



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Beyond Benchmarks: Assessing LLM Code Generation Reliability and Developer Trust Calibration for Real-World Software Engineering Tasks

Submitted By:

MD Azizul Khan

Student ID: 2514708

Module Code: ACFA7004

Submitted to:

Seena Joseph

Research Methods and Professional Development

TABLE OF CONTENTS

1	Introduction	3
1.1	Context and Background	3
1.2	Research Problem.....	4
1.3	Research Objectives	4
1.4	Significance of the Research	5
2	Literature Review	6
2.1	Scope and Purpose of the Review	6
2.2	Evaluation Frameworks and Benchmarks for LLM Code Generation	6
2.3	Reliability, Non-Determinism, and Code Quality in Practice.....	7
2.4	Small-Scale and Practical Software Engineering Task Evaluation.....	7
2.5	Trust Calibration in AI-Assisted Coding Contexts	8
2.6	Theoretical and Technical Frameworks.	9
3	Research Design and Methodology	9
3.1	Research Design.....	9
3.2	Research Methods	12
3.3	Research Materials	13
3.3.1	Data Sources and Collection.....	13
3.3.2	Evaluation Metrics	13
3.3.3	Software, Libraries and Frameworks.	14
3.3.4	Hardware Requirements.....	15
4	Expected Outcomes and Limitations	15
4.1	Anticipated Results	15
4.2	Implications.....	16
4.3	Limitations	16
5	Project Timeline.....	16
	References.....	18

1 INTRODUCTION

1.1 Context and Background

Integrating large language models (LLMs) into software engineering processes is one of the most important changes in computing in recent years. Codex, ChatGpt, Claude and GitHub Copilot are examples of models that have demonstrated the capability of producing syntactically plausible code when prompted with natural language prompts [1]. AI code-generation systems have advanced rapidly, moving from early neural models to today's transformer-based architectures in only a short time, with modern LLMs being scaled out to billions of parameters trained on large open-source code repositories [2]. A seminal step in the context of evaluating functional correctness was the introduction of Codex by Chen et al. [3] which established a benchmark of HumanEval and the pass@k metric as the key standard of assessing functional correctness, which remains a dominant paradigm of evaluation in the literature.

1.2 Research Problem

The practical utility of code generation aided by LLM is spread over a variety of dimensions of software productivity. Anand et al. [4] demonstrated that these tools can speed up the logic creation, boilerplate effort is reduced and inline documentation is provided, making these tools increasingly essential members of the software development team. Ahmed et al. [5] have designated AI-assisted software engineering as a research priority with far-ranging implications in both industry and society, with the greater challenge posed by the need to have an unbiased assessment criterion and a new code of digital ethics. In line with this view, surveys of enhancement methods such as fine-tuning, retrieval-augmented generation, and prompt engineering validate that enhancing reliability has been a key open problem across all major LLMs [6]. There is also a high rate of developer adoption, with Lavanya et al. [7] reporting significant productivity improvements especially among junior developers, albeit with a moderate level of trust (~38%), which also indicates that the issue of reliability continues to be a concern in practice even among users of these tools on a regular basis.

1.3 Research Problem

Although model capability is rapidly advancing, there are still persistent questions regarding the reliability of the code generated by LLM in real-world scenarios, and even more so in the context of small-scale, practical, everyday tasks as opposed to the algorithmic interview-style problems that dominate published benchmarks. Ouyang et al. [8] have shown that ChatGPT is severely non-deterministic on the same prompt, and establish most tasks as producing zero semantically identical

responses across repeated execution, and critically determined that the use of zero temperature does not ensure reproducibility. This instability essentially calls into question the credibility of the code generated by LLM in the context of professional software engineering.

This is worsened by the evaluation frameworks employed in majority of the published studies. Liu et al. [9] showed that the popular system HumanEval systematically overestimates the ability of LLM: when test suites were augmented with EvalPlus, pass@k scores dropped by up to 28.9% across 26 tested models, revealing a discrepancy between measured and actual reliability. This result was confirmed by Agarwal et al. [10], who argued that the HumanEval tasks are identical in difficulty to simple algorithmic interview questions that are completely unrelated to the tasks of software engineering in the real world that involve interaction with a variety of files, methods and system components. Catir et al. [11] confirmed that the performance of LLM significantly reduces with respect to the advanced programming tasks as compared to the introductory ones, but the difference between the benchmark performance and realistic reliability on realistic small scale programming tasks has not been well measured.

Another and under-researched aspect is the trust of developers. Meier [12] showed that human-like human-AI interfaces breed over-trust, in which users blindly accept AI outputs without proper verification. Okamura and Yamada [13] formally defined the concepts of over-trust and under-trust in human-AI cooperation, showing that poorly calibrated trust leads to significant efficiency losses and safety risks. However, no current research empirically links measures of the reliability of the codes used in it, with the trust calibration behavior in a software engineering context, and leaves developers lacking an evidence base to appropriately tune their dependence on these tools.

1.4 Research Aim

To evaluate the reliability of code generated by multiple large language models across structured small-scale software engineering tasks, and to critically examine how observed reliability patterns relate to developer trust calibration in AI-assisted coding contexts.

1.5 Research Objectives

Objective 1. To critically analyze existing literature on LLM code generation evaluation methods, reliability metrics, and trust calibration frameworks to identify gaps specific to small-scale software engineering task contexts.

Objective 2. To design and implement a structured task suite of small-scale Python software engineering tasks that reflect realistic developer workflows, going beyond standard benchmark-style problems such as HumanEval and MBPP.

Objective 3. To empirically evaluate the functional correctness, output consistency, exception handling, and code quality of outputs generated by a selected set of LLMs using automated unit testing and static analysis tools.

Objective 4. To analyse and compare reliability patterns across LLMs using quantitative metrics including pass@k, syntax accuracy rate, exception handling success rate, and static code quality scores.

Objective 5. To critically evaluate the alignment between empirically measured LLM reliability and established trust calibration principles, identifying conditions likely to produce over-trust or under-trust in AI-assisted code generation for small-scale software engineering.

1.6 Significance of the Research

This research is important at the point of convergence between academic rigour and practical need of the developer. Technically, it fills an existing gap in the literature on software engineering evaluation by creating an experimental framework specifically designed to measure the reliability of LLM in small-scale realistic software engineering tasks, but not in simplified benchmarks [5], [10]. This leads to the greater task of creating unbiased, task-oriented evaluation criteria listed by Ahmed et al. [5] as key to responsible AI adoption in software engineering.

Practically, the results will provide developers and engineering teams with quantitative data which they can use to make informed decisions regarding which LLMs to deploy to perform small-scale tasks, and how much trust to place in their results. Accuracy should not be the only measure of the code generators as argued by Anand et al. [4] that the code generators should be evaluated on the basis of validity, correctness, security, reliability and maintainability. The trust calibration analysis is an extension of this practical value, which offers practical advice on how to reduce the over-trust risks observed in the human-AI interaction literature, which in turn can benefit practitioners who are already in the business of utilizing the code generated by LLM in the context of professional software development.

2 LITERATURE REVIEW

2.1 Scope and Purpose of the Review

This literature review discusses the available body of research that will be relevant in determining the reliability of the code generated by LLM, as well as the trust implications of such a system on the developers in software engineering scenarios. The review includes primary literature published between 2020 and 2026, but with a focus on peer-reviewed journal articles and conference proceedings on IEEE Xplore and the ACM Digital Library. The scope includes four thematic areas: the evaluation frameworks and evaluation standards, reliability and non-determinism in practice, the assessment of small-scale and tasks of practical software engineering, and trust calibration in human-AI coding situations. The review is themed to allow critical synthesis across the studies, finding contradictions and agreements and positioning this study very clearly against the already existing body of knowledge.

2.2 Evaluation Frameworks and Benchmarks for LLM Code Generation

Since the introduction of HumanEval and Codex in 2021, the evaluation of the capabilities of the LLM code generation has been dominated by a few benchmark datasets. HumanEval contains 164 problems that are Python functions-level problems with unit tests and pass@k, with the probability of finding a correct solution quickly became the standard correctness measure of the field. Subsequently Xu et al. [14] demonstrated with PolyCoder that open-source models could approach the performance of Codex models in particular languages, and also emphasised that the Python-centricity of HumanEval would not generalise to other programming environments. Umama et al. [1] confirm this language bias in a systematic review of 58 studies guided by PRISMA that demonstrated that HumanEval, MBPP, and APPS are the predominant benchmarks despite their algorithmic simplicity and limited scope.

The process of the development of these initial schemes into more rigorous assessment has been very extensive, but structurally incomplete. After reviewing 291 SE benchmarks, Hu et al. [15] concluded that existing designs do not reflect adequately industrial workflows, real world risk or the full complexity of the task of software engineering, and proposed rigorous, transparent and collaborative evaluation criteria. The most direct empirical evidence supporting benchmark inadequacy is the work of Liu et al. [9], who showed that adding more powerful and more systematic generated test cases via

EvalPlus to HumanEval reduced pass@k scores by up to 28.9% across 26 LLMs, showing that a significant proportion of previously credited correct solutions were actually incorrect under more rigorous testing. All of these findings together make it structurally biased to simplistic tasks, making claims about the reliability of LLM in wider software engineering contexts epistemically dubious.

2.3 Reliability, Non-Determinism, and Code Quality in Practice

Experimental studies of the reliability of the LLM demonstrate that the instability remains constant when conditions that practitioners face frequently are disrupted. The most extensive evidence is provided by Ouyang et al. [8], the analysis of 829 coding tasks demonstrated that over 75 percent of them employed zero semantically identical output across repeated runs even with temperature set to zero, which directly undermines the concept of reproducibility and confidence of developers. A controlled repeated-trials design, generating solutions across ten Python problems at three iterations per model was adopted by Palla and Slaby [16], who found meaningful variance in Claude, GPT, and Gemini families, establishing that reliability is a metric that should be measured across iterations and not as a single-shot metric. In Ságodi et al. [17] ChatGPT was found to outperform Copilot across 25 program synthesis problems, although neither model produced consistently acceptable quality when functional testing, static analysis, and developer acceptance were all evaluated at the same time, explaining the need to have a multi-dimensional evaluation.

On a larger scale, Huynh and Lin [6] found that syntactic and semantic error rate, resource constraints, and security vulnerabilities were consistent across all major LLMs, and surveyed metrics beyond pass@k including CodeBLEU, and ICE-Score which capture quality dimensions not seen by correctness-only models. Gülmez [2] extended this point of view into history and found that even with large-scale model development basic problems with correctness verification and consistency at the repository level have remained unresolved. Such intersections of evidence across studies suggests that the reliability of LLM does not solely depend on benchmark score but rather on consistency, quality and accuracy across various dimensions of tasks.

2.4 Small-Scale and Practical Software Engineering Task Evaluation

One common limitation to the benchmark-centred literature is the lack of correspondence between the assessed tasks and the small, practical programming tasks that comprise much of the daily work of most developers. Agarwal et al. [10] brought this criticism into the open, by stating that real-world software engineering evaluation needs to take into consideration interaction with multiple files, APIs, and methods, contexts that are completely absent in HumanEval-style problems. Empirical evidence

was given by Catir et al. [11] who found that LLMs perform well on novice level program tasks, with partial or heuristic solutions on advanced tasks.

Rahman et al. [18] validated this issue by evidence of a 324,843 class semantic similarity score of 0.89 but a functional pass rate of only 0.33 which validates that the code can look plausible but still unfunctional. A 100-participant developer survey paired with MBPP benchmark evaluation that Lavanya et al. [7] conducted found that the productivity gains were present but trust was moderate with the participants expressing concerns of reliability even as they continued to use LLMs on a regular basis. Ahmed et al. [5] put this in terms of a systemic issue that needs large, unbiased datasets as a precondition to responsibly advance AI-assisted software engineering, a gap that is directly filled by this study through its purpose-built small-scale task suite.

2.5 Trust Calibration in AI-Assisted Coding Contexts

The quality of the generated code as perceived and reacted to by its developers is the human aspect of the reliability of LLM that is addressed by an increasing but still relatively small number of studies. Okamura and Yamada [13] have given the experimental framework on which the study will be based, formally defining over-trust as the situation when user confidence is greater than actual AI reliability, and showing that adaptive trust-calibration cues can be more effective in shaping the outcome of human-AI cooperation than transparency approaches as a standard. Even though they mentioned that their empirical study was carried out in a drone cooperation scenario, theoretical constructs that they have developed can be directly applied to AI-assisted coding. Meier [12] has shown the applicability of these constructs to LLMs in particular, showing that anthropomorphic AI interfaces invariably give rise to over-trusting behaviors in users that result in the disclosure of sensitive information and uncritical acceptance of the information presented by the anthropomorphic AI interfaces.

In a software engineering context, Shin [19] showed that textual similarity measures are weakly correlated with dynamic adequacy measures, meaning that developers who use a textual approach to similarity measurements instead of an execution approach are systematically at risk of over-relying on the outputs of LLM. In another dimension Mathews [20] found that buggy code can be validated by the buggy code itself, and that bug-revealing tests can be rejected by the buggy code itself. Combined, these studies prove that the reliability of the code in terms of reliability metrics is a substantive and quantifiable issue in AI-assisted software engineering, but no study has empirically linked the reliability metrics of the code with trust calibration conditions within a controlled experimental design.

2.6 Theoretical and Technical Frameworks.

The theoretical basis of this study refers to the Design Science Research (DSR) framework that frames the task suite and evaluation harness as the artifacts of research that should be used to solve an identified problem and assessed in terms of predefined goals. The conceptual framework of Okamura and Yamada [13] is used as the conceptual framework to look at the findings of reliability. Technically, the study is based on literature of automated unit test generation: Zhang et al. [21] in a systematic review of 116 studies demonstrated that LLMs are more effective in semantic understanding in test generation however inconsistent datasets and metrics hinder fair comparison, a finding that supports the need to have a controlled and consistent evaluation protocol as proposed herein. Bektas [23] noted that evaluation reliability of SE studies based on LLM is moderate and advised that evaluation design should explicitly consider the variability of the temperature and sampling behavior of the LLM that should be accounted in evaluation design. According to Joel et al. [22], a small-scale task approach that emphasizes Python-based applications proves to be relevant.

2.7 Literature Summary

The three gaps that converge and are not adequately largely addressed are established in the literature. First, benchmark evaluation infrastructure is structurally flawed to measure the reliability of LLM on practical small-scale software tasks, overestimating the correctness and underestimating non-determinism. Second, the reliability studies which have been conducted in the past either employ simplistic benchmarks, or test only one or two models without repetition or multi-dimensional measures. Third, in any software engineering research, trust calibration has not empirically been linked to any of the code reliability measures. This paper has filled out all three gaps by developing a structured set of tasks of realistic Python problems, testing the four LLMs on five reliability dimensions in automated testing, and systematically mapping the results of finding reliability to the known conditions of trust calibration.

3 RESEARCH DESIGN AND METHODOLOGY

3.1 Research Design

This paper uses a positivist research philosophy, in which we take the reliability of the generated code outputs as an objective, measurable attribute of the generated code outputs which can be quantified

without having to interpret the researcher. It uses a deductive method: the existing theory and previous empirical research before data collection is formed the basis of deriving hypotheses about the differences in inter-LLM reliability. The combination has been long-established in the research of computational software engineering, and is consistent with the approach successfully used by Palla and Slaby [16] and Ságodi et al. [17] to characterize the differences in reliability in LLM through controlled experimentation.

The study design is a comparative experiment. The LLM is considered as the independent variable that is under controlled prompting conditions throughout a task suite, and five measures of reliability are the dependent measures. It is an experimental strategy since the study had a goal of determining causal or quasi-causal relationships between model choice and measurable reliability outcomes under controlled conditions, which cannot be established using solely a survey or case study design. Figure 1 and Figure 2, which were created to illustrate the research methodology flowchart and the architecture of the system of the experiment, respectively.

Research Methodology Flowchart

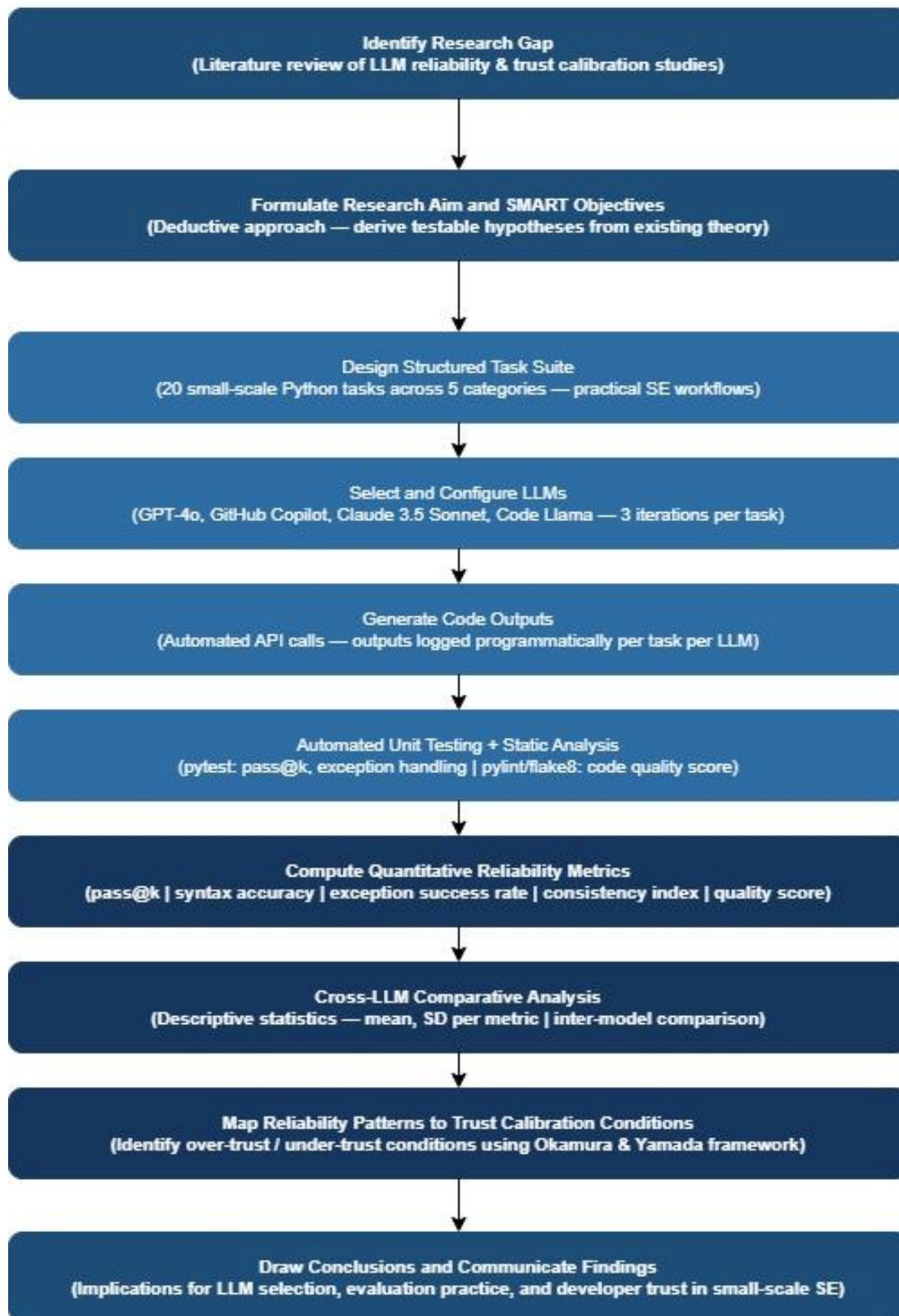


Figure 1: Research Methodology Flowchart

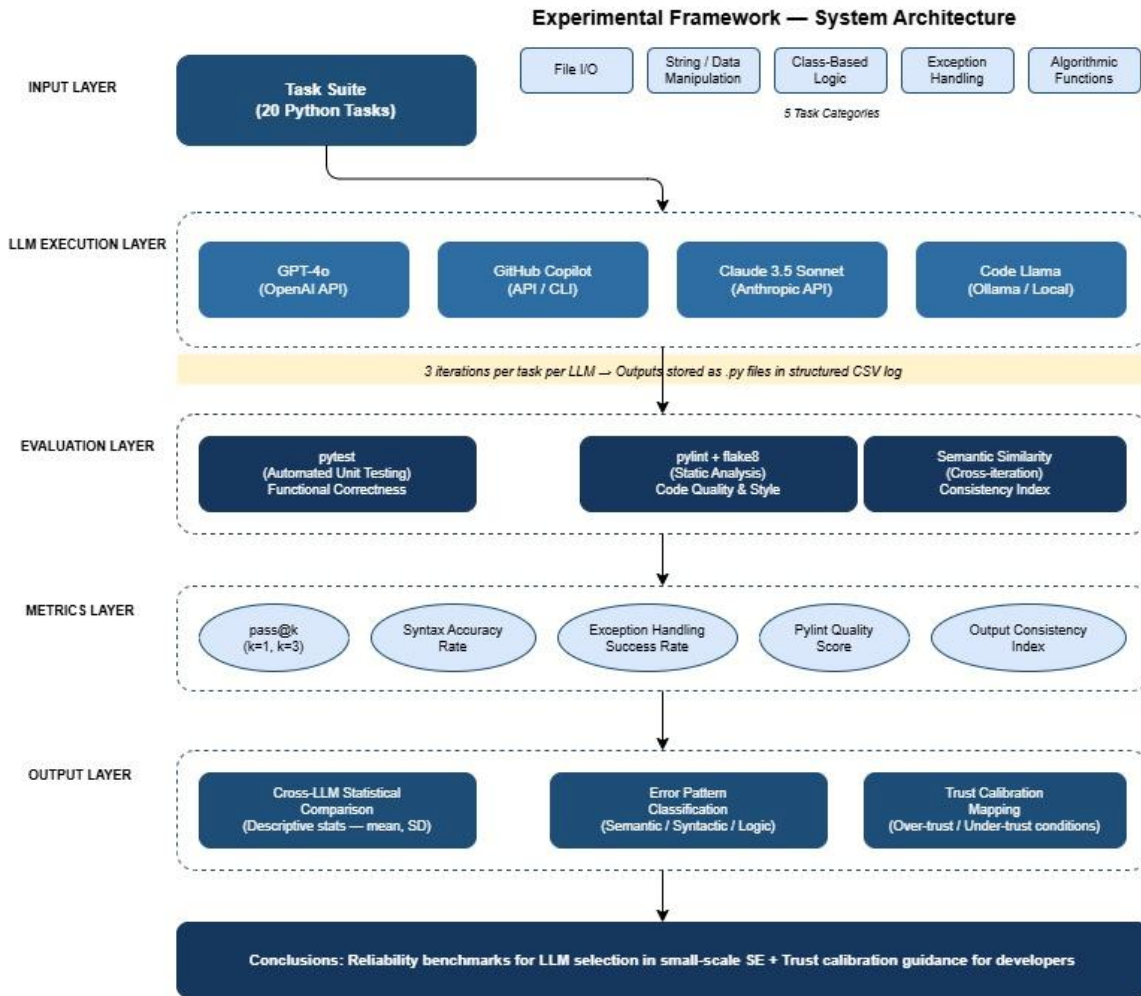


Figure 2: Experimental System Architecture

3.2 Research Methods

It is a quantitative and a mono-method. Each LLM code output is evaluated using automated evaluation tools, which yield numerical measures that are then analyzed using descriptive statistics analysis and compared with other models. Quantitative methods are suitable as the research questions involve the differences in the correctness rates and consistency, and quality scores across the LLMs, which are all the numbers and need to be compared statistically [16]. The research approach is quantitative experiment, which is in line with the philosophy of positivism and deductive approach that was established above.

The small-scale Python tasks are designed in 20 tasks that represent five categories that reflect real-world workflows by the developer: file input/output operations, data and string manipulation, object logic (classes), exception handling, and basic algorithmic functions. These categories were chosen to represent task types that seem to be common to professional software engineering environments yet to

be represented in HumanEval-style benchmarks, directly responding to the constraints found by Agarwal et al. [10] and Catir et al. [11]. These four LLMs are assessed: GPT-4o (OpenAI API), GitHub Copilot, Claude 3.5 Sonnet (Anthropic API), and Code Llama via Ollama. Each LLM produces an output of a solution to each task three times independently, following the repeated-iteration protocol of Palla and Slaby [16], and resulting in a total of 240 code outputs ($4 \text{ LLMs} \times 20 \text{ tasks} \times 3 \text{ iterations}$). The data analysis will involve descriptive statistics (mean and standard deviation of each metric per LLM and task category) and will then compare the data across LLM. The patterns of reliability are then qualitatively mapped to the over-trust and under-trust conditions as defined by Okamura and Yamada [13].

3.3 Research Materials

3.3.1 Data Sources and Collection

The main data source is the task suite that is specifically designed and described in Section 4.2. Every task comes with a hand-written unit test suite to which the outputs of LLM are compared. MBPP is designed to be used as a partial reference baseline to contextualise passk findings with respect to current benchmark literature, following the precedent of Lavanya et al. [7]. None of the code in the program are explicitly written, but programmatically generated via API calls to each LLM, whose outputs are captured as structured Python files and logged to a CSV record containing the identifier of the LLM, the identifier of the task, the iteration number, and the raw output. There are no pre-existing datasets as the main evaluation inputs that can eliminate the issue of data provenance and licensing. All the tasks are developed in a manner that they can be solved by small-scale developers using Python, keeping the difficulty of the task realistic in the small-scale SE environment under study.

3.3.2 Evaluation Metrics

Table 1 gives the five quantitative measures of reliability used in this study along with their measurement instruments and justifications based on the confirmed literature.

Table 1: Quantitative Reliability Metrics for LLM Code Evaluation

Metric	Tool	What It Measures	Justification
pass@k (k=1, k=3)	pytest	Functional correctness across k attempts	Foundational metric from Chen et al. [3]; strengthened by EvalPlus framework of Liu et al. [9]
Syntax Accuracy Rate	Python interpreter / pytest	Proportion of outputs free from syntax errors	Captures baseline executability across task types [6]
Exception Handling Success Rate	pytest (custom harness)	Correct behaviour under edge-case inputs	Key reliability dimension identified by Palla and Slaby [16]
Static Code Quality Score	pylint / flake8	Style compliance, complexity, and maintainability	Addresses quality beyond correctness as argued by Ságoti et al. [17]
Output Consistency Index	diffliib (semantic similarity)	Agreement across three independent runs	Motivated by non-determinism findings of Ouyang et al. [8]

3.3.3 *Software, Libraries and Frameworks.*

The Python 3.11 is the implementation language of the evaluation harness. Pytest as an automated unit testing framework has a layer of custom pytest fixtures to carry out exception-handling policies. Pylint and flake8 are used as the tools of the static analysis, with the quality scores of pylint normalised to a 0-10 scale. The OpenAI Python SDK (GPT-4o), the Anthropic Python SDK (Claude 3.5 Sonnet), and Ollama are used to gain access to API. The consistency of output is calculated using Python diffliib SequenceMatcher, which is a normalised similarity ratio between 0 and 1 across the three iterative outputs per task. The results are aggregated with help of the pandas library, and are exported to CSV to be analyzed by statistics.

3.3.4 Hardware Requirements

No specialized equipment is needed in this study. GPT-4o, GitHub Copilot, and Claude 3.5 Sonnet are all inferred via cloud API endpoints, and only require access to the Internet and valid API credentials. Code Llama is implemented on Ollama, on a normal development laptop with at least 16 GB of memory. The harness of evaluation, data recording and statistical analysis scripts are minimal and can be executed in any contemporary development machine.

Ethical Considerations

The current research does not imply the use of any human subjects and does not require gathering any personal information. Any use of LLM API is in compliance with published terms of service of each provider, including the OpenAI, Anthropic, and GitHub API agreements. Outputs of generated code are only run in isolated virtual environments (Python venv) to eliminate the possibility of any risk of unintended system interaction. There is no code generated using LLM that is implemented into production systems. The selection of the LLMs under evaluation are not accompanied by any conflicts of interest; all models are considered as independent experimental conditions.

4 EXPECTED OUTCOMES AND LIMITATIONS

4.1 Anticipated Results

This research project expects to determine statistically significant differences in reliability between the four LLMs on various measures in line with the previous comparative studies [16], [17]. The value of Pass@k is supposed to be meaningfully differentiated by the type of task, with file I/O and class-based tasks proving to be more difficult than the manipulation functions of strings, which represents the difficulty gradient proposed by Catir et al. [11]. Non-determinism, quantified by the index of the output consistency, is expected across all models a significant proportion of tasks, replicating the results of Ouyang et al. [8] in a practical task setting. It is hoped that the trust calibration mapping will be able to show certain conditions of over-trust, especially in the case of tasks where the outputs are syntactically correct and pass the superficial inspection but fail the unit tests of the edge cases, in line with the mechanisms described by Okamura and Yamada [13] and the empirical patterns identified by Mathews [20].

4.2 Implications

The first theoretical implication is the extrapolation of the trust calibration frameworks in general human-AI cooperation settings to the specific area of the LLM-assisted code generation. The practical implication is an evidence-based framework of the developers and the small engineering teams working on practical python software project to select and give trust guidance on selecting an LLM. The task suite, harness, and metric suite are themselves a methodological contribution that can be extended to other languages, LLMs, or levels of task complexity, directly responding to the need to have evaluations that can be replicated identified by Bektas [23] and Hu et al. [15].

4.3 Limitations

A number of restrictions have to be recognized. There might be restrictions on API access, which could then restrict the number of planned LLMs included, therefore limiting the generalizability of comparative results. The task suite which is intended to represent real-world workflows, cannot possibly capture the entire range of complexity in real-world software engineering such as multi-file projects and integration with external systems. The unit testing is an automated process that cannot identify all the semantic errors, or architectural vulnerabilities as Mathews [20] showed. Moreover, since the capabilities of LLM are rapidly changing as a result of constant updates of models and releases of new versions, the empirical results might need to be replicated due to the constant changes in models and the introduction of new versions into the market, a limitation that is widely recognized in the evaluation literature [23]. Lastly, trust calibration analysis is an analysis, not an empirically validated analysis through developer studies, which is a direction to be taken in future studies.

5 PROJECT TIMELINE

The research project will be done over a period of twelve weeks. Where possible phases overlap to keep the momentum going: literature review is continued as the early stages of task design and writing commences before data collection is fully accomplished to keep the submission timeline on track. The Gantt chart is shown in Table 2. The key milestones are: by the end of Week 4: the task suite is complete and ready to run; by the end of Week 6: all code outputs are in; by the end of Week 8: metric computation and analysis are complete; by the end of Week 12: the full draft is ready to be reviewed.

Table 2: 12-Week Project Gantt Chart

Ph	Task / Milestone	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12
1	Systematic Literature Review												
2	Task Suite Design												
3	Experiment Setup (APIs, harness, logging)												
4	Code Generation (data collection)												
5	Automated Testing & Static Analysis												
6	Metric Computation & Statistical Analysis												
7	Trust Calibration Mapping												
8	Writing and Drafting												
9	Review, Refinement and Submission												

REFERENCES

- [1] A. Umama et al., "LLM-Based Code Generation: A Systematic Literature Review With Technical and Demographic Insights," *IEEE Access*, vol. 13, pp. 194915-194939, 2025, doi: 10.1109/ACCESS.2025.3631952.
- [2] B. Gülmez, "Code Generation with Large Language Models: A Survey from Neural Program Synthesis to Autonomous Software Development," *Applied Intelligence*, vol. 56, p. 200, 2026, doi: 10.1007/s10489-026-07230-0.
- [3] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.
- [4] A. Anand, S. Chopra, and M. Arora, "Analysis of LLM Code Synthesis in Software Productivity," in *Applied Intelligence and Computing*, pp. 247-259, SCRS, 2025, doi: 10.56155/978-81-955020-9-7-24.
- [5] I. Ahmed et al., "Artificial Intelligence for Software Engineering: The Journey So Far and the Road Ahead," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, Art. 119, 2025, doi: 10.1145/3719006.
- [6] N. Huynh and B. Lin, "Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications," arXiv:2503.01245, 2025.
- [7] R. Lavanya et al., "Machine Learning-Assisted Code Generation: Exploring the Role of Large Language Models in Automating Software Development Tasks," *Int. J. Adv. Signal Image Sci.*, vol. 12, no. 3s, pp. 1825-1837, 2026.
- [8] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An Empirical Study of the Non-Determinism of ChatGPT in Code Generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, Art. 42, 2025, doi: 10.1145/3697010.
- [9] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," in *Proc. 37th Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2023.
- [10] A. Agarwal et al., "Copilot Evaluation Harness: Evaluating LLM-Guided Software Programming," arXiv:2402.14261, 2024.
- [11] E. Catir, R. Claesson, and R. M. Tsoupidi, "Evaluating Code Generation of LLMs in Advanced Computer Science Problems," arXiv:2504.14964, 2025.

- [12] R. Meier, "Balancing Minds and Data: The Privacy Dilemma of LLMs and Anthropomorphism in LLMs," *J. Social Comput.*, vol. 6, no. 3, pp. 173-183, 2025, doi: 10.23919/JSC.2025.0014.
- [13] K. Okamura and S. Yamada, "Empirical Evaluations of Framework for Adaptive Trust Calibration in Human-AI Cooperation," *IEEE Access*, vol. 8, pp. 220335-220349, 2020, doi: 10.1109/ACCESS.2020.3042556.
- [14] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A Systematic Evaluation of Large Language Models of Code," in *Proc. ACM SIGPLAN Int. Symp. Mach. Programming (MAPS '22)*, San Diego, CA, USA, 2022, doi: 10.1145/3520312.3534862.
- [15] X. Hu et al., "Assessing and Advancing Benchmarks for Evaluating Large Language Models in Software Engineering Tasks," *ACM Trans. Softw. Eng. Methodol.*, 2025, doi: 10.1145/3786771.
- [16] D. Palla and A. Slaby, "Evaluation of Generative AI Models in Python Code Generation: A Comparative Study," *IEEE Access*, vol. 13, pp. 65334-65347, 2025, doi: 10.1109/ACCESS.2025.3560244.
- [17] Z. Ságodi, I. Siket, and R. Ferenc, "Methodology for Code Synthesis Evaluation of LLMs Presented by a Case Study of ChatGPT and Copilot," *IEEE Access*, vol. 12, pp. 72303-72316, 2024, doi: 10.1109/ACCESS.2024.3403858.
- [18] M. Rahman, S. H. Khatoonabadi, and E. Shihab, "OpenClassGen: A Large-Scale Corpus of Real-World Python Classes for LLM Research," in *Proc. Evaluation and Assessment in Software Engineering (EASE)*, 2026.
- [19] J. Shin, "Investigating the Effectiveness of Large Language Models in Automated Software Engineering," Ph.D. dissertation, York Univ., Canada, 2025.
- [20] N. S. Mathews, "Code Generation and Testing in the Era of AI-Native Software Engineering," M.Math thesis, Univ. of Waterloo, Canada, 2024.
- [21] J. Zhang, X. Hu, C. Gao, X. Xia, and S. Li, "Enhancing Automated Unit Test Generation with Large Language Models: A Systematic Literature Review," *ACM Trans. Softw. Eng. Methodol.*, 2026, doi: 10.1145/3802827.
- [22] S. Joel, J. J. Wu, and F. Fard, "A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages," *ACM Trans. Softw. Eng. Methodol.*, 2025, doi: 10.1145/3770084.

[23] A. Bektas, "Large Language Models in Software Engineering: A Critical Review of Evaluation Strategies," M.Sc. thesis, Freie Univ. Berlin, Germany, 2025.