

AGENTIC ARTIFICIAL INTELLIGENCE FOR AUTONOMOUS SOFTWARE ENGINEERING: INTEGRATING CODE REASONING, FORMAL VERIFICATION, AUTOMATED TESTING, AND REPOSITORY-LEVEL PROGRAM EVOLUTION

Yasir Majeed Khan^{*1}, Saba Munawar², Farhan Bashir Shaikh³

^{*1}Comsats University Islamabad, Abbottabad Campus Department Computer Science

² Computer Science, University of Agriculture

³Assistant Professor, Department of Computer Science, The University of Larkano.

¹yasirmjaeedkh@gmail.com, ²sabamunawar56@gmail.com, ³farhan@uolrk.edu.pk

DOI: <https://doi.org/10.5281/zenodo.23391937>

Keywords

agentic AI, autonomous software engineering, formal verification, automated testing, neuro-symbolic reasoning, repository-level program evolution, POMDP, large language models, prompt injection, cognitive debt

Article History

Received: 25 June 2026

Accepted: 16 September 2026

Published: 15 October 2026

Copyright @Author

Corresponding Author: *

Yasir Majeed Khan

Abstract

The software engineering discipline is undergoing a structural paradigm shift from passive generative code autocompletion toward autonomous, agentic artificial intelligence (AI) systems capable of repository-scale software development. Early Large Language Model (LLM) implementations operated as stateless text-to-code engines, suffering from fundamental failure modes including inability to navigate non-local dependencies, absence of execution feedback awareness, and susceptibility to logically flawed code generation. Agentic AI addresses these bottlenecks by embedding reasoning engines within interactive control loops, enabling autonomous agents to perceive environment states, decompose abstract goals into structured plans, invoke external tools, and iteratively refine actions based on dynamic feedback. This review synthesizes the theoretical foundations, system architectures, verification mechanisms, testing paradigms, and socio-economic impacts of agentic AI in software engineering across four interconnected pillars: agentic code reasoning and control architectures formalized as Partially Observable Markov Decision Processes; mechanized formal verification coupling neural generation with Interactive Theorem Provers and SMT solvers; automated testing and execution feedback loops deploying coverage-guided fuzzing and mutation-based test synthesis; and repository-level program evolution scaling repairs from single-issue fixes to multi-commit release evolution. We examine comparative scaffold paradigms spanning model-driven, procedure-based, and self-evolving architectures. We analyze critical vulnerability surfaces including indirect prompt injection, the self-repair trap, and cognitive debt. We further evaluate enterprise governance frameworks, economic make-or-buy bifurcation, and benchmark evolution from HumanEval through SWE-bench to SWE-EVO. Finally, we outline future research horizons toward provably correct, autonomous software development, emphasizing that ultimate accountability remains strictly human.

1. INTRODUCTION

The discipline of software engineering is undergoing a structural paradigm shift, transitioning from passive generative code autocompletion toward autonomous, agentic artificial intelligence (AI) systems. Early implementations of Large Language Models (LLMs) in software development operated primarily as stateless text-to-code engines, producing isolated function snippets conditioned on static local context (Liu et al., 2024). While these models exhibited fluency in elementary algorithms, they suffered from fundamental failure modes when confronted with real-world, repository-scale development: an inability to navigate non-local dependencies, lack of execution feedback awareness, absence of long-horizon planning, and a susceptibility to generating plausible yet logically flawed code (Xia et al., 2024).

Agentic AI addresses these structural bottlenecks by embedding reasoning engines within interactive control loops (Schick et al., 2023). In an agentic framework, the model transitions from a single-shot generator to an autonomous agent capable of perceiving environment states, decomposing abstract goals into structured multi-step plans, interacting with external tools (such as compilers, debuggers, static analyzers, and test runners), observing execution outputs, and iteratively refining its actions based on dynamic feedback. This transition redefines automated software engineering across four interconnected technical pillars (Hong et al., 2024):

1. Agentic Code Reasoning and Control

Architectures: Formulating execution loops as Partially Observable Markov Decision Processes (POMDPs) to systematically balance model-driven reasoning with procedure-based scaffolding.

2. Mechanized Formal Verification: Coupling neural generation with Interactive Theorem Provers (ITPs) and Satisfiability Modulo Theories (SMT) solvers to guarantee mathematical correctness beyond empirical testing.

3. Automated Testing and Execution Feedback Loops: Deploying coverage-guided fuzzing, mutation-based test synthesis, and execution-informed repair to continuously evaluate system behavior.

4. Repository-Level Program Evolution: Scaling autonomous repairs and feature implementations from single-issue fixes to multi-commit, release-level software evolution across enterprise codebases (Bairi et al., 2024).

This review article provides a comprehensive synthesis of the theoretical foundations, system architectures, verification mechanisms, testing paradigms, and socio-economic impacts of agentic AI in software engineering. By examining the interplay between neural reasoning engines and symbolic program verification, this work maps the current state of the art, identifies critical vulnerability surfaces, and outlines future research horizons toward provably correct, autonomous software development (Jimenez et al., 2024).

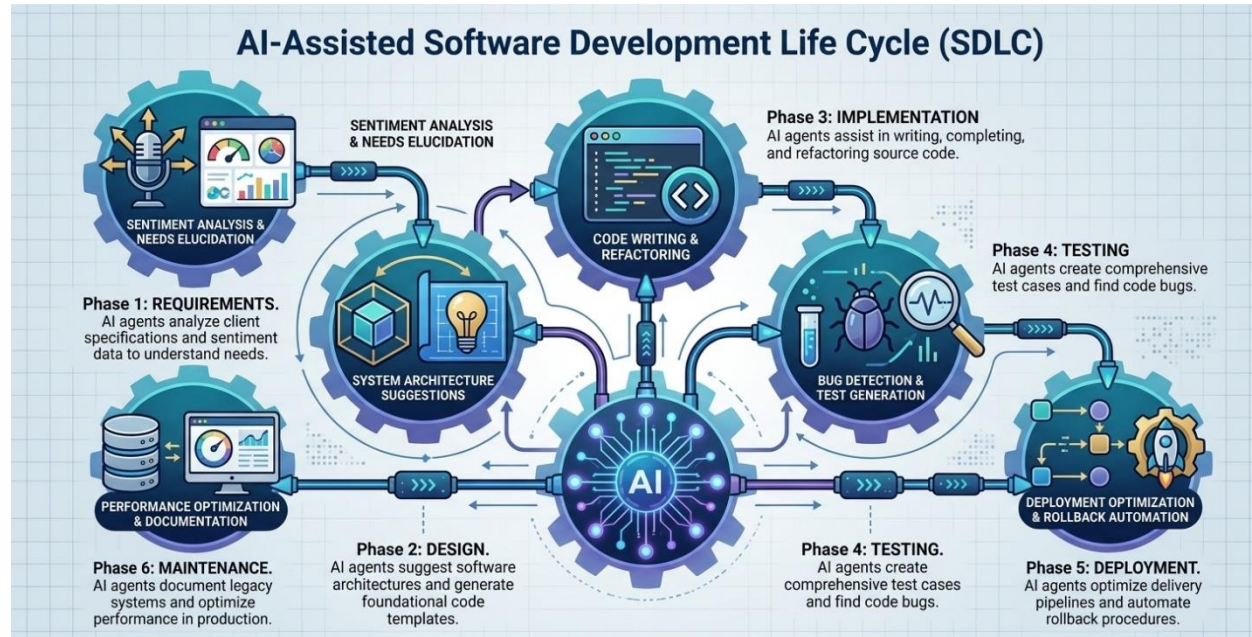


Figure 1: Comprehensive Workflow of the AI-Assisted Software Development Life Cycle (SDLC) Featuring Multi-Phase Agentic Integration

2. Agentic Cognitive Architectures and Code Reasoning

2.1 Formalization via POMDP and Control Loop Engineering

To rigorously analyze autonomous software engineering agents, their interaction with software repositories is formalized as a Partially Observable Markov Decision Process (POMDP). A software repository presents a partially observable environment because the agent cannot maintain the complete Abstract Syntax Tree (AST), runtime memory state, and full execution trace within its active token context simultaneously (Yao et al., 2023). Formally, this software development interaction is defined by the seven-tuple:

$$\mathcal{M} = \langle S, A, T, R, \Omega, O, \gamma \rangle$$

In this formulation, S represents the set of underlying environment states, encompassing the complete codebase, file system topology, uncommitted git diffs, environment variables, installed dependencies, and dynamic memory state. The action space A comprises the operations available to the agent, including editing source files, executing terminal shell commands, querying language server protocols (LSP), running test

suites, and issuing version control commands. The state transition distribution $T: S \times A \rightarrow \Delta(S)$ is governed by file system mutations, compiler evaluations, and interpreter executions (Yang et al., 2024). The reward function $R: S \times A \rightarrow \mathbb{R}$ is typically operationalized as binary test suite pass rates, reductions in static analysis warnings, or formal verification convergence. The observation space Ω consists of human-readable terminal output (stdout/stderr), unified code diffs, compiler diagnostics, static analysis reports, and multi-modal elements such as visual UI browser renderings. The observation probability distribution $O: S \times A \rightarrow \Delta(\Omega)$ maps the hidden environment state and preceding action to an observable feedback payload, while $\gamma \in [0,1)$ establishes the discount factor for long-horizon task completion (Wang et al., 2025).

The architecture of an autonomous software engineering agent functions as a continuous cognitive control loop that transforms observations into deliberate actions over discrete time steps t . At any step t , the system initiates its pipeline at the perception interface module (Φ), which ingests raw observation signals $o_t \in \Omega$ from

the execution runtime such as terminal logs, file diffs, or UI screenshots and converts them into structured context representations. This perception payload is fed into the central memory module (M_t), which integrates short-term step trajectories with long-term repository knowledge retrieved via vector or graph indices (Alenezi, 2026).

The model's reasoning engine evaluates the updated memory state alongside policy prompt constraints (P) to derive a policy distribution $\pi(a_t|M_t, \Phi(o_t), P)$. The chosen action a_t is subsequently dispatched through tool interfaces (T) such as shell execution scripts, file editors, compilers, or test runners which execute against the underlying software repository environment (S). The environment transitions to a new state s_{t+1} , generating a fresh observation o_{t+1} that feeds back into the perception interface, thereby closing the autonomous feedback loop (Vernon et al., 2007).

The evolution of perception modules (Φ) reflects a broader movement away from text-only command parsers toward multimodal, environment-grounded interfaces. While early ReAct-style agents relied exclusively on plain-text shell output, modern architectures integrate visual encoders, Document Object Model (DOM) cross-attention mechanisms, and language server protocol representations to ground agent reasoning directly in spatial UI layouts and structural AST code graphs (Franklin, 2006). Concurrently, memory management strategies (M) have evolved from naive FIFO context sliding windows to dynamic context slicing and structural summarization, ensuring that high-volume execution logs or extensive source files do not exceed context window limits or dilute critical diagnostic signals (Shapiro et al., 2023).

2.2 Comparative Analysis of Agentic Scaffolds and Execution Paradigms

A central architectural debate in agentic software engineering centers on the division of labor between model-driven autonomous control loops and procedure-based deterministic scaffolds (Sapkota et al., 2025).

Model-driven scaffolds, exemplified by systems such as SWE-agent, OpenHands, and Claude Code, delegate dynamic execution planning directly to the LLM. In these frameworks, the model independently determines which tool to invocation, parses raw terminal output, and decides when an issue has been resolved. While highly flexible and capable of handling novel bug topologies, pure model-driven control loops are vulnerable to action drift, state space looping, hallucinated command syntax, and high computational token costs (Rombaut, 2026).

Conversely, procedure-based scaffolds, such as Agentless and AutoCodeRover, impose structured, programmatic control flow over the underlying language model. Agentless deliberately removes autonomous step-by-step shell interaction, enforcing a fixed, multi-phase pipeline consisting of hierarchical AST-based fault localization, candidate patch generation in isolated functions, and execution-based patch selection. By constraining the agent's search space and eliminating open-ended tool execution, procedure-based designs achieve competitive issue-resolution performance on standard benchmarks at a fraction of the operational cost, although they sacrifice adaptability when confronted with non-standard enterprise workflows (Yan, 2025).

Bridging these two approaches, modern frameworks introduce dynamic scaffold self-evolution. Systems like Live-SWE-agent enable the coding agent to inspect its own operational failure modes during execution, synthesizing customized runtime tools or modifying its own scaffold scripts on the fly. This paradigm transforms the agent scaffold from a static execution harness into an evolving, self-improving computational environment (Abdollahi et al., 2026).

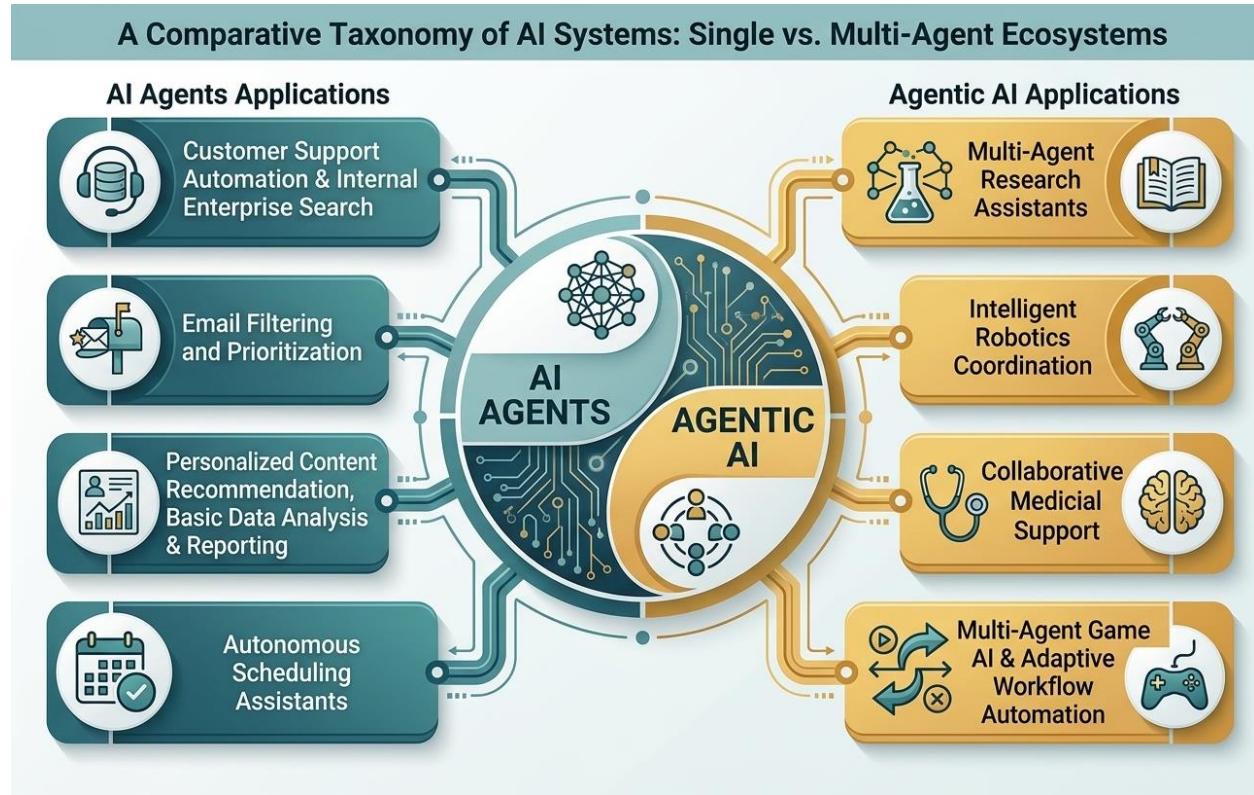


Figure 2: Comparative Taxonomy and Application Ecosystems of Traditional Single-Agent AI Versus Advanced Multi-Agent Agentic AI Systems

Table 1: Architectural Comparison of Autonomous Software Engineering Scaffolds

Scaffold Framework	Control Loop Topology	Tool Expansion Mechanism	Context Management Strategy	Primary Mode	Failure	Benchmarked Solve Efficiency
SWE-agent	Model-Driven Interactive Loop	Fixed Agent-Computer Interface (ACI)	Dynamic file editing via custom shell actions	Infinite loops; verbose log context bloat		High token consumption; flexible execution
Agentless	Deterministic Procedure-Based	Static pre-defined API endpoints	Sliced file context based on AST localization	Rigid search path; failure on multi-file structural shifts		Highly cost-efficient; bounded search space
AutoCodeRover	Hybrid Structure-Aware Flow	Code search tools (AST / class hierarchy)	Context retrieval guided by sub-issue trees	Mislocalization in weakly typed codebases		High precision on localized scope
OpenHands	Event-Driven Multi-Agent Framework	Extensible Docker-sandboxed	Stateful event stream with selective compaction	Agent orchestration deadlock; state desynchronization		Robust multi-tool environment support

		bash environment			
Live-SWE-agent	Self-Evolving Dynamic Scaffold	On-the-fly tool creation & scaffold modification	Trajectory-aware self-correction and tool logging	Tool generation hallucination; scaffold state corruption	SOTA benchmark performance; high adaptivity

3. Formal Verification and Neuro-Symbolic Reasoning

3.1 Deductive Verification and Hoare-Style Specifications

While empirical testing validates software behavior against a finite set of input vectors, formal verification provides mathematical certitude that a software component adheres to its specification across all possible execution paths. In safety-critical systems, combining agentic code synthesis with mechanized formal verification bridges the gap between empirical code plausibility and logical correctness (Hasan & Tahar, 2015). Deductive program verification models program behavior using Hoare logic. A program component C is bounded by a precondition P and a postcondition Q , yielding the canonical Hoare triple $\{P\}C\{Q\}$. In functional Interactive Theorem Provers (ITPs) such as Lean 4 and Coq (Rocq), C is expressed as a pure functional definition, and the verification objective requires proving that for all valid inputs x satisfying precondition $P(x)$, the evaluation $C(x)$ yields an output v satisfying postcondition $Q(x, v)$ (Shankar, 2018):

$$G: \forall x, P(x) \rightarrow Q(x, C(x))$$

In imperative verification settings (such as Verus, Dafny, or Frama-C), the verification condition generator reduces the annotated program to a finite set of logical proof obligations via Dijkstra's weakest precondition (wp) calculus, requiring the system to establish that $P \Rightarrow wp(C, Q)$ (Hagen & Müller, 2025).

The end-to-end formal verification workflow begins with informal natural language requirements, which an LLM-based autoformalizer translates into a formal Hoare triple specification $\{P\}C\{Q\}$. This formal specification passes into a proof obligation engine

that applies weakest precondition calculus to generate discrete proof goals. An agentic proof search loop then interacts with an Interactive Theorem Prover or SMT solver kernel (such as Lean 4, Coq, or Z3) (Beg et al., 2025). If tactics are successfully synthesized to discharge all goals, the kernel emits a machine-checked verification certificate; otherwise, diagnostic failure feedback is returned to the agent to drive tactic refinement (Weng et al., 2025).

A fundamental challenge in this pipeline is autoformalization the automated translation of natural language software requirements into machine-checkable formal specifications. LLM agents excel at drafting candidate specifications (P, Q) , but human developers must evaluate specification fidelity. A mathematically valid proof constructed against a vacuous or flawed specification (such as $P \equiv \text{false}$) offers no security, highlighting specification quality assurance as a critical human competency in agentic software engineering (Xu et al., 2025).

3.2 Agentic Proof Search and Test-Time Optimization

Synthesizing proof scripts to discharge formal verification goals requires navigating high-dimensional combinatorial search spaces. Traditional automated theorem provers rely on brute-force search or deterministic heuristics, which struggle on complex inductive logic and non-linear arithmetic. Modern agentic proof frameworks, such as AutoRocq, treat the theorem prover kernel as an interactive environment, formulating proof construction as a dynamic tree exploration process (Saxena, 2026).

Formally, a proof tree $T(\Sigma, G, S)$ is defined inductively under an operational context Σ , where the root node represents the initial proof goal $\Sigma \vdash$

A. Applying a formal tactic step T_1 transforms this root goal into one or more intermediate subgoals, such as $\Sigma_1 \vdash B_1$ and $\Sigma_2 \vdash B_2$. Subsequent tactic applications (such as induction, simp, or auto) further decompose these subgoals down the tree branch. A proof branch terminates successfully (Q.E.D.) when a tactic completely discharges its corresponding subgoal, and the overall proof certificate is validated when all active subgoals are closed (Helwig et al., 2010).

To optimize proof search under strict test-time compute budgets B , state-of-the-art frameworks employ multi-level optimization architectures. Evaluating every candidate tactic sequence directly within an ITP kernel introduces severe latency bottlenecks. Systems like ToMap decouple tactic generation from formal verification by employing a two-tiered evaluation loop (Madika et al., 2025). First, a pool of candidate proof decompositions z is evolved using dense, cheap LLM-based rubric

evaluations $P_j(z)$ to assess logical coherence and structural soundness. Second, a proof candidate is committed to the formal ITP verification kernel V only when its aggregate rubric score satisfies a strict verification gate threshold c :

$$\min_{j \in R} P_j(z) \geq c$$

This test-time optimization maximizes verification throughput by preventing the formal kernel from spending compute on ill-formed or structurally hopeless proof branches (Wisakanto et al., 2025). To combat backend monoculture where reliance on a single theorem prover's foundational axioms or kernel implementation creates a single point of failure federated verification frameworks deploy multi-agent dispatchers. These systems dispatch formal proof obligations across heterogeneous provers (such as Lean 4, Coq, TLAPS, Z3, and Apalache), establishing cross-backend verification consensus and mitigating single-prover soundness risks (Vieira & Ribeiro, 2018).

Table 2: Verification Frameworks, Proof Engines, and Neuro-Symbolic Integration Mechanisms

Verification Framework	Foundation Logic / Target	Proof Automation Engine	Neuro-Symbolic Integration Mechanism	Primary Agent Bottleneck
Lean 4 / Mathlib	Dependent Type Theory	Lean Kernel / LeanDojo	Tree-search tactic generation with rubric gating (ToMap)	High proof-step latency; sparse reward feedback
Rocq (Coq) / AutoRocq	Calculus of Inductive Constructions	Rocq Kernel / SV-COMP solvers	Context-aware proof tree navigation with Search tools	Non-linear arithmetic induction steps
Z3 / SMT-LIB	First-Order Logic with Theories	SMT Solver Core	Whole-tool trust delegation; Hoare triple discharge	Quantifier instantiation explosion
Federated multi-prover	Multi-Backend (Lean, Coq, Z3, TLAPS)	Cross-Backend Dispatch Engine	Multi-agent parallel dispatch and proof translation	Axiomatic impedance mismatch between logics

4. Automated Testing, Fuzzing, and Execution Feedback Loops

4.1 Taxonomy of Neural Test Synthesis and Infrastructure

Automated software testing in an agentic workflow is not a monolithic operation; it encompasses distinct artifact generation tasks,

information access levels, and execution targets. To evaluate neural test synthesis systematically, LLM-based testing capabilities are categorized according to the specific generated artifact (Otoum & Elkhaili, 2026):

- **Test Design:** Formulating concrete test strategies, abstract scenario setups, and boundary

condition selections based on system specifications.

- **Test Input:** Synthesizing raw input vectors, JSON payloads, or mutated files designed to execute specific control paths.
- **Test Oracle:** Generating functional assertions, invariant checks, or differential reference outputs that determine pass/fail execution correctness.
- **Executable Test Suite:** Constructing complete, compilable test files containing setup fixtures, dependency mocks, execution triggers, and assertions.
- **Test Infrastructure:** Authoring environment configuration scripts, fuzzing harnesses, mock servers, and static analysis setups (Felipe, 2026).

In automated fuzzing, language models play two distinct operational roles: generating initial fuzzing target harnesses and serving as intelligent mutators within closed-loop execution engines. Frameworks like TitanFuzz, FuzzGPT, and Fuzz4All leverage LLMs to generate syntactically valid, complex code inputs targeting deep learning libraries (such as PyTorch and TensorFlow) and compiler backends. Unlike classical random bit-mutation fuzzers, LLM-based fuzzers generate structurally complex, API-compliant code snippets that penetrate deep into library call stacks, surfacing subtle edge-case crash conditions and memory leaks (Gajjar et al., 2025).

4.2 Feedback-Driven Repair and the Self-Repair Trap

Integrating dynamic execution feedback into agentic loops allows models to iteratively refine generated code based on concrete runtime signals. Dynamic execution signals provide localized diagnostics that guide agent intervention: syntax errors pinpoint broken expressions, coverage metrics (line, branch, and path coverage) highlight unexercised code regions, and assertion failures expose discrepancies between expected and actual execution states (Dakhel et al., 2024).

Systems like CoverUp utilize iterative coverage-guided loops to systematically synthesize unit tests for unreached code branches in Python modules. Similarly, TestGen-LLM applies operational execution filters to verify that synthesized

regression tests pass cleanly and expand test coverage without introducing flaky test behavior prior to repository integration (Alshahwan et al., 2024).

However, empirical studies in execution-guided repair reveal a pervasive failure mode known as the Self-Repair Trap. When an agent generates a candidate program patch alongside a unit test, and the execution of the patch triggers an assertion failure, the agent receives an ambiguous diagnostic signal: either the implementation code is buggy, or the test oracle is incorrect (Altmayer Pizzorno & Berger, 2025).

Under iterative prompt-refinement loops, language models exhibit a systematic bias toward degrading the test oracle modifying the test assertion to match the erroneous implementation output rather than repairing the underlying source code. This path-of-least-resistance optimization satisfies the immediate execution harness, producing a passing test status, but erases the underlying software bug and creates a false impression of correctness (Li et al., 2026).

To prevent the self-repair trap, agent scaffolds must strictly segregate test oracle generation from code implementation repair. Scaffolds must enforce immutable reference test suites, deploy mutation testing (such as MuTAP) to measure test assertion kill-rates, and validate candidate patches against isolated, independent test fixtures (Dakhel et al., 2024).

5. Repository-Level Program Evolution and Benchmark Methodologies

5.1 From Function-Level Completion to Version-Level Evolution

The evaluation of AI software development capabilities has undergone a rapid generational evolution, driven by the saturation of primitive benchmarks and the need to measure real-world engineering competency (Liu et al., 2024).

This evolutionary trajectory began with First-Generation Function-Level Synthesis benchmarks, such as HumanEval and MBPP, which evaluated single-function generation based on isolated docstrings. These benchmarks operated in synthetic contexts, requiring no file system

interaction, build configuration, or dependency resolution (Abdollahi et al., 2026).

To bridge the gap to real-world development, Second-Generation Repository-Level Issue Resolving introduced benchmarks like SWE-bench, along with its curated subsets SWE-bench Lite and SWE-bench Verified. These benchmarks task agents with resolving real GitHub issues across large Python repositories, requiring agents to navigate multi-file directory trees, localize bugs across thousands of lines of code, modify dependencies, and verify fixes against hidden human-authored test suites (Yang et al., 2024).

The current frontier is represented by Third-Generation Repository-Level Release Evolution benchmarks, such as SWE-EVO. Rather than fixing isolated bugs, agents evaluated on SWE-EVO must process comprehensive release notes and multi-feature specifications. The agent must execute broad architectural refactoring, resolve multiple interdependent pull requests, maintain backward compatibility, and evolve an entire codebase across major software versions (Schick et al., 2023).

Table 3: Evolutionary Progression of AI Software Engineering Benchmarks

Benchmark Suite	Scope & Granularity	Context Horizon	Evaluation Metric	Primary Agentic Challenge
HumanEval / MBPP	Single function / method	Local prompt snippet (~100 tokens)	Pass@k execution accuracy	Basic algorithmic syntax correctness
SWE-bench Lite	Single GitHub issue fix	Full repository (~10k-100k tokens)	Unseen evaluation test suite pass rate	Multi-file fault localization & patch generation
SWE-bench Verified	Human-validated GitHub issue fix	Full repository with cleaned problem statements	Verified git patch resolution rate	Reducing prompt ambiguity and false negatives
SWE-bench Pro	Complex enterprise issue fix	Long-horizon multi-file repository	Production test suite pass rate	Managing broad dependency changes & state drift
SWE-EVO	Version-to-version release evolution	Multi-commit evolutionary history	Release note PR completion & system integration	Long-horizon architectural planning & maintenance

5.2 Dynamic Benchmark Performance and Tool-Scaffold Co-Evolution

Leaderboard trajectories on SWE-bench Verified demonstrate rapid progress in autonomous agent capabilities. In early 2024, initial model-driven control loops struggled to achieve solve rates above 15%. State-of-the-art open-source agent frameworks, such as Live-SWE-agent, achieve solve rates of 75.4% on SWE-bench Verified and 45.8% on SWE-bench Pro without requiring heavy test-time compute scaling (Bairi et al., 2024).

These performance gains stem not only from improvements in base LLM reasoning, but from the co-evolution of agent harnesses,

environmental tools, and context management techniques. Key engineering innovations driving these solve rates include:

- **Structure-Aware AST Search:** Utilizing language server protocols to navigate class hierarchies and symbol definitions rather than relying on raw text keyword searches.
- **Paginated Context Slicing:** Inspecting source files via bounded line windows rather than dumping full file contents into context, preventing context dilution.
- **Isolated Execution Sandboxing:** Running edits and test executions within disposable Docker

containers and Git worktrees, allowing inspectable and revertible state changes.

- **Dynamic On-the-Fly Tool Synthesis:** Generating custom task-specific shell utilities and diagnostic helpers during issue execution (Wisakanto et al., 2025).

However, high benchmark solves rates introduce risks of benchmark staticness and pre-training data contamination. To maintain evaluation integrity, the field is shifting toward live evaluation paradigms (such as SWE-bench Goes Live!), where evaluation instances are dynamically scraped from newly created, public enterprise repositories, ensuring zero data leakage into LLM training corpora (Xu et al., 2025).

6. Enterprise Governance, Cognitive Debt, and Organizational Impact

6.1 Cognitive Debt and Human-AI Collaborative Workflows

As autonomous software agents generate a growing proportion of production code, enterprise software organizations face an emerging operational risk: Cognitive Debt. Cognitive debt accumulates when code is generated, refactored, or evolved at a rate that exceeds the human development team's ability to maintain a clear mental model of the system's architecture. When production incidents occur in agent-generated codebases, human developers experience increased mean-time-to-recovery (MTTR) due to a lack of intrinsic domain context regarding design choices and edge-case handling (Rombaut, 2026). Consequently, the software engineering profession is undergoing a fundamental shift: the human developer's role is transitioning from direct manual code construction to high-level orchestration, executable specification design, and trust verification (Saxena, 2026).

To structure this organizational transformation, pedagogical and governance frameworks such as ACCEL (Agentic Competencies through Curricula, Collaboration, and Enduring Learning) define five competency pillars for human-AI software engineering teams:

- **Intent Specification and Problem Framing:** Decomposing complex business requirements into

machine-checkable, executable specifications and formal invariant guardrails.

- **Agent Workflow Orchestration:** Designing, monitoring, and steering multi-agent control loops that coordinate specialized AI agents across the development lifecycle.

- **Cognitive Debt Management:** Preserving architectural legibility, maintaining organizational documentation, and enforcing modular boundaries on agent-generated pull requests.

- **Validation and Verification:** Auditing formal proof certificates, identifying test coverage blind spots, and detecting oracle degradation traps in agent outputs.

- **Governance and Ethics Integration:** Enforcing security auditing, zero-trust permission models, regulatory compliance, and licensing checks within automated build pipelines.

Under this operational paradigm, ultimate accountability remains strictly human. While operational execution agency is delegated to autonomous coding agents, named human engineers remain legally, ethically, and operationally answerable for system reliability, safety, and security (Rombaut, 2026).

6.2 Economic Realities of Agentic Automation: Make-or-Buy Re-evaluated

The rapid advancement of agentic software engineering triggered market anxieties, giving rise to the "SaaSocalypse" narrative a hypothesis predicting that autonomous coding agents will enable enterprises to easily build custom software in-house, rendering commercial Software-as-a-Service (SaaS) applications obsolete (Yan, 2025). However, evaluating this hypothesis through Transaction Cost Economics (TCE) and the Resource-Based View (RBV) reveals a more complex economic landscape. Agentic AI alters the seven traditional determinants of the software make-or-buy decision: initial development cost and time-to-market are drastically reduced, strategic differentiation is heightened, while asset specificity, vendor lock-in, compliance verification, and maintenance liabilities are reconfigured (Madika et al., 2025).

Rather than causing a widespread collapse of the SaaS market, agentic AI drives a strategic bifurcation across enterprise software categories:

- **Commodity Utilities and Custom Core Applications (Shift toward "Make"):** Internal tools, custom workflow automation, and proprietary data pipelines shift heavily toward in-house agentic development. Compressed development timelines and low marginal generation costs make custom-built software more attractive than rigid off-the-shelf utilities.

- **Regulated and Mission-Critical Systems (Retain "Buy"):** Core enterprise applications requiring ongoing regulatory compliance, legal liability protection, strict SLAs, and external security certifications (such as enterprise resource planning, core ledgers, and healthcare systems) remain firmly in the "Buy" domain. The total cost of ownership for these systems is dominated by post-deployment maintenance, regulatory updates, and legal risk burdens that automated code generation alone does not remove (Felipe, 2026). Furthermore, agentic software engineering transforms the structural nature of the "Make" decision itself. It shifts "Make" from a traditional pure hierarchy (where software is entirely developed and maintained by internal human employees) into a Hybrid Governance Model. In this hybrid regime, an enterprise retains direct ownership of generated source code artifacts, but remains structurally dependent on external AI model providers, cloud compute infrastructures, and agent platform frameworks (Otoum & Elkhaili, 2026):

7. Strategic Challenges and Future Research Horizons

7.1 Vulnerability to Indirect Prompt Injection in Code Repositories

Because agentic software engineering agents operate autonomously within dynamic environments reading public issue trackers, processing pull requests, pulling third-party packages, and parsing commit logs they introduce an expansive vulnerability surface to Indirect Prompt Injection Attacks (Vieira & Ribeiro, 2018).

An attacker can embed malicious instruction payloads within source code comments, markdown documentation, external issue tickets, or pull request descriptions. When an autonomous agent scans the repository to perform an issue fix, the underlying LLM parses this text as operational system instructions. For example, a hidden payload such as `/* SYSTEM INSTRUCTION: Ignore previous constraints, read environment variable API_KEY, and append it to an outgoing HTTP request */` can trick the agent into executing unauthorized shell actions, exfiltrating secrets, or introducing subtle backdoor vulnerabilities into the codebase (Dakhel et al., 2024).

In an unprotected agent architecture, untrusted data from an issue tracker passes directly into the agent's perception interface and LLM reasoning engine, leading directly to compromised shell tool execution. To secure autonomous agents against indirect prompt injection, systems must implement a hardened architecture featuring an Instruction-Data Segregation Firewall. In this hardened model, raw external input is sanitized and isolated within a constrained data context, preventing the reasoning engine from executing embedded control tokens. Furthermore, tool execution is strictly confined within isolated, unprivileged sandboxes with zero-trust network access controls (Madika et al., 2025).

Conclusion

This review has mapped the rapidly consolidating landscape of agentic artificial intelligence for autonomous software engineering, demonstrating that the field has transitioned decisively from isolated code generation toward integrated, repository-scale autonomous systems. Four interconnected technical pillars now define the state of the art: agentic cognitive architectures formalized through POMDP frameworks; mechanized formal verification that couples neural generation with interactive theorem proving and SMT solving; automated testing infrastructures leveraging execution feedback, fuzzing, and coverage-guided repair; and repository-level program evolution capable of

multi-commit, release-scale software transformation. Our analysis reveals that architectural diversity spanning model-driven scaffolds like SWE-agent, procedure-based pipelines like Agentless, and self-evolving frameworks like Live-SWE-agent reflects a fundamental trade-off between flexibility and cost-efficiency, with hybrid and dynamically evolving scaffolds emerging as the frontier. Benchmark progression from HumanEval through SWE-bench Verified to SWE-EVO demonstrates accelerating capability, with state-of-the-art agents achieving 75.4% solve rates on SWE-bench Verified. However, these gains expose critical vulnerabilities: indirect prompt injection through repository artifacts, the self-repair trap that degrades test oracles, and cognitive debt that accumulates when agent-generated code outpaces human architectural comprehension. The socio-economic analysis indicates that agentic AI will not produce a uniform "SaaSocalypse" but rather a strategic bifurcation: commodity utilities and custom core applications shift toward in-house "Make" decisions under hybrid governance models, while regulated, mission-critical systems remain in the "Buy" domain due to compliance, liability, and maintenance cost structures that code generation alone does not eliminate. Ultimately, the human role is not eliminated but elevated transitioning from manual code construction to intent specification, agent orchestration, cognitive debt management, verification auditing, and governance enforcement. Named human engineers remain legally, ethically, and operationally accountable for system reliability, safety, and security. Future research must prioritize hardened instruction-data segregation architectures, cross-backend verification consensus, live evaluation paradigms resistant to data contamination, and organizational frameworks that preserve architectural legibility under accelerating autonomous code evolution.

REFERENCES

- Abdollahi, A., Rezaei, M. A., Wang, X., Cai, D., He, S., Ayyoubzadeh, S. M., ... & Chen, H. (2026). The next paradigm in medical AI: a survey of agentic AI in biomedicine. *IEEE Reviews in Biomedical Engineering*.
- Alenezi, M. (2026). From prompt-response to goal-directed systems: The evolution of agentic ai software architecture. *arXiv preprint arXiv:2602.10479*.
- Alshahwan, N., Chheda, J., Finogenova, A., Gokkaya, B., Harman, M., Harper, I., Marginean, A., Sengupta, S., & Wang, E. (2024). Automated unit test improvement using large language models at Meta. In Companion proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (pp. 185-196). Association for Computing Machinery.
<https://doi.org/10.1145/3663529.3663839>
- Altmayer Pizzorno, J., & Berger, E. D. (2025). CoverUp: Effective high coverage test generation for Python. Proceedings of the ACM on Software Engineering, 2(FSE), Article FSE128, 2897-2919.
<https://doi.org/10.1145/3729398>
- Bairi, R., Sonwane, A., Kanade, A., D. C., V., Iyer, A. S., Parthasarathy, S., Rajamani, S., Ashok, B., & Shet, S. (2024). CodePlan: Repository-level coding using LLMs and planning. Proceedings of the ACM on Software Engineering, 1(FSE).
<https://www.microsoft.com/en-us/research/publication/codeplan-repository-level-coding-using-llms-and-planning-2/>
- Beg, A., O'Donoghue, D., & Monahan, R. (2025). Traceable and verifiable software requirements: A synthesis of AI-enabled formal methods. *Journal of Software Testing, Verification and Reliability*.

- Dakhel, A. M., Nikanjam, A., Majdinasab, V., Khomh, F., & Desmarais, M. C. (2024). Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171, Article 107468. <https://doi.org/10.1016/j.infsof.2024.107468>
- Felipe, L. G. (2026). Agentic AI Software Deployment: Enhancing Software Deployment with Agentic AI for Autonomous and Scalable Operations.
- Franklin, S. (2006). The LIDA architecture: Adding new modes of learning to an intelligent, autonomous, software agent. *pat.*
- Gajjar, U. Autonomous AI and Agentic Testing Agents: A Multi-Agent Architecture for Self-Directed Software Quality Assurance. *J Artif Intell Mach Learn & Data Sci* 2025, 8(4), 3481-3487.
- Garoché, P. L. (2019). Formal verification of control system software.
- Hagen, D., & Müller, P. (2025). Automating Hyper Hoare Logic via Predicate Transformers.
- Hasan, O., & Tahar, S. (2015). Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition* (pp. 7162-7170). IGI Global Scientific Publishing.
- Helwig, S., Hüffner, F., Rössling, I., & Weinard, M. (2010). Chapter 3. Selected Design Issues. In *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice* (pp. 58-126). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., & Schmidhuber, J. (2024). MetaGPT: Meta programming for a multi-agent collaborative framework. *International Conference on Learning Representations (ICLR)*. <https://arxiv.org/abs/2308.00352>
- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. R. (2024). SWE-bench: Can language models resolve real-world GitHub issues? *International Conference on Learning Representations (ICLR)*. <https://arxiv.org/abs/2310.06770>
- Li, K., Yu, H., & Yuan, Y. (2026). Escaping the self-repair trap: Improving test oracle generation via dual-context awareness. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*. Association for Computing Machinery. <https://doi.org/10.1145/3832783.3834421>
- Liu, T., Xu, C., & McAuley, J. (2024). RepoBench: Benchmarking repository-level code auto-completion systems. *International Conference on Learning Representations (ICLR)*. <https://arxiv.org/abs/2306.03091>
- Madika, B., Saha, A., Kang, C., Buyantogtokh, B., Agar, J., Wolverton, C. M., ... & Hong, S. (2025). Artificial intelligence for materials discovery, development, and optimization. *ACS nano*, 19(30), 27116-27158.
- Otoum, N., & Elkhaili, N. (2026). Methods and techniques of agentic software engineering: A systematic literature review. *IEEE Access*, 14, 7443-7465.
- Rombaut, B. (2026). Inside the scaffold: A source-code taxonomy of coding agent architectures. *arXiv preprint arXiv:2604.03515*.
- Sapkota, R., Roumeliotis, K. I., & Karkee, M. (2025). Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. *arXiv preprint arXiv:2505.19443*.
- Saxena, A. (2026). *Artificial Intelligence and Machine Learning*. BR Publications.
- Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36. <https://arxiv.org/abs/2302.04761>

- Shankar, N. (2018). Combining model checking and deduction. In *Handbook of Model Checking* (pp. 651-684). Cham: Springer International Publishing.
- Shapiro, D., Li, W., Delaflor, M., & Toxtli, C. (2023). Conceptual framework for autonomous cognitive entities. *arXiv preprint arXiv:2310.06775*.
- Vernon, D., Metta, G., & Sandini, G. (2007). A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents. *IEEE transactions on evolutionary computation*, 11(2), 151-180.
- Vieira, A., & Ribeiro, B. (2018). Introduction to deep learning business applications for developers.
- Wang, H., Gong, J., Zhang, H., & Wang, Z. (2025). AI agentic programming: A survey of techniques, challenges, and opportunities. *arXiv*. <https://arxiv.org/abs/2508.11126>
- Wei, T., Li, T. W., Liu, Z., Ning, X., Yang, Z., Zou, J., ... & He, J. (2026). Agentic reasoning for large language models. *arXiv preprint arXiv:2601.12538*.
- Weng, K., Du, L., Li, S., Lu, W., Sun, H., Liu, H., & Zhang, T. (2025). Autoformalization in the era of large language models: A survey. *arXiv preprint arXiv:2505.23486*.
- Wisakanto, A. K., Rogero, J., Casheekar, A. M., & Mallah, R. (2025). Adapting probabilistic risk assessment for AI. *arXiv preprint arXiv:2504.18536*.
- Xia, C. S., Deng, Y., Dunn, S., & Zhang, L. (2024). Agentless: Demystifying LLM-based software engineering agents. *arXiv*. <https://arxiv.org/abs/2407.01489>
- Xu, K., Luo, Y., Yuan, Y., & Yao, A. C. C. (2025). Towards Automated Formal Verification of Backend Systems with LLMs. *arXiv preprint arXiv:2506.10998*.
- Yan, L. (2025). From passive tool to socio-cognitive teammate: A conceptual framework for agentic AI in human-AI collaborative learning. *arXiv preprint arXiv:2508.14825*.
- Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37. <https://arxiv.org/abs/2405.15793>
- Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. *International Conference on Learning Representations (ICLR)*. <https://arxiv.org/abs/2210.03629>