Towards Autonomous Software Engineering

Hao Wang[1] **Ruijie Meng**[2] **Zhe Ye**[1] **Alex Gu**[3] **Naman Jain**[4]

Xiaoyuan Liu[1] **Swarat Chaudhuri**[5] **Thomas Zimmermann**[6] **Sumit Gulwani**[7]

Armando Solar-Lezama[3] **Ion Stoica**[1] **Dawn Song**[1]

[1]UC Berkeley [2]CISPA [3]MIT [4]Cursor
[5]The University of Texas at Austin [6]UC Irvine [7]Microsoft

{hwang628,zhey,xiaoyuanliu,istoica,dawnsong}@berkeley.edu
meng@cispa.de gua@mit.edu naman@anysphere.co swarat@cs.utexas.edu
tzimmer@uci.edu sumitg@microsoft.com asolar@mit.edu

Abstract

Rapid advances in large language models (LLMs) and coding agents are fundamentally reshaping the landscape of software engineering. While recent work has explored AI-assisted coding and automated program repair in isolation, a systematic understanding of how AI systems automate and transform the entire software development life cycle has yet to emerge. **In this position paper, we argue that the research community should systematically study software engineering as a trajectory of substitutions for increasing autonomy, where responsibilities, risks, and failure modes fundamentally shift.** To make this concrete, we introduce a three-level taxonomy inspired by autonomous driving that distinguishes degrees of autonomy along a roadmap from today's AI-assisted development workflows to fully autonomous software engineering in which AI systems autonomously identify demands and design, implement, verify, and maintain software without human oversight. Across these levels, we identify a unifying concern: *preserving and faithfully executing human intent* as direct human control recedes. Building on this taxonomy, we examine key structural shifts and outline a research agenda organized around specification understanding, security and verification, system-level concerns, human-agent interaction, and economic modeling. We hope this framework provides a shared vocabulary for the research community and motivates targeted effort in the problems that will most critically gate progress towards safe, reliable, and fully autonomous software engineering.

1 Introduction

Rapid advances in frontier AI, particularly large language models (LLMs) and AI agents, have begun to fundamentally reshape how software is developed. Traditionally, software engineering has been organized around a human-led lifecycle, in which developers translate often ambiguous requirements into deployed systems through stages such as design, implementation, testing, and maintenance [46]. In today's AI era, AI assists developers in a wide range of tasks (e.g., code and test generation [107, 80]). Emerging AI agents further extend these capabilities by autonomously augmenting human developers across multiple stages of the software development lifecycle [18].

This trend points towards a natural next stage: autonomous software engineering, in which AI systems assume responsibility for end-to-end development. In this paradigm, AI systems autonomously identify high-level demands (e.g., new product features, patching bugs and vulnerabilities, or redesigning due to a shift in user behavior) and translate them into production-ready software with minimal or no human intervention. Such systems not only generate code, but also make decisions about system design, validate correctness through testing, manage deployment, and adapt to evolving requirements and environments. Early signals of this transition are already visible: frontier agents today can autonomously reason over repositories, author and execute tests, identify vulnerabilities, and coordinate multi-stage tasks [22, 8, 117]. This shift positions AI systems as potential substitutes—rather than mere assistants—for human developers.

The transition of AI from assistance to autonomy raises fundamentally new questions. For example, how should responsibility be allocated between the AI users and providers? How can we reason about reliability, verification, and accountability when decisions are made by autonomous systems? What new risks emerge when control shifts away from human developers? While a growing body of work discusses software engineering in the AI era [97, 132, 51, 127, 104, 46], it largely focuses on short-horizon, task-level autonomy within human-led development workflows, implicitly assuming that humans remain in the lifecycle. As a result, many questions introduced by increasing AI autonomy remain underexplored.

As AI systems begin to take full ownership of software engineering, the central challenge shifts. Instead of asking how AI assists humans with specified tasks, we should understand how responsibility can be gradually and safely transferred from humans to AI across the entire development lifecycle. **Therefore, we argue that the community should systematically study software engineering as a trajectory of increasing autonomy, characterizing how autonomy evolves and how it fundamentally reshapes responsibilities, risks, and failure modes as control transitions from human developers to AI systems.** Without such a framework, progress toward autonomous software engineering risks being ad hoc, poorly understood, and misaligned with the requirements of building trustworthy and accountable software systems. We stress that we do not project this roadmap as inevitable. Rather, we argue that stage-level responsibility transfer is a real, currently operating phenomenon that deserves a shared vocabulary, independent of whether the trajectory terminates in full autonomy. A unifying concern cuts across this trajectory: *preserving and faithfully executing human intent* as direct human control recedes. We treat intent preservation [113] as the core problem around which the structural shifts and research directions in this paper are organized.

To this end, we propose a structured framework for understanding this transition. We begin by revisiting traditional software engineering practices and decomposing them into core activities—requirement curation, system design, implementation, testing, and deployment—along with their dependencies and feedback loops. Building on this decomposition and drawing inspiration from the established levels of autonomous driving [105], we organize the progression toward autonomy into three levels (Section 2) from code autonomy to pipeline autonomy, and ultimately to demand autonomy, each defined by the extent to which responsibility shifts from humans to AI systems.

For each level, we characterize how responsibilities are redistributed between humans and AI and identify the capabilities AI systems would need to possess. We then systematically analyze the structural shifts introduced by increasing AI autonomy (Section 3), such as specifications becoming the primary development artifacts, full-stack abstraction breakage, and organizational transformations. Building on this analysis, we outline a broader research agenda, including specification understanding, verification, security and governance, system and language design for autonomous agents, human–agent collaboration interfaces, and economic modeling of hybrid development (Section 4). Throughout, we surface a set of concrete *predictions* about how software engineering will evolve as autonomy deepens, intended to provide some targets to organize research around.

Our goal is not to advocate for unbounded or accelerated autonomy, but to provide a precise conceptual framework for reasoning about its trajectory. By making explicit how responsibility, risk, and control evolve as autonomy increases, we aim to enable the development of software systems that are not only more capable but also trustworthy, secure, and aligned with human and societal objectives.

2 A Taxonomy of Autonomy in Software Engineering

2.1 Software Development Lifecycle (SDLC)

We anchor our taxonomy in the recurring stages of the software development lifecycle (SDLC) [102]. The lifecycle begins with **requirement curation**, where new demands or requests (e.g., feature requests, dependency updates, or prototypes) are packed into a product requirements document to initiate the development process. It is followed by **system design**, in which developers plan the layout and abstractions needed to ensure maintainability and efficiency. Next is **implementation**, where the planned changes are realized in code. This is followed by **testing**, where the implementation is validated and audited to ensure functionality, regressive correctness, security, and other desired properties. Finally, in **deployment**, the proposed change is merged into the main codebase and rolled out to production. Traditionally, this lifecycle is human-led, and AI assists at various stages in modern development practice. For illustration, we depict these stages as a linear pipeline in Figure 1, and a more general view of the same activities and their feedback structure is deferred to Figure 2 in Section A.1.

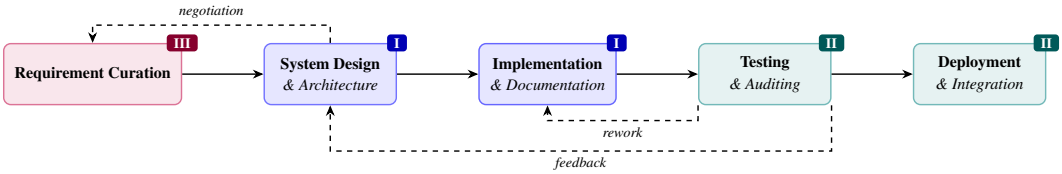

Figure 1: Linear view of the software development lifecycle, annotated with our taxonomy of AI autonomy. Solid arrows indicate hard dependencies, and dashed arrows indicate feedback and negotiation (where the requirement is modified to bootstrap to a feasible design) loops that break strict left-to-right ordering. Box colors and the Roman numeral at each box's top-right corner indicate when that stage transitions to full AI autonomy: **Level I** automates implementation and design; **Level II** propagates autonomy to testing and deployment; **Level III** completes the transition by automating demand. For clarity, the figure places testing before deployment. In practice, especially for large systems, substantial validation like canary releases and pressure testing may continue after deployment and integration. A more general view of the same activities is given in Figure 2 (Section A.1).

2.2 Three-Level Taxonomy of Autonomy

We envision that AI autonomy propagates outward from the most mechanistic tasks to those demanding a broader context, judgment, and consideration of societal implications. Drawing inspiration from the taxonomy of driving automation [105], we frame the autonomy in software engineering as a temporal progression through three levels, each characterized by which stages of the SDLC transition from human responsibility to full AI control. Figure 1 illustrates this progression across the SDLC stages. We emphasize that the **responsible and safe** use of AI is central to our vision.

Level I—Code Autonomy represents the first step towards autonomy. At this level, AI takes full ownership of system design and implementation, without requiring line-by-line human approval. Humans retain responsibility for oversight-critical stages, including testing, deployment, and demand proposal and curation. The key difference from modern development practice is that humans only review AI-generated output at the pull-request granularity, rather than co-authoring code. Current AI-assisted coding, where AI augments but does not replace humans, is the precursor to this level.

Level II—Pipeline Autonomy marks a qualitative shift, where AI assumes responsibility for the entire development pipeline from system design and implementation through testing and deployment. At this level, AI determines and executes all intermediate stages. Humans neither author nor review code, and their role is limited to specifying high-level demands, such as articulating what the system is expected to do, accepting the resulting behavior, and repeating the lifecycle. Therefore, this level presumes a sufficiently complete specification of the demand together with comprehensive automated verification, since no human inspects the intermediate artifacts.

Level III—Demand Autonomy completes the end-to-end autonomy across the SDLC, where AI systems not only execute the development pipeline but also determine *what to build*. This can include decisions ranging from adding a new feature per request or a complete refactoring of the entire

project. AI agents proactively identify demands from operational signals, user behavior, and security advisories rather than relying on externally specified demands. This eliminates the final human role in the loop, enabling fully autonomous software engineering. Even at this level, autonomy is still bounded by a founding mission typically defined by humans at inception. Self-identified demands are expected to extend and refine the system in service of this mission rather than redefine it. Reconciling emergent demands with this initial intent is itself a core challenge at this level.

While our taxonomy outlines a principled progression across levels, real-world adoption may not be smooth. The appeal of rapid prototyping and efficiency gains often outpaces the development of safeguards (e.g., verification and accountability), creating pressure for premature deployment of higher-autonomy systems. In practice, we observe Level II-style end-to-end pipelines deployed where outputs cannot be reliably verified, and Level III-style demand autonomy prototyped without understanding downstream impacts [111, 112, 39, 6]. We also observe a mispractice in which humans adopt Level I or even Level II-style autonomy but place unchecked trust in AI agents, deploying outputs they have neither reviewed nor verified without meaningful oversight. Skipping levels without addressing the underlying issues can produce severe failures without sufficient visibility, including specification drift, security regressions, cascading ecosystem effects, and accountability gaps. Therefore, responsible adoption requires explicit *level gating*: progression should occur only after the corresponding challenges have been resolved. Going beyond the technical considerations, other societal gating criteria are also important, including labor displacement and deskilling. An oranization should not progress to a higher autonomy level merely because it is technically feasible, as the resulting labor and societal costs may not be offset by commensurate benefits. Domain risk further calibrates the attainable level as an orthogonal axis rather than a stage every system passes through. High-assurance domains such as medical devices and avionics ark likely to plateau at Level I or below indefinitely, gated by certification regimes that presuppose human authorship and liability, while low-assurance domains are already approaching Level II (Section C.4).

Our taxonomy targets the autonomous development of software that is long-lived and continuously maintained, including major systems, services, and SaaS applications that are deployed, depended upon, and evolved over long horizons. However, we do note that a large fraction of AI-written code today is instead *single-use*. This can be throwaway scripts, one-off analyses, and disposable glue that a human commissions for an immediate result. This regime is largely orthogonal to our central concerns, as the identified challenges like long-horizon specification, maintenance, and accountability are not applicable due to the single-use nature of the code. However, this does not make single-use autonomy risk-free: misalignment with the user's intent, hallucination, and unintended side effects can still lead to real hazards, which reinforce our central concern for human intent preservation and execution.

2.3 Cross-Cutting Dimensions of Autonomy

While our three-level taxonomy is organized along a single primary axis—the degree of autonomy across SDLC stages, autonomy in software engineering also varies along several additional dimensions. These dimensions are inherently orthogonal, and no single level fully captures or resolves them. We now discuss three key cross-cutting dimensions and their implications across levels.

Specification granularity and completeness consider the level of detail in the instructions provided to AI [47], and determines the degree of autonomy required. Well-structured specifications, such as bug reports with tests and explicit acceptance criteria [50], constrain the problem space and reduce the need for independent judgment. In contrast, sparse or underspecified specifications require AI to infer scope, design decisions, and success criteria [12, 38, 139]. As a result, weaker specifications push even lower-level systems toward challenges associated with higher autonomy, while stronger specifications mitigate autonomy demands across levels.

Temporal autonomy captures how long AI systems can operate coherently without human intervention [70, 81]. This spectrum ranges from *ticket-level* (handling a single task), through *sprint-level* (operating over multiple days on a backlog), to *release-level* (managing an entire release cycle), and finally *continuous* operation (sustained, self-directed operation over months or even years without predefined endpoints). Temporal autonomy is orthogonal to the three-level taxonomy; for example, a Level I system may achieve sprint-level operation if appropriately scaffolded, while a Level II system may remain limited to the ticket level due to constraints such as the context window.

Oversight mode describes the nature of human oversight at each level. Across all levels, human oversight may take several forms [4, 116, 88, 45]: *collaborative co-specification* (human and agent jointly shape the product requirements document), *action-level approval* (human approves every AI action before execution), *plan-level approval* (human reviews and approves the AI's plan or pull request without inspecting intermediate actions), boundary/guardrail autonomy (AI operates independently within predefined policy constraints), and *monitor-only / smoke-test-and-rollback* (automated monitoring and rollback mechanisms provide the safety net). Oversight mode is sensitive to the domain of application, which cannot be treated as a monolithic property of each level. We analyze in Section C.4 how domain risk profiles calibrate both the attainable autonomy level and the appropriate oversight mode.

3 Structural Shifts Under Autonomy Taxonomy

Advances towards autonomous software engineering are poised to induce fundamental structural shifts in today's software engineering paradigm. In this section, we systematically examine the key shifts that emerge as AI systems progress toward higher levels of autonomy. We begin by examining how these shifts manifest across stages of the SDLC model (i.e., demand, system design, implementation, testing, and deployment). To this end, we discuss the evolving role of specification, the erosion of abstraction boundaries in design, the emergence of new software forms, and the increasing heterogeneity in testing and auditing. We then extend the discussion to two cross-cutting shifts: multi-agent orchestration and organizational transformation.

3.1 Demand: Specifications as Primary Development Artifacts

As autonomy advances, we expect humans to engage progressively less with code, with AI systems bridging the gap between human intent and concrete implementations. Specifications will become the dominant artifact at the human-AI interface. Within a development cycle, the specification needs to fully capture the proposed demand and the corresponding testing strategy, both explicitly through natural language and implicitly through conventions and mental models. At Level II, AI systems operate from specifications without end-to-end human verification, exposing two structural problems: natural-language demands are ambiguous and underspecified [49, 87, 125, 77, 113], and many quality dimensions (e.g., maintainability, UI/UX suitability, or refactoring correctness) lack gold oracles for automated evaluation [83, 71, 84]. At Level III, the burden shifts further, where AI systems must synthesize specifications. These specifications must be precise enough for implementation and testing while still consistent with the deployed behavior over long-horizon evolution, which current methods can only partially satisfy without human verification [35, 131, 76, 28]. We stress that specification synthesis at Level III is not envisioned as an agent silently inventing intent from static artifacts or isolated self-play. Rather, we envision it to be similar to the a negotiation process, where agents proactively engage stakeholders, customers, and regulators to author and refine demands through intent formalization and ambiguity resolution [49, 87, 125].

A practical difficulty is that maintaining specifications by hand is often harder than maintaining the interaction history that produced it. Therefore, we also anticipate *specification distillation*: rather than authoring a full product requirements document upfront, developers converse with agents to synthesize a coherent, durable specification from the accumulated histories. This practice bridges Levels II and III, as although humans still supply intent through dialogue, the agent will compile the dialogue into the specification that a Level III system would eventually maintain on its own.

Looking forward, we envision that specifications may eventually be the *only* approach that humans control software engineering. As the software grows, the specification of the software becomes a compilation of historical demands and design choices. We expect specifications to approach a near *one-to-one* correspondence with the resulting system, such that the entire software artifact is reconstructible from specifications alone. Moreover, specifications concern not only functional correctness, but also *customization*: software encodes preferences, conventions, architectural invariants, and design rationale. We also note emerging efforts in this direction [68].

3.2 System Design: Abstractions as Permeable Defaults

Modern software is built atop deep stacks of abstractions (e.g., programming languages, frameworks, and libraries) to manage the cognitive limits of human developers. We argue that a key benefit of AI is the superior attention span and context capability, which we expect to continue improving with advances in models and agent harnesses [57, 138]. As a result, the traditional rationale for rigid abstraction layers may weaken. Instead, abstractions may evolve from load-bearing structures into *permeable defaults* that AI systems can selectively bypass when end-to-end optimization across the stack is beneficial. This shift has two immediate consequences. First, the software supply chain will shift from shared *implementations* towards shared *protocols*: agents may modify or re-implement dependencies on demand, treating high-quality reference implementations as adaptable starting points rather than products consumed as-is. Second, it recasts long-term maintenance. When an agent can faithfully reproduce a non-trivial application in hours, rewriting may emerge as a practical alternative to refactoring for a non-trivial class of software. This argument does not apply uniformly. Systems built on tacit knowledge or a long tail of difficult demand integrations can be hard to reconstruct from a specification. However, for the class of software where rewriting is feasible, the cost of producing software, the dependency structure, and the reconstruction criteria become substantially more fluid.

3.3 Implementation: Software Takes on New Forms

Beyond transforming how software is developed, advancing autonomy also challenges the notion of what software is. Today's software is largely a static, hand-authored artifact with interfaces and code paths fixed at release time. As AI systems become both code producers and runtime participants, we expect new forms of software to emerge: *neurosymbolic*, *self-evolving*, *conversational*, and *ephemeral* software.

Neurosymbolic software tightly couples neural and symbolic components by construction, on the premise that neither alone is sufficient for trustworthy autonomous development. In such software, deterministic code handles the predictable, paved paths of computation, while neural components handle the novel and flexible ones. Concrete realizations include runtime monitors and policy engines with AI orchestrators [96, 109, 20] and hybrid pipelines that dynamically route requests across models. *Self-evolving software* [128] treats the deployed system itself as a continuously edited artifact, where AI observes usage and demands, and generates new modifications with minimal human intervention. This dissolves the traditional release boundary, as the system becomes the joint product of its initial specification and the trajectory of agent-authored adaptations applied since deployment However, this carries a caveat for auditing, which requires pinning the exact version or checkpoint even as the system continues to evolve. *Conversational software* [103] replaces fixed UIs and APIs with natural-language conversations as the primary interface, in which users express intent, and the system acts accordingly, tailored to the request. The boundary between using and modifying the software blurs: the same dialogue can drive a query, a customization, or a feature addition, all mediated by an underlying agent that translates intent into vetted execution. These forms are complementary to each other, as a mature deployment may be neurosymbolic in its core, self-evolving at its periphery, and conversational at its interface. Realizing them requires careful balancing of guarantees, utility, and cost.

Ephemeral software [33, 72], on the other hand, is generated on demand to serve a single task and then discarded rather than maintained as a durable artifact. We envision that it will emerge as regenerating the software becomes cheaper than reusing existing pieces of code. As discussed in Section 2, this regime sheds most of the lifecycle concerns that dominate persistent systems. However, intent misalignment, hallucinated behavior, and unintended side effects remain its principal risks, which is why faithful intent capture and precise specification matter even when nothing about the artifact is meant to last.

3.4 Testing and Auditing: From Artifact-Level Validation to Agent-Level Assurance

As autonomy advances, testing and auditing practices will become more complex and more heterogeneous. AI-generated code is structurally diverse: even when given the same specification, different agents can produce implementations that diverge substantially [7, 10, 95]. This structural dispersion challenges traditional testing and auditing approaches, which often rely on recurring implementation patterns, stable conventions, and predictable control/data flows [43]. When functionally equivalent

AI-generated programs vary widely in structure, reviewers and tools must reason over a larger and less standardized space, reducing coverage and increasing the chance that defects or vulnerabilities escape detection. At the same time, the burden of writing and validating tests shifts to AI [83, 71, 84], rendering artifact validation alone to be untrustworthy, as an agent that writes both the implementation and the tests can be consistent yet wrong. This is precisely where reward hacking becomes pervasive [129], and it is fundamentally hard to detect without inspecting the code itself. Even delegating the check to a separate verifier agent would not help, as the verifier and the generator can talk past each other and fail to converge, or co-adapt until the tests merely ratify the implementation's bugs. Therefore, assurance must extend from the artifacts to the agents that produce them. Concretely, this means auditing the agent's specifications, skills, tool permissions, memory, and traces as first-class objects [96, 109, 20].

3.5 Multi-Agent Orchestration: Human-Mirrored Hierarchies to Native Protocols

Autonomous software engineering is naturally multi-agent. Component-wise, different agents operate on different services, each carrying their own designs, specifications, skills, and security policies [7, 10]. Within a single component, agents may also collaborate with specialized roles such as managers and reviewers, that operate on a shared task [22, 79]. As agent collaboration dynamics become more complex, coordination failures will compound [23], such as miscommunication, silent disagreements, and specification drift. New failure modes also surface, such as slopsquatting [112] and toxic-skill propagation [111]. Additionally, we envision that an AI-native coordination hierarchy will emerge. Today's orchestration solutions largely mirror human organizational structures, which are shaped by constraints on bandwidth, attention, and span of control. AI agents are not bound by these factors as they can fork and merge states, broadcast context, and run protocols (e.g., consensus) that are impractical for human teams. We thus predict that new forms of management, collaboration, and accountability, designed around agents' actual capabilities rather than human organizational constraints, will appear, and that these will be critical to the adoption of higher-autonomy software engineering.

3.6 Organization: Human Contribution as Specification, Governance and Oversight

Progression along the taxonomy also implies a redistribution of human contribution, as higher autonomous levels enables smaller teams focusing on specification, security, governance, and accountability rather than implementation [118, 104]. This is a more fundamental compression than that introduced by high-level languages, version control, or cloud infrastructure [19], as entire categories of human labor are absorbed, not merely accelerated.

We envision that several organizational functions absent at scale today will become pivotal, including AI governance, model evaluation, agent trustworthiness assessment, and audit-trail tooling. This also requires new categories of firms, such as third-party AI code auditors and certification bodies for AI-generated software. As the cost of producing functional software drops dramatically, the dynamics of today's SaaS markets will undergo drastic changes. The market may shift toward bespoke, organization-specific AI-generated alternatives, where the resulting long tail of tools may further amplify security and reliability challenges [101].

The economics of running the software also enter the agent's consideration. When a system carries recurring infrastructure cost (like model API costs for neurosymbolic softwares discussed in Section 3.3), decisions about pricing, ad placement, or usage limits become part of sustaining it. Whether such a monetization strategy falls within an autonomous agent's obligations, or remains a reserved human decision, is an open governance question distinct from the development-cost modeling of Section 4. Software engineering education must shift accordingly, with more focus on specification writing, testing, auditing, and system-level reasoning. We expect implementation skills to remain valuable as a foundation rather than as the central output of professional training.

4 Research Directions

The structural shifts discussed in Section 3 surface a spectrum of open problems, from fundamental technical barriers to practical and institutional concerns. In this section, we outline the research directions that we believe are best positioned to address these shifts and to enable a smoother transition

along the autonomy taxonomy. We organize the agenda into five themes, ordered roughly from the most fundamental technical questions to the most applied. Each theme is relevant at multiple levels of the taxonomy, but its urgency, framing, and binding constraints shift as autonomy deepens.

4.1 Specification, Verification, and Behavioral Understanding

The absence of trustworthy oracles is the deepest technical barrier exposed by this taxonomy, and it sharpens at each successive level: at Level II, agents must operate against specifications no human verifies; at Level III, they must synthesize those specifications from first principles. As an example, AI may need to infer from production telemetry and user behavior that a service needs a rate-limiting policy, and then author the precise specification and tests that define its correct behavior.

Autonomous specification synthesis and behavioral understanding. A fully autonomous agent requires methods for synthesizing structured, machine-checkable specifications from first principles without relying on human-provided requirements. Research directions include formal and semi-formal specification languages that are both expressive and amenable to automated verification [60], techniques for inferring behavioral invariants directly from implementations [35, 34], and approaches to maintaining specification–implementation consistency across successive development cycles. Equally important are tractable program analysis methods that allow an autonomous agent to construct a reliable model of its own software's behavior, detect behavioral regressions, and identify emergent interactions between components before they manifest in production [29, 126]. Recent LLM-driven extensions of these classical lines, like using language models to propose program invariants [100], infer specifications from existing code [131, 76], and translate natural-language intent into formal postconditions or preconditions through atomic requirement decomposition and traceable refinement [145], suggest a viable architecture, with VERINA [144] providing a contemporary baseline for end-to-end function-level synthesis. However, three structural obstacles remain. At realistic granularity, repository-scale specification benchmarks substantially outpace what current frontier models can produce [26], indicating that synthesis at production granularity is much harder than function-level results suggest. At the methodological level, recent analysis argues that any autoformalization of natural-language intent must ultimately defer to a human to confirm the specification matches intent [28], raising the question of what a genuinely Level-III pipeline could even look like. At the systems level, few published methods address specification drift across many development cycles, and the benchmarks that begin to probe this regime report sharp quality degradation over successive iterations [94], despite this being precisely the regime where autonomous agents would operate. Progress on all three obstacles hinges on an evaluation infrastructure that barely exists today. Therefore, we explicitly call for repository-scale specification-synthesis datasets, benchmarks that measure specification-implementation consistency and drift across many development cycles. Without such datasets and evaluations, any claims of Level-II and Level-III capability cannot be substantiated.

Specification-grounded verification of LLM-generated code. Even when code passes all available tests, hidden defects may remain that cannot be detected automatically without formal verification tools [74, 82]. Scalable, trustworthy methods for formally or empirically verifying the correctness of LLM-generated code are an urgent research need [142, 144, 143]. Promising directions include combining LLM-generated code and formal specifications in proof assistants such as Dafny, Lean, or Coq [86, 120, 89, 2, 1], using invariant inference and SMT-based techniques to find counterexamples [100, 24, 53], and developing domain-specific verification oracles [14, 15]. There has been early proof of feasibility. [114] demonstrates that closed-loop systems combining LLM proposals with proof-assistant feedback accept the majority of correct Dafny programs while rejecting flawed ones on small benchmarks. [133, 16] show that neuro-symbolic loop-invariant inference combined with LLMs obtains strong coverage on classical invariant benchmarks. [2] shows that tree-search-based bootstrapping yields steadily improving Verus pipelines without large amounts of labeled data. Bridging the gap between LLM-based code generation and formal methods is a promising but largely unexplored direction, with several barriers remaining. First, capability is highly uneven across verifiers: end-to-end pass rates on identical algorithmic tasks reach roughly >80% on Dafny but only 27% on Lean [21], with the most expressive logic consistently the hardest. Additionally, performance collapses on programs requiring global invariants, ghost state, or rich algebraic reasoning [149], and even when local verification succeeds, composing specifications and proofs across modules remains brittle [41, 135]. At repository scale, [143] further shows that frontier agents can discharge many local proof obligations yet still fail to reason about cross-module invariants or construct the reusable

lemma libraries required for complete verification. Finally, natural-language descriptions appear to provide surprisingly little verification uplift [21], suggesting the bottleneck lies in formal reasoning capacity rather than in retrieval or context.

4.2 Security, Governance, and Accountability

Security becomes a first-class research problem once agents author and deploy code without human review, and accountability questions follow close behind. Progress is needed on both technical defenses and the institutional scaffolding that surrounds them.

Trustworthy automatic code auditing. Reliable pipelines for security and quality review that do not require continuous human-in-the-loop involvement are a prerequisite for Level II. Research on LLM-assisted static analysis [64, 62], automated code review [123, 63, 122, 115], and audit trail generation for AI-produced code is needed. Hybrid and agentic pipelines have begun to demonstrate concrete uplift over traditional analyzers [17, 69, 134, 64]. Static-analyzer feedback on regenerated code reduces residual vulnerabilities on Python security benchmarks [3], and structured secure-coding-practice reasoning lifts model security rates across families [98]. A trustworthy auditing system must not only catch known vulnerability patterns but generalize to novel attack surfaces introduced by AI-generated code, which may differ structurally from human-authored code in ways that evade existing static analysis tools [27, 55]. We identify two primary limits that constrain progress in this direction. First, current systems still inherit the rule sets and per-project specifications of their underlying analyzers, so novel structural patterns in AI-generated code can evade them. Second, defenses against adaptive prompt-injection attacks remain weak: a recent systematization of dozens of studies finds that adaptive attack success against state-of-the-art defenses remains substantial [78].

Legal and institutional frameworks for AI-authored software. Extending existing liability and accountability frameworks to accommodate fully autonomous development agents requires collaboration across law, ethics, and organizational design. Research directions include new standards for audit trails and attribution in AI-generated codebases, institutional models for assigning accountability when no human reviewed the relevant code, and regulatory frameworks that create meaningful incentives for safe autonomous development. In practice, liability concerns are likely to keep a human in the loop. If an agent-built application leaks customer data, the accountability should be borne by the developer or organization that deployed the agent, not by the provider of the underlying model. This is itself a strong incentive to retain meaningful human oversight as autonomy increases. An initial scaffolding now exists: OWASP's Top 10 for Agentic Applications [96], Microsoft's open-source Agent Governance Toolkit [109], and AWS's Agentic AI Security Scoping Matrix [20] provide first-pass policy and identity primitives, while regulatory deadlines—EU AI Act high-risk obligations from August 2026 and the Colorado AI Act from June 2026—are forcing enterprises to operationalize them. On the legal side, the GitHub Copilot class action established that breach-of-license claims around AI-trained code can survive dismissal even when DMCA copyright claims fail [52], but core questions of liability allocation when no human reviewed the deployed code remain unresolved, and audit-trail standards for AI-authored code remain fragmented across vendors.

4.3 System and Language Design for Autonomous Agents

Current programming languages, version control conventions, project structures, and design patterns were built around the assumptions of human authorship. As agents become primary authors, these foundations deserve systematic re-examination.

Agent-centric coding models. New coding models are needed, e.g., abstractions that play to the strengths of AI agents while making agent-generated code more legible to human reviewers. At the limit, an agent could directly author machine code with no intermediate source language. This includes research into code representations [99], literate-programming-style outlines for code [108], and commit and comment structures that facilitate both AI generation and human comprehension. Practical progress has been concentrated on agent-facing artifacts rather than on languages themselves: external memory and retrieval architectures such as repository-level planning [11] materially improve agent grounding on large codebases, and emerging design-pattern proposals—filesystem-based task locking and `git` as a conflict-resolution primitive in the recent 16-agent C-compiler experiment [22], behavioral-contract protocols such as SEMAP [79]—begin to codify reusable structures for autonomous development. The unresolved question is whether genuinely agent-native programming

languages and design patterns are needed, or whether structured artifacts on top of conventional languages will suffice; no published taxonomy yet plays the role for agents that classical pattern catalogs [146] played for human teams.

Versioned reasoning and long-horizon planning. Agents must be able to track the provenance of their decisions, recognize when prior reasoning is invalidated by subsequent changes, and update their understanding accordingly. Research on decision provenance systems, long-horizon planning for software evolution [67], and mechanisms for preserving architectural coherence across agent reinitializations or model updates is directly relevant to enabling Level III systems to sustain software over multi-year timescales. Recent benchmarks have begun to instrument the gap directly. EvoClaw [31] evaluates agents across dependency-constrained milestones from real OSS repositories and reports that frontier agents retain feature-implementation capability but fail to prevent regressions, with overall scores collapsing from above 80% on isolated tasks to at most 38% in the continuous-evolution setting; SWE-CI [25] formalizes the same phenomenon as continuous-integration-style cycles. Production tooling that maintains conversation history and cross-repository dependency maps provides a partial mitigation but does not address specification problems like implementation drift across model updates or agent reinitializations, leaving long-horizon coherence as the binding constraint.

Another related open question concerns the unit of versioning itself. If, as we argue in Section 3.1, the artifact becomes reconstructible from its specification alone, it is no longer obvious whether teams should version the specification, the generated code, or both. How branching, merging, and provenance are re-conceived for this setting is an open direction closely matched to the shift toward autonomous development.

Multi-agent coordination protocols. Enabling multiple agents to collaborate without conflicting changes or mutually incompatible assumptions requires new protocols for inter-agent coordination, conflict resolution, and shared world-modeling [59]. The ambiguity and the lack of machine-checkable semantics of natural language render it unsuitable as the media, since agents may silently disagree, drift, or talk past one another. Research into structured communication standards for agent teams, formal models of agent objective hierarchies, and mechanisms for maintaining a consistent product vision across autonomous subsystems represents a critical open direction at higher levels of autonomy. Initial protocols have demonstrated measurable gains: SEMAP, combining behavioral contracts with structured messaging and an explicit lifecycle state machine, reports a substantial reduction in failures at function-level and deployment-level coding [79], and standardized message-format proposals [59] represent first steps toward interoperability. However, MAST [23] shows that tactical fixes (e.g., better prompts) yield only modest improvements and leave overall failure rates above acceptable production thresholds, indicating that structural protocol redesign is the binding constraint. Moreover, all current protocols presuppose a single team's trust boundary, leaving the cross-organizational case raised by the ecosystem-evolution challenge largely unaddressed.

4.4 Interfaces for Human–Agent Collaboration and Comprehension

Through Levels I and II, the principal human role shifts from authoring to reviewing. Supporting this shift requires progress on knowledge transfer into agents, interpretability of agent reasoning, and tooling for human comprehension of agent-authored code.

Co-design interfaces for human-agent collaboration. Designing seamless, low-friction interfaces for human–agent knowledge transfer is an open research problem spanning both HCI and systems design [121, 65]. This includes structured onboarding protocols, persistent agent memory for project-specific conventions, and interfaces that allow developers to express architectural constraints and style preferences in a form the agent can reliably act on [58, 36, 7]. A central but underexplored axis here is *personalization*: as code becomes cheap to (re)generate, the personalization layer transferred to an agent becomes the durable artifact of software engineering, increasingly more so than the code itself. Research on multi-scale personalization (individual, team, organization), progressive elicitation that infers preferences from accept/reject and review signals rather than asking developers to specify them upfront, and portable, vendor-neutral representations that prevent personalization from being silently locked into a single agent ecosystem is therefore directly on the critical path for Level I and Level II adoption.

Furthermore, an underexplored axis is *human-human* collaboration mediated by agents. When multiple developers each drive their own Level II-or-higher agents, it is unclear which coordination practices remain effective, or even reasonable. Developments may collapse toward one-person teams or fully agent-native coordination structures in bigger groups. Characterizing effective practices for agent-mediated collaboration among humans is an open problem alongside the human-agent interface design.

Human understanding of agent-maintained codebases. When AI agents are the primary authors and maintainers of a codebase, preserving human comprehensibility requires deliberate design [61, 116]. This includes automatically generated architectural overviews, agent-produced change rationale, and visualization tools that help developers orient themselves in AI-generated code. Industrial case studies provide useful yet mixed starting evidence. [116] finds that AI-generated code is broadly comparable in readability to human-written code. In contrast, [61] documents agent-driven projects requiring substantially more effort to comprehend than their human-developed counterparts. Initial interpretability work on the code domain (e.g., token-level rationales [56] and anchored explanations of generated code [137]) begins to address the problem at the snippet level. However, faithfulness remains a serious obstacle: chain-of-thought reasoning traces are not guaranteed to reflect the model's underlying computation [75, 136], and the kinds of explanations that most matter at the SE scale (like design-tradeoff justifications, test-strategy choices, and architectural-decision rationales) are largely unaddressed by existing interpretability methods. Ensuring AI-generated software is comprehensible to humans must be treated as a first-class engineering concern, not an afterthought.

4.5 Economic Modeling of Hybrid Development

Even Level I autonomy will be adopted at scale only when its returns exceed its costs, yet rigorous cost models for agentic development pipelines are largely absent from the literature.

Return on investment of autonomy. Measuring the productivity gains of autonomous development against its costs requires careful experimental design and realistic benchmarks that go beyond toy tasks. The empirical picture is currently contested: some studies report speedups [150], while others suggest long-term performance decay [48, 44], and a randomized controlled trial found that early-2025 AI tools *increased* experienced developers' task completion time by roughly 19% despite a self-reported 20% speedup [13]. This hints that the absence of metrics that jointly capture cost, supervision burden, and maintenance overhead is the central open problem [37]. We therefore argue that cost, supervision burden, and maintenance overhead should be reported jointly with success rate, rather than as separate, decontextualized metrics. Understanding the true cost of autonomous development, both immediate ones like token cost and long-horizon ones like technical debt, is essential for rational investment decisions.

Resource allocation in hybrid teams. Given a budget of human and computational resources, a key problem is the distribution of effort between human developers and AI agents across the stages of the development pipeline [40]. Optimization models for resource allocation in hybrid human-AI teams represent a largely unexplored research direction with direct practical implications for team structure and tooling decisions. Building such a basis is a prerequisite for the rational deployment decisions our taxonomy assumes, and it cuts across the cost-effectiveness, human-agent collaboration, and benchmarking themes above [30].

5 Alternative Views

Although we describe AI autonomy as the organizing axis of the future of software engineering, two alternative positions deserve serious consideration.

The durable human–AI partnership view. The first alternative holds that software engineering will remain a durable human–AI partnership, in which agents and humans specialize in different parts of the lifecycle rather than one progressively delegating to the other. The strongest version of this position is not that today's tools happen to resemble partnerships, but that certain activities may be *categorically non-delegable* regardless of capability. For example, bearing legal or organizational accountability for a system's consequences requires not only competence but standing to bear consequences, and curating demands can rest on tacit knowledge, negotiation, and institutional legitimacy that exist nowhere in text form. We acknowledge that this is a structural difficulty and

an open problem (Section 4.2) However, we believe that our position remains valuable due to two clarifications. First, partnership and autonomy are not mutually exclusive. Oversight mode cuts across all levels of our taxonomy (Section 2.3), so even Level II systems may operate under human governance, and a system can run autonomously while an organization retains legal responsibility for deploying and guardrailing it. Second, the partnership view should be read as a concrete, falsifiable prediction about *which classes of software* remain human-governed rather than an abstract endpoint. We expect partnership to persist the longest (Section C.4) in domains with either strict certification regimes or extensive dependency on knowledge that are hard to reconstruct from a specification (Section 3.2).

The scaling view. The second alternative accepts our roadmap but argues that the challenges we identify will be resolved through continued scaling. In this view, the verification bottlenecks of Section 4.1 merely reflect the limitation of current training data and regimes rather than structural limits. Eventually, the reliance on the topics identified in this paper like verification, auditing, and human-AI interaction will dissolve as models improve. Against this, we first note that although we agree that models will continue to improve in the projected future, we believe that formal guarantees will still remain necessary for many fields that have ultra-low error tolerance (Section C.4). Scaling does not provide any guarantees by itself, and given the massive volume of code and the stochastic nature of the models, it is very dangerous to fully rely on an agent without any guarantees, either formally or empirically. Additionally, while scaling may still significantly improve model capability, it cannot fully resolve the challenges brought by the collaborative nature of software engineering, like specification ambiguity, human-AI collaboration, and maintenence difficulties. Our position emphasizes these domains outside of the pure capability domain since, although they are central to the software development life cycle, they are more overlooked compared to the recent progress on pure implementation-based domains.

Overall, the structural shifts we highlight remain worth studying under either hypothesis. Whether the trajectory aligns more with the full autonomy roadmap or terminates in a stable partnership, responsibility over the SDLC stages is already migrating. Recent developments such as "vibe coding" workflows and systems like Claude Code [8] already emphasize intent specification and end-to-end autonomy with minimal human intervention. Our work provides a shared vocabulary and safeguards to track this migration process in the hope of ensuring a responsible transition.

6 Conclusion

In this position paper, we systematically study the trajectory towards autonomous software engineering. We introduce a three-level taxonomy of AI autonomy in software engineering and the corresponding cross-cutting challenges and predicted shifts. Across all levels, we identify a unifying challenge of preserving and faithfully executing human intent as direct human control over the software lifecycle recedes. We also analyze and propose promising research directions for resolving these challenges and achieving a higher level of autonomy. We hope this work helps to understand the current landscape of autonomous software engineering, provides insights about the future evolution of autonomous software engineering, and encourages future research in these directions.

References

[1] Shubham Agarwal, Alexander Krentsel, Shu Liu, Mert Cemri, Audrey Cheng, Rui Meng, Tomas Pfister, Chun-Liang Li, Sylvia Ratnasamy, Aditya Parameswaran, Matei Zaharia, Ion Stoica, and Mohsen Lesani. Inductive deductive synthesis: Enabling ai to generate formally verified systems, 2026.

[2] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. Alphaverus: Bootstrapping formally verified code generation through self-improving translation and treefinement. *arXiv preprint arXiv:2412.06176*, 2024.

[3] Kamel Alrashedy, Abdullah Aljasser, Pradyumna Tambwekar, and Matthew Gombolay. Can llms patch security issues? *arXiv preprint arXiv:2312.00024*, 2023.

[4] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi T. Iqbal, Paul N. Bennett, Kori Inkpen, and Jaime Teevan.

Guidelines for human-ai interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

[5] Anthropic. Disrupting the first reported ai-orchestrated cyber espionage campaign. https://www.anthropic.com/news/disrupting-AI-espionage, nov 2025. Accessed: 2026-05-02.

[6] Anthropic. Mitigating the risk of prompt injections in browser use. https://www.anthropic.com/news/prompt-injection-defenses, nov 2025. Accessed: 2026-05-02.

[7] Anthropic. Agent skills. https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview, 2026. Accessed: 2026-05-02.

[8] Anthropic / Community Sources. Claude code. https://digitalocean.com/resources/articles/claude-code-alternatives, 2026. Terminal-based AI coding assistant built on Anthropic's Claude models.

[9] Anysphere / Community Sources. Cursor (ai coding platform). https://zoer.ai/posts/zoer/cursor-vs-copilot-vs-claude-code-2026, 2026. AI-centric code editor and multi-purpose coding assistant.

[10] Augment Code. #1 open-source agent on SWE-bench Verified by combining Claude Sonnet 3.7 and O1. https://www.augmentcode.com/blog/1-open-source-agent-on-swe-bench-verified-by-combining-claude-3-7-and-o1, 2025. Accessed: 2026-05-02.

[11] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698, 2024.

[12] Muneera Bano. Addressing the challenges of requirements ambiguity: A review of empirical literature. In *2015 IEEE Fifth International Workshop on Empirical Requirements Engineering (EmpiRE)*, 2015.

[13] Joel Becker, Nate Rush, Beth Barnes, and David Rein. Measuring the impact of early-2025 ai on experienced open-source developer productivity. https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/, 07 2025.

[14] Dirk Beyer. Competition on software verification and witness validation: Sv-comp 2023. In *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings, Part II*, page 495–522, Berlin, Heidelberg, 2023. Springer-Verlag.

[15] Dirk Beyer and Alexander K. Petrenko. Linux driver verification. In *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Applications and Case Studies - Volume Part II*, ISoLA'12, page 1–6, Berlin, Heidelberg, 2012. Springer-Verlag.

[16] Varun Bharti, Shashwat Jha, Dhruv Kumar, and Pankaj Jalote. Loop invariant generation: A hybrid framework of reasoning optimised llms and smt solvers. *arXiv preprint arXiv:2508.00419*, 2025.

[17] Google BigSleep. From naptime to big sleep: Using large language models to catch vulnerabilities in real-world code. 2024.

[18] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.

[19] Frederick P. Brooks. No Silver Bullet – Essence and Accidents of Software Engineering. In *Information processing '86*. Elsevier Science Publishers B.V., 1986.

[20] Aaron Brown and Matt Saner. The agentic AI security scoping matrix: A framework for securing autonomous AI systems. https://aws.amazon.com/blogs/security/the-agentic-ai-security-scoping-matrix-a-framework-for-securing-autonomous-ai-systems/, nov 2025. Accessed: 2026-05-02.

[21] Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, et al. A benchmark for vericoding: formally verified program synthesis. *arXiv preprint arXiv:2509.22908*, 2025.

[22] Nicholas Carlini. Building a c compiler with a team of parallel claudes. `https://www.anthropic.com/engineering/building-c-compiler`, Feb 2026. Anthropic Engineering Blog; Accessed: 2026-03-03.

[23] Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

[24] Saikat Chakraborty, Shuvendu K Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. Ranking llm-generated loop invariants for program verification. *arXiv preprint arXiv:2310.09342*, 2023.

[25] Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. Swe-ci: Evaluating agent capabilities in maintaining codebases via continuous integration. *arXiv preprint arXiv:2603.03823*, 2026.

[26] Zaoyu Chen, Jianbo Dai, Boyu Zhu, Jingdong Wang, Huiming Wang, Xin Xu, Haoyang Yuan, Zhijiang Guo, and Xiao-Ming Wu. Codespecbench: Benchmarking llms for executable behavioral specification generation. *arXiv preprint arXiv:2604.12268*, 2026.

[27] Domenico Cotroneo, Cristina Improta, and Pietro Liguori. Human-written vs. ai-generated code: A large-scale study of defects, vulnerabilities, and complexity, 2025.

[28] Aaron Councilman, David Jiahao Fu, Aryan Gupta, Chengxiao Wang, David Grove, Yu-Xiong Wang, and Vikram Adve. Towards formal verification of llm-generated code from natural language prompts. *arXiv preprint arXiv:2507.13290*, 2025.

[29] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[30] Dominik Dellermann, Adrian Calma, Nikolaus Lipusch, Thorsten Weber, Sascha Weigel, and Philipp Ebel. The future of human-ai collaboration: a taxonomy of design knowledge for hybrid intelligence systems. *arXiv preprint arXiv:2105.03354*, 2021.

[31] Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, et al. Evoclaw: Evaluating ai agents on continuous software evolution. *arXiv preprint arXiv:2603.13428*, 2026.

[32] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

[33] Chris Ellis. Ephemeral software: Ui, data, and functions in an ai-first world, 2025. Engineered Intelligence.

[34] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, pages 213–224, 1999.

[35] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[36] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, 2024.

[37] Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie M Zhang, Pinjia He, and Ahmed E Hassan. Swe-effi: Re-evaluating software ai agent system effectiveness under resource constraints. *arXiv preprint arXiv:2509.09853*, 2025.

[38] Henning Femmer, Daniel Méndez Fernández, Stefan Wagner, and Sebastian Eder. Rapid quality assurance with requirements smells. *ArXiv*, abs/1611.08847, 2016.

[39] Forcepoint X-Labs. Indirect prompt injection in the wild: X-labs finds 10 ipi payloads. `https://www.forcepoint.com/blog/x-labs/indirect-prompt-injection-payloads`, apr 2026. Accessed: 2026-05-02.

[40] Andrew Fuchs, Andrea Passarella, and Marco Conti. Optimizing risk-averse human-ai hybrid teams. In *2024 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 117–124. IEEE, 2024.

[41] Robert Joseph George, Carson Eisenach, Udaya Ghai, Dominique Perrault-Joncas, Anima Anandkumar, and Dean Foster. Bridge: Building representations in domain guided program verification. *arXiv preprint arXiv:2511.21104*, 2025.

[42] GitClear. Coding on Copilot: 2023 data suggests downward pressure on code quality (incl. 2024 projections). `https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality`, 2024. Accessed: 2026-05-02.

[43] Pavlína Wurzel Gonçalves, Pooja Rani, Margaret-Anne Storey, Diomidis Spinellis, and Alberto Bacchelli. Code review comprehension: Reviewing strategies seen through code comprehension theories. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pages 589–601. IEEE, 2025.

[44] Google Cloud. 2024 accelerate state of DevOps report (DORA). `https://cloud.google.com/blog/products/devops-sre/announcing-the-2024-dora-report`, 2024. Accessed: 2026-05-02.

[45] Google SRE. Canarying releases. `https://sre.google/workbook/canarying-releases/`, 2018. Accessed: 2026-05-02.

[46] Alex Gu, Naman Jain, Wen-Ding Li, Manish Shetty, Yijia Shao, Ziyang Li, Diyi Yang, Kevin Ellis, Koushik Sen, and Armando Solar-Lezama. Challenges and paths towards ai for software engineering, 2025.

[47] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[48] Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. Speed at the cost of quality: How cursor ai increases short-term velocity and long-term complexity in open-source projects. *arXiv preprint arXiv:2511.04427*, 2025.

[49] Haoxiang Jia, Robbie Morris, He Ye, Federica Sarro, and Sergey Mechtaev. Automated repair of ambiguous problem descriptions for llm-based code generation. *arXiv preprint arXiv:2505.07270*, 2025.

[50] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.

[51] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

[52] Joseph Saveri Law Firm and Matthew Butterick. Github copilot litigation. `https://githubcopilotlitigation.com/`, 2024. Accessed: 2026-05-02.

[53] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models. *arXiv preprint arXiv:2311.07948*, 2023.

[54] Andrej Karpathy. Post on x (twitter). https://x.com/karpathy/status/2024583544157458452, 2025. Accessed: 2026-03-09.

[55] Mohammed Kharma, Soohyeon Choi, Mohammed AlKhanafseh, and David Mohaisen. Security and quality in llm-generated code: A multi-language, multi-model analysis. *arXiv preprint arXiv:2502.01853*, 2025.

[56] Dipin Khati, Daniel Rodriguez-Cardenas, David N Palacio, Alejandro Velasco, and Denys Poshyvanyk. Enabling global, human-centered explanations for llms: From tokens to interpretable code and test generation. *arXiv preprint arXiv:2503.16771*, 2025.

[57] Yuri Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. Babilong: Testing the limits of llms with long context reasoning-in-a-haystack. *Advances in Neural Information Processing Systems*, 37:106519–106554, 2024.

[58] Shuvendu K Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, et al. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950*, 2022.

[59] Cheryl Lee, Chunqiu Steven Xia, Longji Yang, Jen-tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R Lyu. A unified debugging approach via llm-based multi-agent synergy. *arXiv preprint arXiv:2404.17153*, 2024.

[60] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[61] Mark Levison. Aigenerated code quality and the challenges we all face, 2026. Blog post, published February 2026.

[62] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An LLM-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1), 2024.

[63] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047, 2022.

[64] Ziyang Li, Saikat Dutta, and Mayur Naik. Llm-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238*, 2024.

[65] Jenny T Liang, Chenyang Yang, and Brad A Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[66] Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The swe-bench illusion: When state-of-the-art llms remember instead of reason. *arXiv preprint arXiv:2506.12286*, 2025.

[67] Wilson Lin. Scaling long-running autonomous coding. https://cursor.com/blog/scaling-agents, January 2026. Cursor Blog.

[68] littleround. OPPF: Open-prompt project format. https://github.com/camelop/oppf, 2026. Accessed: 2026-07-21.

[69] Dongge Liu, Oliver Chang, Jonathan metzman, Martin Sablotny, and Mihai Maruseac. OSS-Fuzz-Gen: Automated Fuzz Target Generation, May 2024.

[70] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

[71] Kaibo Liu, Zhenpeng Chen, Yiyang Liu, Jie M Zhang, Mark Harman, Yudong Han, Yun Ma, Yihong Dong, Ge Li, and Gang Huang. Llm-powered test case generation for detecting bugs in plausible programs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 430–440, 2025.

[72] Shu Liu, Alexander Krentsel, Shubham Agarwal, Mert Cemri, Ziming Mao, Soujanya Ponnapalli, Alexandros G. Dimakis, Sylvia Ratnasamy, Matei Zaharia, Aditya Parameswaran, and Ion Stoica. The time is here for just-in-time systems: Challenges and opportunities, 2026.

[73] Yue Liu, Yanjie Zhao, Yunbo Lyu, Ting Zhang, Haoyu Wang, and David Lo. " your ai, my shell": Demystifying prompt injection attacks on agentic ai coding editors. *arXiv preprint arXiv:2509.22040*, 2025.

[74] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. Dafnybench: A benchmark for formal software verification. *arXiv preprint arXiv:2406.08467*, 2024.

[75] Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 305–329, 2023.

[76] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. Specgen: Automated generation of formal program specifications via large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 16–28. IEEE, 2025.

[77] Zhi Ma, Cheng Wen, Zhexin Su, Xiao Liang, Cong Tian, Shengchao Qin, and Mengfei Yang. Bridging natural language and formal specification–automated translation of software requirements to ltl via hierarchical semantics decomposition using llms. *arXiv preprint arXiv:2512.17334*, 2025.

[78] Narek Maloyan and Dmitry Namiot. Prompt injection attacks on agentic coding assistants: A systematic analysis of vulnerabilities in skills, tools, and protocol ecosystems. *International Journal of Open Information Technologies*, 14(2):1–10, 2026.

[79] Zhenyu Mao, Jacky Keung, Fengji Zhang, Shuo Liu, Yifei Wang, and Jialong Li. Towards engineering multi-agent llms: A protocol-driven approach. *arXiv preprint arXiv:2510.12120*, 2025.

[80] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 2024 Network and Distributed System Security Symposium (NDSS)*, 2024.

[81] METR. Task-completion time horizons of frontier ai models. `https://metr.org/time-horizons/`, apr 2026. Accessed: 2026-05-02.

[82] Md Rakib Hossain Misu, Cristina V Lopes, Iris Ma, and James Noble. Towards ai-assisted synthesis of verified dafny methods. *Proceedings of the ACM on Software Engineering*, 1(FSE):812–835, 2024.

[83] Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. Test oracle automation in the era of llms. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–24, 2025.

[84] Davide Molinelli, Luca Di Grazia, Alberto Martin-Lopez, Michael D Ernst, and Mauro Pezze. Do llms generate useful test oracles? an empirical study with an unbiased dataset. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 278–290. IEEE, 2025.

[85] Alfred Santa Molison, Marcia Moraes, Glaucia Melo, Fabio Santos, and Wesley K. G. Assuncao. Is llm-generated code more maintainable & reliable than human-written code?, 2025.

[86] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer, 2021.

[87] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering*, 1(FSE):2332–2354, 2024.

[88] National Institute of Standards and Technology. Artificial intelligence risk management framework: Generative artificial intelligence profile. `https://nvlpubs.nist.gov/nistpubs/ai/NIST.AI.600-1.pdf`, 2024. Accessed: 2026-05-02.

[89] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[90] Oasis Security Research. Claude.ai prompt injection data-exfiltration vulnerability ("claudy day"). `https://www.oasis.security/blog/claude-ai-prompt-injection-data-exfiltration-vulnerability`, mar 2026. Accessed: 2026-05-02.

[91] OpenAI. Openai codex. `https://en.wikipedia.org/wiki/OpenAI_Codex`, 2021. AI model for code translation and generation, foundational to GitHub Copilot.

[92] OpenAI Frontier Evals Team. Why we no longer evaluate swe-bench verified. `https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/`, 2026. Accessed: 2026-05-02.

[93] OpenClaw Contributors. Openclaw: Open-source autonomous ai agent. `https://openclaw.ai/`, 2026. Accessed: 2026-03-17.

[94] Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. Slopcodebench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026.

[95] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of chatgpt in code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–28, 2025.

[96] OWASP Gen AI Security Project. Owasp top 10 for agentic applications for 2026. `https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/`, dec 2025. Accessed: 2026-05-02.

[97] Ipek Ozkaya. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software*, 40(3):4–8, 2023.

[98] Rupam Patir, Keyan Guo, Haipeng Cai, and Hongxin Hu. Fortifying llm-based code generation with graph-based reasoning on secure coding practices. *arXiv preprint arXiv:2510.09682*, 2025.

[99] Indraneil Paul, Goran Glavaš, and Iryna Gurevych. Ircoder: Intermediate representations make language models robust multilingual code generators. *arXiv preprint arXiv:2403.03894*, 2024.

[100] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *International Conference on Machine Learning*, pages 27496–27520. PMLR, 2023.

[101] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*, pages 2785–2799, 2023.

[102] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2014.

[103] Nicole M Radziwill and Morgan C Benton. Evaluating quality of chatbots and intelligent conversational agents. *arXiv preprint arXiv:1704.04579*, 2017.

[104] Abhik Roychoudhury, Corina Pasareanu, Michael Pradel, and Baishakhi Ray. Ai software engineer: Programming with trust. *arXiv preprint arXiv:2502.13767*, 2025.

[105] SAE International. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. Technical Report J3016_202104, SAE International, 2021. SAE Recommended Practice.

[106] Amirali Sajadi, Kostadin Damevski, and Preetha Chatterjee. How safe are ai-generated patches? a large-scale study on security risks in llm and agentic automated program repair on swe-bench. *arXiv preprint arXiv:2507.02976*, 2025.

[107] Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. Using ai-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Information and Software Technology*, 178:107610, 2025.

[108] Kensen Shi, Deniz Altınbüken, Saswat Anand, Mihai Christodorescu, Katja Grünwedel, Alexa Koenings, Sai Naidu, Anurag Pathak, Marc Rasi, Fredde Ribeiro, et al. Natural language outlines for code: Literate programming in the llm era. *arXiv preprint arXiv:2408.04820*, 2024.

[109] Imran Siddique. Introducing the agent governance toolkit: Open-source runtime security for AI agents. `https://opensource.microsoft.com/blog/2026/04/02/introducing-the-agent-governance-toolkit-open-source-runtime-security-for-ai-agents/`, apr 2026. Accessed: 2026-05-02.

[110] Snyk Security Research. How "clinejection" turned an ai bot into a supply chain attack. `https://snyk.io/blog/cline-supply-chain-attack-prompt-injection-github-actions/`, feb 2026. Accessed: 2026-05-02.

[111] Snyk Security Research. ToxicSkills: Snyk audit finds security flaws in 36% of agent skills, including 76 confirmed malicious payloads. `https://snyk.io/blog/toxicskills-malicious-ai-agent-skills-clawhub/`, 2026. Accessed: 2026-05-02.

[112] Joseph Spracklen, Raveen Wijewickrama, AHM Nazmus Sakib, Anindya Maiti, and Bimal Viswanath. We have a package for you! a comprehensive analysis of package hallucinations by code generating {LLMs}. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3687–3706, 2025.

[113] Ion Stoica, Matei Zaharia, Joseph Gonzalez, Ken Goldberg, Koushik Sen, Hao Zhang, Anastasios Angelopoulos, Shishir G. Patil, Lingjiao Chen, Wei-Lin Chiang, and Jared Q. Davis. Specifications: The missing link to making the development of llm systems an engineering discipline, 2024.

[114] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification*, pages 134–155. Springer, 2024.

[115] Tao Sun, Jian Xu, Yuanpeng Li, Zhao Yan, Ge Zhang, Lintao Xie, Lu Geng, Zheng Wang, Yueyan Chen, Qin Lin, et al. Bitsai-cr: Automated code review via llm in practice. *arXiv preprint arXiv:2501.15134*, 2025.

[116] Wannita Takerngsaksiri, Chakkrit Tantithamthavorn, Micheal Fu, Jirat Pasuksmit, Kun Chen, and Ming Wu. Code readability in the age of large language models: An industrial case study from atlassian, 2025.

[117] OpenHands Team. Introducing the openhands index. `https://openhands.dev/blog/openhands-index`, January 2026. Accessed: 2026-03-03.

[118] Valerio Terragni, Annie Vella, Partha Roop, and Kelly Blincoe. The future of ai-driven software engineering. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–20, 2025.

[119] Minh VT Thai, Tue Le, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

[120] The Coq Development Team. The Coq reference manual – release 8.19.0. `https://coq.inria.fr/doc/V8.19.0/refman`, 2024.

[121] Christoph Treude and Marco A Gerosa. How developers interact with ai: A taxonomy of human-ai collaboration in software engineering. *arXiv preprint arXiv:2501.08774*, 2025.

[122] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *Proceedings of the 44th international conference on software engineering*, pages 2291–2302, 2022.

[123] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 163–174. IEEE, 2021.

[124] Veracode. 2025 GenAI code security report: Assessing the security of using LLMs for coding. `https://www.veracode.com/wp-content/uploads/2025_GenAI_Code_Security_Report_Final.pdf`, 2025. Accessed: 2026-05-02.

[125] Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. Ambig-swe: Interactive agents to overcome underspecificity in software engineering. In *The Fourteenth International Conference on Learning Representations*, 2026.

[126] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. Can large language models write good property-based tests? *arXiv preprint arXiv:2307.04346*, 2023.

[127] Yao Wan, Zhangqian Bi, Yang He, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip Yu. Deep learning for code intelligence: Survey, benchmark and toolkit. *ACM Computing Surveys*, 2024.

[128] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[129] Hao Wang, Hanchen Li, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. Do androids dream of breaking the game? systematically auditing ai agent benchmarks with benchjack, 2026.

[130] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, 2025.

[131] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*, pages 302–328. Springer, 2024.

[132] Man-Fai Wong, Shangxin Guo, Ching-Nam Hang, Siu-Wai Ho, and Chee-Wei Tan. Natural language generation and understanding of big code for ai-assisted programming: A review. *Entropy*, 25(6):888, 2023.

[133] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. *arXiv preprint arXiv:2310.04870*, 2023.

[134] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[135] Xu Xu, Xin Li, Xingwei Qu, Jie Fu, and Binhang Yuan. Local success does not compose: Benchmarking large language models for compositional formal verification. *arXiv preprint arXiv:2509.23061*, 2025.

[136] Haoran Xue, Gias Uddin, and Song Wang. An empirical study of reasoning steps in thinking code llms. *arXiv preprint arXiv:2511.05874*, 2025.

[137] Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. Ivie: Lightweight anchored explanations of just-generated code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2024.

[138] Yuchen Yan, Yongliang Shen, Yang Liu, Jin Jiang, Mengdi Zhang, Jian Shao, and Yueting Zhuang. Inftythink: Breaking the length limits of long-context reasoning in large language models. *arXiv preprint arXiv:2503.06692*, 2025.

[139] Di Yang, Xinou Xie, Xiuwen Yang, Ming Hu, Yihao Huang, Yueling Zhang, Weikai Miao, Ting Su, Chengcheng Wan, and Geguang Pu. Assessing the impact of requirement ambiguity on llm-based function-level code generation. *arXiv preprint arXiv:2604.21505*, 2026.

[140] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[141] John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024.

[142] Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal mathematical reasoning: A new frontier in ai, 2024.

[143] Zhe Ye, Hantao Lou, Yuechun Sun, Peiyang Song, Zhengxu Yan, Timothe Kasriel, Qingyang Zhang, Kaiyu Yang, Soonho Kong, Jingxuan He, and Dawn Song. Vero: Can AI agents build formally verified software repositories?, 2026.

[144] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*, 2025.

[145] Zhe Ye, Aidan ZH Yang, Huangyuan Su, Zhenyu Liao, Samuel Tenka, Zhizhen Qin, Udaya Ghai, Dawn Song, and Soonho Kong. Intent-aligned formal specification synthesis via traceable refinement. *arXiv preprint arXiv:2604.10392*, 2026.

[146] Artem Zakirullin. Cognitive load is what matters. 2024.

[147] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025.

[148] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. Explainability for large language models: A survey. *ACM Transactions on Intelligent Systems and Technology*, 15(2):1–38, 2024.

[149] Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V Veeravalli, Aarti Gupta, and Sanjeev Arora. Algoveri: An aligned benchmark for verified code generation on classical algorithms. *arXiv preprint arXiv:2602.09464*, 2026.

[150] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Measuring github copilot's impact on productivity. *Communications of the ACM*, 67(3):54–63, 2024.

A A Taxonomy of AI Involvement in Software Engineering

In this section, we introduce a three-level taxonomy that provides a roadmap for automated software engineering. The levels form a progression from today's augmentation tools toward fully autonomous development systems. The structure of this section is as follows. We first provide a review of the traditional software development lifecycle (SDLC). Next, we introduce formally the three levels in the taxonomy

A.1 A More General View of the SDLC model

As aforementioned, the waterfall rendering of Figure 1 is a deliberate simplification. It lines up the five core activities—requirement curation, system design, implementation, testing, and deployment—in a single canonical left-to-right order, which is convenient for understanding and aligning the AI-autonomy levels of Section 2 with a recognizable pipeline. In practice, however, very few production teams follow strict waterfall: requirements evolve through conversation with stakeholders, prototypes often precede detailed design, and testing routinely feeds back into both implementation and architecture. Other modern SDLC models—agile, spiral, V&V, and DevOps—differ precisely in how they sequence these activities, how tightly they couple the feedback loops, and how often they revisit earlier phases.

Agile as an example. Under an agile process, development proceeds in short sprints organized around continuous stakeholder engagement rather than a one-shot specification. The ordering of activities is effectively reversed relative to waterfall at several points: developers may prototype first and refactor the design afterward (so implementation informs system design rather than the other way around); demand is continuously re-negotiated with stakeholders throughout a sprint (so the requirement curation and system design phases remain in active dialogue rather than closing out before implementation begins); and testing, auditing, and review generate rework that feeds into either implementation or design, depending on what the finding surfaces. As a consequence, no single topological ordering of the five activities is universally correct; the activities themselves and their *dependencies* are the stable structure, while their *sequencing* is a choice made by the SDLC model.

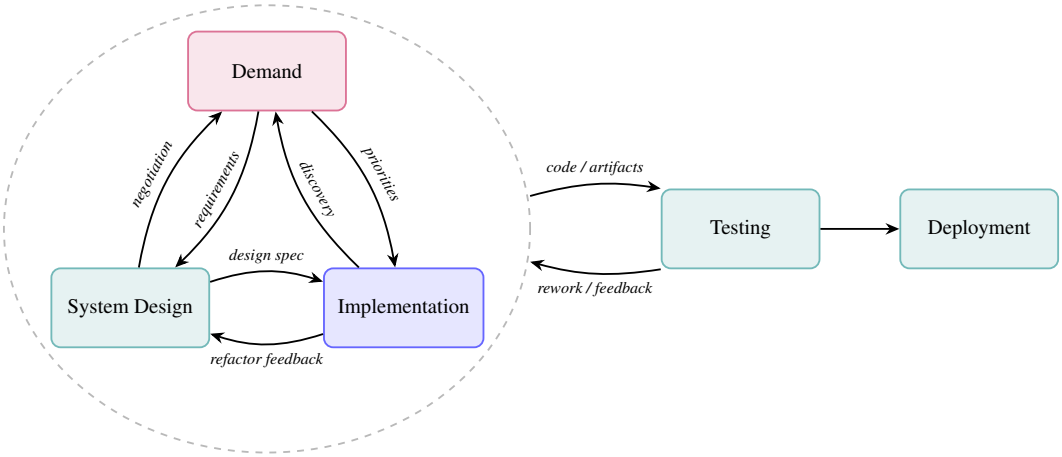

Figure 2: Generalized view of AI involvement in software engineering. The dashed ellipse encloses the iterative pre-deployment loop—demand, system design, and implementation—whose activities engage in continuous back-and-forth (requirements and priorities flow forward while negotiation, refactor feedback, and discovery flow back). The verified artifact then flows to testing and on to deployment.

A generalized SDLC model. Figure 2 makes this structure explicit by separating the pre-deployment loop from the downstream assurance and deployment stages. The dashed ellipse on the left encloses the three activities—demand, system design, and implementation—that in an agile setting are in continuous back-and-forth with one another. Every pair of activities inside the ellipse is connected by a pair of arrows capturing the typical direction of communication in each orientation: requirements flow from demand to system design, while negotiation flows back (scope and priorities are renegotiated as technical constraints become clear); design specifications flow from system design

to implementation, while refactor feedback flows back (implementation experience reshapes the architecture); and priorities flow from demand to implementation, while discovery flows back (what the developer learns while building can reshape what the stakeholder asks for). The cluster as a whole then exchanges code and artifacts with testing, which returns rework and feedback, and once testing is satisfied the verified artifact moves on to deployment.

We next provide an overview of the three levels in our taxonomy. Overall, as AI capabilities mature, we envision that autonomy will propagate outward from the most mechanistic activity toward those demanding broader context, judgment, and consideration of societal implications. With this principle in mind, we envision that the automation implementation will be the first to happen; then gradually all the phases except for demand proposal will be automated; and finally the end-to-end software engineering will become a full self-evolving autonomy. Next, we introduce the details for the levels in our taxonomy, starting from Level 0, the present SDLC model of the modern AI era.

A.2 Level 0: AI-Assisted Development (the present)

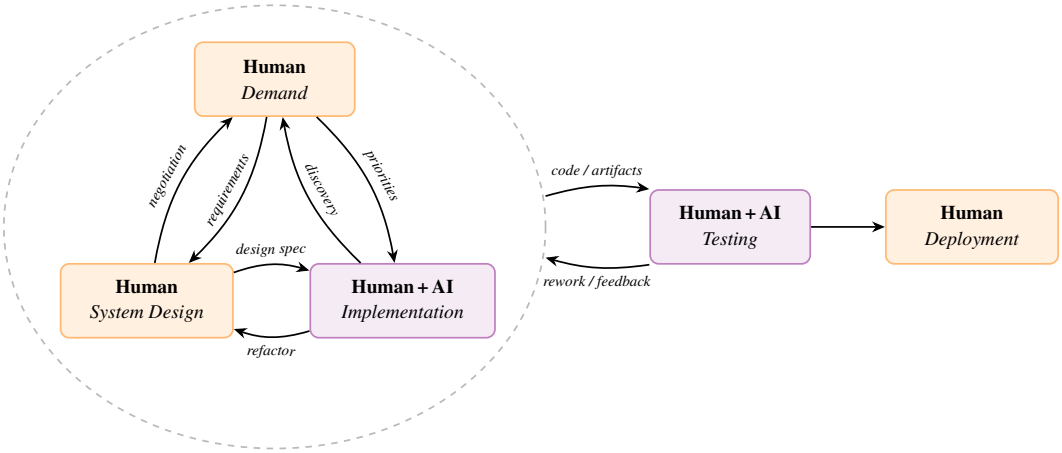

Figure 3: Level 0 pipeline. Orange boxes are purely human, violet boxes are human–AI collaboration, and blue boxes are purely AI (none at this level). All five SDLC phases remain distinct: humans own requirement curation, system design, and deployment, while AI assists humans during implementation and testing.

Level 0 describes the current state of practice, in which AI tools like coding agents and code completion assistants purely augment human developers without replacing them in any end-to-end sense. The developer retains ownership of system design and architecture, and AI assistance is confined largely to the implementation and testing phase.

This level is well-understood and broadly deployed [54, 8, 91, 9]. AI-generated code tends to exhibit a lower incidence of functional bugs and bug severity relative to human-written code [85].

A system has exceeded Level 0 when AI begins to autonomously produce design artifacts, or when quality-assurance decisions are delegated to AI without per-action human approval.

A.3 Level I: Developer-Supervised AI Development

At Level I, the AI agent assumes primary responsibility for both system design and implementation. Given a demand, the AI autonomously produces a system design, generates code and documentation, and submits the result as a pull request for human review. The developer's role is reduced to *engineering oversight*: reviewing AI-generated artifacts for correctness, conformance to project conventions, and adherence to established quality standards, as well as running and interpreting tests and conducting security audits. This paradigm is already emerging in practice through agentic frameworks that accept a task description and produce a pull request [140, 130].

Specification granularity at Level I. A critical but often underappreciated variable at this level is the *granularity of the specification* provided to the AI agent. The demands that enter a Level I system span an enormous range of abstraction: from a precisely scoped bug report with a failing test

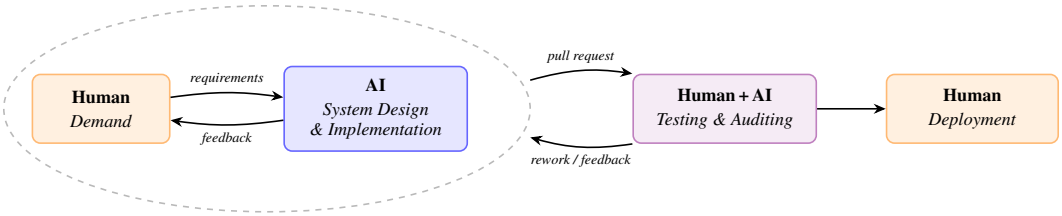

Figure 4: Level I pipeline. Orange boxes are purely human, violet boxes are human–AI collaboration, and blue boxes are purely AI. The AI agent owns system design and implementation jointly, producing a pull request. Humans still curate the demand, share testing and auditing oversight with the AI, and gate deployment via PR approval.

case and a pointer to the offending line, to a high-level request to "add multi-tenancy support" with no further elaboration. These extremes place very different burdens on the AI. A tightly specified demand requires little more than competent implementation; a vague one requires the agent to make consequential design decisions, resolve ambiguities, and infer unstated constraints—tasks that are qualitatively harder and more error-prone.

This spectrum is already visible in current practice. For example, Carlini et al. [22] report building a 100K-line Rust-based C compiler using orchestrated AI agents, a project that succeeded in large part because each agent task was carefully scoped and the overall architecture was human-directed. Even so, the resulting compiler initially failed on some of the simplest programs and relied heavily on LLVM as a reference implementation—illustrating that even well-specified agent-driven development can produce brittle results when the specification does not fully capture the intended behavior. As the level of specification becomes coarser, the gap between what is stated and what is intended widens, and the risk of such failures grows.

We therefore identify *specification engineering*—the practice of crafting specifications that are precise enough to guide AI agents while remaining tractable for human authors—as a key enabling discipline for Level I. Effective specifications may be multimodal, combining natural language descriptions with API sketches, example inputs and outputs, test cases, and visual mockups. The ability to provide and interpret such multi-format specifications is an important capability for both agents and the humans who direct them.

Existing effort. There has been extensive research and benchmarking efforts on attaining and evaluating capabilities related to this level. For example, SWE-Bench [50] is one of the first comprehensive benchmarks to evaluate the capability of resolving real-world GitHub issues. Although SWE-Bench, especially the verified subset of SWE-Bench, contains high-quality verifiable problems, it only covers a part of the capabilities needed in Level I, as it only covers mostly bug fixes, feature additions, and regression issues. There are other emerging benchmarks [141], but similarly, they only evaluate a subset of the demands mentioned in Section 2.1 and do not capture the full challenge of Level I. Regarding this level, there has already been some research effort, including the orchestration of multi-agent systems with access to multiple types of tools.

Realizing Level I reliably requires that AI agents possess sufficient context about the target codebase, the team's coding conventions, and the broader software architecture. It also demands that the cost and latency of agentic pipelines be competitive with human implementation. We discuss the corresponding challenges in Section B.1 and the research directions that address them in Section 4.

Acceptance Criteria

A system operates at Level I when *all* of the following hold:

(1) Given a structured demand (e.g., a GitHub issue or ticket), the AI autonomously produces a complete pull request—including design rationale, code, and documentation—without human involvement during generation.

(2) Human involvement is restricted to the review and oversight stage, occurring only *after* the AI has produced its output; the developer does not co-author or steer the design during generation.

(3) Human review remains a required gate: the proposed change requires explicit human approval (e.g., PR merge) before reaching production.

(4) The AI can handle all four demand categories (corrective, adaptive, perfective, preventive) across the majority of real-world issues without escalation to human-led implementation.

A system has exceeded Level I when human review is no longer required at the assurance stage, or when deployment proceeds without any human approval step.

A.4 Level II: Autonomous End-to-End Development

Figure 5: Level II pipeline. Orange = purely human; violet = human–AI collaboration (none at this level); blue = purely AI. The human role is limited to supplying the initial demand; the AI agent owns design, implementation, testing/auditing, and deployment end-to-end. The backward arrow captures the AI surfacing ambiguities back to the human for clarification before proceeding.

Level II marks a qualitative shift: the AI agent assumes full responsibility for the *entire* development pipeline, from system design and implementation through testing, auditing, and deployment. The defining characteristic of this level is that neither code authorship nor code review is performed by human stakeholders; the AI agent bears sole responsibility for all intermediate stages. The human role is strictly limited to high-level demand specification. Managers articulate their requirements; the AI determines the design, implementation, verification, and deployment of the resulting system.

This level requires not only strong implementation capability but also the ability to produce meaningful tests, perform security and correctness audits, and self-evaluate the quality of produced artifacts—all without human guidance. The open problems at this level are substantially harder than those at Level I, and we discuss them in Section B.2 and the corresponding research directions in Section 4.

Code as the authoritative specification. A maintainability challenge specific to Level II raises a deeper question about the relationship between code and specification. Natural language is hopelessly imprecise and ambiguous as a specification medium: the same sentence can be interpreted differently by different readers, and natural language requirements routinely admit multiple valid implementations with divergent behavior. In traditional development, the *code itself* serves as the authoritative, precise specification of what the system does—it is the one artifact that is, by definition, unambiguous with respect to the machine's behavior. Human-readable code thus functions as both an executable artifact and a communication medium: developers read it to understand what the system does, debug it, extend it, and reason about its correctness.

At Level II, where AI agents are the primary authors and no human reviews the code before deployment, this dual role is under threat. If the code is structured in ways that are natural for AI generation but opaque to human readers, the system loses its function as a comprehensible specification—even if it runs correctly. This concern is not merely about readability in the superficial sense of formatting and naming; it is about whether the code can serve as a medium of communication between the AI system and the humans who must ultimately understand, trust, audit, and regulate the software.

One response to this challenge is to require that AI-generated code conform to human-readable conventions, potentially at the cost of some efficiency—treating legibility as a hard constraint on generation rather than an optional property. A more ambitious direction, explored in the software design community, asks whether there exist higher levels of abstraction—intermediate notations between natural language and code—that could serve as the specification medium: precise enough to be machine-checkable, but more compact and conceptually transparent than raw source code. Such representations could serve as the primary human-facing artifact, with the underlying implementation treated as a derived product.

Another complementary approach is to leverage the AI agent itself as an explainer: if the agent can generate the code, it can also generate on-demand explanations, architectural summaries, and design rationale for any component, potentially making the codebase more comprehensible than a

comparable human-written system. Whether this constitutes genuine understanding or merely creates an additional layer of potentially unreliable abstraction is an open question, but the capability is already emerging in practice and deserves careful study.

Acceptance Criteria

A system operates at Level II when *all* of the following hold:

(1) Given a natural-language demand, the AI produces, tests, audits, and deploys a change with no human involvement between demand specification and the deployed output.

(2) The AI reliably detects ambiguous or underspecified demands and either resolves them autonomously or surfaces them to the human for clarification *before* proceeding—rather than silently misinterpreting.

(3) The deployed output satisfies functional, security, and maintainability requirements without post-deployment human correction under normal operating conditions.

(4) The agent maintains architectural coherence across multiple successive demands, without human intervention to enforce design consistency.

A system has exceeded Level II when it begins to identify and prioritize its own development demands, rather than waiting for human-provided demand specification.

A.5 Level III: Fully Autonomous Development

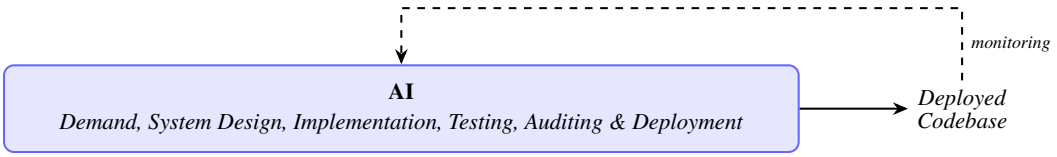

Figure 6: Level III pipeline. Orange = purely human (none at this level); violet = human–AI collaboration (none at this level); blue = purely AI. A single AI agent autonomously identifies demands from operational signals and carries out design, implementation, testing, auditing, and deployment; the dashed arc shows the monitoring feedback through which the deployed system informs subsequent autonomous cycles.

Level III represents a qualitative shift beyond execution autonomy. At Levels 0 through II, the AI operates in a paradigm of *execution autonomy*: given a demand specified by a human, it carries out some or all stages of the development pipeline. Level III introduces *goal autonomy*: the AI system independently determines *what* to build and *why*—proactively identifying demands from operational signals, user behavior, security feeds, and the system's own evolving state, rather than waiting for external direction.

Concretely, goal autonomy manifests in capabilities that have no analogue at lower levels: autonomously responding to newly disclosed security vulnerabilities by identifying affected code, generating patches, and deploying fixes without human initiation; proactively identifying performance bottlenecks from production telemetry and designing, implementing, and evaluating optimizations; designing and executing gray-scale experiments to evaluate feature hypotheses, iterating on the basis of user behavior and A/B test results; and performing preventive dependency migrations ahead of end-of-life deadlines, driven by continuous monitoring of the dependency ecosystem. These behaviors require the system not just to execute a given plan, but to form plans autonomously from raw operational signals.

This level presupposes that all challenges of Levels 0 through II have been sufficiently addressed and adds a further set of uniquely difficult sociotechnical problems, discussed in Section B.3, with corresponding research directions in Section 4.

Auditability and the liability gap. Even though the software is autonomously developed, specification engineering remains essential at Level III—arguably more so than at any lower level. If the software is intended for human use, human stakeholders must be able to understand what the system does, why it makes the decisions it makes, and what guarantees it provides. Self-generated specifications must therefore be not only machine-checkable but also human-auditable: they must

be expressed in a form that regulators, users, and oversight bodies can inspect. Comprehensive logging of the agent's decision-making process—what demands it identified, what design alternatives it considered, what tradeoffs it made—is a prerequisite for any meaningful accountability framework.

The challenge is compounded by the current state of software liability law. In most jurisdictions, software is distributed under licenses that disclaim virtually all liability for defects, and courts have generally not held software vendors to the strict liability standards applied to physical products. This regime evolved under the assumption that human developers and human oversight provide a meaningful (if imperfect) check on software quality. At Level III, where no human is in the loop, the absence of software liability creates a troubling gap: an autonomous agent may generate and deploy software that causes real harm, and no entity—neither the agent's developer, nor the deploying organization, nor the agent itself—bears clear legal responsibility for the consequences. Without liability frameworks that assign meaningful accountability for autonomously generated software, there is little institutional incentive to invest in the safety, verification, and auditability measures that Level III demands. Developing such frameworks is not only a legal challenge but a prerequisite for responsible deployment of fully autonomous development systems.

Acceptance Criteria

A system operates at Level III when *all* of the following hold:

(1) The system identifies development demands autonomously from operational signals (logs, telemetry, user behavior, security advisories) without any human prompting or demand specification.

(2) The system sustains coherent architectural evolution over timescales of weeks to months without human intervention to maintain consistency or set direction.

(3) The system generates and maintains its own specifications that accurately describe its deployed behavior, and updates them as the system evolves without human-authored requirements.

(4) The system responds to operational incidents (e.g., newly disclosed vulnerabilities, performance regressions) autonomously—including diagnosis, patch generation, and redeployment—without waiting for a human to file a demand.

B Challenges

Next, we highlight the prominent challenges to achieve autonomous software engineering shown in Section 2 and defer the full list to Section B.

B.1 Challenges Towards Implementation Autonomy (Level I)

Cost-Effectiveness. Current agentic workflows can be expensive in terms of computation, latency, and operational cost, especially when models are large, iterative, or prone to entering unproductive loops. A recent randomized controlled trial of 16 experienced open-source developers across 246 tasks found that early-2025 AI tools *increased* task completion time by approximately 19% even though developers self-reported a roughly 20% speedup [13]. In practice, the cost of running and supervising automated systems may outweigh their productivity gains, particularly for smaller projects or tasks that require frequent human intervention. The end-to-end cost of orchestrated agent teams can be substantial: a recent demonstration of 16 parallel coding agents collaboratively building a 100K-line Rust-based C compiler reported approximately 2,000 sessions and $20K in API costs to produce a working artifact [22], and cost-aware re-evaluations of agentic scaffolds reveal large variance in cost-per-resolved-task that aggregate leaderboards typically hide [37]. Achieving cost-effective autonomy thus remains a key practical barrier to widespread adoption.

Human–Agent Knowledge Transfer. Current agents are capable of design- and implementation-level tasks but typically lack project-specific knowledge: the preferred coding style, the architectural invariants the team has established, the rationale behind legacy decisions, and the conventions that make a pull request acceptable. Imparting this knowledge requires significant upfront effort from developers, and the interface for doing so remains poorly designed. Unlike a new team member who

can be onboarded through conversation, code review, and documentation, an AI agent must acquire preferences and constraints through whatever communication channel the system exposes. The lack of a settled interface is reflected in the proliferation of incompatible project-context conventions across major coding agents—`CLAUDE.md`, `AGENTS.md`, `.cursorrules`, and Skills directories [7]—each implementing a different progressive-disclosure scheme. Industry context-engineering reports document meaningful gains on SWE-bench Verified attributable primarily to repository-level context plumbing rather than to model improvements [10], suggesting that knowledge transfer, not raw capability, is often the binding constraint at Level I.

The deeper consequence is that personalization itself is being relocated within the software value chain. As the marginal cost of generating code falls and the rationale for many shared abstractions weakens (Section C.5), the residual durable artifact of software engineering is no longer the code, but the corpus of preferences, conventions, architectural invariants, and constraints that distinguish one organization's—or one developer's—software from another's. The same trend can be seen at the user-facing layer: when an organization can cheaply re-implement a generic SaaS-style application against a shared protocol, what differentiates its instance from anyone else's is precisely the personalization layer fed to the agent. Knowledge transfer interfaces therefore play a dual role: they are the channel by which agents acquire enough context to act correctly, and they are the persistent representation of an organization's accumulated software identity. This dual role surfaces challenges that current interfaces do not address. First, personalization operates at multiple, partially overlapping scales—individual developer style, team-level conventions, repository-wide invariants, and organization-wide architectural commitments—and an interface designed for any one scale tends to handle the others poorly. Second, personalization must be *progressive* rather than one-shot: the relevant preferences are revealed over months of accept/reject decisions, code reviews, and incident postmortems, not specified upfront, yet today's context files are largely static documents that developers must remember to update by hand. Third, the absence of cross-vendor portability standards means that personalization encoded for one agent ecosystem typically cannot be reused by another, creating a vendor-lock-in risk whose severity grows with the depth of personalization—and therefore directly opposes the trend toward agents as commodity infrastructure. Addressing these three gaps—multi-scale personalization, progressive elicitation, and portability—is increasingly central to making Level I autonomy practical at scale.

Opacity of Agent Reasoning. As agents make increasingly consequential design and implementation decisions, developers need tools to understand *why* an agent made a particular choice—not merely *what* it produced. Without such transparency, meaningful human oversight becomes difficult to achieve even when the developer is nominally in the review role. This is non-trivial: prior studies have shown that the natural-language reasoning traces produced by chain-of-thought prompting are not guaranteed to faithfully reflect a model's underlying computation [75, 136], and traditional interpretability methods scale poorly to frontier-sized models [148]. Initial efforts to adapt interpretability to the code domain—such as token-level rationales [56] and anchored explanations of just-generated code [137]—are encouraging but cover only a narrow slice of the design and architectural decisions developers actually need to audit.

Agent Behavior Debugging. When an autonomous development agent produces an incorrect or undesirable output, diagnosing the root cause is substantially harder than debugging conventional software. At Level I, where the developer still reviews output, such debugging capability is essential for closing the feedback loop and improving agent performance over time. Recent large-scale trace analysis suggests that agent failures are qualitatively different from traditional software bugs: across roughly 1,600 execution traces from seven popular multi-agent frameworks, Cemri et al. [23] identify 14 distinct failure modes spanning specification, inter-agent misalignment, and verification, with the substantial majority of failures stemming from coordination and specification issues rather than from raw model capability. Tactical fixes such as prompt engineering or topology changes yield only modest improvements and leave overall failure rates that remain too high for production deployment, indicating the need for fundamentally new debugging primitives.

Level-I Benchmarks. Today's coding benchmarks, notably SWE-bench [50] and its variants [141], capture only a narrow slice of Level I by measuring whether an agent resolves a well-scoped issue against a fixed test suite. The narrowness has now become acute enough that OpenAI's Frontier Evals team has publicly stopped reporting SWE-bench Verified, announcing in February 2026 that they

no longer evaluate frontier models on it, citing both flawed test cases that reject functionally correct solutions and substantial training-data contamination, and recommending SWE-Bench Pro as the replacement [92, 32]. Diagnostic probes confirm the contamination concern independently—state-of-the-art models reach up to 76% accuracy at identifying buggy file paths from issue descriptions *alone*, falling to roughly 53% on out-of-distribution repositories [66], and continuous-update pipelines that admit only post-cutoff issues see frontier models drop substantially relative to SWE-bench Verified [147]. These benchmarks also say little about the cost, review burden, or long-term maintainability of agent output—precisely the dimensions on which Level I adoption decisions turn. Benchmarks that jointly measure task success, inference and supervision cost, and the burden imposed on human reviewers are a prerequisite for evidence-based deployment at this level.

B.2 Challenges Towards Pipeline Autonomy (Level II)

Ambiguous and Underspecified Demand. Demands expressed in natural language are inherently ambiguous, incomplete, and context-dependent. Even experienced human developers rely on iterative clarification, implicit assumptions, and domain intuition to resolve such ambiguity. An automated agent, however, may misinterpret intent, fill gaps with incorrect assumptions, or overfit to literal phrasing—producing solutions that technically satisfy the stated requirements while violating unstated but essential expectations. This challenge is exacerbated in early project phases when stakeholders themselves are uncertain about the full requirements. The effect is large enough to be measured even on benchmarks that were curated for clarity: SpecFix [49] found that 43.6% of HumanEval+, MBPP+, and LiveCodeBench problem descriptions are repairably ambiguous, and automatically clarifying them yields up to 30.9% pass@1 improvement on the modified subset and 4.1% absolute improvement across the whole benchmark. Earlier work on interactive clarification likewise raised GPT-4 pass@1 from 70.96% to 80.80% on MBPP-sanitized by detecting ambiguity through code-consistency checks [87], and recent studies show that LLMs often fail to detect ambiguity in user requests at all unless explicitly forced to [125]. On industrial requirements with deeply nested temporal constructs, even the strongest auto-formalization pipelines still struggle with temporal interpretation, conditional structure, and implicit shared engineering knowledge that the original requirement leaves unstated [77].

Lack of Trustworthy Oracle. For many software quality dimensions—readability, maintainability, UI/UX suitability, documentation quality, and the correctness of refactoring—there is no reliable oracle against which to evaluate AI-generated outputs. Even when unit tests exist, verifying that automatically generated tests are correct, comprehensive, and free from systematic bias is itself a non-trivial problem [83, 71]. Empirical evidence quantifies the gap: when LLMs are prompted to generate test cases for tricky bugs in plausible programs, the precision of the generated tests can be as low as 6.3%, with the majority of false positives stemming from incorrect oracles rather than incorrect inputs [71]. The best-performing oracle-generation pipelines on Java benchmarks remain bounded well below human accuracy [83, 84], and the steep dependence on model scale suggests this is capacity-limited rather than knowledge-limited, precluding straightforward solutions through retrieval or prompt engineering [84]. In the absence of trustworthy feedback signals, automated systems risk optimizing against weak or proxy metrics, reinforcing superficial correctness while missing deeper quality issues.

Lack of Provable or Verifiable Guarantees. In safety-critical and security-critical domains, software must satisfy strong correctness requirements that ideally admit formal verification. Decades of programming-languages research have built the foundations for this—abstract interpretation [29], deductive verifiers such as Dafny [60], interactive theorem provers such as Lean [86], Coq [120], and Isabelle/HOL [89], dynamic invariant detection [35, 34], and SMT-based invariant inference [24, 53]—and connecting these foundations to LLM-based code generation is now an active frontier. However, automatically verifying complex software systems remains an open research problem, particularly when systems interact with external environments, legacy components, or sources of undefined behavior. Current automated development approaches cannot provide formal proofs or exhaustive verification of generated code, making it difficult to establish the level of assurance required for deployment in high-stakes settings. Concrete numbers illustrate the gap: on the VERINA benchmark for end-to-end verifiable code generation [144], frontier models achieve substantially higher pass rates on code generation than on specification or proof generation, with end-to-end one-shot proofs in the low single digits even on basic problems. Across verification systems, off-the-shelf LLMs

achieve roughly 27% success on Lean tasks, 44% on Verus tasks, and 82% on Dafny tasks [21], with pure Dafny verification rising from approximately 68% to 96% over a single year [74, 21]. Performance still collapses on classical algorithms requiring global invariants and ghost state [149], and even when local verification succeeds, composing specifications and proofs across modules remains brittle [41, 135]. On the Vero benchmark for repository-scale joint implementation and proof synthesis [143], the strongest evaluated agent closes 81% of individual specifications yet fully implements and verifies only 25 of 43 multi-module instances, revealing a sharp gap between local proof completion and end-to-end repository verification.

Lack of Security Guarantees. Guaranteeing that automatically generated code is secure by design is extremely difficult. Modern software vulnerabilities frequently arise from subtle interactions between components, misuse of APIs, incomplete threat modeling, or insecure defaults—areas in which automated agents may lack adversarial awareness. Recent large-scale audits of LLM-generated code make the magnitude concrete: industry studies report that a substantial fraction of generated code contains OWASP Top 10 vulnerabilities, with the rate concentrated in particular language ecosystems [124], and a study of 20,000+ GitHub issues found that LLM-generated patches introduce *new* vulnerabilities with patterns absent from human-authored fixes [106]. Automated dependency suggestions hallucinate non-existent packages on average about 5.2% of the time for commercial models and 21.7% for open-source models, and the hallucinated names recur across reruns frequently enough to be squattable—a phenomenon now widely termed *slopsquatting*, formalized at scale across more than half a million generated samples [112], that creates a plausible supply-chain attack path security reports have begun to observe. Code that is functionally correct may still expose exploitable attack surfaces or fail to implement appropriate defenses. Beyond generation-time issues, the behavior of LLMs is difficult to constrain formally: external anomalies such as prompt injection or adversarial inputs may cause agents to produce code with unexpected and potentially dangerous properties [73, 90]. Empirical evidence shows that indirect prompt injection attacks against agentic coding editors achieve very high success rates [73], and recent systematizations find that adaptive attack success against state-of-the-art defenses remains substantial [78]. The threat is no longer hypothetical: a widely reported in-the-wild prompt-injection incident against an AI coding assistant emerged in early 2026 [110], and a contemporaneous audit of the agent-skills ecosystem found a non-negligible fraction of catalogued skills containing critical security issues, with malicious skills frequently combining prompt injection with traditional malware delivery [111].

Agent Governance. As AI agents take on more autonomous roles, mechanisms for controlling, auditing, and constraining their behavior become essential. This includes policy frameworks for what actions agents are permitted to take autonomously versus which require human approval, logging and audit trail standards for agent-generated changes, and rollback and recovery protocols for erroneous agent behavior. An emerging body of standards and tooling now begins to formalize these requirements—OWASP's Top 10 for Agentic Applications [96], Microsoft's open-source Agent Governance Toolkit [109], and AWS's Agentic AI Security Scoping Matrix [20]—and concrete regulatory deadlines (the EU AI Act high-risk obligations from August 2026, the Colorado AI Act from June 2026) are already forcing enterprises to operationalize them. Without such governance infrastructure, deploying Level II systems in production environments carries unacceptable risk.

Level-II Benchmarks. Level II requires benchmarks that reach well beyond the pass-a-test-suite format of Level I: there is no human-authored specification to enforce, no curated test oracle to trust, and no assumption that the deployed artifact has been reviewed. A meaningful Level II benchmark must evaluate whether an agent can resolve ambiguous demands without silently misinterpreting them, produce tests whose correctness and coverage are themselves trustworthy, generate code that remains secure and maintainable under later modification, and surface its own limitations rather than failing silently. Initial steps in this direction include CodeSpecBench [26] for repository-level executable behavioral specification generation, and the DafnyComp- and Vericoding-style cross-module benchmarks [41, 21, 135] that quantify the composability gap when local verification succeeds but module integration does not. Designing such benchmarks is itself an open problem—one that deserves explicit research attention, because without them the field cannot measure progress toward pipeline autonomy or calibrate when a Level II system is ready to deploy.

B.3 Challenges Towards Demand Autonomy (Level III)

Specification Generation and Full Software Understanding. At lower levels, a human-authored specification serves as the authoritative statement of what the software is supposed to do. At Level III, no such artifact is supplied externally: the agent must produce it autonomously. Current specification-synthesis methods—ranging from dynamic invariant detection [35, 34] to LLM-driven approaches that infer specifications from existing code [131, 76]—require human verification of the inferred specifications, and recent analysis suggests this is fundamental rather than incidental: any auto-formalization of natural language must ultimately ask the user to confirm the specification matches intent [28]. This self-generated specification must go beyond an informal description of intended behavior; it must be precise enough to guide implementation, serve as a ground truth for testing, and provide an auditable record of the system's intended semantics. Crucially, the specification must evolve coherently alongside the software itself—as the system grows, the agent must update its stated intentions in a way that remains consistent with the deployed behavior, or deliberately record and justify any intentional divergence. A prerequisite for maintaining this consistency is that the agent must be capable of *fully understanding* the behavior of the software it produces. This is a non-trivial demand: Rice's theorem establishes that any non-trivial semantic property of a program is undecidable in general, which means that complete behavioral understanding is computationally intractable for arbitrary systems.

Managing Competing Priorities and Tradeoffs. Software engineering in practice is governed as much by business context as by technical requirements. Decisions about when to ship, how much technical debt to accept, when to prioritize security over usability, and how to allocate engineering effort are fundamentally shaped by organizational priorities, time-to-market pressures, and risk tolerance. A fully autonomous AI system must internalize not only the technical context of the software it develops but the full constellation of business constraints bearing on each decision. These constraints are not static: they shift with market conditions, team capacity, regulatory environment, and stakeholder priorities. Initial empirical signals already suggest that current AI-assisted development can worsen the technical-vs-business balance rather than improve it: in Google's 2024 cross-sectional DORA survey, a 25% increase in reported AI usage is associated with a 7.2% *decrease* in delivery stability and a 1.5% decrease in delivery throughput [44], and longitudinal analysis of agent-assisted development shows that initial velocity gains can decay over months as accumulated technical debt offsets short-term productivity [48, 42].

Responsibility and Accountability. Software is a social system before it is a technical one. When AI-generated software causes harm—through a security breach, a financial error, an incorrect medical recommendation, or a discriminatory outcome—the question of who bears responsibility is not merely philosophical but legal and institutional. Current frameworks for software liability assume human authorship and human oversight, and do not straightforwardly extend to settings where no human reviewed the relevant code. Open litigation already illustrates the friction: the 2022 GitHub Copilot class action established that even when DMCA copyright claims fail, breach-of-license claims around AI-trained code can survive dismissal, leaving the legal exposure of AI-generated code partially unresolved [52]. Concurrent industry disclosures of state-sponsored actors weaponizing coding agents to target multiple organizations [5] make the question of liability allocation increasingly urgent rather than academic.

Long-Term Maintenance and Versioned Reasoning. Software systems exist and evolve over timescales of years to decades. A fully autonomous development agent must manage not only the current state of the codebase but its trajectory: anticipating dependency obsolescence, managing accumulating technical debt, adapting to shifting requirements, and preserving the coherence of the system's architecture across many successive changes. Recent benchmarks designed to evaluate agents under continuous software evolution rather than isolated tasks expose this gap directly. EvoClaw [31] evaluates agents across 98 dependency-constrained milestones from real OSS repositories and finds that frontier agents retain feature-implementation capability but fail to prevent regressions, with overall performance scores collapsing from above 80% on isolated tasks to at most 38% in the continuous-evolution setting; SWE-CI [25] formalizes the same phenomenon as continuous-integration-style cycles. Empirical longitudinal studies of agent-assisted development likewise document multiple signatures of declining code quality across the AI-adoption period [42, 48]. Human engineers accomplish this through a combination of institutional memory, shared conventions,

and deliberate design stewardship. Replicating this capacity in an autonomous agent—particularly one that may be reinitialized, retrained, or replaced over the system's lifetime—raises deep questions about continuity of intent and long-horizon planning.

Multi-Agent Coordination. Multi-agent architectures—in which multiple specialized agents handle different pipeline stages or work in parallel—are an implementation strategy that can be employed at any level of the taxonomy, from Level I teams of reviewer and implementer agents onward. Already at Levels I and II, multi-agent failures account for the majority of observed coordination errors: across roughly 1,600 traces from seven popular frameworks, the substantial majority of failures stem from specification or coordination issues rather than from capability limits [23]. Even in tightly orchestrated settings, conflict patterns are striking—the 16-agent C-compiler experiment [22] required an external GCC oracle for ground-truth disambiguation precisely because the agents repeatedly encountered identical bugs when treating the kernel as a single task, motivating per-file parallel debugging mediated by the oracle. At Level III, however, the coordination problem becomes qualitatively harder: when agents not only execute the pipeline but autonomously set goals and priorities, ensuring that they share a consistent product vision, resolve conflicting objectives, and do not produce mutually incompatible changes requires mechanisms that go well beyond standard software engineering conventions. Questions of which agent's objectives take precedence, how disagreements are resolved autonomously, and how a coherent long-term trajectory is maintained across many interacting autonomous subsystems remain substantially open.

Ecosystem-Level Evolution and Cross-Project Coordination. The challenges above are framed around a single software system, but most software is embedded in an ecosystem of interdependent projects: libraries, services, platforms, and downstream consumers that evolve asynchronously and constrain one another. At Level III, demand generation lifts this interdependence into the control loop—a change autonomously introduced in project Y may trigger a cascade of new demands in project X, which may in turn propagate back to Y, producing cyclic, potentially unstable update chains that no single project's agent can see in full. Ecosystem-level goals—backward compatibility across a dependency graph, shared security posture, coordinated deprecation—also need not align with any individual project's local objectives, and may conflict with them. Early empirical signals suggest such ecosystem-level coupling is already forming: the agent-skills ecosystem has begun to mirror traditional package registries as an attack surface [111], and slopsquatted dependencies hallucinated by AI assistants have created cross-project propagation paths that no single project's agent or maintainer can monitor in isolation [112]. Managing this is closely tied to the multi-agent coordination problem above, but it operates across organizational boundaries rather than within a single team, so the protocols, trust assumptions, and conflict-resolution mechanisms that suffice for a single project's agents are insufficient at ecosystem scale.

Trigger-Surface Security. When demand identification is automated, the channels through which demands enter the development loop become a new attack surface. An adversary who can manipulate the telemetry, logs, user-behavior signals, dependency advisories, or issue streams that a Level III agent monitors can induce the agent to author and deploy changes of the attacker's choosing—a supply-chain attack that requires no human developer to be compromised, because the demand itself is fabricated. Such attacks are already being observed in the wild against Level-I and Level-II agents: surveys of production AI assistants document indirect prompt-injection payloads embedded in webpages and APIs designed to exfiltrate API keys, initiate unauthorized payments, or execute shell commands when ingested by agents during routine tasks [39, 6]. Even the strongest published defenses—including reinforcement-learning-based hardening of browser agents—reduce, but do not eliminate, adaptive attack success [6]. Level II security properties do not subsume this threat: they assume demands are human-authored and focus on securing the pipeline downstream of the demand. Level III demands authenticated, integrity-checked demand sources, provenance tracking from operational signal to code change, and policies that gate which categories of signal are allowed to autonomously initiate modifications to production systems.

Level-III Benchmarks. Benchmarking a system that chooses its own objectives is fundamentally different from benchmarking one that executes given ones. There is no fixed task list to score against, and success must be measured over long horizons in terms of whether the agent identified the right demands, maintained architectural coherence, avoided destabilizing ecosystem partners, and recognized the limits of its own judgment. Candidate designs include long-running simulated

environments with evolving user behavior, adversarial ecosystems that probe an agent's response to malicious triggers, and longitudinal evaluations that track specification–implementation drift over many development cycles—but none of these is well-established, and building them is itself an open research problem. Initial efforts include EvoClaw [31] and SWE-CI [25], which evaluate maintenance and regression resistance over multi-milestone evolution streams, and SWE-EVO [119], which targets multi-file changes from release notes; however, all of these still presuppose externally curated milestones and therefore do not yet evaluate the goal-setting capability that defines Level III. Without such benchmarks, claims about Level III readiness cannot be empirically adjudicated, and the organizational and regulatory decisions that depend on those claims will be made on inadequate evidence.

C Organizational Implications and Adoption Dynamics

The taxonomy described in the preceding sections is not merely a technical road map; it is also a sociotechnical one. Each successive level of automation does not simply change the tools used in software engineering—it changes who does what, how teams are structured, and what capabilities firms must cultivate. In this section we discuss four cross-cutting implications of the taxonomy that extend beyond the technical challenges identified at each level: the transformation of team structures and professional roles, the changing shape of software engineering organizations, the dynamics of adoption that will govern how quickly—and how unevenly—these transitions unfold, and the broader market and societal consequences that follow from a dramatic reduction in the cost of software production.

C.1 Transformation of Roles and Team Structures

The progression from Level 0 to Level III represents a fundamental redistribution of responsibility across the software engineering pipeline, and each step redefines what human contributors are expected to do.

At Level 0, the team structure familiar today largely persists. Implementors focus on writing and modifying code, architects on system decomposition and design, QA engineers on testing, and security practitioners on auditing. AI tools reduce effort at the implementation layer but do not substantially alter the professional role structure.

At Level I, the shift becomes more pronounced. As AI agents assume primary responsibility for design and implementation, the human role contracts toward oversight and review. The ability to write code fluently becomes less important than the ability to evaluate code critically—to detect subtle correctness issues, assess architectural coherence, and recognize when an agent's design decisions diverge from project conventions. This shift favors architectural judgment and evaluative expertise over implementation throughput. The work that remains for humans—reviewing complex pull requests, assessing whether an agent's design choices are consistent with the system's long-term trajectory, identifying subtle specification gaps—requires the kind of deep codebase knowledge and cross-cutting awareness that comes from sustained engagement with a system's evolution. It is not that these tasks are intrinsically "senior"; rather, they demand a particular cognitive profile: the ability to reason about systems holistically, to spot what is *missing* rather than what is *wrong*, and to evaluate tradeoffs that are not captured in any test suite. These capabilities are distinct from (and complementary to) implementation skill, and organizations will need contributors who specialize in them regardless of how those roles are titled. New hybrid roles may emerge: *specification engineers* who translate high-level demands into the structured formats agents require; *AI integration engineers* responsible for configuring, onboarding, and maintaining agentic pipelines; and *AI code reviewers* with particular expertise in diagnosing agent failure modes and evaluating the trustworthiness of agent-generated artifacts. The distinction that matters is not seniority per se, but the difference between *implementors* (who produce code) and *architects* (who evaluate, constrain, and direct the system's evolution)—and as AI absorbs the former role, the latter becomes the primary human contribution.

At Level II, these transformations deepen. When AI agents handle the full pipeline from design through deployment, the human contribution narrows to demand specification and audit. The product manager and the security auditor become more central; the implementor becomes peripheral. Organizations that currently invest heavily in implementation capacity—large pools of developers

executing well-specified tasks—will face structural pressure to reallocate those resources toward upstream activities (requirements elicitation, architectural governance) and downstream activities (quality assurance, deployment monitoring, and incident response).

At Level III, the human role in software engineering shrinks further still, toward policy setting and accountability governance. Engineering headcount for equivalent system output may contract dramatically, while the cognitive profile of the remaining contributors shifts toward strategic, evaluative, and regulatory competencies rather than technical implementation. The key scarce human skill becomes not the ability to build software, but the ability to specify what should be built and to hold autonomous systems accountable for the result.

C.2 The Future Shape of Software Engineering Organizations

Beyond individual team roles, the taxonomy carries implications for how software engineering organizations as a whole will be structured.

Smaller, higher-leverage teams. As AI agents become capable of executing increasing fractions of the development pipeline, the size of the engineering team required to maintain a given software system will shrink. This is not a novel dynamic: the introduction of high-level programming languages, version control systems, and cloud infrastructure each enabled smaller teams to operate at greater scale. The transition through Levels I and II, however, represents a more fundamental compression—not just that individual tasks become faster, but that entire categories of human labor are absorbed by automated systems. The resulting teams will be smaller on average, with a higher proportion of roles concentrated in architecture, product strategy, security, and governance rather than implementation.

Capital structure rebalancing and the AI supply chain. Traditional software engineering organizations are labor-intensive. As AI automation advances, the cost of developing and operating software systems will shift from human labor toward compute infrastructure. This rebalancing changes the economic profile of software firms: capital expenditure on AI inference, model evaluation, and monitoring replaces payroll at the margin.

The future economics of AI-driven development are shaped by a multi-layered supply chain: energy and land costs determine the price of compute hardware, which determines the cost of training and serving AI models, which in turn determines the price of development tools and applications built on top of them. Each layer of this chain introduces its own dynamics: energy costs are subject to geopolitical and environmental constraints; hardware supply is concentrated among a small number of manufacturers; model training is dominated by a handful of well-capitalized labs; and the application layer is where the benefits ultimately reach software organizations. Shifts at any layer—a breakthrough in chip efficiency, a regulatory change affecting energy markets, a new open-source model competitive with proprietary offerings—can dramatically alter the cost structure of AI-driven development. The uncertainty in this supply chain makes it difficult to predict whether AI development tools will become cheap enough to be universally accessible or will remain expensive enough to create structural advantages for well-capitalized firms, potentially accelerating consolidation in parts of the software industry. How this supply chain evolves will have outsized influence on the future shape of software engineering organizations and the pace at which different levels of the taxonomy become economically viable.

Emergence of new organizational functions. Several organizational functions that do not currently exist at scale will become necessary as firms advance along the taxonomy. AI governance teams will be responsible for defining the scope of agent autonomy, reviewing agent behavior logs, and responding to incidents caused by autonomous agents. Model evaluation and selection will become a strategic function analogous to vendor management today. Firms operating at Level II or above will need dedicated capacity for assessing the trustworthiness of AI agents, specifying behavioral constraints, and maintaining audit trails—activities that do not map cleanly onto any existing organizational function.

New market entrants and service providers. The transition also creates opportunities for new categories of firms. Third-party AI code auditing services—analogous to today's security consultancies or financial auditors—may emerge to provide independent verification of AI-generated software.

AI-native development shops may offer end-to-end autonomous delivery as a service, competing with traditional staff-augmentation models. Regulatory and certification bodies will develop standards for AI-generated software, creating demand for compliance and assurance services. These new entrants will reshape the competitive landscape of the broader software industry.

C.3 Adoption Dynamics: Diffusion Will Be Gradual and Uneven

Even if the technical challenges at each level of the taxonomy are eventually resolved, it is difficult to predict the pace at which the organizational transformation described above will occur. Historically, major technological transitions in software engineering—from waterfall to agile methodologies, from on-premises to cloud infrastructure, from procedural to object-oriented programming—have unfolded over timescales of one to two decades, driven by gradual shifts in tooling maturity, workforce skills, institutional norms, and economic incentives. However, this time it could be faster, as we have seen coding AI being much more universally adopted and given more responsibility within only a few months.

Domain and sector variation. Regulatory requirements in safety-critical domains—aviation, healthcare, nuclear systems, financial infrastructure—impose certification and liability standards that will significantly delay adoption of higher-autonomy levels. Regulators will require demonstrated reliability, interpretable audit trails, and well-defined accountability chains before permitting AI-generated software to operate in contexts where failures could cause serious harm. Greenfield software projects, where legacy constraints are absent and risks are lower, will adopt higher levels of automation earlier; large, safety-critical, or compliance-heavy systems will lag by years or decades.

Workforce transition and implications for education. The professional workforce involved in software engineering has been trained, credentialed, and socialized around a set of practices, tools, and norms that center on human authorship. Universities, bootcamps, and professional certifications will need to adapt their curricula to prepare developers for roles that are fundamentally evaluative and supervisory rather than generative—a shift that affects not only what students learn but how software engineering is conceptualized and taught as a discipline. This transition will take time, and the interim period will be characterized by a workforce that is unevenly prepared for the demands of human–AI hybrid or AI-supervised development.

The implications for software engineering education deserve particular attention. Today, students learn software engineering primarily by building: writing code, debugging it, and gradually developing intuition about large-scale system behavior through hands-on implementation. A recurring concern is that if AI handles most implementation, newcomers to the field will lack the formative experience of reading, navigating, and reasoning about large codebases—the very experience that builds the evaluative judgment that the remaining human roles demand.

However, this concern has a counterpoint. If the cost of development falls dramatically, students and early-career practitioners may actually gain *more* experience with large-scale systems, not less: they can direct AI agents to build ambitious projects that would previously have been infeasible for a single person or a small team, gaining exposure to the architectural, integration, and maintenance challenges that come with scale. The bottleneck shifts from "can I build this?" to "can I evaluate, test, and maintain this?"—which may be a more authentic preparation for the roles that will persist.

This suggests a fundamental reorientation of curricula. In the traditional model, students learn to implement and are assessed on their ability to produce correct code; testing, auditing, and architectural evaluation are treated as secondary skills acquired later through experience. In an AI-driven world, this order inverts: the primary skills become specification writing, code review, testing strategy, security auditing, and system-level reasoning—precisely the competencies needed to supervise and evaluate AI-generated artifacts at Level I and beyond. Implementation skill remains valuable as a foundation for understanding, but it is no longer the central output of professional training. Universities that recognize this shift early and restructure their programs accordingly will produce graduates better prepared for the emerging landscape.

Institutional and cultural inertia. Organizations develop practices, processes, and management structures adapted to their current mode of production. Transitioning to AI-supervised or autonomous development is not merely a technical upgrade—it requires revising hiring practices, performance

metrics, team topologies, and governance processes. Established firms with deep institutional commitments to existing processes will typically adopt these changes more slowly than new entrants building from scratch. Cultural resistance to AI substitution—from engineering leadership, labor organizations, and individual contributors who have invested in traditional skills—will further mediate the pace of adoption.

Implications for the research community. The pace of adoption will be shaped in part by the maturity of the research agenda outlined in this paper. Progress on trustworthy oracles, interpretable agent reasoning, and agent governance frameworks is not merely technically important—it is also a prerequisite for the organizational trust required to advance to higher levels of autonomy in practice. Research that produces deployable artifacts—auditing tools, governance frameworks, specification interfaces—alongside theoretical contributions will have outsized influence on the rate at which organizational adoption occurs. Conversely, unresolved safety and accountability gaps will, appropriately, serve as a brake on adoption, reinforcing the urgency of the research directions identified at each level of the taxonomy.

C.4 Domain Risk Profile and Deployment Constraints

The appropriate level of AI autonomy in software engineering is not a universal answer—it depends critically on the risk profile of the domain in which the software operates. Two broad categories reveal sharply different adoption trajectories.

High-assurance domains. Safety-critical and compliance-heavy sectors—including medical devices, avionics, financial infrastructure, and industrial control systems—impose regulatory certification requirements, mandatory audit trails, and liability frameworks that assume human authorship and oversight at every stage. In these domains, AI-generated code must typically pass the same certification processes as human-written code, a requirement that current AI systems are rarely equipped to satisfy without extensive human involvement. Even Level I—where a human reviews all AI-generated artifacts before deployment—may require regulatory approval before AI-authored changes can be deployed in contexts governed by standards such as IEC 62304 (medical software), DO-178C (avionics), or PCI-DSS (payment systems). Progress toward higher levels in these domains will be conditional on formal verification capabilities (Section A.4), interpretable agent reasoning (Section A.3), and the development of certification frameworks that explicitly accommodate AI-authored code. In practice, high-assurance domains are likely to plateau at Level I or below for the foreseeable future, and organizations operating in these domains should treat the taxonomy as a long-horizon roadmap rather than a near-term deployment guide.

Low-assurance domains. Internal tooling, developer utilities, data processing scripts, and personal or small-scale applications present a fundamentally different risk profile. The cost of failure is lower, rollback is easier, and regulatory constraints are absent or minimal. These domains are already beginning to see deployments approaching Level II: end-to-end agentic pipelines that accept a natural-language description of a task and produce a tested, working application with minimal human oversight (examples including ClaudeCode [8] and OpenClaw [93]). For this class of software, the transition through the levels is likely to be faster and less contested, constrained primarily by technical capability rather than regulatory or institutional barriers. The gradual expansion of Level II and Level III practices in low-assurance domains will generate the operational experience, tooling, and trust that may eventually inform safe adoption in more demanding settings.

Intermediate domains. Many domains lie between these extremes: e-commerce, content management, enterprise productivity tools, and social platforms operate at substantial scale and involve significant financial or reputational risk, but are not subject to the same formal certification requirements as safety-critical systems. For these domains, the key determinant of appropriate autonomy level is often the reversibility of errors and the organization's capacity for rapid incident response rather than formal regulatory compliance. As monitoring, rollback, and automated testing infrastructure matures, these domains are well-positioned to advance toward Level II, provided that the governance frameworks discussed in Section A.4 are in place. The domain risk profile thus functions as a practical calibration tool: organizations should locate themselves on the taxonomy not only according to their technical capability but according to the failure tolerance of their specific deployment context.

C.5 Abstraction Collapse and the Rewrite Economy

A subtler consequence of the taxonomy, distinct from the role and market shifts discussed above, concerns the fate of the abstractions on which contemporary software engineering depends. Modern software is built atop a deep stack of abstractions—programming languages, frameworks, libraries, container runtimes, orchestration layers, third-party services—each of which exists, in part, to manage the cognitive limits of human developers. Abstractions encapsulate complexity so that an engineer can build a payment system without reasoning about TCP retransmission, or write a web application without managing memory layout. When AI agents acquire sufficient context and capability to operate across the whole stack, the rationale for many of these abstractions weakens, and several distinct dynamics may follow.

Stack-wide optimization. A useful analogy is the practice of writing a custom CUDA kernel: when performance demands justify the cost, an expert reaches below the standard abstractions and writes specialized code tailored to a specific workload and hardware configuration. The cost of this approach is the human expertise required to reason simultaneously about algorithms, memory hierarchies, and hardware-level details—expertise that few developers possess. If AI agents can reliably reason across the entire stack, this kind of cross-cutting optimization need not remain exceptional. Agents could, in principle, perform bottom-to-top customization throughout an application, bypassing abstraction layers when doing so yields better performance, security, or fit to a specific use case. The consequence is not the disappearance of abstractions but a change in their role: rather than serving as load-bearing structures that humans must respect, they become defaults that agents can selectively pierce when warranted.

From dependencies to protocols. This dynamic has implications for the software supply chain. Today, software systems are assembled from a vast graph of dependencies—libraries, frameworks, and services produced by other organizations and embedded as opaque black boxes. This graph reflects the economics of human development: it is far cheaper to depend on someone else's authentication library than to write one from scratch. If the marginal cost of writing customized code falls dramatically, the calculus changes. Dependencies may shift from being shared *implementations* toward being shared *protocols*: standardized interfaces and behavioral contracts that agents can re-implement on demand, customized to local requirements, rather than artifacts that must be ingested wholesale. A plausible intermediate form is a model in which high-quality, open-source reference implementations of common software categories—a messaging platform, a project tracker, a CRM—serve as starting points that organizations modify extensively to fit their specific workflows, rather than as products consumed as-is. The economic logic of this transition is uncertain—if code itself loses much of its value as a product, the structure of the software supply chain becomes difficult to predict—but the underlying theme is clear: a shift from shared implementations toward personalization, mediated either by reimplementation or by new abstractions designed for plug-and-play customization.

The rewrite as an alternative to maintenance. The same dynamic suggests a partial answer to a recurring concern about agents in large codebases: the worry that long-term technical debt, accumulated complexity, and codebase idiosyncrasies will erode the productivity gains agents otherwise provide. The conventional response is to invest in better tooling for navigating, refactoring, and incrementally improving existing systems. An alternative response, made more credible by stack-wide agent capabilities, is to rewrite rather than refactor. If an agent can reproduce the functionality of a non-trivial application in a matter of hours—a small internal dashboard, a moderately complex SaaS-style tool—then the accumulated debt of the existing implementation becomes less binding: the system can simply be rebuilt cleanly, paying a one-time cost in exchange for substantial future benefits. This argument does not apply uniformly. Systems whose value lies in tacit knowledge, hard-won corner-case handling, or integrations with a long tail of external services are not easily reproduced from a specification, and the rewrite carries real risk. SaaS-style applications, where the behavior is comparatively well-defined and externally checkable, are far more amenable to this approach than safety-critical systems or systems whose behavior has been shaped by years of incremental adjustment to operational reality. But within the class of software where rewriting is feasible, the rewrite economy reframes technical debt as a problem with a discrete, bounded cost rather than an open-ended drag on productivity—and it reinforces the broader picture in which abstractions, dependencies, and even codebases themselves become more fluid as the cost of producing software falls.

C.6 Broader Market and Societal Consequences

The organizational shifts described above are first-order effects—changes within software engineering teams and firms. The taxonomy also predicts second-order consequences that extend well beyond the software industry itself: structural disruption to the market for software products, a fundamental shift in who can produce software, and the prospect of an unprecedented proliferation of software systems. We discuss these consequences in turn, acknowledging that they are speculative predictions rather than established findings, but predictions that deserve serious analytical attention.

Disruption of the SaaS business model. The software-as-a-service model rests on a specific economic foundation: because building and maintaining software is expensive, it is efficient to amortize that cost across a large customer base, charging each customer a subscription fee for access to a shared product. As AI agents drive the marginal cost of building and maintaining a custom application lower and lower, this economic logic weakens significantly. Organizations that today pay per-seat subscriptions for productivity tools, project management software, or domain-specific platforms may find it cheaper—and more effective—to instruct an AI agent to build a tailored equivalent, calibrated precisely to their workflows and data. This pressure will fall most acutely on the long tail of SaaS products serving specialized or niche markets, where the per-seat price is high relative to the complexity of the underlying functionality. Horizontal platforms that derive their value from network effects, shared data, or deep integrations across an ecosystem may be more resilient; a customer relationship management system is only useful if counterparties also use it, and that value cannot be replicated by a bespoke build. Nevertheless, the overall trajectory suggests a contraction of the addressable market for many categories of commercial software, as the barriers to custom development collapse.

Democratization of software engineering. The history of computing is in part a history of progressive democratization: mainframes gave way to personal computers, command-line interfaces gave way to graphical ones, and the web gave way to mobile platforms, each transition expanding the population of people who could productively use computing technology. Autonomous AI development represents a potential inflection point in this trajectory that extends democratization to the production side: not merely who can *use* software, but who can *build* it. At Level II and beyond, a domain expert—a physician, a logistics coordinator, a musician, a small-business owner—could in principle describe a software system in natural language and receive a deployed, functional application without any programming knowledge. This parallels the effect that spreadsheets had on financial modeling: a capability previously restricted to specialists became accessible to a much broader population, fundamentally changing who could engage in quantitative analysis. The analogy is instructive, however, in one important respect: spreadsheets did not eliminate the need for expertise, they redistributed it. The democratization of software engineering through AI is unlikely to mean that anyone can build *correct, secure, and maintainable* software without expertise; it is more likely to mean that the relevant expertise shifts from implementation to specification, evaluation, and judgment.

Proliferation of software. If the cost of producing a functional software system falls dramatically, supply-side economics predicts a corresponding explosion in the number of software systems produced. Software today is characterized by a winner-take-most dynamic: because development is expensive, markets tend to converge on a small number of dominant products that spread their fixed costs across the largest possible user base. When those fixed costs decline toward zero, the incentive to converge disappears. The result may be a long tail of highly customized, narrowly scoped, or even single-user software systems—tools built for one team's specific workflow, scripts generated for a single data processing task, or personal utilities tailored to an individual's preferences. Such proliferation has potential benefits: software that today does not exist because no commercial market is large enough to justify its development could become ubiquitous. But it also introduces challenges. A world with vastly more software is also a world with vastly more attack surface, more unmaintained codebases, more opportunities for subtle defects to propagate into consequential systems, and more difficulty for any governance regime to provide meaningful oversight. The security and reliability challenges identified at Level II become correspondingly more urgent as the population of deployed AI-generated systems grows.

Software as an everyday commodity. Taken together, the preceding predictions point to a structural shift: software engineering is becoming an everyday commodity, with costs falling to negligible levels

even for medium and large-scale systems. What is today a significant line item in organizational budgets—and a major barrier to entry for individuals and smaller organizations—will instead resemble a ubiquitous utility, accessible on demand at minimal cost. This transformation extends beyond the software industry. As software functions as a general-purpose input across nearly all sectors, making it cheap and abundant effectively lowers the cost of operating and innovating in the broader economy. Organizations that historically could not afford custom or sophisticated systems—ranging from small businesses to public institutions—will be able to deploy capabilities previously reserved for well-resourced enterprises, including increasingly complex applications. At the same time, competitive advantages rooted in the ability to build software will erode, as even substantial systems can be produced cheaply and routinely. Differentiation will shift toward data, distribution, trust, and brand, while implementation itself becomes commoditized. Although the pace and impact of this transition will vary across domains, the overall trajectory—toward abundant, low-cost software as a baseline economic good—appears durable.