From Correctness to Consistency: Redefining Reliability for the Agentware Era

Xue Qin
Villanova University
Villanova, United States
xue.qin@villanova.edu

Maurício Gruppi
Villanova University
Villanova, United States
mgouveag@villanova.edu

Abstract

The rise of "vibe coding" has marginalized requirement analysis, allowing unverified "Make it Work" assumptions to accumulate into complex defects that evade standard testing. In the absence of a reliable ground truth oracle, we advocate a paradigm shift from correctness verification to logic consistency. We introduce Differential Logic Analysis (DLA), a framework that utilizes Logical Inference to detect internal contradictions across parallel and sequential development workflows. Preliminary simulations demonstrate that DLA successfully intercepts logic drift, such as contradictions and unspecified assumptions, and offers a new perspective on reliability assessment in the AI-assistant development era.

CCS Concepts

• **Software and its engineering → Formal software verification**.

Keywords

Vibe Coding, LLM-assisted Development, Software Verification, Logic Consistency, Differential Logic Analysis

ACM Reference Format:
Xue Qin and Maurício Gruppi. 2018. From Correctness to Consistency: Redefining Reliability for the Agentware Era. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The proliferation of Large Language Models (LLMs) like GitHub Copilot [10] and ChatGPT [18] is transforming software construction, enabling non-experts to build complex systems via natural language [11, 17]. This has given rise to "vibe coding," where developers orchestrate AI components through rapid iteration rather than rigorous planning [20]. However, this speed-driven approach has created a critical gap: the systematic abandonment of Requirement Analysis. Practitioners frequently bypass formal specifications, only to find that their initial instant success [23, 28] often degrades into prolonged struggles with prompt revision and complex bugs. As a result, developers increasingly report abandoning projects entirely rather than attempting to debug AI-generated logic [7].

This volatility stems from the stochastic non-determinism of LLMs [24]. To bridge the gap between prompts and execution, models inject "Make it Work" assumptions that are often explicitly or implicitly flawed [14]. This gap arises because prompts provide high-level, often underspecified instructions, while the resulting implementations require implicit assumptions to become executable. Consequently, iterations compound hallucinations [31] rather than refining them. Testing code generated by an LLM-assistant fails to break this cycle due to an inherent logical bias: when the same model generates both code and tests, the latter inherits the same unverified assumptions [15], creating a self-validating loop.

Moreover, the gap between prompt and execution exposes a deeper challenge of the Agentware era: the absence of a reliable ground truth. In traditional software engineering, implementation correctness can be validated against formal specifications and detailed requirements. An AI coding assistant may make assumptions that do not reflect the developer's intent and are not grounded in an explicit specification. There is no authoritative reference against which the assumptions can be verified. As a result, the intended system behavior becomes ambiguous and potentially conflicted. We therefore advocate a paradigm shift from correctness verification to *logic consistency*. Rather than asking if artifacts are "correct" relative to a missing specification, we examine whether their internal logical commitments remain coherent. To operationalize this, we introduce **Differential Logic Analysis (DLA)**, a logic inference framework [27] that treats prompts and code as comparable premises to detect logical divergence. By identifying contradictions and underspecified interactions, DLA intercepts drift across both parallel and sequential development scenarios. To explore the viability of this framework, we address two core research questions:

- RQ1: Can DLA identify logical contradictions in LLM artifacts that bypass auto-generated test suites?
- RQ2: How does logic divergence manifest differently across various workflows?

To answer the questions, we conducted two case studies: 1) Parallel Development: Where multiple agents independently implement related features, creating the potential for logic divergence [13] in isolation (e.g., assumptions made by agents work locally but conflict globally). 2) Sequential Iteration: Where a single agent refines a codebase over multiple turns, creating the potential for logic drift over iterations (e.g., new-turn logic diverges from prior logic). We then manually applied DLA rules to the artifacts generated in these simulations. By treating the natural language prompts and code

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

segments as logical premises, we tested whether Logical Inference, specifically *Entailment* (logical consequence), *Contradiction*, and *Neutrality* [16], could successfully contextualize software logic. Our analysis confirmed that DLA could systematically identify "logic divergence" and "unspecified assumptions" that auto-generated tests failed to catch, proving that **Logic Consistency** is a viable and measurable proxy for reliability in the absence of a ground truth. In this vision paper, we contribute:

- **A New Perspective (Consistency > Correctness):** We advocate shifting correctness-based verification with logic consistency checks to ensure coherence when formal oracles are unavailable.
- **The Differential Logic Analysis (DLA) Framework (RQ1):** We introduce a logic-inference methodology adapted from formal methods [12] to detect conflicting assumptions in parallel and sequential AI-assisted development.
- **Feasibility Evidence (RQ2):** We provide empirical evidence that logic divergence is a distinct, detectable failure mode that persists even when code passes syntax and runtime checks.

2 Study Methodology

2.1 Logic-Centric View of AI-Assisted Artifacts

Traditional software engineering relies on rigorous specifications to bridge the gap between intent and code. In contrast, AI-assisted development often bypasses formal requirements, relying on Large Language Models (LLMs) to bridge sparse natural language prompts to executable instructions. This shift introduces a "Logic-Consistency" problem where the LLM must fill specification gaps with unverified assumptions. To support this new development lifecycle, we propose a *Differential Logic Analysis (DLA)* to audit the alignment among artifacts. In particular, we define the software artifact into three interconnected logic layers:

Intent Logic (P, Requirements): The high-level "what." This layer is often underspecified and lacks the formal constraints (e.g., state-priority rules) necessary for robust execution [8]. P consists of a set of logical statements (*premises*) that can be traced back to the user prompt.

Operational Logic (H, Code): The "how." A hybrid logic where LLM inference formalizes P. To be executable, H often embeds *unverified assumptions* that may diverge from user intent. H contains the set of premises P and may contain additional statements assumed by the agent.

Validation Logic (V, Tests): The verification boundary. Because tests are often generated by the same context, V typically inherits the same implicit assumptions as H. This leads to *Oracle-Implementation Coupling*, in which the test suite reflects the implementation's internal logic rather than the original requirements, thereby masking fundamental logic gaps.

The proposed DLA performs cross-artifact verification to ensure collective operational logic remains consistent with the aggregate intent. By applying *logical inference rules*, DLA identifies conflicts and unverified assumptions, such as contradictory premises or unsupported expansions, across multi-agent and multi-turn software evolutions. The goal of DLA is to identify cases where a statement must be verified by the user. When a logical statement S in the generated program does not come directly from the premises in P, DLA will check whether: (i) if S can be logically deduced from the premises in P and (ii) if S contradicts any of the other premises. If either (i) or (ii) fail, the system issues a notice prompting the user to verify if S is a desired assumption.

2.2 Study Design

To empirically observe how logic inconsistencies emerge in AI-assisted workflows, we designed two controlled experiments that mimic standard software development patterns: *Parallel Development*, where multiple distinct features are built simultaneously by different agents (simulating a collaboration team), and *Sequential Development*, where features are built iteratively on top of previous code (simulating an agile workflow). We selected two representative scenarios to cover different categories of software logic: Design A represents a Hospital Door Control System (safety-critical access-control logic), and Design B represents an E-commerce Wallet (transactional state-management logic). These domains allow us to examine how logic divergence emerges under underspecified requirements, where implicit assumptions introduced during development can significantly impact system behavior. By covering distinct logic categories, we aim to demonstrate that DLA is applicable across different types of software systems rather than being limited to a specific domain.

2.2.1 Design A: Parallel Feature Development (Logic Divergence in Isolation). **Objective:** This design investigates *Logic Divergence* in multi-agent environments. When agents implement separate features independently, they often infer conflicting rules to bridge gaps in shared-state interactions. We hypothesize that this results in components that are locally correct but globally contradictory. **Instantiation:** We simulate two LLM agents building a *Hospital Door Control* system. Agent A implements an "Emergency Override" (fail-open on Fire), while Agent B implements a "Biohazard Protocol" (fail-shut on Virus). The prompts are not synchronized across agents, as we assume the two developments are independent. **Measurement:** We employ DLA to detect logical contradictions between the operational behaviors of independent agents (H_A and H_B). The analysis evaluates whether integrating these agents into a **shared state** introduces state-transition conflicts that satisfy local verification but compromise global system consistency.

Parallel Prompt Requirements:

Agent A: Implement an emergency override. If a 'Code Red' is active, all electronic doors must **fail-open** for safety.

Agent B: Implement a quarantine protocol. If a 'Biohazard' alert is active, all electronic doors must **lock** to prevent spread.

2.2.2 Design B: Sequential Feature Development (Logic Drift in Iteration). **Objective:** This design investigates *Logic Drift* in iterative development. We hypothesize that agents evolve code by injecting *unverified assumptions* to bridge feature gaps, leading to a silent override of foundational rules to satisfy new requests without developer awareness. **Instantiation:** We simulate a single LLM agent evolving an *E-commerce Wallet* across two turns. In Turn 1, the agent implements a baseline withdrawal logic. In Turn 2, it is tasked with adding a security constraint (fraud-prevention lock). The agent maintains access to the codebase and prompt history from the previous turn. **Measurement:** We employ DLA to detect **unverified**

logic injected when the agent adapts the initial operational behavior (account withdrawal) to accommodate new constraints (account locking). Specifically, we evaluate logic drift by comparing the Turn 2 implementation (H) against the accumulated requirements ($P_{T1} \wedge P_{T2}$) to determine whether the system has introduced implicit assumptions that diverge from the established specifications.

Sequential Prompt Flow:

Turn 1: Write a `UserWallet` class. Users must be able to withdraw funds instantly as long as they have a positive balance.

Turn 2: Update the class to include password changes. To prevent fraud, a password change must **lock** the account for 24 hours. The user can also check the account status.

2.3 DLA via Logical Inference

To evaluate logic consistency beyond functional correctness, we employ DLA grounded in formal logical inference. Because auto-generated oracles in validation logic (V) frequently suffer from operational logic bias, DLA provides an independent audit by treating implementation and requirements as structured **Logic Artifacts**. Unlike traditional ground truth-oriented correctness checking (where Code is checked against Requirements [9, 21]), we map these artifacts to a **Premise (P)** and a **Hypothesis (H)** to evaluate their *logical alignment* based on the following relationship categories. Note that these category terms originate from NLI, and we explain them in the SE context. For DLA, we perform a manual audit to identify the core rules and safety constraints within the code. This allows us to pinpoint exactly where the program's behavior changes or drifts away from the requirements:

- **Entailment (Strict Alignment):** The operational rules in H are logically necessitated by P. The implementation strictly adheres to the provided specifications logic without further verification.
- **Contradiction (Logic Conflict):** The logic in multiple H contradict with reference of P.
  - In **Design A**, we compare H_A with H_B to detect whether independent agents propose incompatible transitions for a shared state (e.g., $Open \wedge Locked$).
- **Neutral (Unverified Assumption):** H introduces operational rules that are consistent with but **not** necessitated by P. This means the implementation code is performing unverified actions.
  - In **Design B**, we evaluate whether the most updated implementation logic H after iterations can be inferred from all the requirements (P_{T1} and P_{T2}).

3 Pilot Study Results

3.1 Study Setup

In this preliminary study, we use GPT-5.2 with a clear history and memory to generate all the code. In a multi-agent scenario, multiple GPT models were initialized to complete each task independently, without sharing context with one another.

Each design scenario was executed once per workflow configuration (parallel development and sequential iteration). The analysis then evaluated whether DLA could identify logical relationships, such as entailment, contradiction, and neutrality, across these artifacts. We agree that evaluating an LLM result typically requires multiple runs to demonstrate the approach's robustness. However, our primary goal is to demonstrate the feasibility of logic-consistency analysis as a verification signal, rather than to estimate the frequency of inconsistencies. Therefore, we did not focus on repeating code generation.

The test cases were generated using the same LLM model as the code generation, but from different instances in separate temporary chat sessions. After generating the code artifact, we initiated a new prompt asking the model to produce unit tests by providing both the original task prompt and the generated source code. This setup reflects a common AI-assisted workflow where developers rely on the same model but different instances or sections to generate tests for implementations.

3.2 Logic Conflict in Parallel Development

Design A evaluates logic divergence across isolated agents during the parallel development of a door control system. Each agent was tasked with a distinct safety feature, but the global state management was left underspecified.

Conflict Observation: Each agent successfully implemented its local requirement but "hallucinated" a priority rule regarding its dominance over the system state. As shown in Listing 1, Agent A (Safety) implements a *fail-open* logic that assumes `CodeRed` overrides all locks. Simultaneously, Agent B (Security) implements a *fail-shut* logic that assumes `Biohazard` overrides all opens. This creates a hidden collision: each agent treats its assigned condition as an absolute mandate, blindly overriding the global state without awareness of the competing constraint.

Listing 1: Conflicting Logic Snippets (Design A)

```
# Agent A (Safety-First)
if self.code_red_active:
    door.is_locked = False
    door.is_open = True
    return

# Agent B (Security-First)
if self.biohazard_alert_active:
    door.is_locked = True
    return False
```

Alert State	Agent A (H_A)	Agent B (H_B)	DLA Verdict
$CR = 1, BH = 0$	**Open**	–	Consistent
$CR = 0, BH = 1$	–	**Locked**	Consistent
$CR = 1, BH = 1$	**Force Open**	**Force Locked**	**Contradiction ($\perp$)**

Table 1: Logic Divergence in Parallel Development

Differential Logic Analysis: We formalize this divergence by extracting the operational logic from the generated code into hypotheses (H_A and H_B). Let the system states be *CodeRed* (CR) and *Biohazard* (BH).

$$\begin{aligned} H_A : CR &\implies \neg\text{Locked} \wedge \text{Open} \\ H_B : BH &\implies \text{Locked} \wedge \neg\text{Open} \end{aligned} \tag{1}$$

Individually, each agent's logic (H_A and H_B) appears correct. However, when both alert states occur simultaneously ($CR \wedge BH$), the combined system is forced into a contradiction where ($H_A \wedge H_B$) implies (Locked $\wedge$ ¬Locked). This represents a critical **Logic Conflict** where local correctness masks global incompatibility, as illustrated in Table 1.

Listing 2: Logic Drift Snippets (Design B)

```
1  # --- Turn 1 ---
2  def withdraw(self, amount):
3    if self._balance <= 0:
4      raise RuntimeError("Balance not positive")
5    self._balance -= amount; return amount
6
7  # --- Turn 2 ---
8  def withdraw(self, amount):
9    """ Instant withdrawal is always allowed...
10   even if account is 'locked' (platform rule). """
11   if self._balance <= 0: # Logic unchanged (
         Symptom 1)
12     raise RuntimeError("Balance not positive")
13   ...
14
15 def change_password(self, ...):
16   """ NOTE: Lock does NOT block withdrawals (
         Symptom 2)
17   (per platform rule) """
18   ...
19   self._lock_until = now + timedelta(hours=24)
```

3.3 Logic Drift in Iterative Refinement

Design B illustrates how logic divergence emerges during iterative development. The same agent was tasked to evolve a wallet system by adding a "Lock" feature (Turn 2) to an existing "Withdraw" feature (Turn 1).

Conflict Observation: We observe two distinct symptoms where the agent attempts to reconcile the invariants of Turn 1 with the new constraints of Turn 2. *Symptom 1:* Despite the introduction of a locking mechanism in Turn 2, the `withdraw` method remains logically identical to its Turn 1 version (Listing 2, line 1 and line 7). The agent resolves the ambiguity between high availability and security by maintaining the status quo, thereby silently allowing the original logic to override the new state constraint. *Symptom 2:* In the newly generated `change_password` method, the agent explicitly documents this decision as a "platform rule" (lines 15-16). This comment represents a hallucinated interaction rule that was never specified in the user prompt but was fabricated to resolve the logical tension. Without further verification, these symptoms represent a high-risk ambiguity: the code's operational behavior (allowing withdrawal) directly contradicts the semantic implication of the term "Locked."

Differential Logic Analysis: To understand the root of these symptoms, we analyze the logical relationship between the Requirements (P) and the Implementation (H). A reasonable implementation should be a logical consequence of the requirements ($P \vdash H$). However, our analysis reveals a critical gap:

$$\begin{aligned} P_{\text{T1}} &: \text{has\_balance} \implies \text{can\_withdraw} \\ P_{\text{T2}} &: \text{pw\_changed} \implies \text{is\_locked} \\ H_{\text{Code}} &: \text{is\_locked} \implies \text{can\_withdraw} \therefore \{P_{\text{T1}}, P_{\text{T2}}\} \quad \nvdash H_{\text{Code}} \end{aligned} \tag{2}$$

The derivation highlights a critical specification gap: the two requirements (P_{T1} and P_{T2}) define the features individually but remain silent on their interaction. When facing this ambiguity, the agent injects a hidden assumption to make the code executable. In Logical Inference, this is defined as *Assumption of Independence* (A_{gap}): *Locking the account does not disable withdrawal.* Hence, the agent *assumes* H_{Code}. By formalizing this gap, we identify that the "bug" is not a coding error, but an unverified design decision.

To demonstrate that this behavior is an unverified assumption rather than a logical bug, we map the derivation path in Table 2. While the individual requirements (P_{T1} and P_{T2}) are verified permises, their interaction is undefined in the prompt. The resulting logic in H_{Code} (shown in Implementation row) is classified as an **Unverified Assumption**. Notably, even the safety alternative (the account is locked and withdrawal is blocked) would similarly require verification, as it too relies on an unstated priority rule.

Context	Logic Proposition	Status
Turn 1 (P_{T1})	has_balance $\implies$ can_withdraw	✓Premises
Turn 2 (P_{T2})	pw_chg $\implies$ is_locked	✓Premises
Gap	$P_{T1} \wedge P_{T2} \implies$?	**? Undefined**
Impl. (H)	is_locked $\implies$ can_withdraw	**! Assumption**
Alternative	is_locked $\implies$ ¬can_withdraw	*! Assumption*

Table 2: Explicit Requirements vs. Unverified Assumptions

3.4 Why Tests Fail to Reveal Logic Errors

Traditional software quality metrics, specifically line coverage and test pass rates, provide a false sense of security in LLM-assisted development. We demonstrate that high coverage is often an illusion of correctness, as generated test cases frequently mirror the logical biases inherent in the implementation.

Subject	Tests	Pass	Cov.
Design A (Agent A)	7	100%	94.4%
Design A (Agent B)	7	100%	95.8%
Design B (Turn 1)	5	100%	100%
Design B (Turn 2)	12	100%	98.2%

Table 3: Automated test code metrics

3.4.1 The Illusion of Coverage. As shown in Table 3, all agents produced robust-looking test suites that, in a traditional CI/CD pipeline, would likely trigger automatic approval for deployment.

The reported coverage reflects standard structural coverage (e.g., statement or branch coverage), indicating that most executable paths in the code are exercised by the generated tests. However, high coverage in this context does not guarantee that the underlying logic is correctly validated. Instead, it only confirms that the tests execute the implemented behavior, which may already embed incorrect or incomplete assumptions. Because the test cases are generated from the same prompt and code artifacts, they tend to align with the implementation's implicit logic rather than independently validating intended behavior. As a result, even with near-complete coverage and all tests passing, logical inconsistencies and requirement violations can remain undetected.

Listing 3: Bias Test Case Design

```
# Agent A (Design A): Asserts "Always Open" during Code Red
self.ctrl.set_code_red(True)
self.assertTrue(self.ctrl.doors["D1"].is_open)

# Agent B (Design A): Asserts "Always Locked" during Biohazard
self.controller.set_biohazard_alert(True)
self.assertTrue(self.controller.doors["ER-1"].is_locked)

# Design B: Verifying "Withdrawals remain instant while locked."
w3.change_password("A1!", "B2!", now=t0)
assert w3.get_status(now=t0).state == "locked"
got = w3.withdraw(5) # withdraw should still work
assert got == 5
```

3.4.2 Deceptive Tests: Validating the Bias Logic. As shown in Listing 3, by providing the same prompt and operational code, the auto-generated test cases follow the implementation's bias and falsely validate the agent's intention. This phenomenon illustrates **Oracle-Implementation Coupling**, where the Validation Logic V directly copies the Operational Logic H. As shown in Listing 3 from lines 9-13, the test validates a *hallucinated* platform rule from Design B rather than probing the missing security constraint.

4 DLA Approach Discussion

Feasibility and Scalability. In the manual analysis, our approach does not attempt to enumerate or verify all possible assumptions; instead, it focuses on identifying logical inconsistencies among generated artifacts, such as prompts and code. In practice, the analysis operates on observed artifacts rather than the full state space of possible behaviors, which distinguishes it from exhaustive verification techniques that may suffer from state explosion. The goal is to detect contradictions, mismatches, or implicit assumptions that arise as development progresses. Therefore, DLA can be applied incrementally as part of iterative development workflows. The analysis can be triggered when new prompts, code updates, or test artifacts are generated, allowing consistency checks to focus only on newly introduced or modified logic. This incremental design helps keep the computational workload manageable even for larger systems with extensive codebases or complex specifications.

Generalizability Across Development Frameworks. The two designed studies simulate Parallel Development and Sequential Iteration to capture two coordination patterns in software development. These patterns remain fundamental in modern agentic software engineering systems, where multiple agents may work concurrently on related components, and artifacts are iteratively refined through successive reasoning and generation steps. Conceptually, this abstraction allows DLA to be compatible with a wide range of development frameworks beyond the specific workflows. For example, it can be integrated as a validation step between agent outputs or as part of a continuous integration process to examine logical relationships across generated artifacts.

Automation and Integration. In this study, we illustrate how DLA identifies relationships such as entailment, contradiction, and neutrality across artifacts. The primary goal is to demonstrate the feasibility of logic-consistency analysis as a verification signal in LLM-assisted development. However, DLA is designed to be automatable. The analysis consists of three steps that can be later integrated into automated workflows: (1) extracting logical premises from prompt, code, or tests; (2) forming premise pairs across artifacts (e.g., prompt–code, code–test, or code–code); and (3) applying logical inference to classify relationships such as entailment, contradiction, or neutrality. The first two steps can be automated using established NLP and program analysis techniques, such as natural language–to–logic translation [30] and code analysis [5], while the final step can be automated using natural language inference models or logic-based reasoning techniques.

5 Related Work

Prior work shows that LLMs struggle with test generation under limited context [29]. Testing-driven workflows with intent clarification can improve outcomes [6], while strong oracle generation is observed when rich context is available [1]. More broadly, LLMs support a range of software engineering tasks, including automated program repair, code generation, and development activities such as implementation, testing, and debugging [22, 25]. Together, these findings indicate that accurate LLM-driven test generation depends on rich context, which vibe coding often lacks. LLM-based specification generation translates natural-language requirements into structured artifacts and has shown promise in domains such as embedded and automotive software [2, 3, 19]. However, developers increasingly adopt informal "vibe coding" [7] due to its convenience, despite its limited support for rigorous logical specification.

Formal logic has been successfully applied to a wide range of software and hardware verification tasks, enabling rigorous reasoning about system correctness. Recent advances illustrate how deductive systems – such as Hoare logic – continue to evolve [4]. Formal deductive methods have also been applied to software engineering workflows [27], improving the usability of specification languages; and new dataset-driven approaches that apply model checking to large corpora of AI-generated programs for security analysis, integrating formal logic with modern machine-learning pipelines [26].

6 Conclusion and Future Work

This paper identifies logic drift as the primary risk of "vibe coding," in which LLMs introduce unverified assumptions to bridge specification gaps. Since AI-generated tests often mirror the implementation's logical biases, the AIware community must shift from checking correctness against missing oracles to ensuring internal **Logic Consistency**. Our Differential Logic Analysis (DLA) framework operationalizes this by detecting contradictions that execution-based QA misses. Future work will expand this vision through larger empirical studies and the application of DLA to complex, multi-developer workflows to ensure coherence in heterogeneous agent-led environments.

References

[1] Adam Bodicoat, Gunel Jahangirova, and Valerio Terragni. 2025. Understanding LLM-Driven Test Oracle Generation. In *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*. 29–39. doi:10.1109/AIware69974.2025.00011

[2] Zehan Chen, Long Zhang, Zhiwei Zhang, JingJing Zhang, Ruoyu Zhou, Yulong Shen, JianFeng Ma, and Lin Yang. 2026. Beyond Basic Specifications? A Systematic Study of Logical Constructs in LLM-based Specification Generation. arXiv:2602.00715 [cs.SE] https://arxiv.org/abs/2602.00715

[3] Zehan Chen, Long Zhang, Zhiwei Zhang, JingJing Zhang, Ruoyu Zhou, Yulong Shen, JianFeng Ma, and Lin Yang. 2026. Enhancing LLM-based Specification Generation via Program Slicing and Logical Deletion. arXiv:2509.09917 [cs.SE] https://arxiv.org/abs/2509.09917

[4] Thibault Dardinier and Peter Müller. 2024. Hyper hoare logic:(dis-) proving program hyperproperties. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1485–1509.

[5] Shijia Dong, Haoruo Zhao, and Paul Harvey. 2026. Code vs Serialized AST Inputs for LLM-Based Code Summarization: An Empirical Study. arXiv:2602.06671 [cs.SE] https://arxiv.org/abs/2602.06671

[6] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2254–2268. doi:10.1109/TSE.2024.3428972

[7] Ahmed Fawzy, Amjed Tahir, and Kelly Blincoe. 2025. Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook – a Grey Literature Review. arXiv:2510.00328 [cs.SE] https://arxiv.org/abs/2510.00328

[8] Molly Q. Feldman and Carolyn Jane Anderson. 2024. Non-Expert Programmers in the Generative AI Future. In *Proceedings of the 3rd Annual Meeting of the Symposium on Human-Computer Interaction for Work* (Newcastle upon Tyne, United Kingdom) *(CHIWORK '24)*. Association for Computing Machinery, New York, NY, USA, Article 15, 19 pages. doi:10.1145/3663384.3663393

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[10] GitHub. 2026. GitHub Copilot. https://github.com/features/copilot Accessed: 2026-02-12.

[11] Philipp Haindl and Gerald Weinberger. 2024. Does ChatGPT Help Novice Programmers Write Better Code? Results from Static Code Analysis. *IEEE Access* PP (01 2024), 1–1. doi:10.1109/ACCESS.2024.3445432

[12] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. doi:10.1145/363235.363259

[13] Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and Reasoning about Systems* (2nd ed.). Cambridge University Press, USA.

[14] Sadia Jahan and Xiaoyin Wang. 2025. How Does ChatGPT Make Assumptions When Creating Erroneous Programs?. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://conf.researchr.org/details/ase-2025/ase-2025-nier-track/6/How-Does-ChatGPT-Make-Assumptions-When-Creating-Erroneous-Programs- NIER Track, To appear.

[15] Zihan Ma, Taolin Zhang, Maosong Cao, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. 2025. Rethinking Verification for LLM Code Generation: From Generation to Testing. arXiv:2507.06920 [cs.CL] https://arxiv.org/abs/2507.06920

[16] Bill MacCartney and Christopher D. Manning. 2008. Modeling semantic containment and exclusion in natural language inference. In *Proceedings of the 22nd International Conference on Computational Linguistics - Volume 1* (Manchester, United Kingdom) *(COLING '08)*. Association for Computational Linguistics, USA, 521–528.

[17] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[18] OpenAI. 2026. ChatGPT. https://chat.openai.com Accessed: 2026-02-12.

[19] Minal Suresh Patil, Gustav Ung, and Mattias Nyberg. 2025. Towards Specification-Driven LLM-Based Generation of Embedded Automotive Software. In *Bridging the Gap Between AI and Reality: Second International Conference, AISoLA 2024, Crete, Greece, October 30 – November 3, 2024, Proceedings* (Crete, Greece). Springer-Verlag, Berlin, Heidelberg, 125–144. doi:10.1007/978-3-031-75434-0_9

[20] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv:2302.06590 [cs.SE] https://arxiv.org/abs/2302.06590

[21] Michael Rath, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Mäder. 2018. Traceability in the Wild: Automatically Augmenting Incomplete Trace Links. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 834–845. doi:10.1145/3180155.3180207

[22] Maria Deolinda Santana, Cleyton Magalhaes, and Ronnie de Souza Santos. 2025. Software Testing with Large Language Models: An Interview Study with Practitioners. arXiv:2510.17164 [cs.SE] https://arxiv.org/abs/2510.17164

[23] Tyler Shields. 2024. My First Attempt at Vibe Coding. https://www.techtarget.com/searchapparchitecture/opinion/My-first-attempt-at-vibe-coding. Accessed: 2026-02-12.

[24] Yifan Song, Guoyin Wang, Sujian Li, and Bill Yuchen Lin. 2025. The Good, The Bad, and The Greedy: Evaluation of LLMs Should Not Ignore Non-Determinism. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 4195–4206. doi:10.18653/v1/2025.naacl-long.211

[25] Hidetake Tanaka, Haruto Tanaka, Kazumasa Shimari, and Kenichi Matsumoto. 2025. Understanding the Characteristics of LLM-Generated Property-Based Tests in Exploring Edge Cases. arXiv:2510.25297 [cs.SE] https://arxiv.org/abs/2510.25297

[26] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C Cordeiro, and Vasileios Mavroeidis. 2023. The formai dataset: Generative ai in software security through the lens of formal verification. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*. 33–43.

[27] Nan Wu, Yingjie Li, Hang Yang, Hanqiu Chen, Steve Dai, Cong Hao, Cunxi Yu, and Yuan Xie. 2024. Survey of machine learning for software-assisted hardware design verification: Past, present, and prospect. *ACM Transactions on Design Automation of Electronic Systems* 29, 4 (2024), 1–42.

[28] Angela Yang. 2025. Noncoders Are Using AI Prompts and V̈ibe Coding T̈o Build Apps. https://www.nbcnews.com/tech/tech-news/noncoders-ai-prompt-ideas-vibe-coding-rcna205661. Accessed: 2026-02-12.

[29] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1607–1619. doi:10.1145/3691620.3695529

[30] Jiansong Zhang and Nora M. El-Gohary. 2017. Integrating semantic NLP and logic reasoning into a unified system for fully-automated code checking. *Automation in Construction* 73 (2017), 45–57. doi:10.1016/j.autcon.2016.08.027

[31] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA022 (June 2025), 23 pages. doi:10.1145/3728894