Prompts Are Not Causally Closed: Why Source Code Remains Essential in the AI Era

SVEN APEL, Saarland University & Max Planck Institute for Software Systems, Germany

The use of AI in software engineering invites an appealing analogy: Perhaps source code is becoming the new machine code – an implementation artifact that software developers no longer need to read. We argue that this analogy is misleading. Compilers hide machine code behind programming languages with defined semantics and tool chains that largely preserve those semantics. Natural-language prompts and generative AI models offer no comparable boundary. They do not form a *causally closed abstraction*: Whereas source-level reasoning suffices for a large class of questions about program behavior, a prompt and an AI model do not let developers reliably explain, predict, and control the behavior that matters for the task at hand. In other words, because prompt-to-code generation has no semantic-preservation contract, prompt-level reasoning lacks causal closure. Consequently, *source code remains a crucial part of the interface between human intent and machine behavior*. The rise of AI therefore increases – rather than removes – the need for software-engineering principles that define good code and constrain how tools generate it.

CCS Concepts: • **Software and its engineering** → **Software development methods**; • **Computing methodologies** → *Artificial intelligence*.

Additional Key Words and Phrases: Causal closure, AI for software engineering, AI programming assistants

1 Introduction

The claim that "source code is the new machine code" captures a genuine shift in software engineering.[1] Software developers increasingly describe a desired change to a software system in natural language and let an AI assistant produce the implementation. If this trajectory continues, why should humans read source code at all? We rarely inspect the machine instructions emitted by a compiler. Perhaps generated source code will soon be equally irrelevant: not necessarily readable, merely executable.

This conclusion confuses two very different abstraction boundaries. We can usually ignore machine code because programming languages, compilers, and their tools provide a sufficiently closed chain of reasoning from source-level constructs to machine behavior – the translation *preserves semantics* by design. A prompt and a generative AI model provide no equivalent closure or semantic-preservation contract. Although the model can produce an implementation, the prompt does not fully define that implementation's semantics, and the model does not provide a stable set of rules or contract from which its behavior can be derived. Tests might be generated to validate the implementation, but they might repeat the same logical mistakes as the code [1, 6]. Source code is therefore not an incidental residue of AI-assisted software engineering. It remains the most precise, inspectable account of the behavioral choices made by the generated program, even though requirements and specifications remain the authoritative account of what we intended it to do.

2 Causal Closure Through Semantics-Preserving Translation

Calling an abstraction *causally closed* means that the information and rules available at one abstraction level are sufficient for reasoning about the relevant effects at the level below.[2] The important point is not that the lower level ceases to exist, but that it needs not enter every explanation or prediction made at the higher level.

[1] See, for example, Mark Harman's post on LinkedIn or Elon Musk's post on X.

[2] There is a distinction between causal closure and causal inference: Causal closure is concerned with whether reasoning at one level of abstraction suffices to account for relevant effects at another, while causal inference asks which effects follow from interventions and how those effects can be identified from data.

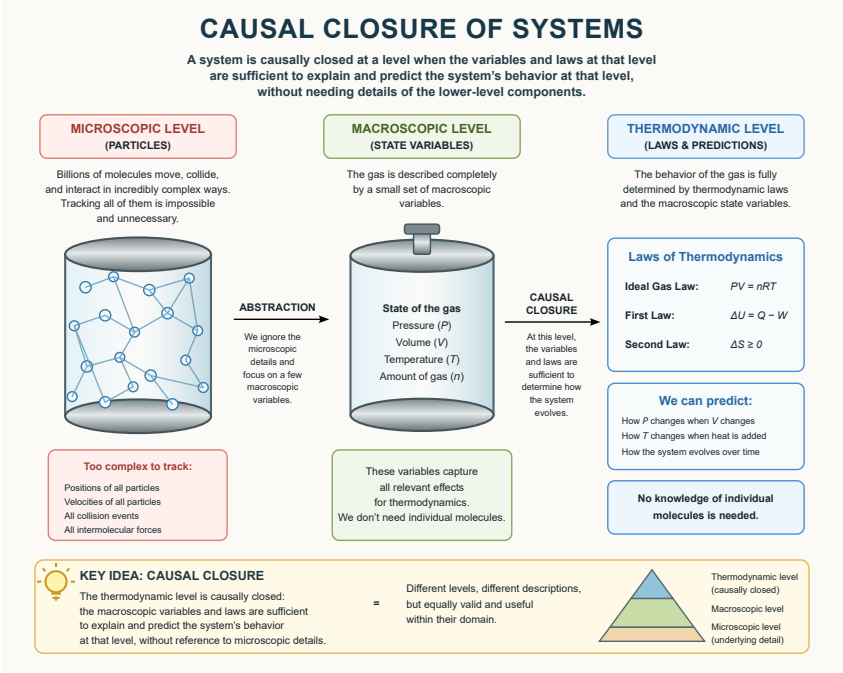

Fig. 1. Thermodynamics is causally closed for a large class of questions about gases. The laws of thermodynamics relate macroscopic quantities such as pressure, volume, and temperature without requiring us to reconstruct the trajectory of every particle.

Thermodynamics offers the canonical example for causal closure, as illustrated in Figure 1. A gas consists of an enormous number of particles, each with a position and momentum and, ultimately, a description governed by the laws of quantum mechanics. Nevertheless, to predict what happens when a gas is compressed or heated, we ordinarily reason with aggregate state variables such as pressure, volume, and temperature. The laws of thermodynamics relate these macroscopic quantities (higher level of abstraction) without requiring us to reconstruct the trajectory of every particle (lower level of abstraction). Microscopic dynamics realizes the macroscopic process, and statistical mechanics explains why the aggregate laws work, but neither must be consulted for each thermodynamic calculation. For the class of macroscopic questions covered by the theory, reasoning in terms of pressure, volume, and temperature is *causally closed*.

A programming language has a grammar and a semantics. A compiler is expected to preserve that semantics, while type checkers, debuggers, model checkers, and tests provide independent ways to challenge the translation. When a C expression evaluates incorrectly, a developer can usually reason from the rules of the C language without studying machine code. When those rules are insufficient, the escape hatch of looking into the generated machine code remains available, but it is exceptional (e.g., when addressing performance or security issues). The abstraction succeeds because both the contract and the translation machinery are subject to precise technical scrutiny.

Note that the analogy to thermodynamics is deliberately loose in one respect: Thermodynamic closure *emerges* from statistical aggregation over microstates, whereas the closure from source code to machine code is *engineered* through semantics-preserving translation. What carries over is not the mechanism but the epistemic consequence. In both cases, the lower level need not enter the

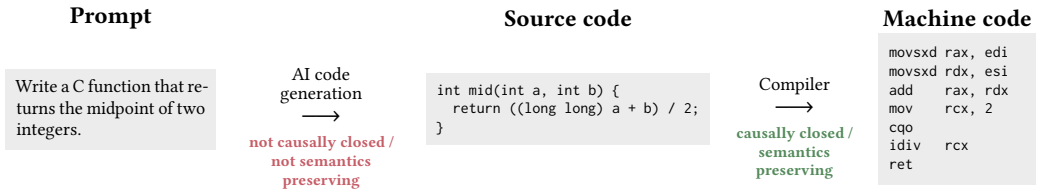

Fig. 2. The two abstraction boundaries are fundamentally different. The prompt underspecifies the source code: It fixes neither the result type (integer or floating point), the rounding direction for negative operands, nor whether overflow must be avoided, detected, or otherwise handled. Different variants of generated source code therefore compute genuinely different values, yet each plausibly satisfies the request. The source code shown avoids signed overflow and rounds toward zero. Once this source program and its compilation setting are fixed, programming-language, compiler, and instruction-set semantics determine the machine program's functional behavior.

reasoning for a well-delineated class of questions. This explains why machine code does not need to be readable by most software engineers. Its producer – the compiler – is constrained by a formal language on one side and an instruction set on the other. We do not trust a compiler because it tends to emit plausible binaries. We trust it because its output is governed by explicit semantics, extensive validation, and decades of engineering around semantic preservation [7].

It is important to note that causal closure is always relative to a set of phenomena and desired precision. Thermodynamics does not answer questions about an individual molecule, and it can break down for very small systems or far from equilibrium. Likewise, a programming-language abstraction does not need to conceal every hardware effect to be useful. Optimizer defects, undefined behavior, hardware faults, and timing effects sometimes force programmers to inspect assembly. Yet, for a large class of ordinary tasks and problems, the abstraction holds well enough: Its concepts and laws capture the causes that matter for the questions software engineers are asking.

3 Prompt-to-Code: No Semantic Preservation, No Causal Closure

Open-ended natural-language prompts generally lack these properties at the level required for reliable software construction. A request such as "make retries robust" leaves open the retry policy, failure taxonomy, idempotency assumptions, observability, latency budget, and compatibility constraints. An AI assistant resolves such gaps using patterns learned from data and context. Its answer may be excellent, but it is a *plausible completion*, not the *semantic consequence* of the prompt [5]. Figure 2 makes the asymmetry concrete. The prompt constrains the desired result but does not determine a unique algorithm, error policy, representation, or implementation. By contrast, the generated source program states a precise choice whose functional consequences can be followed through a semantics-preserving compiler to machine instructions. This closure is scoped, as before: questions about timing, compiler defects, or hardware faults may still require reasoning below the source-code level.

The gap cannot be closed merely by using a larger model or a longer prompt. Software requirements are incomplete because the world is incomplete, with relevant facts distributed across source code, tests, production behavior, organizational practice, and unstated stakeholder expectations [8]. Human developers face the same incompleteness, but what differs is how it is resolved. A human decision is not inherently correct or transparent, but a developer or stakeholder can have authority to clarify intent and can be held accountable for the choice. An AI model has no independent authority over stakeholder intent. When it fills a gap with a *plausible completion*, that choice therefore requires external justification. Its output is also probabilistic, shifting with context, configuration,

and model version. Even when an AI model can explain its output, that explanation is not a proof that the generated software has the intended effects. In fact, AI models can produce plausible but unfaithful explanations for their own behavior [12].

Verification and validation tools can close particular parts of the gap. Types can rule out classes of errors, tests can establish observed behavior, and formal specifications can prove selected properties. But these tools work by replacing ambiguous intent with more precise artifacts. They do not make natural language prompts themselves causally closed. Indeed, once a specification is formal and complete enough to delimit acceptable behavior, we have introduced another specification or programming language rather than eliminated such languages. The specification need not determine a unique implementation. It must be precise enough to judge the relevant effects. This is, in fact, what domain-specific languages such as SQL, P4, or G-CODE do. They replace a natural-language prompt with programs written in a more precise, domain-specific language that supports reasoning about relevant behavior without requiring inspection of every implementation detail.

One might object that we stopped reading machine code not because source code became more readable, but because compilers earned our trust through validation, and that sufficient validation of an AI model would likewise let us ignore its source-code output. The analogy breaks at the point where trust is anchored. A compiler can earn reusable *process-level* assurance: Arguments and tests about semantic preservation apply across many compilations rather than beginning anew for each output. This assurance remains conditional on defined source behavior, the compiler version and configuration, the target platform, and the absence of tool chain defects. An off-the-shelf AI model offers no comparable semantic-preservation contract. Because the prompt does not determine the output, and the mapping shifts with context, configuration, and model version, trust cannot be discharged once over the process. It must be re-earned for each generated artifact. This does not imply that a human must always read source code: A proof certificate, an executable specification, or independently derived behavioral evidence could instead justify trust in a particular artifact. But such evidence must be strong enough to cover the consequential behavior left open by the prompt. General-purpose AI development does not currently produce such evidence systematically, and real-world code remains enormously challenging for contemporary verification tools.

The practical consequence is that, in the absence of such evidence and artifacts, source code remains a crucial artifact for reasoning. A future representation might differ from current textual languages, or sufficiently strong evidence might make direct source inspection unnecessary. In any case, opaque generated code plus a conversational assurance is not an abstraction layer – it is deferred uncertainty. Where AI does deliver trustworthy artifacts, it does so by exactly this route. Recent advances in AI for mathematics succeed because they are anchored to a rigorous problem formulation and a sound reasoning engine – most notably the LEAN theorem prover[3] – that mechanically checks every step [11]. However, such checking establishes that a proof follows from its *formal* statement. It does not by itself establish that an AI-generated formalization faithfully translates the original natural-language claim. Bastounis et al. [3] demonstrate practical mismatches between natural-language proofs and LEAN formalizations and show that resolving the ambiguities required for semantically faithful autoformalization is, in their formal analysis, harder than the Halting problem. Recent work on using AI to verify complex software reinforces this point [4]: Verifying key security properties of SIGNAL required 26 000 lines of trusted specification and AARCH64 semantics, representing substantial human effort that current automation does not eliminate and possibly never fully will.

[3] https://lean-lang.org/

4 AI Makes Code Quality and Design More Important

As software developers must continue to reason about consequential generated code, readability is not cosmetic. Software architecture, modular boundaries, design patterns, design rationale, and code style make human reasoning possible. They let a reviewer connect a requirement to the implementation and trace a change to its consequences. Generated source code that passes today's tests but obscures those connections creates a liability for tomorrow's change. This shifts the central question for software engineering in the AI era: It is not only whether an AI model can produce code that works. It is whether tools can produce code that humans and other tools can understand, maintain, and evolve. Letting AI agents generate code without explicit policies and human oversight creates substantial engineering risk [9, 10, 13].

Using AI in software engineering demands explicit policies: respect architectural boundaries, prefer local and reviewable changes, expose assumptions, preserve established contracts, generate tests that probe behavior rather than mirror the implementation, and so on. Software engineering supplies the vocabulary for these policies. Concepts such as cohesion, coupling, information hiding, contracts, traceability, and maintainability describe qualities that cannot be recovered from a binary "works/does not work" judgment [2]. They should shape AI model context, generation, evaluation, and review. AI assistants should be optimized not just for task completion but also for producing evidence that makes their changes intelligible. They should communicate that evidence through well-structured code and supporting artifacts and incorporate human feedback.

This view also clarifies the software developer's changing role in the AI era. Developers need not manually type every line, just as they need not manually encode every machine instruction. But delegating coding is not delegating responsibility. Software developers must formulate intent, choose constraints, examine implementations, and decide whether the available evidence justifies trust. AI may move effort from code writing to reading and validation – it does not make those activities obsolete.

5 Conclusion

Source code cannot become the new machine code merely because AI can generate it. Machine code became safely ignorable in many settings only after programming languages and compilers established a rigorous and heavily tested abstraction boundary. Natural-language prompts and generative AI models provide no comparable *causal closure*: Because prompt-to-code generation lacks a stable semantic-preservation contract comparable to that of a programming language and compiler, developers must reopen and inspect the generated artifact.

The goal, therefore, should not be software that no person ever needs to understand. It should be software whose behavior and implementation can be understood at the right level of abstraction, with clear routes to inspect and challenge every consequential step. Unless and until AI-assisted software engineering offers that kind of accountable abstraction, source code remains crucial under the hood: In the absence of equivalent evidence, programmers still need to open it, because *prompts are not causally closed.*

References

[1] Youssef Abdelsalam, Norman Peitek, Anna-Maria Maurer, Marvin Wyrich, and Sven Apel. 2026. A Mechanistic Lens on Semantic Conflicts: Using Activation Patching to Understand LLM Behavior. arXiv:2607.05587 [cs.SE] https://arxiv.org/abs/2607.05587

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer.

[3] Alexander Bastounis, Fabian Circelli, and Anders C. Hansen. 2026. Navier–Stokes Lost in Translation: Why Lean Verification of AI Autoformalisation Does Not Guarantee Correct Natural Language Proofs. arXiv:2610.08144 [math.AP] doi:10.48550/arXiv.2610.08144

[4] Swarat Chaudhuri, Andrew Ferraiuolo, Anton Kovsharov, Tej Chajed, Sam Schlesinger, Karn Seth, Jonathan Katz, Charles Sutton, and Pushmeet Kohli. 2026. *AlphaProof Code: Verifying Signal's Security at the Machine-Code Level.* Technical Report. Google DeepMind.

[5] Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. 2024. Detecting Hallucinations in Large Language Models Using Semantic Entropy. *Nature* 630 (2024), 625–630. doi:10.1038/s41586-024-07421-0

[6] Michael Konstantinou, Florian Tambon, and Mike Papadakis. 2026. On the Risk of Coding Before Testing: An Empirical Study on LLM-based Test Generation Workflow. arXiv:2607.05139 [cs.SE] https://arxiv.org/abs/2607.05139

[7] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814

[8] Bashar Nuseibeh and Steve Easterbrook. 2000. Requirements Engineering: A Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. ACM, 35–46. doi:10.1145/336512.336523

[9] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (Jan. 2025), 96–105. doi:10.1145/3610721

[10] Gilberto Recupito, Fabiano Pecorelli, Gemma Catolino, Valentina Lenarduzzi, Davide Taibi, Dario Di Nucci, and Fabio Palomba. 2024. Technical Debt in AI-enabled Systems: On the Prevalence, Severity, Impact, and Management Strategies for Code and Architecture. *J. Syst. Softw.* 216, C (Oct. 2024), 22 pages. doi:10.1016/j.jss.2024.112151

[11] George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco J. R. Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, Lei Yu, Aja Huang, Miklós Z. Horváth, Andrew Ferraiuolo, Henryk Michalewski, Edward Lockhart, Codrut Grosu, Thomas Hubert, Matej Balog, Pushmeet Kohli, and Swarat Chaudhuri. 2026. Advancing Mathematics Research with AI-Driven Formal Proof Search. arXiv:2605.22763 [cs.AI] https://arxiv.org/abs/2605.22763

[12] Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. 2023. Language Models Don't Always Say What They Think: Unfaithful Explanations in Chain-of-Thought Prompting. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 74952–74965.

[13] Alisa Welter, Niklas Schneider, Tobias Dick, Kallistos Weis, Christof Tinnes, Marvin Wyrich, and Sven Apel. 2025. An Empirical Study of Knowledge Transfer in AI Pair Programming. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 166–177. doi:10.1109/ASE63991.2025.00022