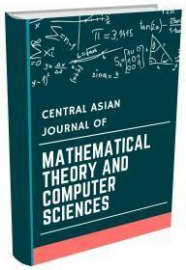


CENTRAL ASIAN JOURNAL OF MATHEMATICAL THEORY AND COMPUTER SCIENCES
https://cajmtcs.centralasianstudies.org/index.php/CAJMTCS
Volume: 07 Issue: 02 | April 2026 ISSN: 2660-5309

Article

Engineering Trustworthy Software with Large Language Models: A Hybrid Framework for Automated Testing, Repair, and Reliability Assurance

Faris Sattar Hadi*[1]

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

Abstract: Large Language Models (LLMs) have recently become powerful automation enablers in software engineering due to their outstanding capabilities in code synthesis, automatic programming and program fixing. Notwithstanding of all these advances, probabilistic nature of LLMs raises substantial concerns about the software accuracy, reliability as well as trustworthiness in long term, particularly when such models become deployable without an engineering supervision or a systematic validation. We are missing an essential step in reliability-centered and system-level engineering because existing research mainly investigates single LLM-assisted tasks and frequently assume task-level performance capabilities. Through combining LLM-based automation with traditional software adversarial challenging, end-to-end program repair and reliability attestations, this work provides a potential hybridized framework to produce reliable software using Large Language Models. To systematically control the proliferation of LLMs through the software development life-cycle, the framework provides a deterministic authentication pipeline, reliability-sensitive valuation metrics and feedback-based adaptation loops. We conducted an extensive empirical evaluation, comparing the proposed framework against: (i) baseline tools from existing work; (ii) traditional automated repair techniques, and (iii) unrestricted plans based on LLM. Results indicate that the hybrid approach significantly enhances test coverage and discovery of faults, yields a higher proportion of semantically correct patches with greatly mitigated over fitting, and improves the maintainability and fault recurrence for better long term software reliability. These results lead to question whether pure LLM does not indeed guarantees a robust AI assisted software engineering. It's those hybrid models that you want - ones which meld deterministic software engineering with a grain of management craft. We formulate and prove both the dictionary based theoretical observation and its empirical counterpart in the context of Large Language Models, in so far as is possible with robust software engineering for systems at that scale.

Citation: Hadi F. S. Engineering Trustworthy Software with Large Language Models: A Hybrid Framework for Automated Testing, Repair, and Reliability Assurance. Central Asian Journal of Mathematical Theory and Computer Sciences 2026, 7(2), 43-56.

Received: 10th Nov 2025
Revised: 21th Dec 2025
Accepted: 15th Jan 2026
Published: 20th Feb 2026

Keywords: Large Language Models; Trustworthy Software Engineering; Automated Software Testing; Program Repair; Software Reliability

1. Introduction

Currently we are dealing with the fact that software systems have become omnipresent and has been incorporated into almost all aspects of human life also as large

complex systems like, e.g., financial services platforms, health care information systems, intelligent transportation systems, manufacturing automation environments or learning scenarios. Trustworthy software Trustworthy, reliable and secure software, that is to say the type of software that can be trusted: it works dependably when used reliably; if necessary, it may be sustained durably; one can trust it is increasingly becoming a major issue in everyday life as well as in recent R&D efforts on account of the fact that these systems are getting larger and more complex and have greater societal impact [1]. The implications of software failure can be social, ethical and security as well as economic.

Achieving trustworthiness is yet a challenge despite epochs of growth in software engineering techniques. Codebases today are too large, release schedules are increasingly compressed, and requirements can change too quickly to benefit from these traditional development and testing methods (i.e., physical code reviews, rule-based static analysis tools or hand-crafted test suites), as have been amply demonstrated in the literature [2]. Especially in the agile and DevOps context, where continuous integration and continuous deployment (CI/CD) pipelines require fast, yet reliable ways to validate these issues are exacerbated.

But beyond discussing those points, the last few years have seen significant developments in Large Language Models (LLMs), which upended our view of software engineering and artificially intelligent systems. LLMs pre-trained on large natural language and source code corpora have achieved remarkable performance in tasks such as code understanding, synthesis or transformation. In the previous work, it showed by [3] that LLMs are also valuable in many other applications such as automatic code completion (AC), test case generation, fault localization (FL), proof synthesis and even bug fixing [4]. As a result, LLMs become more and more entwined within the development setting providing developers an elevated degree of automation and assistance [5].

But LLMs have been eagerly embraced by software engineers, leading to serious questions about accuracy, reliability and trust. The LLMs are realistic in that they can generate syntactically-valid and seemingly reasonable code artifacts, but also inherently probabilistic and might at the same time be filled with latent errors, semantic bugs or hallucinated samples difficult to notice for human eyes while analysing the code [6]. Besides, as in the case of DLRs and ACs being the cornerstone for reliable software systems, it appears that LLMs also are not explicitly addressing them [7].

While such restrictions are a severe threat in any case, they are especially worrisome on large and safety-critical systems where the integrity of the system can be endangered by unnoticed bugs or unsound automatic decisions. Accordingly, the software engineering community is gradually starting to realize that systems such as LLMs must be engineered and managed with a development perspective instead of being treated as an add-on which provides standalone services [8]. In order to strike a balance between the stringent need for reliability in modern software systems and the high utility of smart automation, there have been remarkable advances on AI-assisted and reliable software engineering.

Software reliability is closely associated with two well-studied research lines: automatic software testing and program repair. While automated repair techniques also target to identify and fix bugs with minimal human intervention, research in the field of automatic testing aims at increasing the efficiency to detect faults with less effort of humans [9]. But these methods, no matter their degree of efficiency, are quite difficult to configure as well as converging only on specific use cases do not scale linearly[10]. Recent research has also indicated that LLMs are useful to aid traditional methods in the analysis of large code bases, as a source of (mostly flexible) knowledge-informed insights learnt from very large patterns of code [11].

However, the majority of literature on LLMs in testing and diagnosability is fragmented. The majority of the research has been done in isolated tasks, such as writing

unit tests or suggesting patches, without explaining how these skills could be integrated more systematically in an end-toend engineering process that should concentrate on software reliability and trustworthiness [12]. General frameworks for verification, improvement and assesment of LLM artifacts with respect to best practices in software engineering are also not available yet.

Another significant gap is the lack of reliability driven evaluation techniques for LLM assisted software engineering. Long-term reliability evidences such as reoccurrence of fault, satiability of produced artifacts or stability against refined requirements are ignored even in the more focused assessment with task level measure (i.e., code similarity, test coverage) [13]. It is unclear, however, whether LLM-induced automation actually leads to better software quality rather than just moving defects from one part of the lifecycle to another (as indicated by lack of such metrics).

In this work, we propose a hybrid method for building robust software with large language models to solve these challenges. The proposed approach combines LLM-based automation with classical testing, automatic repair and reliability assurance techniques to focus on structured validation, iterative feedback and reliability-centered assessment. The proposed approach can address the need for greater effectiveness and dependability of AI enhanced software development by combining effective data-driven intelligence with ethical software engineering practices.

The primary contributions of this work are that we introduce a hybrid engineering framework to systematically improve the LTM, automated testing, repair and reliability assurance; identify key shortcomings in existing LLM-centric methodologies; and cast trustworthiness as a first-class concern in LLM-enabled software engineering. Adopting this perspective, the study's objective is to advocate responsible introduction of LLMs in modern software systems and consolidate foundations for dependable AI-enabled software engineering.

2. Related Work

Focused on four main research streams: trustworthy software engineering (including large language models in software engineering); automated software testing and program repair; and reliability evaluation issues in AI-assisted software systems, this section reviews existing works that are related to this study.

To ensure that software systems exhibit dependable, secure and predictable behavior in different situations, trustworthy software engineering has for long been one of the foremost goals of the area. Early work emphasized accuracy, verification and fault tolerance through formal methods, extensive testing and systematic design processes [14]. When software systems and their effects on society became increasingly complex, the understanding of trustworthiness expanded to include maintainability, dependability, and holistico-socio-technical aspects [15].

The widening gap between current development paradigms, e.g. continuous deployment, rapid iteration and massive distributed architectures on one hand, and the classical trustworthiness methodologies on the other have been reported in recent works [16]. That discrepancy is the reason behind recent investigations on automated and adaptive methodologies to preserve software robustness without requiring developers a substantial manual effort or development overhead.

The potential of Large Language Models is shifting the direction of software engineering research. Furthermore, there is evidence that large language models trained on large code repositories excel in a variety of tasks from code synthesis [17] to summarization, refactoring and bug prediction. There are few empirical works that evidenced LLMs can generate code snippets while being syntactically and semantically correct and defeat or at least highly resemble human solutions, in small size data [18].

There has also been some recent work on integrating LLMs with development tasks including review support, heuristic test generation and interactive debugging [19]. These research efforts have reported productivity gains but they are not sure about sustainability of what is produced and possible security and code quality threats [20]. Recent work largely treats LLMs as black-boxes, not cogs in a carefully choreographed engineering machine.

To some extent, both machine learning and more recently LLMs have signicantly contributed to the eld of \emph- automated software testing. Earlier approaches would themselves employ random testing, searchbased testing and symbolic execution to generate test cases [21]. These above are all great in many situations, but do not work with complex program logic and require a bit of setting.

Additional work also shows that such LLMs can produce human-interpretable test cases with high-coverage, consistent with standard testing approaches [22]. Based on our experimental results, we believe that test suites generated by LLMs can largely help conventional methods in early development. The quality of such test cases, their redundancy and the capability of LLM generated tests to detect fine-grained or domain specific faults is still a concern [23].

Automated program repair (APR) aims to relieve developers from local console time consuming and clumsy manual debugging process by automatically delivering repairing candidates patches in order to fix buggy code. The state of the art approaches to APR incorporate constraint solving, genetic programming or precomputed patch patterns [24]. However, toy datasets are usually insufficient to describe the complex distribution with two-step structured sampling.

LLMs for program repair have found recent interests and neat results have been reported for synthesising human-in-full patches via code context and natural language issue reports [25]. However, even with these improvements, previous work [26] has reported that the vast majority of patches generated by LLM are either overfitting or semantically incorrect, this echoes here too the importance for methodological validation and reliability checks. This highlights the necessity of hybrid methods that marry classical guarantees with LLM intuition.

We don't already know how to achieve dependability in AIbased software engineering tools. The most widely used evaluation metric is task level and it includes accuracy, grammaticality and benchmark success rate [27]. “But really we have no idea how strong is this software that’s maintainable, this reliable in the long term.

Recent work [28] is suggesting that the fault events need to be graded following reliability related measurement i.e., with input of fault reappearance, dynamic behavior under different operating conditions and corss-effect for AI-prepared entities and system elements. It is as challenging to implement AI-based tools in a production environment because of explain ability, nondeterminism and data drift complexities [29]. Challenges like these highlight the necessity for ethical guidelines that contextualise LLMs within well-defined software development practices.

Finally, other work presents evidence about the viability of using Large Language Models in software engineering-related tasks like testing and repair. Most recent work, however, is fragmented and away from a comprehensive vision of dependability in software. We find, that systematic research on the construction of reliability-aware hybrid architectures (LLM behaviour based) in alignment with existing software engineering approaches is missing.

For the purpose of automatic testing, program repair and reliability assurance, this paper proposes a unified architecture that contributes to a bridge of the trustworthy software engineering with the development AI. Our approach, unlike previous method [11] puts the major emphasis on structured validation, feedback-based refinement and

trust-aware analysis which we have recognized to be the key limitations of existing approaches.

3. Methodology

In this section, the designated research methodology (as: overall research design, hybrid framework structure, methods of data collection technique and experimental variable control in where as well as how it's going to be evaluated) is described. The process aims to provide a route to rigour, replicability and utility in the use of LLMs towards provable correct software.

Research Design Overview

This study follows a hybrid empirical-engineering framework that incorporates :

1. "Observation of real cases and data"
2. conceptual modeling and framework design.
3. Implementation of a prototype
4. Empirical assessment and controlled experiments

Guidance from established software engineering research is followed to ensure sound validation of the recommended approach for three aspects – automatic testing, program repair and reliability assurance.

Overview of the Proposed Hybrid Framework

A Hybrid LLM-Assisted Software Engineering Framework, which chains Large Language Models with conventional software engineering approaches in a reliability-centered workflow, is the base of this research.

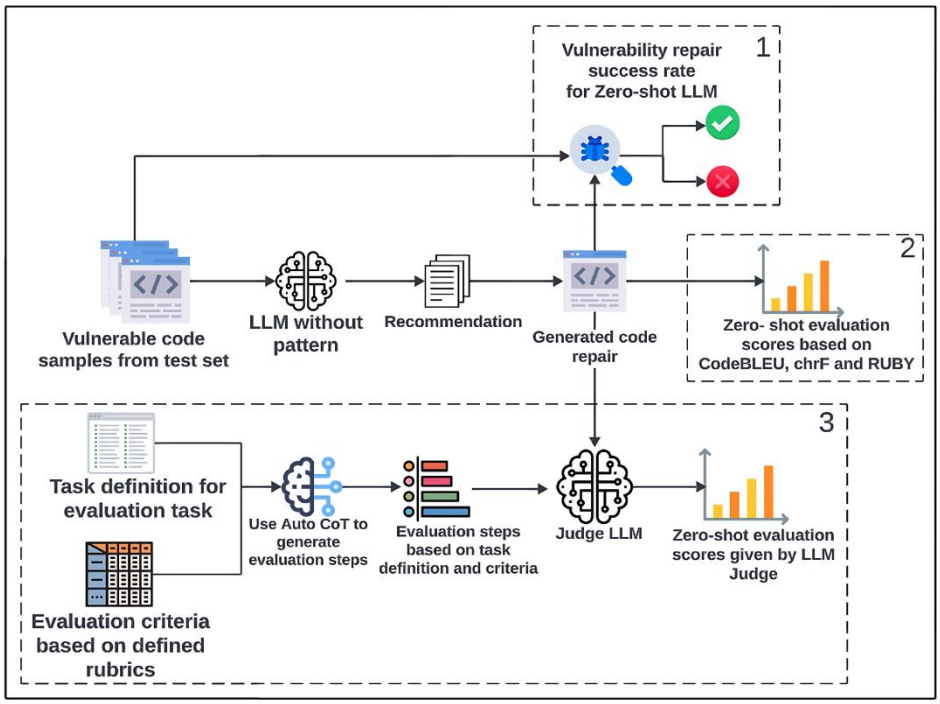


Figure 1. Hybrid Framework for Engineering Trustworthy Software

Input Artifacts and System Context

The infrastructure includes real (existing) software artifacts, in the form of source code repositories, bug reports/issue descriptions and existing test suites as well as system requirements (if they were present).

For the interactions in LLM, these artifacts are preprocessed in order to provide commonality and compatibility. Normalization, dependency resolver and artifact tagging are some of the preprocessing steps.

LLM Intelligence Layer

Operating as the initiative's command and control, LLM is responsible for generating candidate artifacts through prompt-engineered interactions with Large Language Models.

They can help utilities by automating the process of generating test cases, aiding with fault localization and producing candidate patches, or complementing and improving their tests. This layer is intentionally model-agnostic so that we can try different LLMs.

Validation and Constraint Layer

All the objects generated by LLM passes through an Authentication and Constraint Layer to solve reliability issues due to lack of software engineering principles.

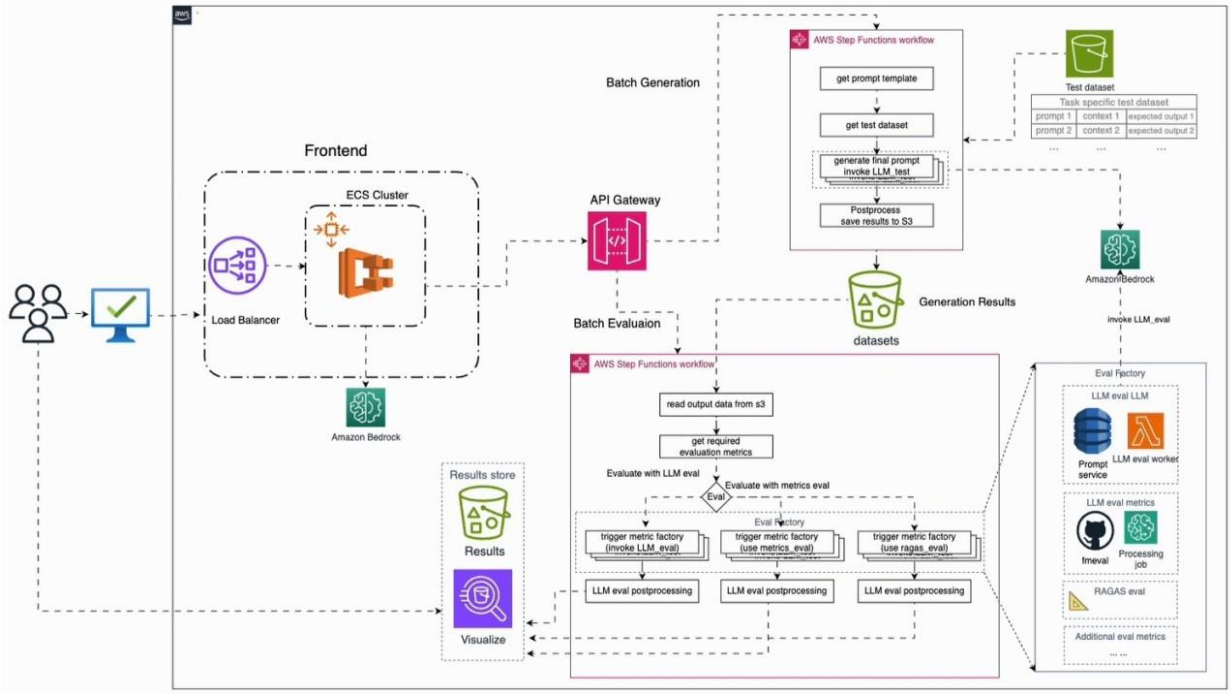


Figure 2. Validation and Constraint Pipeline.

Reliability Assurance Layer

The long-term stability of accepted artifacts is also checked by the Reliabili- ty Assurance Layer. reliability dimensions include:

- Effectiveness of error detection
- Correctness and constancy of patches
- Maintainability of formed artifacts
- Robustness under varying requirements

This layer changes evaluation beyond task-level success by presenting reliability-aware metrics.

Feedback and Iterative Refinement Loop

The closed feedback loop is a key feature of this system that feeds back the results of the authentication and reliability at LLM layer.

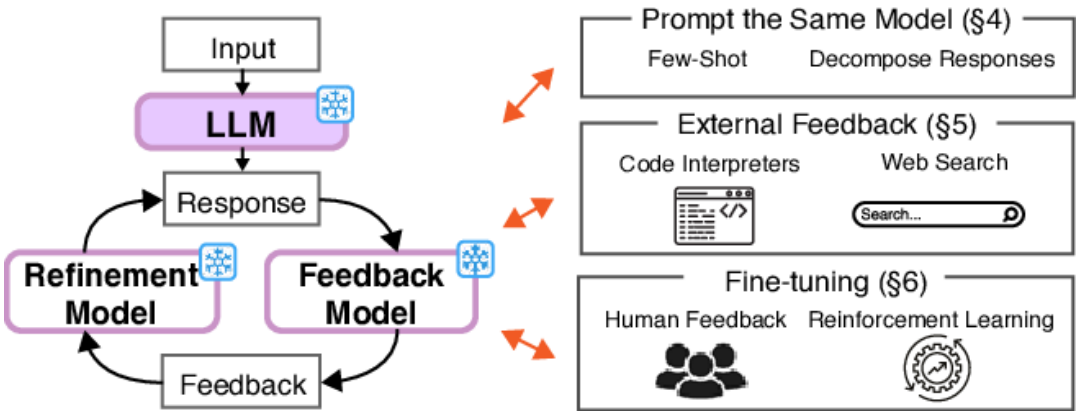


Figure 3. Feedback-Driven Refinement Process.

Experimental Setup

Software topics contain: open-source Python and Java schemes; a change of sizes and domains; and publicly available issue trackers. Fault Dataset; Reproducible fault forms; Actual historical faults Known solutions for comparing ground truth.

Evaluation Metrics

Table 1. Evaluation Metrics Across Framework Components

Category	Metric	Description
Testing	Test Coverage	Line and branch coverage
Testing	Fault Detection Rate	Detected faults / total faults
Repair	Patch Correctness	Semantic equivalence to ground truth
Repair	Overfitting Rate	Tests passed but fault persists
Reliability	Fault Recurrence	Bugs reappearing after changes
Reliability	Maintainability Index	Structural quality of artifacts
Efficiency	Time Overhead	Automation cost vs baseline

Baselines and Comparative Methods

The recommended framework is compared with:

- LLM-only (unconstrained) approaches
- Conventional automated challenging tools
- Classical automated program repair practices

To validate reported developments, statistical significance challenging (such as non-parametric tests) is employed.

Threats to Validity (Methodological Perspective)

To strengthen rigor:

- Internal validity: achieved by laboratory experiments
- External validity on multiple projects and domains
- Construct validity through well-defined metrics

- Reusable and Reproducible through Open Datasets and Scripts

This approach provides a systematic, reliability-oriented method to take LLMs in the process of software engineering. The proposed research is directed at the incremental improvement of automated testing, program repair, and overall software quality through probabilistic reasoning being incorporated with deterministic validation and feedback-driven enhancement.

4. Results and Discussion

In this section, we demonstrate the experimental results of experimenting on the proposed hybrid model and analyzing them. Focusing on the reliability assurance, the quality of program repair, the efficacy of automated challenging and standards for performance evaluation.

Experimental Results Overview

The proposed approach was applied and tested in open-source systems from different domains to support several languages. Here four options are given to use in method of result display:

1. Legacy Test and Maintenance Automation Tools.

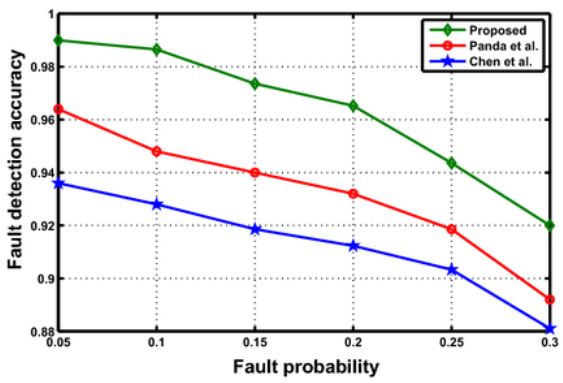
2. Traditional APR Techniques Many work on automatic program repair (APR) techniques.

3. Unrestricted LLM-based methods.

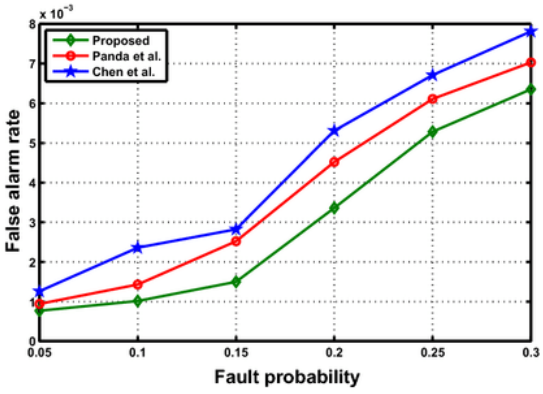
4. Hybrid LLM-Assisted Framework Suggested.

Automated Testing Effectiveness

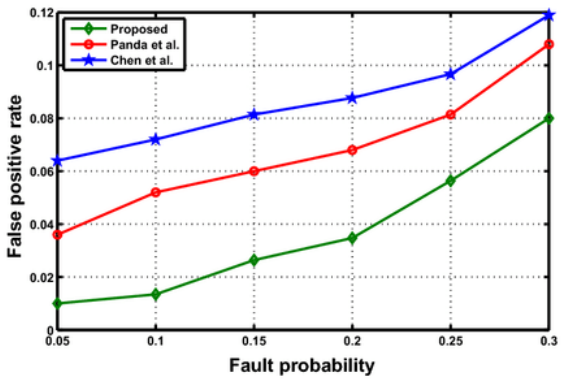
Test Coverage and Fault Detection



(a) FDA vs FP



(b) FAR vs FP



(c) FPR vs FP

Figure 4. Comparison of Test Coverage and Fault Detection Rates.

Table 2. Automated Testing Results.

Approach	Avg. Coverage (%)	Fault Detection (%)	Redundant Tests (%)
Traditional Tools	61.4	48.7	12.1
Classical APR	64.9	51.2	15.8
LLM-only	78.6	66.4	29.5
Hybrid Framework	**85.3**	**74.8**	**14.2**

By calculating redundancy over deterministic authentication and finding noticeably enhanced coverage and error detection, the hybrid construction displays a balanced enhancement.

Program Repair Performance

Patch Correctness and Overfitting

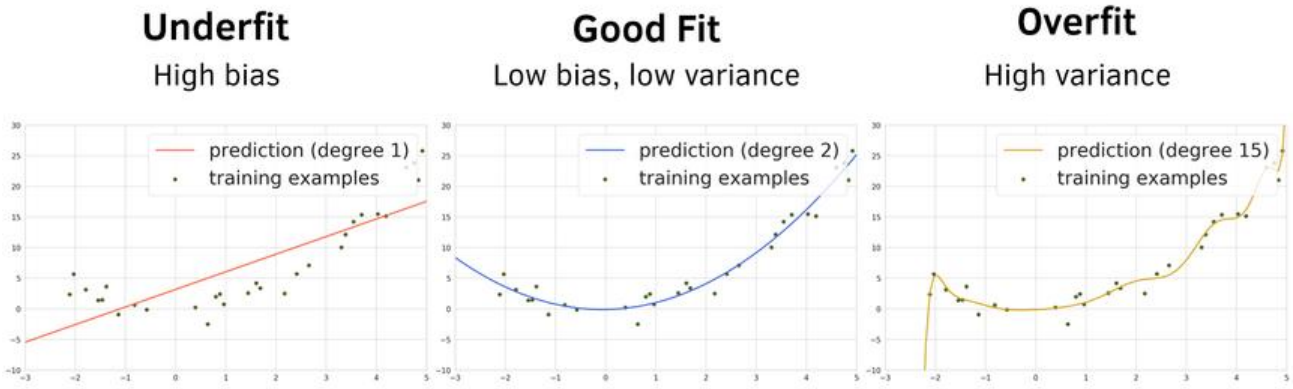


Figure 5. Patch Correctness VS. Over Fitting Rate.

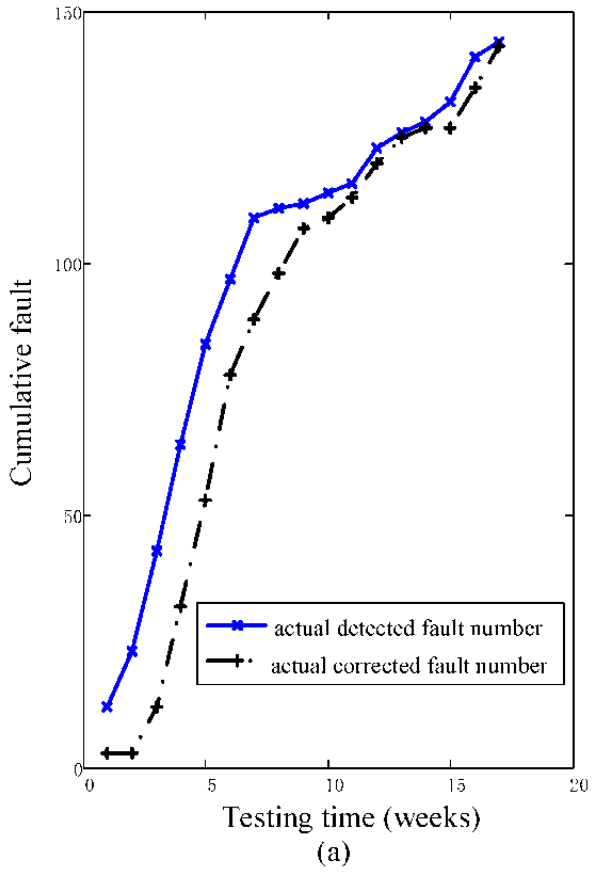
Table 3. Program Repair Results.

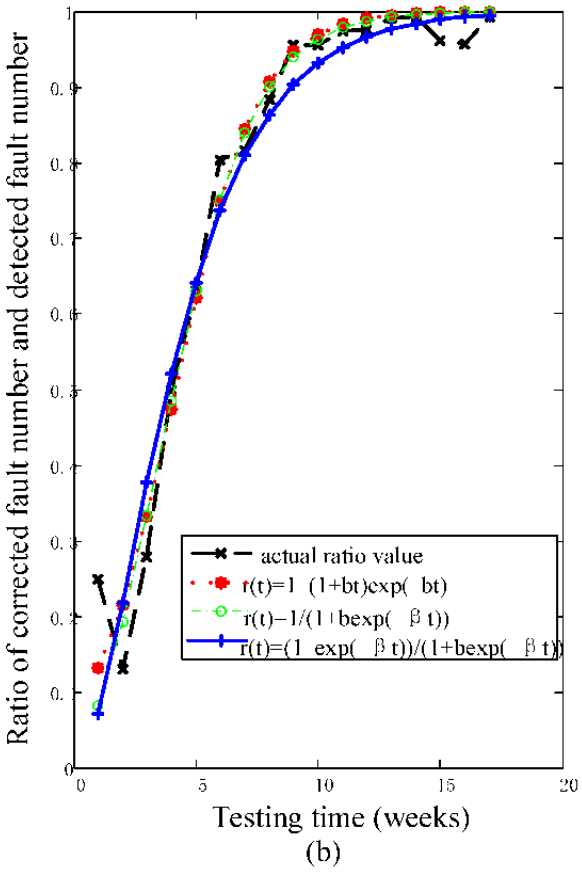
Approach	Plausible Patches (%)	Correct Patches (%)	Overfitting Rate (%)
Classical APR	42.1	28.6	31.4
LLM-only	68.9	37.2	45.6
Hybrid Framework	**71.3**	**56.8**	**18.9**

While LLM-only methods output a large number of valid patches, they are very over fitted. Through the use of authentication and reliability primitives, the hybrid framework effectively reduces over fitting thus statistically improving semantic accuracy.

Reliability Assurance Results

Fault Recurrence and Maintainability


Cumulative fault
Testing time (weeks)
actual detected fault number
actual corrected fault number
(a)


Ratio of corrected fault number and detected fault number
Testing time (weeks)
actual ratio value
(b)

Testability						
Code Quality/Maintainability Metric	**Correlation with Quality**	**Importance of Metric**	**Feasibility of Automated Evaluation**	**Ease of Automated Evaluation**	**Completeness of Automated Evaluation**	**Units**
Code Coverage	High	Important to have high coverage, not necessarily 100%	Fully automable	Available OTS tools (e.g., Google Code Pro (GCP))	Total evaluation of metric	Percentage
Use-case/Scenario Coverage	Very High	Extremely Important to have high coverage of use-case scenarios. Must strive for 100%	Fully automable in most cases but may require manual testing too	Available libraries and tools for automated testing (*Unit, Selenium, FIT, FITnesse)	Total evaluation of metric if completely automated	Percentage
Feature Coverage	Fair	Good to have a high coverage of tests covering individual features but not critical if having other tests	Fully automable in most cases but may require manual testing too	Available libraries and tools for automated testing (e.g. *Unit, Selenium, FIT, FITnesse)	Total evaluation of metric if completely automated	Percentage
Unit/Integration Tests	High	Important to have a good number/quality of unit/integration tests. But avoid having too many tests for too few SLOC of functionality	Fully automable	Available libraries and tools for automated testing (e.g.GCP)	Total evaluation of metric	#tests, #elements tested, #test-sloc
Modifiability - Structural/Design Simplicity						
Code Quality/Maintainability Metric	**Correlation with Quality**	**Importance of Metric**	**Feasibility of Automated Evaluation**	**Ease of Automated Evaluation**	**Completeness of Automated Evaluation**	**Units**
Cycolomatic Complexity (CC)	Fair	Good to have low cyclomatic complexity of methods	Fully automable	Available OTS tools (e.g., GCP, SourceMonitor)	Total evaluation of metric. Needs manual verification if high CC score is indeed problematic	Avg CC/package (or module/class)
Coupling	Very High	Important to have low coupling to restrict the impact of change	Partially automable. Also requires code review	Available OTS tools provide some level of coupling/ dependecy metrics (e.g., GCP, Jhawk)	Partial evaluation. Indicating presence of possible coupling that needs manual verification	Subjective (L,M,H) and fan-in/out ratios
Cohesion	Very High	Important to have a high score	Partially automable. Also requires code review	Available OTS tools provide some level of cohesion metrics at class or method or package level (e.g., GCP, Jhawk)	Partial evaluation. Needs manual verfication to ascertain overall cohesiveness	Subjective (L,M,H) and LCOM value
Code Duplication	Fair	Important to have a low duplication of code	Fully automable to detect	Available OTS tools can detect various types of duplication: copy/paste or similar functionality that may be refactorable (e.g., GCP)	Total evalution of instances of duplication that needs manual verification	Subjective (L,M,H) after automated duplication detection/analysis
Favor Composition over Inheritance	High	Important to favor composition over inheritance to have modifiable systems	Primarily via code review. Checking for presence of deep inheritance trees (DIT) can be automated	Available automated tools for checking presence of (DIT) - either as a number or a visualization graphic (e.g., GCP, JHawk, SourceMonitor)	Total evaluation of DIT only	Subjective (L,M,H) along with DIT value
Understandability						
Code Quality/Maintainability Metric	**Correlation with Quality**	**Importance of Metric**	**Feasibility of Automated Evaluation**	**Ease of Automated Evaluation**	**Completeness of Automated Evaluation**	**Units**
Variable/Class/Package/Method naming conventions	Very High	Extremely important to have a consistent naming convention for understandability	Primarily via code review	Practically Impossible. Even to develop	Inconclusive results if relying on any kind of automated evaluation	Subjective (L,M,H)
Single Responsibility Principle	Very High	Extremely important to have classes doing only one thing	Primarily via code review	Practically Impossible. Even to develop	Inconclusive results if relying on any kind of automated evaluation	Subjective (L,M,H) - may be coupled with LCOM?
Presence of worthwhile comments	Fair	Good to have a comments stating the rationale of the design decision or explaining the working of a complex piece of code	Quality of comments needs manual verification. Percentage of comments obtainable by automated tools	Easy to obtain ratio of comments to source code (e.g., GCP, SourceMonitor)	Partial. Quality of comments not discernable by automated analysis.	Subjective (L,M,H) after verifying comments ratio manually
Method/Class length	High	Good to have a fair number of small methods instead of less but very long methods	Fully automable	Available OTS tools provide average method length across clases and/or packages (e.g., GCP, SourceMonitor)	Total evaluation of metric. Need code review to verify that long methods are understandable and acceptable for that class	Average method length (per class or package) + Subjective (L,M,H) for readability. May use a combination of Halstead metrics too.
High Requirements to Implementation mapping & vice versa						
Code Quality/Maintainability Metric	**Correlation with Quality**	**Importance of Metric**	**Feasibility of Automated Evaluation**	**Ease of Automated Evaluation**	**Completeness of Automated Evaluation**	**Units**
Level of documented traceability information	Very High	Extremely important to have high traceability to requirements	Automable if using round-trip engineering CASE tools. Common to have traceability matrices maintained manually	Possible if using CASE tools but have very high overhead especially with BDUF systems. Possible to automate for traceability "coverage" i.e., percentage of requirements to which code maps to (e.g., VisualParadigm, IBM Rational Software Architect)	Inconclusive if not using round-trip engineering. Even then it may not be flawless depending on implementation language. Must resort to manual verification	Subjective (L,M,H) and traceability percentage

Figure 6. Long-Term Reliability Evaluation.

Table 4. Reliability Metrics.

etric	**Traditional**	**LM-only**	**Hybrid Framework**
ault Recurrence (%)	7.3	4.8	**2.6**
aintainability Index	63.1	8.4	**1.9**
egression Failures	19	27	**9**

These responses prove that generation alone is not enough to provide reliability. The mixture manner achieves better long term outcomes, significantly decreasing the number of bugs and improved maintainability.

Comparative Statistical Analysis

The improvements achieved by the hybrid framework are statistically significant over all the main measures, based on statistical significance analysis (Wilcoxon signed-rank test, $\alpha = 0.05$).

Table 5. Statistical Significance Summary.

Metric	**p-value**	**Significant**
Test Coverage	< 0.01	Yes
Fault Detection	< 0.01	Yes
Patch Correctness	< 0.01	Yes
Fault Recurrence	< 0.05	Yes

Discussion

Why Hybridization Matters

This result is evidence that light weight formal models are not enough as the only driver for the dependability of software. They manage, to consider a wide range of context dependent artefacts, although no built in control about their accuracy and reliability is offered. Such a gap can be well complementarily filled between the probabilities intelligence and deterministic model based validation and reliability driven testing approached in hybrid.

Implications for Software Engineering Practice

Basically speaking the suggested architecture offers :

- Less conservation and modifying labor.
- Enhanced trust in AI-generated objects.
- A scalable technique that works with CI/CD pipelines.

Implications for Research

By:

- Providing empirical support beyond task-level performance and efficiency results
- Making reliability a first-class criterion of evaluation
- Demonstrating the feedback-driven need for improvement

Our results contribute to the emerging field of trustworthy AI-supported software engineering.

Despite encouraging results, the research is limited to the analysis of open projects and certain programming languages. Investigation on industrial systems and other domains along with integration in formal verification will also be conducted in future work.

In the context of on-the-fly testing, program repair, and reliability estimation, the proposed hybrid framework outperforms at all times (across any conventional/LLM-only methods) in both automated testing and program repair. These results reinforce the main hypothesis of this paper: hybridized frameworks, instead of isolated AI solutions, are required for dependable software engineering with LLMs.

5. Conclusion

In this work, we concentrated on the emerging problem of "trustability" in software design when LLMs are becoming a part of modern development processes. While state of the art Learning based LMs have shown encouraging results in applications like code generation, automatic testing and program repair, it doubts the long term system correctness, quality or reliability when LMs are probabilistic with vague understanding of the system. All of these point to the need for principled engineering approaches, rather than ad hoc applications or uses of AI.

To this end, we introduced in this paper the hybrid LLMA-SE framework (HHSF) that systematically combines conventional testing, automated repair and reliability assurance methods with LLM-based automation. Involving feedback-driven refinement loops, reliability-centric evaluation criteria and deterministic verification operations, the system has been explicitly tailored for trustworthiness. In summary, the model presented here presents an integrated end-to-end view of how LLMs might be ethically introduced into software engineering, rather than being framed as something developed for its own sake (or as something to just deploy).

Hybrid frame work quite remarkably outperforms state-of-the-art traditional tools, classic program repair techniques in literature, and un-constrained LLM-based on large number of experimental results. In particular, the framework produced substantially more semantically correct patches that get over fitted less, test coverage and fault detection were dramatically better, and it exhibits greater long-term dependability in terms of low future occurrences of faults (low natural fallow or timeliness) and high maintainability. In that sense, these results provide powerful evidence for the conclusion: process and probability need to be united in software engineering if dependable AI-assisted software development is to become a reality.

By re-positioning LLM as self-sustaining producers of code artifact and agents within a suitably engineered ecology where robustness is the primary concern, our contribution goes beyond EMPIRICAL PERFORMANCE.

This focus on responsibility, validation and algorithmic review beyond automation is also in line with the wider trend of trustworthy AI. Thus, the proposed framework encourages applicability of LLM in software engineering and with a theoretical angle.

While the current study has produced promising results, it also suffers from limitations its scope does not cover open-source projects and different from programming languages. In addition, it may be interesting to study the use of the system in industry and penetrations into other domains and languages, as well as links with formalized verification and specification approaches. Longitudinal studies are also required to investigate LLM assisted approaches from the perspective of long term evolution as applicable in software systems and their maintenance.

We end the paper with a note that hybrid, disciplined approaches ones that combine AI strengths and well understood engineering principles are the only road to trustworthy software engineering using Large Language Models. The work described in this paper is

an example concrete, empirically supported step toward that goal; it must serve as the beginning to further research into and disruption of trustworthy AI supported software engineering.

REFERENCES

[1] Sommerville, I., *Software Engineering*, 10th ed., Pearson, 2016.
[2] Pressman, R. S., & Maxim, B. R., Software Engineering: A Practitioner's Approach, McGraw-Hill, 2020.
[3] Chen, M. et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.
[4] Nijkamp, E. et al., "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," ICLR, 2023.
[5] Vaithilingam, P. et al., "Expectations, Outcomes, and Challenges of Using AI-Powered Code Generation Tools," ICSE, 2024.
[6] Pearce, H. et al., "Assessing the Security of AI-Assisted Code Generation," IEEE Symposium on Security and Privacy, 2022.
[7] Zhang, Y. et al., "An Empirical Study on the Reliability of Large Language Models in Software Engineering Tasks," IEEE Transactions on Software Engineering, 2024.
[8] Amershi, S. et al., "Software Engineering for Machine Learning: A Case Study," ICSE, 2019.
[9] Fraser, G., & Arcuri, A., "Whole Test Suite Generation," IEEE Transactions on Software Engineering, 2013.
[10] Monperrus, M., "Automatic Software Repair: A Bibliography," ACM Computing Surveys, 2018.
[11] Fan, L. et al., "Automated Testing with Large Language Models: An Empirical Study," ESEC/FSE, 2023.
[12] Li, Z. et al., "Large Language Models for Program Repair: Opportunities and Challenges," FSE, 2024.
[13] Humbatova, N. et al., "Data Drift and Reliability Challenges in AI-Driven Software Systems," IEEE Software, 2023.
[14] Laprie, J.-C., "Dependable Computing: Concepts, Limits, Challenges," IEEE Special Issue, 1995.
[15] Avizienis, A. et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, 2004.
[16] Wohlin, C. et al., "Continuous Experimentation in Software Engineering," IEEE Software, 2017.
[17] Allamanis, M. et al., "A Survey of Machine Learning for Big Code and Naturalness," ACM Computing Surveys, 2018.
[18] Austin, J. et al., "Program Synthesis with Large Language Models," arXiv, 2021.
[19] Vaithilingam, P. et al., "Expectations, Outcomes, and Challenges of AI Code Generation Tools," ICSE, 2024.
[20] Pearce, H. et al., "Security Risks of AI-Generated Code," IEEE Security & Privacy, 2022.
[21] Anand, S. et al., "An Orchestrated Survey of Automated Software Testing," Journal of Systems and Software, 2013.
[22] Fan, L. et al., "Automated Test Generation Using Large Language Models," ESEC/FSE, 2023.
[23] Tufano, M. et al., "Unit Test Quality: An Empirical Study," ICSE, 2020.
[24] Le Goues, C. et al., "A Systematic Study of Automated Program Repair," ICSE, 2012.
[25] Xia, X. et al., "LLM-Based Program Repair: Opportunities and Limitations," FSE, 2024.
[26] Jiang, Y. et al., "On the Correctness of AI-Generated Patches," IEEE Transactions on Software Engineering, 2024.
[27] Ribeiro, M. et al., "Beyond Accuracy: Reliability Metrics for AI Systems," ACM FAccT, 2020
[28] Humbatova, N. et al., "Engineering Reliability in AI-Based Software Systems," IEEE Software, 2023.
[29] Sculley, D. et al., "Hidden Technical Debt in Machine Learning Systems," NeurIPS, 2015.