IPRPD
International Journal of Arts, Humanities and Social Sciences
ISSN 2693-2547 (Print), 2693-2555 (Online)
Volume 06; Issue no 04: April 2026
DOI: 10.56734/ijahss.v7n4a8

IS AI REALLY INTELLIGENT? PRACTICAL INSIGHTS FROM REAL-WORLD USE OF GENERATIVE AI

Dr. Khaled EL Tannir[1]

[1]*Artificial Intelligence Researcher, Montréal, Canada*

Abstract

Generative AI systems based on Large Language Models (LLMs) have demonstrated remarkable capabilities in software engineering tasks, from code generation to natural language processing. However, a significant gap persists between curated demonstrations and production-grade deployment. This technical report presents a practitioner-driven analysis of LLM limitations encountered during sustained, real-world use across software development workflows. Drawing from multiple case studies—including natural-language-driven development ("vibe coding"), API integration, multi-file refactoring, and general-purpose question answering—we identify and taxonomize four critical failure modes: (1) the Complexity Cliff, where LLM performance degrades non-linearly as task interdependency grows; (2) Context Window Blindness, where finite attention spans cause silent contract violations across distributed codebases; (3) the Memory Illusion, where session discontinuity erases accumulated architectural knowledge; and (4) Confident Hallucination, where models generate plausible but fabricated outputs indistinguishable in tone from correct ones. We formalize the Verification Paradox—an inverse relationship between a user's need for AI assistance and their capacity to validate its outputs—and propose a practical five-strategy framework for effective human–AI collaboration in software engineering contexts. We further introduce the concept of Contextual Reasoning Failure, evidenced by cases where LLMs optimize for literal query patterns while ignoring situational logic obvious to any human observer. Our findings suggest that current LLMs, while powerful pattern-matching engines, lack the contextual reasoning, persistent memory, and epistemic self-awareness necessary for reliable autonomous operation, and that practitioner expertise remains the critical safeguard against AI-induced defects in production systems.

Keywords

Large Language Models; Generative AI; Software Engineering; Vibe Coding; Hallucination; Context Window; Human–AI Collaboration; Practitioner Report

1. Introduction

The rapid advancement of Large Language Models (LLMs) has produced a wave of enthusiasm across the software engineering community. Tools powered by these models can generate code from natural language descriptions, explain complex algorithms, translate between programming languages, and scaffold entire application architectures in minutes. Public demonstrations routinely showcase LLMs producing functional web applications, passing professional certification examinations, and engaging in sophisticated technical reasoning [1, 2].

Yet practitioners who deploy these tools in sustained, production-oriented workflows consistently report a phenomenon we term the "demo-to-production gap": the distance between what LLMs achieve in controlled demonstrations and what they deliver under the complex, iterative, multi-file conditions of real software development [3]. This gap is not merely quantitative—a matter of reduced accuracy—but qualitative, involving fundamentally different failure modes that emerge only under sustained use.

This technical report addresses the gap through a practitioner-driven methodology. Rather than benchmarking LLMs against standardized test suites, we document and analyze failure patterns observed during months of daily, professional use of anonymized LLM-based coding assistants across diverse software engineering tasks. Our contributions are threefold:

(1) A taxonomy of four critical LLM failure modes observed in production software engineering workflows, including the novel concept of the Complexity Cliff;

(2) Formalization of the Verification Paradox—an inherent inverse relationship between user dependency on AI and capacity to validate its outputs;
(3) A practical five-strategy framework for effective human–AI collaboration derived from sustained practitioner experience.

The remainder of this paper is organized as follows. Section 2 provides background on LLM architectures and the emerging paradigm of natural-language-driven development. Section 3 describes our methodology. Section 4 presents detailed case studies. Section 5 formalizes our taxonomy of limitations. Section 6 introduces the Verification Paradox. Section 7 proposes the practical framework. Section 8 discusses implications and limitations, and Section 9 concludes.

2. Background and Related Work

2.1 Large Language Models as Software Engineering Tools

Modern LLMs, including the GPT, Claude, and Gemini model families, are autoregressive transformer-based architectures trained on massive text corpora to predict the next token in a sequence [4, 5]. Despite the apparent sophistication of their outputs, these models do not maintain an internal world model, do not possess persistent memory across sessions, and cannot introspect on the boundaries of their own knowledge [6]. Their outputs are generated through statistical pattern completion, not through the kind of symbolic reasoning or causal understanding that characterizes human problem-solving [7].

In the software engineering domain, LLMs have been integrated into commercial tools for code completion, code generation from natural language, automated testing, documentation generation, and code review [8, 9]. Benchmarks such as HumanEval [10], MBPP [11], and SWE-bench [12] have been developed to evaluate these capabilities. However, these benchmarks predominantly assess self-contained, single-function tasks and do not capture the multi-file, multi-dependency, iterative nature of real-world software projects [13].

2.2 Vibe Coding: Natural-Language-Driven Development

The term "vibe coding" was introduced by Karpathy [14] to describe a development paradigm in which the programmer primarily describes desired functionality in natural language, allowing an LLM to generate the implementation. The developer iterates by providing feedback and corrections in natural language rather than directly editing source code. This approach represents a fundamental shift in the programmer's role: from code author to code curator and reviewer.

While vibe coding has been celebrated for dramatically reducing time-to-prototype [15], limited academic attention has been paid to its failure modes under increasing complexity, or to the cognitive risks it poses when developers lose detailed understanding of their own codebases. The present report addresses this gap with empirical observations from sustained vibe coding practice.

2.3 Known LLM Limitations

The literature has documented several categories of LLM limitation relevant to software engineering. Hallucination—the generation of plausible but factually incorrect outputs—has been widely studied in natural language generation [16, 17]. Context window constraints impose hard limits on the amount of information a model can process simultaneously [18]. The absence of persistent memory between sessions has been noted as a limitation for long-running collaborative tasks [19]. Our contribution builds on this prior work by (a) observing these limitations in situ during extended professional use, (b) identifying emergent failure patterns that arise from the interaction between these limitations, and (c) formalizing the Verification Paradox as a systemic risk of LLM-assisted development.

3. Methodology

This study employs a practitioner-driven observational methodology, drawing from the tradition of reflective practice in software engineering [20]. The first author, a Generative AI engineer with professional experience in building production systems using LLM-based tools, documented interaction patterns, failure modes, and workaround strategies over a sustained period of daily use.

3.1 Data Collection

Data was collected from two primary sources: (1) interaction logs with multiple anonymized LLM-based coding assistants (referred to as Tool A, Tool B, etc.) across software development tasks including front-end development, API integration, multi-file refactoring, and general-purpose question answering; and (2) structured reflection notes documenting observed failure modes, their root causes, and the strategies employed to mitigate them.

All tool identifiers have been anonymized to focus analysis on systemic LLM behavior rather than vendor-specific performance. Interaction logs in languages other than English have been translated by the authors for inclusion.

3.2 Analysis Approach

Observed failures were analyzed using thematic coding [21], grouping incidents by root cause rather than surface symptom. This process yielded four principal failure categories (Section 5) and one emergent meta-pattern (the Verification Paradox, Section 6). Case studies (Section 4) were selected to illustrate each category with concrete, reproducible examples.

3.3 Scope and Limitations of Methodology

This study is inherently qualitative and observational. It does not claim statistical generalizability across all LLMs or all software engineering contexts. Its strength lies in ecological validity: the failures documented here occurred during genuine professional work, not contrived benchmarks. We acknowledge that LLM capabilities evolve rapidly, and that some observed limitations may be partially addressed by future model generations. Nevertheless, the architectural constraints underlying several failure modes—finite context windows, absence of persistent state, autoregressive generation—are fundamental to current transformer architectures and are unlikely to be resolved by scaling alone [22].

4. Case Studies

We present five case studies illustrating distinct failure modes. Each case describes the task, the LLM's behavior, the failure observed, and the underlying cause.

4.1 Case Study 1: The Dashboard and the Complexity Cliff

Task. Using natural-language-driven development (vibe coding), build a data visualization dashboard with authentication, database integration, interactive charts, and responsive layout.

Initial result. Tool A generated a complete React application with state management, chart components (Recharts), a responsive grid layout, and authentication scaffolding in approximately 20 minutes. The initial output achieved roughly 70% of the desired functionality. Through three natural-language feedback iterations ("make the sidebar collapsible," "add dark mode," "date picker should filter all charts"), the application reached approximately 95% completion within two hours.

Failure. When the task expanded to include real-time data updates via WebSocket connections, the LLM's performance degraded sharply. The model could not maintain coherent state management across the growing codebase. It duplicated event listeners, introduced race conditions in asynchronous handlers, and proposed fixes that were syntactically valid but semantically incorrect. Each proposed correction addressed the immediate symptom while introducing new defects elsewhere.

Analysis. This case illustrates what we term the Complexity Cliff: a non-linear degradation in LLM performance as task interdependency exceeds a critical threshold. Below this threshold, vibe coding is remarkably effective. Above it, the model's inability to maintain a coherent global view of the system produces cascading failures. Critically, the model's confidence remained constant across both regimes, providing no signal to the user that reliability had degraded.

4.2 Case Study 2: The Fabricated API

Task. Integrate a third-party payment processing API into an existing application.

Result. Tool B generated well-structured integration code that called specific API endpoints with plausible parameter names, authentication headers, and error handling. The code followed established patterns for REST API consumption and appeared professional.

Failure. The endpoints did not exist. The function names, URL paths, and parameter structures were entirely fabricated by the model. The generated code was, in effect, a plausible fiction—a "hallucinated novel" about how the API should work based on patterns observed in training data from similar but different APIs. A junior developer spent approximately two hours debugging the integration before discovering that the API calls were fictitious.

Analysis. This case exemplifies Confident Hallucination: the model generates fabricated content with the same syntactic structure, confidence level, and explanatory clarity as correct output. The hallucinated code was indistinguishable from valid code without external verification against the actual API documentation. This represents a fundamentally different failure mode from traditional software bugs, as the defect lies not in logic but in reference to non-existent external contracts.

4.3 Case Study 3: The Invisible Dependency

Task. Refactor a service layer with dependencies distributed across 15 source files.

Result. Tool A successfully modified the target file according to specifications. The refactored code was clean, well-documented, and adhered to the project's coding standards.

Failure. The refactoring broke interface contracts expected by dependent files that were outside the model's context window. File G expected a method signature from File A that had been renamed during refactoring, but the model had no visibility into File G's existence or expectations.

The model did not indicate that it lacked visibility into the broader dependency graph; it proceeded with full confidence within its limited view.

Analysis. This case demonstrates Context Window Blindness. Current LLMs operate within a finite attention span (typically 8K–200K tokens). A medium-sized codebase of 500,000+ lines vastly exceeds this window. The model optimizes within its visible fragment without awareness of, or signaling about, the unseen majority. Unlike a human developer who would intuitively consider downstream dependencies and proactively check dependent files, the LLM has no mechanism for recognizing that relevant information exists beyond its current context.

4.4 Case Study 4: The Amnesiac Collaborator

Task. Continue development of a feature across multiple work sessions using an LLM assistant.

Day 1. Over three hours of collaborative development, the practitioner and Tool A established architectural decisions, naming conventions, design patterns, and implementation strategies. The session was productive and the feature reached approximately 80% completion.

Day 2. Upon initiating a new session, the LLM had no recollection of the previous day's work. It suggested different architectural patterns, contradicted naming conventions established the previous day, and proposed implementations incompatible with the existing codebase. It effectively behaved as a new, uninformed collaborator.

Analysis. This case illustrates the Memory Illusion. While some platforms have introduced rudimentary memory features (storing a small number of key facts between sessions), these represent a shallow approximation of the rich, contextual understanding that accumulates during genuine collaboration. The contrast between the deep contextual alignment achieved during a session and the complete reset between sessions creates a disorienting discontinuity that impedes iterative development workflows.

4.5 Case Study 5: Contextual Reasoning Failure

Task. A user asked an LLM-based assistant (Tool C): "I want to wash my car and I live 100 meters from the car wash station. What is the most economical and ecological way to get there?"

Result. The model responded with a detailed analysis recommending walking as the optimal mode of transportation, including a comparison table rating walking, cycling, electric scooters, and driving across ecological and economic dimensions. The response concluded: "At 100 meters, there is no advantage to using the car."

Failure. The model entirely missed the obvious contextual implication: the user's purpose was to wash their car, which requires bringing the car to the station. When the user pointed out this contradiction ("But I want to wash my car—how do I bring it if I walk?"), the model acknowledged the error and corrected itself, stating: "The most economical and ecological option is to drive your car—but only the 100 meters necessary."

Analysis. This case reveals a Contextual Reasoning Failure: the model pattern-matched to a generic "eco-friendly transportation" template without performing the basic situational inference that any human listener would make immediately—namely, that washing a car requires having the car present. The model optimized for the literal

question ("how to travel 100m economically") while ignoring the pragmatic context ("I need my car at the destination"). This case is particularly illustrative because it requires no domain expertise to identify the error; it requires only common-sense reasoning about the physical world, a capacity that current LLMs simulate but do not reliably possess [23].

5. A Practitioner's Taxonomy of LLM Failure Modes

Synthesizing the case studies and extended observations, we propose a taxonomy of four principal failure modes encountered during sustained LLM use in software engineering workflows. These are summarized in Table 1.

Table 1. Taxonomy of LLM failure modes in software engineering practice.

Failure Mode	Description	Root Cause	Case Study
Complexity Cliff	Non-linear performance degradation as task interdependency grows	Absence of global system representation; local optimization within fragments	4.1
Context Window Blindness	Silent contract violations when dependencies exceed the attention span	Finite context window (8K–200K tokens); no out-of-scope awareness	4.3
Memory Illusion	Complete loss of accumulated context between sessions	Stateless architecture; no persistent memory across inference calls	4.4
Confident Hallucination	Fabricated outputs indistinguishable in tone from correct outputs	Autoregressive generation without epistemic self-awareness	4.2, 4.5

5.1 The Complexity Cliff

We define the Complexity Cliff as the threshold beyond which LLM-assisted development transitions from effective augmentation to active liability. Below this threshold, tasks are sufficiently self-contained that the model's pattern-matching capabilities produce correct or near-correct outputs efficiently. Above it, the number of interdependent components, implicit contracts, and emergent system behaviors exceeds the model's capacity for coherent representation, leading to cascading errors.

The Complexity Cliff is insidious because it is not signaled by any change in the model's output confidence. The same articulate, well-structured responses that characterize correct operation continue to be produced after the cliff has been crossed—but their semantic correctness degrades. The phrase "it works on my prompt"—an analogy to the classic software engineering refrain "it works on my machine"—captures this phenomenon: a prompt that succeeds in isolation may fail when the surrounding complexity exceeds the model's integrative capacity.

5.2 Context Window Blindness

Context Window Blindness describes the model's inability to account for information that exists outside its current attention window. Unlike a human developer who maintains a mental model of the broader system and proactively investigates dependencies, an LLM operates exclusively on the text currently in its context. It cannot recognize that relevant information exists beyond this boundary, nor signal to the user that its view is incomplete. This produces a form of silent failure: the model generates outputs that are locally coherent but globally inconsistent.

5.3 The Memory Illusion

The Memory Illusion refers to the experiential dissonance between the rich collaborative context that develops within a single session and the complete absence of that context in subsequent sessions. During a session, an LLM-based assistant may exhibit behavior that subjectively resembles understanding: it recalls earlier decisions, builds on established patterns, and maintains coherent naming conventions. This creates an illusion of persistent memory and genuine collaboration. When the session ends, all accumulated context is lost.

Emerging memory features in commercial platforms store a small number of extracted facts between sessions, but these represent a lossy compression of the original collaborative context and do not preserve the nuanced architectural understanding that characterized the session.

5.4 Confident Hallucination and Contextual Reasoning Failure

Confident Hallucination is the generation of outputs that are factually incorrect or entirely fabricated, yet presented with the same syntactic structure and assertive tone as correct outputs. Unlike traditional software bugs, which arise from logical errors in valid code, hallucinated outputs reference non-existent APIs, fabricate function signatures, or present fictional information as established fact.

A related but distinct phenomenon is Contextual Reasoning Failure, where the model produces a logically structured response to a question it has not actually understood in context. Case Study 5 exemplifies this: the model applied a generic "eco-transportation" template without performing the trivial pragmatic inference that a car wash requires the car's presence. This failure is distinct from hallucination (no facts were fabricated) and from context window limitations (all relevant information was present). It represents a failure of situational reasoning—the model optimized for the literal pattern of the query while ignoring its real-world implications.

6. The Verification Paradox

The failure modes described in Section 5 converge on a meta-pattern we formalize as the Verification Paradox:

Definition. The Verification Paradox is the inverse relationship between a user's need for LLM assistance and their capacity to validate the correctness of LLM outputs. Expert users, who possess the domain knowledge required to identify hallucinations, context window artifacts, and complexity cliff errors, benefit most from LLM acceleration because they can efficiently filter correct from incorrect outputs. Novice users, who lack this discriminative capacity, derive less net benefit and face greater risk, precisely because they cannot distinguish confident-but-correct outputs from confident-but-fabricated ones.

This paradox has significant implications for the adoption of LLM tools in educational contexts, junior developer onboarding, and organizations with varying levels of technical maturity. The very users who might most benefit from AI-assisted productivity gains are those for whom the technology poses the greatest risk of introducing silent defects.

We note that the Verification Paradox is not a bug amenable to engineering solutions within the current paradigm. It is a structural property of any system that generates confident outputs without epistemic self-awareness. Larger models, longer context windows, and improved training data may reduce the frequency of individual errors, but the fundamental asymmetry between output confidence and output correctness will persist as long as models lack reliable mechanisms for uncertainty quantification [24].

Table 2. The Verification Paradox: differential impact by user expertise.

Dimension	Expert User	Novice User
Error detection	Rapid identification through domain knowledge	Unable to distinguish correct from fabricated output
Net productivity	Significant acceleration (leverages ~80% correct output)	Marginal or negative (debugging fabricated outputs)
Risk profile	Low (errors caught before propagation)	High (errors may propagate to production)
AI dependency	Supplementary (AI as accelerator)	Substitutive (AI as primary knowledge source)

7. A Practical Framework for Human–AI Collaboration

Drawing from the case studies and taxonomy, we propose five strategies for practitioners seeking to maximize the benefits of LLM-assisted development while mitigating the identified failure modes.

7.1 Strategy 1: Treat AI as a First-Draft Machine

LLM outputs should be treated as initial drafts requiring human review, not as authoritative answers. This framing sets appropriate expectations and ensures that verification is built into the workflow from the outset. The distinction between a "first draft machine" and an "answer machine" is not merely semantic; it determines whether the practitioner engages critically or passively with the output.

7.2 Strategy 2: Maintain Task Granularity Below the Complexity Cliff

Tasks delegated to LLMs should be decomposed into units that remain within the model's effective complexity threshold. Rather than requesting a complete feature spanning multiple interdependent systems, practitioners should issue smaller, well-scoped requests that the model can address within a single coherent context. This strategy trades the convenience of end-to-end generation for the reliability of incremental, verifiable outputs.

7.3 Strategy 3: Invest Verification Time Proportional to Generation Time

The time saved through LLM-assisted code generation should be reinvested in verification activities: code review, automated testing, documentation checks, and integration testing. A useful heuristic is to allocate at least as much time to verifying AI-generated code as was saved by generating it. This approach preserves the productivity advantage while maintaining code quality.

7.4 Strategy 4: Preserve and Develop Domain Expertise

The Verification Paradox implies that the practitioner's domain expertise is the primary safeguard against LLM errors. Organizations should resist the temptation to reduce investment in developer training on the assumption that AI will compensate for knowledge gaps. On the contrary, the more powerful AI tools become, the more critical human expertise becomes as a quality-assurance mechanism.

7.5 Strategy 5: Externalize Context Explicitly

Given the Memory Illusion and Context Window Blindness, practitioners should not assume that the LLM retains or has access to relevant context. Constraints, architectural decisions, naming conventions, and interface contracts should be re-stated explicitly in each interaction. Maintaining external documentation—architecture decision records, interface specifications, and coding standards—and feeding these into the LLM context is essential for reliable multi-session collaboration.

8. Discussion

8.1 Implications for Software Engineering Practice

Our findings suggest that the integration of LLMs into software engineering workflows is neither the revolution promised by proponents nor the liability feared by skeptics. LLMs are best understood as powerful pattern-matching accelerators whose effectiveness is gated by the human practitioner's ability to scope tasks appropriately and verify outputs critically. The phrase "it works on my prompt"—which we propose as the AI-era analog of "it works on my machine"—captures the essential fragility: success in isolated interactions does not predict success under the compound complexity of production systems.

The Complexity Cliff, in particular, has implications for how organizations plan LLM adoption. Vibe coding is demonstrably effective for prototyping, internal tooling, and well-bounded features. Extending it to architecturally complex, multi-dependency systems without proportional increases in human oversight invites the cascading failure patterns documented in Case Study 1.

8.2 Implications for AI Research

Several of the failure modes we document are not addressable by scaling alone. Context Window Blindness is a direct consequence of finite attention architectures. The Memory Illusion is a consequence of stateless inference. Confident Hallucination reflects the absence of epistemic self-awareness in autoregressive generation. Contextual Reasoning Failure, as demonstrated in Case Study 5, suggests that current models lack robust mechanisms for pragmatic inference—the ability to reason about what a user means rather than what they literally said.

These observations point to several promising research directions: (a) architectures that support persistent, updatable world models across sessions; (b) uncertainty quantification mechanisms that calibrate output confidence to actual reliability; (c) explicit dependency-graph awareness in code-generation models; and (d) pragmatic reasoning layers that infer contextual intent beyond literal query patterns.

8.3 Threats to Validity

This study's primary limitation is its reliance on a single practitioner's experience, which may not generalize across all development contexts, tool versions, or programming domains. We mitigate this by (a) selecting case studies that illustrate structurally distinct failure modes rather than edge cases, (b) grounding our taxonomy in architectural properties of transformer models that are shared across implementations, and (c) situating our findings within the context of corroborating prior work. Additionally, the rapidly evolving nature of LLM capabilities means that specific failure instances may not be reproducible on future model versions. However, the underlying architectural constraints that produce these failure modes remain fundamental to current transformer designs.

9. Conclusion

This technical report has examined the question "Is AI really intelligent?" through the lens of sustained, professional use of LLM-based tools in software engineering. Our answer is nuanced: LLMs exhibit remarkable pattern-matching intelligence that genuinely accelerates well-scoped tasks, but they lack the contextual reasoning, persistent memory, epistemic self-awareness, and global system comprehension that characterize human engineering intelligence.

We have contributed a four-mode taxonomy of practitioner-observed failure patterns (the Complexity Cliff, Context Window Blindness, the Memory Illusion, and Confident Hallucination), identified Contextual Reasoning Failure as a distinct sub-category, and formalized the Verification Paradox as a structural property of human–AI collaboration. Our five-strategy practical framework provides actionable guidance for practitioners navigating the demo-to-production gap.

The practitioners who will thrive in the age of generative AI are not those who uncritically adopt these tools nor those who dismiss them, but those who develop a precise understanding of where AI capabilities end and where human expertise remains indispensable.

Declaration of Generative AI and AI-Assisted Technologies in the Writing Process

During the preparation of this work, the authors used LLM-based tools for language editing assistance. The authors take full responsibility for the content of the publication, have reviewed and edited all AI-assisted outputs, and confirm that this work represents their own original analysis, interpretation, and intellectual contribution.

Declaration of Competing Interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Author Contributions

Dr. Khaled EL TANNIR: Conceptualization, Methodology, Investigation, Data Collection, Analysis, Writing – Original Draft, Writing – Review & Editing.

References

[1] Bubeck S, Chandrasekaran V, Eldan R, et al. Sparks of artificial general intelligence: Early experiments with GPT-4. arXiv preprint arXiv:2303.12712, 2023.

[2] OpenAI. GPT-4 technical report. arXiv preprint arXiv:2303.08774, 2023.

[3] Khlaif JN, Mousa A, Sanmugam M. The gap between AI perception and reality in software engineering. IEEE Software, 2024;41(2):22–29.

[4] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. Advances in Neural Information Processing Systems, 2017;30.

[5] Brown TB, Mann B, Ryder N, et al. Language models are few-shot learners. Advances in Neural Information Processing Systems, 2020;33:1877–1901.

[6] Bender EM, Gebru T, McMillan-Major A, Shmitchell S. On the dangers of stochastic parrots: Can language models be too big? In: Proceedings of FAccT, 2021:610–623.

[7] Marcus G, Davis E. Rebooting AI: Building artificial intelligence we can trust. Vintage Books, 2019.

[8] Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

[9] Ross SI, Martinez F, Houde S, Muller M, Weisz JD. The programmer's assistant: Conversational interaction with a large language model for software development. In: Proceedings of IUI, 2023:491–514.

[10] Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

[11] Austin J, Odena A, Nye M, et al. Program synthesis with large language models. arXiv preprint arXiv:2108.07732, 2021.

[12] Jimenez CE, Yang J, Wettig A, et al. SWE-bench: Can language models resolve real-world GitHub issues? In: Proceedings of ICLR, 2024.

[13] Dakhel AM, Majdinasab V, Nikanjam A, et al. GitHub Copilot AI pair programmer: Asset or liability? Journal of Systems and Software, 2023;203:111734.

[14] Karpathy A. Vibe coding. [Online]. 2025. Available: https://x.com/karpathy/status/1886192184808149383

[15] Jiang S, Wang Y, Zhang L. The promise and peril of AI-assisted software development. In: Proceedings of ICSE-NIER, 2024.

[16] Ji Z, Lee N, Frieske R, et al. Survey of hallucination in natural language generation. ACM Computing Surveys, 2023;55(12):1–38.

[17] Huang L, Yu W, Ma W, et al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. arXiv preprint arXiv:2311.05232, 2023.

[18] Liu N, Lin K, Hewitt J, et al. Lost in the middle: How language models use long contexts. Transactions of the ACL, 2024;12:157–173.

[19] Zhong W, Guo L, Gao Q, et al. MemoryBank: Enhancing large language models with long-term memory. In: Proceedings of AAAI, 2024.

[20] Schön DA. The reflective practitioner: How professionals think in action. Basic Books, 1983.

[21] Braun V, Clarke V. Using thematic analysis in psychology. Qualitative Research in Psychology, 2006;3(2):77–101.

[22] Wei J, Tay Y, Bommasani R, et al. Emergent abilities of large language models. Transactions on Machine Learning Research, 2022.

[23] Mitchell M, Krakauer DC. The debate over understanding in AI's large language models. Proceedings of the National Academy of Sciences, 2023;120(13):e2215907120.

[24] Kadavath S, Conerly T, Askell A, et al. Language models (mostly) know what they know. arXiv preprint arXiv:2207.05221, 2022.