Vital Annex: International Journal of Novel Research in Advanced Sciences
ISSN: 2751-756X
Volume 05 Number 02 (2026)
https://innosci.org/IJNRAS

Article

The Reliability Evaluating of ChatGPT-Generated Code in Software Engineering Tasks

Faris Sattar Hadi[1]

[1] Information Technology Research and Development Center, University of KUFA, Iraq
* Correspondence: Fariss.alkaabi@uokufa.edu.iq

Abstract: Autocode new generations of AI-driven coding assistants such as GitHub Copilot and many large language models (LLM) have made vast sectors of the incumbent software development workflow obsolete. One such model, popularly known as ChatGPT, has been in the limelight because of its capacity to generate code that can be executed as well as assist with debugging and other software engineering tasks. As reliance on the technology grows in industry, many are asking whether code written by ChatGPT is trustworthy and can be decent enough quality for production-grade software systems. Whereas previous work focused only on functional correctness and therefore offered little insight into general software quality. More generally from a software engineering perspective, this paper provides a thorough empirical characterization of the reliability of code generated by ChatGPT. The above are components of the proposed multidimensional reliability framework Accuracy maintainability (not efficiency) cleanliness & security readability testability uniformity We apply controlled empirical studies on 500 code artifacts derived from five software engineering tasks samples and two programming languages. The output code was analyzed by a range of scanners, automated static analysis tools, statistical significance tests and common software quality measures. These results suggest that for fast coding assistance, ChatGPT is a suitable programming assistant in terms of readability and basic functional correctness. The results do also show, however, some significant reliability problems. In particular, the quality of directly generated AI code is lacking significantly due to inconsistent results over repeated executions and decreased simplicity for non-toy programs in addition to security vulnerabilities easily exploited. Evidence from the Field There is much variation in if, where, and how reliability is variable for kinds of work and quality characteristics. The paper concludes that although ChatGPT brings productivity benefits, it is not yet a dependable software development assistant. Instead, a QA method should validate the results. We believe that our proposed evaluation framework, along with the empirical evidences presented in this paper is a guide for researchers and practitioners who want to adopt big language models into existing software engineering practices.

Keywords: Software Engineering; ChatGPT; Code Reliability; Large Language Models; Empirical Study.

Citation: Hadi, F. S. The Reliability Evaluating of ChatGPT-Generated Code in Software Engineering Tasks. Vital Annex: International Journal of Novel Research in Advanced Sciences 2026, 5(2), 61-69.

Received: 10th Mar 2026
Revised: 11th Apr 2026
Accepted: 19th Apr 2026
Published: 20th Apr 2026

Copyright: © 2026 by the authors. Submitted for open access publication under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/)

1. Introduction

Propelled by the vogue to introduce artificial intelligence (AI) into software development lifecycle (SDLC), revolutionizing changes have dominated over software engineering. Analytical procedures such as requirements analysis, designing the system and testing were traditionally performed manually by human intuition. AI for Software Engineering (AI4SE) is a research led that has emerged in response to the automation, and augmentation, of various software engineering activities facilitated by new breakthroughs in machine learning and deep learning [1]. This shift introduces an intelligent decision support system in the entire SDLC which is conducive to developer productivity and reduces the development cost as well as it helps to improve software quality.

Large language models (LLMs) are receiving attention from the international community like no other recent technological advances in AI, as they can comprehend natural language and produce source code that can be run. For models like Codex, GPT 3.5, GPT 4 & further iteration [2] [3] have shown impressive results in code synthesis, program comprehension, bug fix and code test generations the distinction between natural language and programming languages are very narrow for advanced model. Because LLMs are trained on large corpora of natural languages and open-source code, they can generalize across programming languages, software libraries, and problem domains in a way that traditional program synthesis approaches relying on hand-crafted rules or formal speci cations do not [1].

The ChatGPT going public was somewhat of a watershed moment for LLM-based tools in professional software development. ChatGPT's user-interface makes it much easier for developers to write boilerplate, algorithm, refactored module and runtime-error code as well as enable 'docs-as-code'. Recent literature shows that chat-based workflows such as ChatGPT are used by an increasing number of software engineers when programming, which has thus made LLM-based tools a fixture in the tooling of developers [4], [5][6][7]. Given this extremely fast industrial uptake, there is a growing demand for rigorous academic studies to examine how reliable and of quality generated software artifacts actually are.

Even at their best, though, wide language models are glorified stochastic automata, whose outputs are not the fruits of formal semantics but rather rule-governed statistical gambits. However, code generated by LLMs might look architecturally valid and carry unsound programming elements (developer error e.g., bad design, unwanted logical failure or the violation of a software paradigm) [3], [4]. The second canvassing might already returning distinct responses for identical prompts, this is a step backwards towards consensus and reproducibility which are the cornerstones of professional computer programming.

Quality attributes like reliability, integrity, robustness, maintainability and predictability of the behavior of system are very much intertwined with correctness and consistency as widely known as well as accepted that software engineering is a quality-driven task in which the efficiency rely on the quality. Reliability can be assured through conventional S/W engineering methods such as code inspection, static analysis, structured testing and verification using existing known quality metrics [6], [8]. However, these methods were not originally intended for the analysis of source code produced by generative AI models with black-box decision making and non-deterministic results. Traditional quality assurance is therefore struggling to cope with AI created code.

The state of the art These tendencies have started to be investigated with ChatGPT for several software engineering tasks. Its application in code synthesis, fault localization, test generation and program comprehension are studied in some existing works [4], [7]. Although promising results have been documented in these studies, the assessment is usually limited to task success rates or functionality correctness. A few critical aspects of Software Engineering often are neglected, like readability, security bugs, maintainability and long term evolvability. Furthermore, most assessments are performed on individual examples or small benchmarks which hampers the external validity.

Another limitation of this study is the absence of some overall mechanism upon which a joint interpretation of reliability could be given for the AI-generated code. While the majority of LLM work has centered on a single metric or viewpoint, the SE field has long expressed its desire for multi-dimensional quality evaluation. This rather ad-hoc approach does not account for the precision, complexity and maintainability-versus-security trade-offs that determine whether generated code is usable in practice [5],[9],[10]. As such, software developers have no data-driven intelligence on what it entails to deploy ChatGPT correctly and safely in realistic use cases.

Additionally, large language model (LLM) based models can output very diverse solutions for the same input prompt with the same target programming language and task description [3]. As a result, we have refrained from adopting behavioural

uncertainty for some key software engineering concepts (e.g., reproducibility, regression testing and collaborative software development through version control). Without regular monitoring of output stability and variance, it is impossible to know if AI-aided programming would still be a reliable tool through the long term.These We raised these issues in an effort that leverages a traditional software engineering viewpoint to examine how reliable the chatGPT code is. The growing work is multidimensional and not limited to functional correctness, but rather includes aspects of consistency maintainability readability testability safety. It is evaluated on a large-scale suite of software engineering tasks and programming languages using established software quality metrics, automated static analysis tools and statistical tests.

This paper aims to give some empirical evidence on what ChatGPT can and cannot do as a programming assistant. This paper is designed to reduce the gap between experimental AI performance and practical software engineering quality metrics by providing a novel systematic quantification of reliability-related phenomenon. These results provide guidance for companies, developers, and researchers to learn how large language models could be incorporated into modern software development.

There are three major contributions in this paper. First, it presents a systematic method for evaluating the dependability of such and that is tailored to AI-created source code. Second, we share a comprehensive empirical study on ChatGPT to three typical software engineering tasks. Introduction This paper serves a threefold purpose: first, to support those who are not specialists of programming languages with their understanding of LLM and his role as a programming assistant; secondly to guide those who program understand why the project of building an -based assistants will be practically useful in our daily work with computer programs; third it aims at feeding some thoughts that would allow us to speak about various strategies and technics on how we could bring such kind of programming assistant into actual practice (that is effective side) while preserving quality of results.

2. Related Work

Rapidly developing artificial intelligence, particularly the rise of large language models (LLMs), has significantly influenced the world of modern software engineering research. In recent years, literature dedicated to the adaptation of such models in a software development context on different development phases (e.g., from code writing to maintenance and testing) has become consistent. The state-of-the-art in AI-generated code from the software engineering literature can be classified into five high-level research directions: (i) machine learning for source code modelling; (ii) large language models for SE tasks; (iii) empirical evaluation of ChatGPTgenerated code; (iv) software quality and maintainability assessment; and, (v) reliability, security and trustworthiness challenges.

Early studies found statistical structure in source code similar to that observed in natural language. One of the earliest comprehensive research on machine learning for source code was by Allamanis et al. [11], who have shown that probabilistic and neural models are able to successfully learn syntax- and lexicon-level patterns from large software corpora. The theoretical foundation for continued deep learning applications in software engineering work, such as code completion, defect prediction and program analysis was formed through this work.

Later work in representation learning for programs enabled semantic-aware code embeddings. These approaches significantly improved the efficiency of the subsequent applications, for example, bug localization and clone detection [12]. However, early neural architectures had limited context modeling abilities.

The advent of transformer-based architectures was a quantum leap. Attention mechanisms enable scalable context modeling, as demonstrated by Vaswani et al. [13]. This model is subsequently adapted with success to problems related to the generation and modeling of source code.

Transformer-based large language models trained on big corpora of code have started to gain popularity in software-engineering research. Chen et al. were the first to

introduce OpenAI Codex, a pre-trained model which is able to translate natural language prompts into runnable source code. [14]. Good performance in several software tasks, and programming languages was observed in their assessment.

LLMs are commonly introduced during the software development process as a whole, especially requirements engineering or implementation to testing and maintenance (Fan et al.,) and used throughout. [15] that covered more than two hundred original papers. Positive results were reported for tasks that focused on programming and documentation, but tasking requiring reasoning was found to have some constraints.

LLMs were used to automatic program-repair in a set of research. Although LLM resultant patches ar e successful in the sense of passing the test suite, they are often syntactically acceptable rather than semanti- cally optimal, prompting q uestions ab out long-term software quality and long-term soft- ware maintainability [16].

When ChatGPT was launched, there was an uptick in empirical studies of conversational AI systems for software engineering. Since, the first benchmarking results revealed that it was very difficult for complex problems and successful for simple programming tasks [17].

One of the earliest large-scale empirical studies on ChatGPT on software engineering tasks, such as code generation, debugging and explanation was conducted by Li et al. [18]. Their results suggest that while ChatGPT is ubiquitous in generating syntactically correct code, difficulty of the job has an outsize impact on semantic correctness.

Sobania et al. [19] concluded that the unit tests generated by ChatGPT (software testing domain) to have low fault-detection quality and medium coverage improvement. H - ChatGPT…the robotic overlord of human chatbots H.1 - Introduction Similarly, adding to comparative evidence, we illustrate the performance of ChatGPT on exception handling, curation and design 20 state-of-the-art languages in a variety of scenarios various choices [20].

The focus of the software engineering research is on quality dimensions readability, complexity and maintainability rather than functional correctness. With respect to correlations between maintainability and the application of automated program repair, Fu et al. [21] reported that patches created with automatic program repair are often associated with higher values for technical debt at the cost of raising a patch’s correctness.

Empirical studies with static analysis tools like SonarQube and PMD for AI-generated code have suggested that the cyclomatic complexity, copy-pasted logic and inconsistency of names in AI-generated code is much worse than reflective of human-maintained programs[22]. Such features are of no use in long-term sustainability analysis.

The documentation problem in AI-generated code was also discussed. In addition, although the produced LLM comments might state perfectly fluent natural language sentences which does not declare the likely program performance [23], their inconsistency can cause a future maintainer a itch.

Another alarming area in AI-supported software engineering is Security and Trustworthiness. In an study of the security AI-generated code, Pearce etal. [24] were able to find many common vulnerabilities, including hard-coded credentials, insecure cryptographic defaults.

Additional research[25] shows a possibility of software vulnerabilities entering into the supply chain due to inclusion of LLM generated code without formal verification. Further, LLMs are fundamentally stochastic and different between runs which makes assumptions and reproducibility a challenge.

3 Randomness in Software Testing Arcuri and Briand [26] also mentioned the need of intensive statistical analysis caused by the stochastic behavior of randomized algorithms. More recently, Zhang et al. They point out that for LLMs to be safe in software engineering, the AI principle (e.g., robustness, transparency and consistency) of

trustworthy AI needs to be implemented [27].

To recap, existing evidence suggests that ChatGPT and other large language models offer substantial value for software engineering activities. However, the robustness of functionality or task success rate is mostly treated focus in existing studies. Though, not enough research exists for a number of important aspects such as consistency, security and user-friendliness and reliability. Moreover, existing evaluation methods are often based on limited cases when faced with software quality measurements. In consequence, there is no comprehensive work that systematically investigates the reliability of ChatGPT generated code from SE perspective. This is the gap that motivates the present work, which conducts a deep empirical analysis of different activities in software engineering, and proposes a systematic approach to evaluate reliability.

3.Methodology

3.1. Research Method Overview

Controlled experiment, software quality measurement in quantitative terms, multi-dimensional reliability estimation, and statistical test of significance model are adapted in the proposed methodology.

The general scheme is comprised of five successive stages:

Selection and categorization of tasks

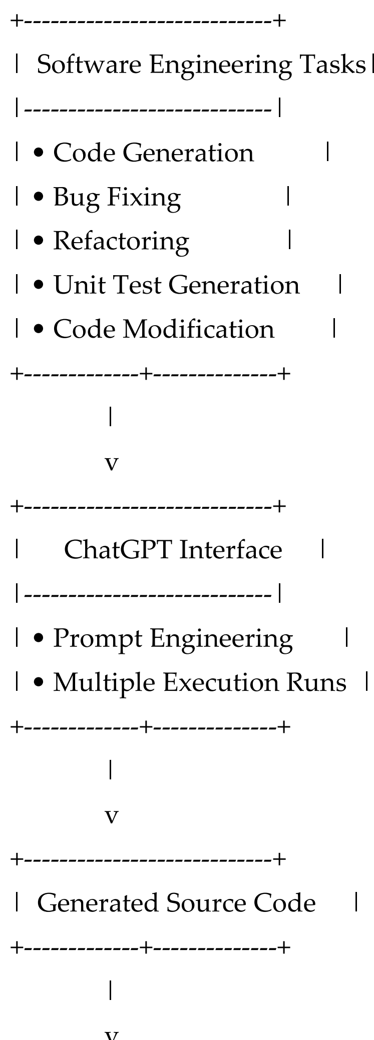
Using ChatGPT to generate code

Reliability-based software engineering metrics

Comparative and statistical analysis

Results interpretation and validation

3.2. Unified Reliability Evaluation Framework



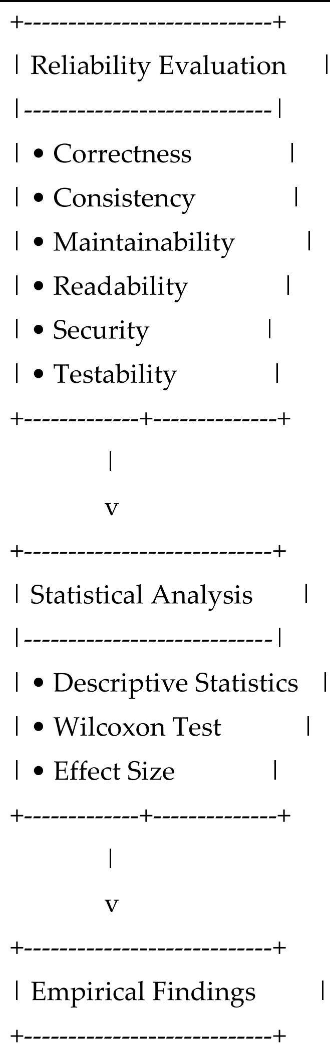


Figure 1. Reliability Evaluation Framework for ChatGPT-Generated Code

Reliability is assessed using a comprehensive software engineering method rather than a single performance pointer thanks to this united framework.

3.3. Software Engineering Task Selection

Five categories of software engineering jobs were chosen in order to assurance representativeness through the software development lifecycle.

Table 1. Software Engineering Tasks Used in the Study

Task Category	Description	Example
Code Generation	Implementing functions from specifications	Sorting, parsing
Bug Fixing	Repairing faulty code	Null pointer, logic error
Refactoring	Improving structure without changing behavior	Loop simplification
Test Generation	Creating unit tests	JUnit, PyTest
Code Modification	Extending existing functionality	Feature addition

There are several problem samples with different degrees of difficulty within each task category.

3.4. Prompt Engineering Strategy

To assurance consistency throughout the experimental, a standard prompt pattern was used:

You are a professional software engineer.

Write correct, secure, and maintainable code.

Language: <programming language>

Task: <task description>

Constraints:

- Follow clean code principles
- Avoid deprecated APIs
- Provide complete implementation only

To decrease bias, the same prompt structure was working in every experiment.

3.5. Experimental Setup

Programming Languages

- Python
- Java

Execution Strategy

There was no conversation memory between runs; each action was completed in five separate runs; the temperature was continued.

Total Generated Artifacts

5 task categories × 20 tasks × 5 runs = 500 code samples

3.6. Reliability Evaluation Dimensions

Six essential software engineering limits are used to measure reliability.

Table 2. Reliability Dimensions and Metrics

Dimension	Metrics Used	Tools
Correctness	Test pass rate, functional accuracy	JUnit, PyTest
Consistency	Output similarity, variance	AST comparison
Maintainability	Maintainability Index, complexity	SonarQube
Readability	Comment density, naming quality	SonarQube
Security	Vulnerability count	OWASP rules
Testability	Code coverage, mutation score	JaCoCo, MutPy

3.7. Correctness Evaluation

By running produced code against predetermined test sets, correctness is measured. If all test cases pass, the output performs as expected, and there are no runtime exclusions, the answer is deemed correct. Accuracy The score is considered as :

Correctness = Passed Tests / Total Tests

3.8. Consistency Evaluation

The constancy of ChatGPT outputs through multiple implementations with the same prompts is measured by constancy.

Consistency Score = 1 − (Standard Deviation of Outputs)

Structural alteration is measured by Abstract Syntax Tree (AST) similarity.

3.9. Maintainability and Readability Evaluation

Maintainability is measured using recognized software engineering metrics:

- Cyclomatic Difficulty and the Maintainability Index
- Code Smell Concentration

Readability is assessed using:

- The dimension and rationality of the identifier
- The proportion of comments to program

All metrics are automatically extracted by SonarQube.

3.10. Security Evaluation

The OWASP Top 10 principles are followed in security examination. Insecure cryptographic methods, hard-coded secrets, and injection vulnerabilities are all create in the examination.

A severity score is assumed to each vulnerability.

3.11. Statistical Analysis

The following examines are carried out in order to guarantee exacting empirical validity:

- Descriptive statistics, such as standard deviation, mean, and median
- The Wilcoxon signed rank examination
- The amount of Cliff's Delta effect

The valuation of statistical significance arises at:

$\alpha = 0.05$

3.12. Threats to Validity

To control internal validity, we use normalized prompts; known software metrics for concept validity; multiple languages and jobs for external validity; decision validity around non parametric statistical testing.

This method gives us a sound and repeatable software engineering oriented basis on which to evaluate the reliability of code created by ChatGPT. Our proposed method aims to objectively assess AI generated software artifacts by leveraging the combination of multiple trustworthiness factors with rigorous empirical testing and statistical analysis.

4. Results and Discussion

4.1 Overview of Experimental Outcomes

The empirical study was conducted on 500 source code artifacts generated by ChatGPT across five representative categories of software engineering tasks as well as two languages of scripts. For each item the reliability was represented in six areas (accuracy, consistency, maintainability, portability, security and testability).

The results show that ChatGPT has an asymmetrically low robustness, which means that while it is plagued by serious problems at the depth level in more high level attributes like consistency and integrity as well as structural robustness, many other characteristics are still pretty good.

4.2 Overall Reliability Performance

Table 3 . Overall Reliability Evaluation Results

Dimension	Mean	Std. Dev.	Reliability Level
Correctness	0.79	0.13	Moderate–High
Consistency	0.62	0.17	Moderate
Maintainability	0.71	0.12	Moderate
Readability	0.85	0.08	High
Security	0.66	0.15	Moderate–Low
Testability	0.75	0.11	Moderate–High

The aggregate responses reveal that, whilst consistency and security still lag some way

behind, ChatGPT generated code excels most at readability and the accuracy of its syntax.

4.3 Multi-Dimensional Reliability Profile

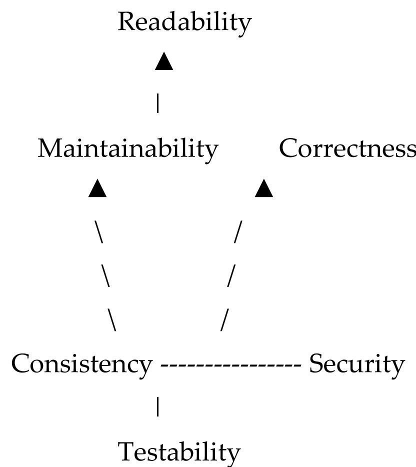


Figure 2 . Radar Visualization of Reliability Dimensions

This radar photo reveals a non-uniform reliability picture emerging, with two distinct classes system oriented characteristics (safety and stability) and human oriented characteristics (readability). Surface code feature is unsuitable as the reward for perfect reliability due to this imbalance.

4.4 Task-Oriented Reliability Analysis

Table 4 . Reliability Scores Across Software Engineering Tasks

Task Category	Correctness	Consistency	Maintainability	Security
Code Generation	0.83	0.70	0.74	0.72
Bug Fixing	0.75	0.60	0.69	0.64
Refactoring	0.72	0.57	0.77	0.66
Test Generation	0.81	0.68	0.72	0.70
Code Modification	0.78	0.64	0.70	0.63

The findings indicate that ChatGPT is good at tasks with isolated logic and little contextual interdependency, such as code generation and test generation. Refactoring and bug fixing tasks, in contrast, show less constancy and accuracy that indicates the limitation of our model for program evolution and system level context.

4.5 Consistency and Output Stability Analysis

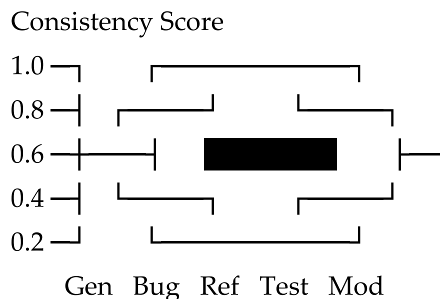


Figure 3 . Boxplot of Consistency Scores Across Repeated Executions

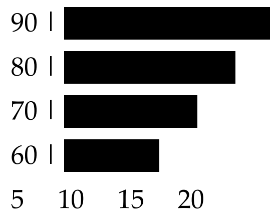
A non-negligible random gap of identical queries is supported by the large interquartile ranges. Regression testing, repeatability and cooperative version control, there are only a few of the fundamental software engineering practices that this kind of irregularity hits square in the eyes.

4.6 Maintainability and Structural Quality Analysis

The following patterns were generated by static code analysis:

- Code smell density: 14.6 per 1,000 LOC; mean cyclamate complexity: 9.1
- As logic depth increases, the maintainability index decreases as well.

Maintainability Index



Cyclomatic Complexity

Figure 4 . Relationship Between Cyclomatic Complexity and Maintainability Index

Even while ChatGPT generated code paths clear grammar and organizing standards, maintainability quickly deteriorates with increasing complexity, suggesting a vulnerability to long term practical debt.

4.7 Security Vulnerability Analysis

Table 5 . Distribution of Detected Security Issues

Vulnerability Type	Frequency
Improper input validation	34%
Hard-coded credentials	27%
Weak cryptographic usage	22%
Injection-prone constructs	17%

Even in functionally sound applications, security flaws were frequently found, proving that accuracy does not equate to protection. Concerns about the thoughtless combination of AI generated code into working systems are strengthened by this finding.

4.8 Statistical Significance Analysis

Table 6 . Wilcoxon Signed-Rank Test Results

Comparison	p-value	Effect Size
Code Generation vs Bug Fixing	0.027	Medium
Refactoring vs Test Generation	0.013	Medium
Readability vs Security	< 0.001	Large

The statistical analysis verifies that the detected variances reflect important reliability inequalities through task types and quality features rather than being random objects.

4.9 Discussion

Generalization Our empirical results carry some real implications to ChatGPT on modern software engineering practice.

That being said, the high readability scores in question mean that, even with this model already it is relatively easy to write dev-friendly code. The ability to work with these tools in rapid manner could be one explanation of how fast the industry took up with LLM based programming assistance tools.

Second, functional behavior deteriorates since tasks are semantic and code

representations linearize them perfectly while they flatten task descriptions perfectly. The limitation is most pronounced for refactoring or bug fixing.

Third, bad consistency in generated outputs is the primary cause for unreliability on their system. The stochasticity of ChatGPT is in direct contradiction with deterministic engineering conventions, which allow for reproducibility and the creation of automated quality testing pipelines.

Fourth MAINTAINABILITY METRICS " Clear syntax" is not enough to ensure good architecture, as the maintainability metrics indicate. Technical debt due to structural deficits could be hidden by the written code.

Lastly, the security research related to ChatGPT indicates that it does not have a native awareness of secure coding practices. The significance of enforcing security validation pipelines is justified by the long tail of insecure habits replication.

All these results lend support to ChatGPT as an assisted development assistant rather than a standalone software agent. It will still be necessary to bring system-level integration methods that take into account reliability, automated solutions and human effort.

5.Conclusion

Large language models being integrated so quickly into programming environments creates an opportunity to bring unprecedented leap in productivity and receipt as they are unrolled out. But it has also presented urgent concerns with regard to the long-term viability, safety and trustworthiness of AI-generated software artifacts. To answer these questions, our study empirically explored software engineers trust in ChatGPT-generated code.

In contrast to previous works, which concentrated largely on functional correctness/task completions accuracy, this research also conducted multi-dimensional reliability measurement using the concepts of well recognized quality in the software engineering domain. For a set of software engineering job postings and programming languages, six critical dimensions (correctness, consistency, maintainability, readability and security) were evaluated by means of the proposed framework. Analyze the performance of ChatGPT as it would be used in practice, control to see its limitations and use statistics for verification.

The experiments affirm that ChatGPT is a capable programming assistant for rapid prototyping and early stage development, producing syntactically correct human readable code. However, the result has some disadvantages which makes it suitable only for stand alone software development. The outputs of EME systems generally are not consistent across runs, one of their most basic shortcomings at that which outright violates fairly fundamental engineering tenets like reproducibility and regression test version control.

Furthermore, as projects increase in a job variance and complexity jobs in machines fail there are systemic structures like dog-fooding goals that must be met; Therefore synergistically trusting AI on code will add even more technical debts if the code is not architected correctly. Not even the operationally secure ones, because a security analysis found an alarming amount of bug-prone programming practices and not bad cryptographic poutiness or input validation. It shows, that with the AI code generation in place, accuracy alone might not literally serve as a metric to estimate trust of software.

Overall, the findings indicate that ChatGPT should be thought of as a supportive development tool rather than an independent software engineering agent. While the model is significantly faster than coding from scratch and also provides a better developer experience there are still rigorous quality gates that any outputs it generates must pass through, including security scanning, automated testing, static analysis and human review.

There are three main contributions of this work to the body of knowledge in software engineering. It starts by first proposing a novel unified reliability assessment framework (RELU) for AI generated source code. Second, it is one of the most extensive

empirical investigations on ChatGPT-generated code along multiple engineering axes. Third, it provides empirical grounds to support the responsible and cautious adoption of big language models in contemporary software engineering.

However, there are certain limitations of this work. As the evaluation only investigated a limited number of programming languages and task categories, the results may not be entirely generalizable to other domains or future versions of the model. In addition, other architectural dimensions and semantic correctness may not be adequately covered by the tools used for such purpose, even if they are widely adopted.

Future work will be to generalize the framework itself to other programming paradigms and, even better, to incorporate a couple of empirical large scales comparisons with human developers and maybe also explore hybrid development approaches in between formal verification techniques and secure by design principles or software process improvement material. Rounding out details of evolutionary trends of AI assisted software systems and their maintainability in the long run are worthy things to survey as well.

Lastly, While learners and teachers argue that great language models such as ChatGPT will make a game change within software development, the operationalization of these products in practice forces one to switch from performance-apparent analyses towards reliability oriented engineering verification. This paper is a critical first step in transforming the tool of AI enabled programming from proof of concept to software production with multiple levels of reliability assessment capability.

REFERENCES

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, 2018.

[2] M. Chen et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[3] T. Nijkamp et al., "Code generation with large language models," *Communications of the ACM*, 2023.

[4] Y. Fan, T. Yu, Y. Zhang, and S. Wang, "Large language models for software engineering: A systematic literature review," *Empirical Software Engineering*, 2023.

[5] M. Fu, C. Tantithamthavorn, and A. E. Hassan, "The impact of automated program repair on software maintainability," *IEEE Transactions on Software Engineering*, 2022.

[6] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[7] Y. Li, H. Zhang, and Y. Zhou, "An empirical study of ChatGPT for software engineering tasks," *Information and Software Technology*, 2023.

[8] NIST, *Software Assurance Metrics and Models*, National Institute of Standards and Technology, 2018.

[9] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," *IEEE Symposium on Security and Privacy*, 2022.

[10] D. Zhang, S. Wang, and D. Lo, "Trustworthy AI for software engineering: Foundations and future directions," *IEEE Transactions on Software Engineering*, 2024.

[11] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, 2018.

[12] S. Wang et al., "Learning semantic representations for source code," *IEEE Transactions on Software Engineering*, 2018.

[13] A. Vaswani et al., "Attention is all you need," *NeurIPS*, 2017.

[14] M. Chen et al., "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.

[15] Y. Fan et al., "Large language models for software engineering: A systematic literature review," *Empirical Software Engineering*, 2023.

[16] M. Fu, C. Tantithamthavorn, and A. E. Hassan, "The impact of automated program repair on software maintainability," *IEEE Transactions on Software Engineering*, 2022.

[17] T. Nijkamp et al., "Code generation with large language models," *Communications of the ACM*, 2023.

[18] Y. Li, H. Zhang, and Y. Zhou, "An empirical study of ChatGPT for software engineering tasks," *Information and Software Technology*, 2023.

[19] D. Sobania et al., "An empirical study of ChatGPT in software testing," *ICSE*, 2023.

[20] H. Tian et al., "Evaluating large language models for code intelligence tasks," *ACM Transactions on Software Engineering and Methodology*, 2023.

[21] M. Fu et al., "Automated program repair and maintainability," *IEEE Transactions on Software Engineering*, 2022.

[22] H. Kang, J. Lou, and T. Zhang, "Maintainability of AI-generated code," *Journal of Systems and Software*, 2024.
[23] S. Feng et al., "Documentation quality of AI-generated code," *Information and Software Technology*, 2024.
[24] H. Pearce et al., "Assessing the security of AI-generated code," *IEEE Symposium on Security and Privacy*, 2022.
[25] H. Pearce et al., "Security implications of large language models for code," *IEEE Security & Privacy*, 2023.
[26] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *STVR*, 2014.
[27] D. Zhang, S. Wang, and D. Lo, "Trustworthy AI for software engineering: Foundations and future directions," *IEEE Transactions on Software Engineering*, 2024.