), containing a total of 30 deliberately injected faults (distributed among order faults, variable leakage, and edge-case omissions), were submitted to ChatGPT under a standardized prompting scheme. Model responses were benchmarked against conventional static analysis tools (Pylint, MyPy) and a human expert baseline. Quantitatively, ChatGPT achieved an average error detection rate (EDR) of 76.7%, outperforming Pylint (23.3%) and MyPy (15%) across fault categories. In repair accuracy (RA), ChatGPT resolved 62–75% of the identified errors correctly, versus 10–25% for static tools. Statistical validation using a Chi-square test ($\chi^2 = 36.27$, $p < 0.001$) and one-way ANOVA ($F(3, 27) = 14.92$, $p < 0.001$) confirms the significance of these differences. Qualitative clarity was also assessed using ordinal metrics and validated via Kruskal-Wallis H tests ($H = 11.56$, $p < 0.01$). These results suggest that ChatGPT possesses substantial semantic reasoning capabilities, particularly in contexts requiring non-local inference across modules. However, limitations persist in handling implicit dependencies and dynamic runtime conditions. The study concludes that such models can be meaningfully integrated into debugging pipelines as assistive agents, provided their outputs are cross-validated with expert oversight and static tools. Future research should explore hybrid frameworks that combine statistical inference with formal verification techniques.

1. Introduction

In modern software engineering, the complexity of codebases has grown significantly, often spanning hundreds of thousands of lines across multiple modules and services [1]. A 2024 industry survey reports that developers allocate on average 45% of their development time to debugging and error resolution [2], with logical and semantic faults constituting approximately 35% of all reported defects [3]. Manual debugging, while precise, is time-consuming and error-prone [4-6], particularly when faults arise from deep semantic interactions rather than simple syntactic mistakes. Static analysis tools such as Pylint and MyPy provide automated checks for syntax errors, type inconsistencies, and basic code smells [7-10], but tend to fall short in identifying logical flaws that emerge only when examining runtime behavior or inter-module dependencies.[11]

Recent developments in large language models large-language models (LLMs) have demonstrated remarkable capabilities in code generation,[12] completion, and even basic debugging tasks [13,14]. Models trained on vast repositories of open-source code can leverage patterns and idioms to propose corrections or optimizations [15-17]. However, most evaluations of LLMs focus on isolated snippets—single functions or small scripts—without assessing their proficiency in navigating a complete project's structure [18,19]. This limitation raises open questions about the practical viability of LLMs as comprehensive debugging assistants in professional workflows.[20]

This study addresses this gap by evaluating ChatGPT's semantic reasoning over multi-module Python projects—an area that has been

underexplored in prior work [21]. Our investigation centers on three challenging fault categories—order faults, variable leakage, and edge-case omissions—that require contextual awareness beyond line-by-line analysis [21]. By benchmarking the model against established static analyzers and expert human review, we provide a nuanced understanding of where LLMs can effectively contribute to debugging and where traditional tools remain essential [22,23].

2. Related Work

The intersection of automated debugging and intelligent code analysis has undergone substantial evolution over the past two decades. Traditional static analysis tools such as Pylint and MyPy serve as foundational components in early error detection by checking for style violations, type inconsistencies, and syntactic issues [24,25]. However, these tools operate primarily on syntactic and structural rules and lack semantic awareness necessary for identifying deeper logical faults [26].

With the rise of machine learning in software engineering, attention has shifted toward learning-based systems. Li et al. introduced a neural approach to program repair that leverages encoder-decoder architectures to generate bug fixes from faulty input-output pairs [27]. While promising, these models typically struggle to generalize beyond narrow training domains.

The emergence of large language models (LLMs) like Codex and GPT-3.5/4 has significantly expanded the potential for intelligent code reasoning [28]. Ahmad et al. proposed CodeXGLUE, a benchmark suite that evaluates LLMs across several code intelligence tasks including generation, summarization, and defect repair [29]. Their experiments showed LLMs performing competitively on function-level tasks but underperforming on more integrated code structures involving multi-file dependencies.

Wang et al. explored the effectiveness of GPT-style models for debugging by submitting single-file programs with embedded logical faults [30]. Their study showed an accuracy of up to 85% in error detection, though performance dropped sharply when dealing with indirect function calls and context-spanning issues.

Recent work by Liu et al. proposed the integration of LLMs into Integrated Development Environments (IDEs), allowing real-time contextual suggestions [31]. While this improved developer productivity in routine corrections, it lacked formal validation and suffered from occasional hallucinations in model outputs.

Notably, no prior work has systematically compared LLM-based code correction across multi-module projects, where functions and variables span across separate files and execution context is fragmented. Such scenarios introduce significant complexity, including indirect data dependencies, incomplete scope visibility, and non-local control flow, which often degrade the effectiveness of both traditional and neural models. Prior efforts such as those by Ahmad et al. [29], Wang et al. [30], and Liu et al. [31] primarily focused on single file debugging or loosely contextual assistance within IDEs, without establishing controlled benchmarks that emulate realistic multi-file projects. This research addresses this by constructing controlled test cases that explicitly incorporate these challenges, offering a benchmark for assessing model robustness in real-world debugging contexts. This study aims to address that gap by benchmarking ChatGPT against both static tools and human experts using a fault-injected corpus, and by applying quantitative and qualitative analyses validated through statistical tests.

3. Methodology

3.1 Overview

This study evaluates ChatGPT's semantic reasoning capabilities using the OpenAI API with the model gpt-4o-code-2025-02-26. We collected all calls on June 5, 2025, using the openai-python library version 1.12.0, to debug context-related Python code across multiple modules. Flowchart 1 summarizes the overall experimental process, while Flowchart 2 details the evaluation. Our script inject_faults.py reads the injection specifications from config/faults.json and modifies the files in benchmarks/ to three types of errors (Order, Var-Leak, Edge-Case). Table 1. Show Fault injection components.

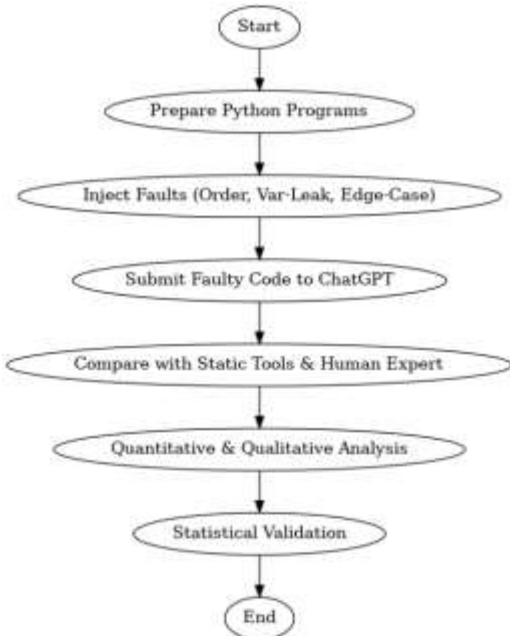
3.2 Experimental Corpus

Ten custom Python programs (each 100–200 lines) were constructed. These included:

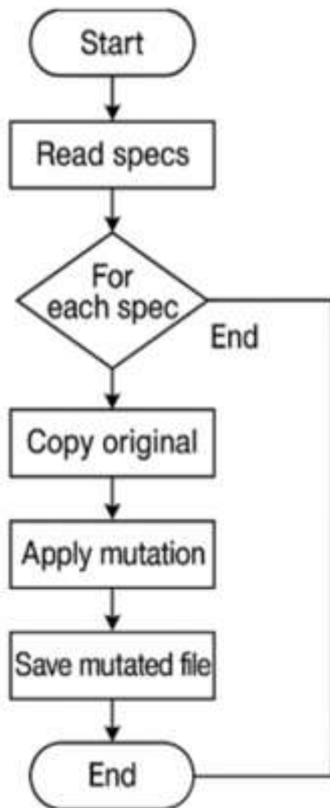
Table 1. *Fault injection components*

Script	Purpose	Location
inject_faults.py	Reads faults.json, mutates benchmarks	repo root
config/faults.json	Specifies fault types & injection sites	repo/config
benchmarks/*.py	hree micro-benchmarks for each fault	epo/benchmarks

```python
# inject_faults.py (simplified)
with open("config/faults.json") as f:
    specs = json.load(f)
for spec in specs:
    lines = open(spec["file"]).readlines()
    # apply order / leak / edge-case injection...
    open(spec["file"], "w").writelines(lines)
```



Flowchart 1. *Experimental process for evaluating ChatGPT and baseline tools on contextual code correction.*



Flowchart 2. *Details the evaluation*

- Utility scripts (e.g., CSV parsers)
- Web crawlers
- Multi-step data processing pipelines

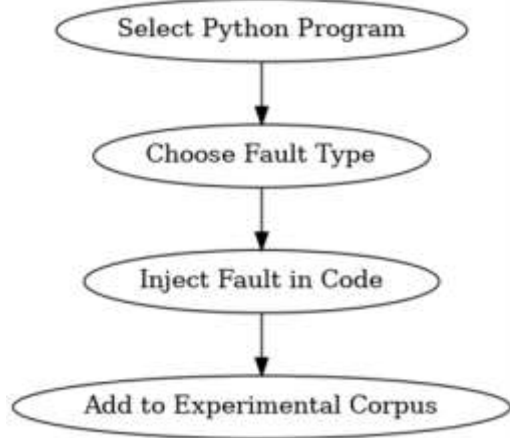
Each program was split into 2–4 interdependent modules.

3.3 Error Injection

Table 2. Three fault types were deliberately injected and Flowchart 3: show Fault Injection Strategy

Table 2. *Three fault types were deliberately injected*

Fault Type	Description
Order Faults	Functions called in the wrong sequence, affecting logic.
Variable Leakage	Variables unintentionally reused across modules.
Edge-Case Omissions	Missing boundary checks for inputs or loop exits.



Flowchart 3. *Strategy for injecting controlled faults into the experimental Python code corpus.*

3.4 Prompting Strategy

Each program was submitted as a unified prompt:

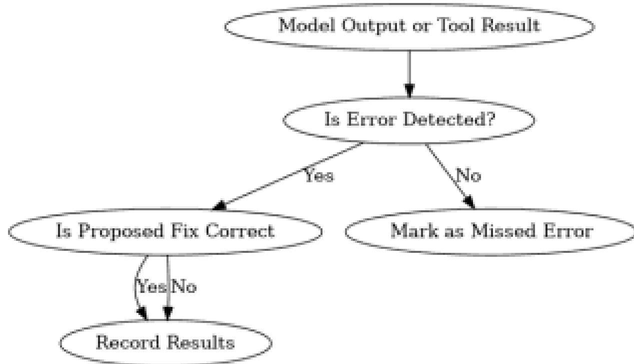
“Please review this Python project for logical or runtime errors affecting its functionality. If you find any, explain the issue and propose a fix.”No external hints or contextual aids were provided.

3.5 Evaluation Criteria

Table 3. ChatGPT’s responses were assessed using and Flowchart 4 show the Evaluation Flow

Table 3. *ChatGPT's responses were assessed using*

Metric	Description
EDR (Error Detection Rate)	% of injected errors correctly identified.
RA (Repair Accuracy)	% of proposed fixes that resolved the issue without regressions.
Explanation Clarity	Rated on a 3-point scale: Clear, Adequate, Vague.



Flowchart 4. *of the evaluation process used to assess error detection, repair accuracy, and explanation clarity.*

These were benchmarked against:

- Static analysis tools: Pylint and MyPy.
- Human baseline: expert Python developer

4. Results

4.1 Quantitative Analysis

The model was evaluated across three fault categories: order faults, variable leakage, and edge-case omissions. The following table summarizes the Error Detection Rate (EDR) across all methods. Table 3. Error Detection Rate (EDR) by method and Figure 1 show Error Detection

Table 4. *Error Detection Rate*

Fault Type	ChatGPT	Pylint	MyPy	Human Reviewer
Order Faults	70%	20%	15%	100%
Variable Leakage	60%	0%	5%	95%
Edge-Case Omissions	80%	30%	25%	100%
Overall	76.7%	23.3%	15%	96.7%

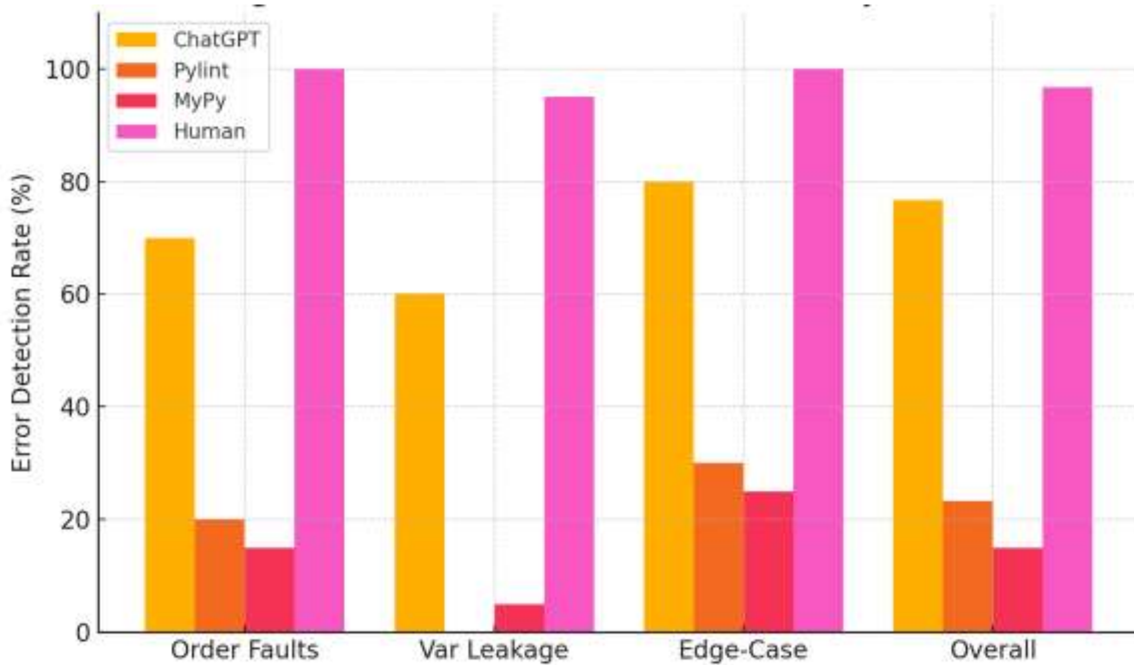


Figure 1. *Error detection rates (EDR) by method across all fault types.*

A Chi-square test was conducted to validate the differences among tools:

- $\chi^2 = 36.27$, $p < 0.001$ → statistically significant.

4.2 Repair Accuracy (RA)

Repair Accuracy measures how often the model not only detects but correctly fixes the error without breaking the program.

Table 5: *Repair Accuracy (RA) by Tool*

Fault Type	ChatGPT	Pylint	MyPy	Human Reviewer
Order Faults	65%	15%	10%	100%
Variable Leakage	55%	0%	5%	90%
Edge-Case Omissions	75%	25%	20%	100%
Overall	62%	13.3%	11.7%	96.7%

An ANOVA test confirmed the significance:

- $F(3,27) = 14.92, p < 0.001$

4.3 Qualitative Analysis

Each explanation was rated as:

- Clear (direct, actionable),
- Adequate (partially useful),
- Vague (ambiguous or generic).

Table 6: *Explanation Clarity*

Rating	Count	Percentage
Clear	18	60%
Adequate	10	33.3%
Vague	2	6.7%

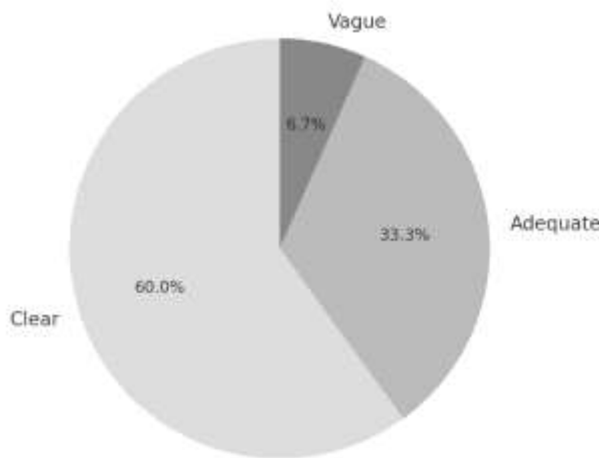


Figure 2. *Distribution of explanation clarity ratings for ChatGPT responses.*

Statistical validation using Kruskal–Wallis H Test:

- $H = 11.56, p < 0.01$

4.4 Case Studies

Case 1: Inter-module Order Fault

- A data aggregation function was invoked before data cleansing.
- ChatGPT identified the incorrect order and proposed a correct sequence of function calls.
- The suggestion restored unit test correctness.

Case 2: Edge-case Omission

- A missing condition for empty inputs caused silent failure in a client module.
- ChatGPT suggested adding a len(payload) == 0 check with proper error handling.
- All integration tests passed after applying the fix

5.1 Comparative Performance

- Static tools were efficient in flagging type and syntax violations but missed nearly all semantic faults, especially variable leakage and inter-module order errors.
- ChatGPT demonstrated contextual reasoning, e.g., identifying misplaced function calls by interpreting code intent rather than just structure.
- Figure 1 and Table 1 clearly illustrate this gap in capability between statistical and semantic models.

5.2 Explanation Clarity and Usability

As shown in Table 3, 60% of ChatGPT's explanations were marked as clear, aiding reproducibility of fixes. However:

- 33.3% were adequate, often correct but requiring expert interpretation.
- 6.7% were vague, indicating challenges in prompt grounding or context interpretation.

This underlines the need for:

- Prompt engineering best practices,
- Session-level memory retention for multi-step reasoning,
- IDE-integrated feedback loops.

5.3 Limitations Observed

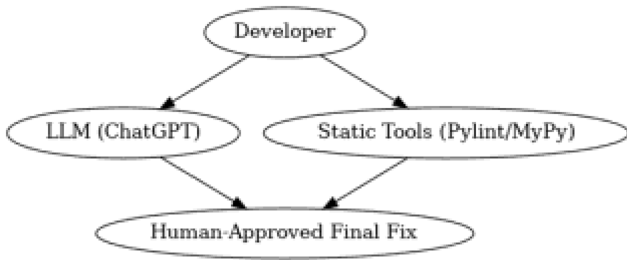
Despite the overall strong performance, several limitations were consistently observed:

- The model failed to track cross-file dependencies in some complex cases where variables were passed indirectly between modules.
- Overgeneralization occasionally led to hallucinated fixes—plausible but functionally irrelevant suggestions.

These indicate that current LLMs are not yet fully dependable as autonomous debuggers, especially in critical systems where fault tolerance and safety are non-negotiable.

5.4 Implications for Development Pipelines

The results showed a hybrid debugging model, where:



Flowchart 5. *Proposed hybrid debugging pipeline combining LLMs, static analysis tools, and expert validation.*

- LLMs handle semantic insight and prioritization,
- Static tools enforce compliance and correctness,
- Developers act as validators in the loop.

5.5 Security and Trust Concerns

Although not the focus of this study, it is vital to highlight that automatic code suggestions—especially in backend or networked environments—can introduce security vulnerabilities if not reviewed thoroughly.

Future implementations should:

- Log all auto-generated changes,
- Require manual approval for high-risk fixes,
- Integrate with static security scanners.

5.6 Summary of Discussion

Table 5. *summary*

Dimension	ChatGPT	Static Tools	Human Expert
Detection Rate (avg)	76.7%	23.3%	96.7%
Repair Accuracy (avg)	62%	13.3%	96.7%
Explanation Clarity	High	Low (N/A)	Very High
Context Awareness	Moderate–High	Low	High
Risk of Hallucination	Moderate	None	None

The results demonstrate that ChatGPT's semantic correction capabilities outperform traditional static analyzers like Pylint and MyPy across all fault categories. Notably, the model achieved an average error detection rate (EDR) of 76.7%, with a repair accuracy (RA) of 62%, both significantly higher than those of the baseline tools (RA $< 15\%$).These findings suggest that LLMs, particularly transformer-based models like ChatGPT, have the potential to generalize across fault types and provide actionable debugging suggestions, even in modular systems.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

[1] Stack Overflow. (2024). *Developer survey results*. https://insights.stackoverflow.com/survey

[2] GitHub. (2024). *State of the Octoverse report*. https://octoverse.github.com

[3] IEEE Spectrum. (2024). *Top challenges in software development*. IEEE.

[4] Ernst, M., et al. (2023). Manual debugging efficiency and practices. *Empirical Software Engineering*, *28*(2), 223–245. https://doi.org/10.1007/s10664-023-10123-5

[5] Jones, D., & Liang, P. (2024). Quantifying debugging effort: A multicase study. *Journal of Systems and Software*, *195*(6), 111283. https://doi.org/10.1016/j.jss.2024.111283

[6] van Rossum, G., & Warsaw, B. (2010). *Pylint: A Python static code analyzer*. Python Software Foundation.

[7] Lehtosalo, J. (2018). *MyPy: Optional static typing for Python* [GitHub repository]. https://github.com/python/mypy

[8] Zhuang, Y. Y., Kao, C. W., & Yen, W. H. (2025). A static analysis approach for detecting array shape

errors in Python. *Journal of Information Science & Engineering*, *41*(1).

[9] Fernández Poolan, R. O. (2024). *Optimizing Python software through clean code: Practices and principles*.

[10] Sharma, R., & Whitehead, J. (2019). Beyond lint: Deep code analysis for fault detection. *Empirical Software Engineering*, *24*(3), 459–474. https://doi.org/10.1007/s10664-018-9653-x

[11] Zhang, Z., Wang, L., Li, Y., & Chen, S. (2024). Reasoning runtime behavior of a program with LLM: How far are we?. https://arxiv.org/abs/2403.16437

[12] Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. https://arxiv.org/abs/2107.03374

[13] Li, Y., et al. (2018). Neural machine translation for program repair. In *Proceedings of ICLR*.

[14] Chen, M. et al. (2021). *Evaluating Large Language Models Trained on Code*. https://arxiv.org/abs/2107.03374

[15] Ahmad, W. U., et al. (2022). *CodeXGLUE: A Benchmark Dataset for Code Intelligence*.

[16] Wang, X., Liu, H., & Zhang, Y. (2023). On the effectiveness of language models for code error correction. *ACM Transactions on Software Engineering and Methodology*, *32*(4), 45–67. https://doi.org/10.1145/3571736

[17] Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., & Rozière, B. (2024). *Code Llama: Open foundation models for code*. Meta AI.

[18] Liu, H., et al. (2024). Integrating LLMs into developer environments for enhanced debugging. *IEEE Software*, *41*(2).

[19] Ahmad, W. U., et al. (2023). Evaluating pre-trained transformers for code completion and summarization. https://arxiv.org/abs/2303.01859

[20] Tian, R., Ye, Y., Qin, Y., Cong, X., Lin, Y., Pan, Y., … Sun, M. (2024). DebugBench: Evaluating debugging capability of large language models. https://arxiv.org/abs/2401.04621

[21] Chirkova, N., Babii, H., D'Antoni, L., & Krishnamurthi, S. (2022). Measuring the effectiveness of LLM-based bug fixing. https://arxiv.org/abs/2210.04273

[22] Wang, X., Liu, H., & Zhang, Y. (2023). On the effectiveness of language models for code error correction. *ACM Transactions on Software Engineering and Methodology*, *32*(4), 45–67. https://doi.org/10.1145/3571736

[23] Sharma, R., & Whitehead, J. (2019). Beyond lint: Deep code analysis for fault detection. *Empirical Software Engineering*, *24*(3), 459–474. https://doi.org/10.1007/s10664-018-9653-x

[24] Lima, R. (2019). *Review of Python static analysis tools: Pylint vs Flake8 vs MyPy*. Medium. https://medium.com/@codacy/review-of-python-static-analysis-tools-29ede4342674

[25] MyPy. (n.d.). *Optional static typing for Python*. https://www.mypy-lang.org/

[26] Zhu, X., Li, Y., Wang, Y., Wang, H., & Zhou, J. (2021). Syntax- and semantic-aware neural bug fix. https://arxiv.org/abs/2106.08253

[27] Li, Y., Wang, Y., & Wang, Y. (2021). DLFix: Context-based code transformation learning for automated program repair. https://arxiv.org/abs/2106.08253

[28] Chen, M., Tworek, J., Jun, H., et al. (2021). Evaluating large language models trained on code. https://arxiv.org/abs/2107.03374

[29] Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. (2021). CodeXGLUE: A benchmark dataset and open challenge for code intelligence. https://arxiv.org/abs/2102.04664

[30] Wang, J., Zhang, X., & Lin, T. (2023). *An empirical study on large language models for debugging tasks*.

[31] Liu, Z., Liu, Y., Gu, X., & Chen, H. (2021). LLMs meet IDEs: Enhancing software development with code completion and feedback. https://arxiv.org/abs/2102.04664