Autonomous Software Engineering Agents: The Next Evolution of AI-Assisted Development

Diego Jimenez-Bosquez
Pontificia Universidad Católica del Ecuador
Sede Esmeraldas
Esmeraldas, Ecuador
ORCID: 0000-0003-1496-2983
diego.jimenez@pucese.edu.ec

Hernan Arturo Rojas Sanchéz
Universidad Estatal de Bolivar
Guaranda, Ecuador
ORCID: 0000-0001-5357-1585
arojas@ueb.edu.ec

Gregorio Sebastian Gualavisi Gonzalez
Pontificia Universidad Católica del Ecuador
Sede Esmeraldas
Esmeraldas, Ecuador
ORCID: 0009-0005-0351-2831
gsgualavasi@pucese.edu.ec

Andres Ricardo Guanolisa Plasencia
Pontificia Universidad Católica del Ecuador
Sede Esmeraldas
Esmeraldas, Ecuador
ORCID: 0000-0001-5906-4832
arguanoluisa@pucese.edu.ec

***Abstract*—The rapid advancement of Artificial Intelligence has led to the emergence of autonomous software engineering agents capable of performing complex development tasks with minimal human intervention. Unlike traditional AI coding assistants, these agents can analyze requirements, generate code, execute tests, identify defects, and iteratively improve software solutions through goal-oriented workflows. This paper explores the architecture, capabilities, and practical applications of autonomous software engineering agents within modern development environments. Particular attention is given to agent-based frameworks, multi-agent collaboration systems, automated debugging, software maintenance, and continuous integration pipelines. To ground the discussion empirically, we conduct a structured synthesis of published experimental evidence on the SWE-bench family of repository-level benchmarks and on multi-agent code-generation frameworks. The synthesized evidence shows that agentic scaffolding raised real-world GitHub issue resolution from under 2% for retrieval-augmented baselines to 12.5% (SWE-agent) and 18–27% on SWE-bench Lite (AutoCodeRover, OpenHands, Agentless), while self-reflective and multi-agent designs raised function-level synthesis above 85% pass@1. The study further examines the benefits and limitations of autonomous agents, including productivity gains, scalability, decision-making reliability, security concerns—notably indirect prompt injection—and ethical considerations. The findings indicate that autonomous software engineering agents represent a significant step toward intelligent software production systems, where human expertise and artificial intelligence collaborate to deliver more efficient, reliable, and scalable software solutions.**

***Index Terms*—Autonomous Agents, Artificial Intelligence, Software Engineering, AI Systems, Intelligent Development, Multi-Agent Systems, Software Automation**

I. Introduction

THE first generation of AI coding tools operated as *assistants*: reactive systems that completed the line a developer was typing or answered an isolated question. Powered by large language models (LLMs) trained on code [1], [2], these assistants demonstrably accelerated implementation work [3], yet the locus of control—deciding what to change, running tests, interpreting failures—remained entirely with the human. A second generation is now emerging: *autonomous software engineering agents* that receive a high-level goal (e.g., a GitHub issue), and then independently localize relevant code, plan a fix, edit multiple files, execute the test suite, observe failures, and iterate until the goal is met [4]–[6].

This shift from suggestion to action is architecturally grounded in three ideas from the language-agent literature: interleaving reasoning with environment actions (ReAct) [7], learning from self-generated feedback across attempts (Reflexion) [8], and equipping models with external tools [9]. It is empirically grounded in a new class of benchmarks—most prominently SWE-bench, built from 2,294 real issue–pull-request pairs across twelve popular Python repositories—that evaluate whether a system can resolve genuine maintenance tasks rather than synthesize toy functions [10].

This paper makes three contributions:

1) A conceptual architecture of autonomous software engineering agents—perception, planning, tool use, memory, and verification—and a taxonomy of single-agent and multi-agent frameworks (Sections III and IV).
2) An evidence synthesis aggregating published experimental results on repository-level issue resolution, multi-agent code generation, and automated program repair, under a transparent extraction protocol (Sections V and VI).
3) A critical analysis of reliability, security, cost, and ethical limitations, with a roadmap for safe human–agent collaboration in production pipelines (Sections VII and IX).

II. Background and Related Work

A. From Assistants to Agents

LLM-based assistants map a local context window to a code suggestion [1]. Agents, by contrast, are closed-loop systems:

the model's output is an *action* (open a file, run a command, apply an edit), the environment returns an *observation* (file contents, test output, compiler errors), and the loop repeats until a stopping condition [7], [11]. Surveys of LLM-based agents for software engineering catalog this rapidly growing design space, spanning requirements analysis, code generation, testing, debugging, and maintenance [12], [13].

B. Benchmarks for Autonomous Software Engineering

Function-level benchmarks such as HumanEval [1] measure isolated synthesis but not the cross-file reasoning that dominates real maintenance. SWE-bench addressed this gap by requiring systems to produce a repository patch that makes previously failing tests pass without breaking existing ones [10]. Its curated subset, SWE-bench Lite (300 instances), has become the de facto arena for comparing agent frameworks [4]–[6], [14].

III. Architecture of Autonomous SE Agents

Despite implementation diversity, published agents share five architectural components.

1) Perception and Context Acquisition: Agents must locate the small fraction of a repository relevant to the task. Strategies range from lexical/semantic retrieval [10], to program-structure-aware search over abstract syntax trees, which AutoCodeRover showed to be markedly more effective than plain file viewing [5], to hierarchical localization from files to classes to edit locations [14].

2) Planning and Reasoning: The ReAct pattern interleaves chain-of-thought reasoning with actions, letting the agent revise its plan as observations arrive [7]. Some frameworks decouple planning from execution entirely; Agentless demonstrates that a fixed three-phase pipeline (localize, repair, validate) can outperform fully autonomous planning at lower cost [14].

3) Tool Use and the Agent–Computer Interface: SWE-agent introduced the notion of an *agent–computer interface* (ACI): purpose-built file viewers, search utilities, and edit commands with guardrails (e.g., syntax checking on every edit). Carefully designed ACIs alone lifted resolution rates far above giving the model a raw shell [4], echoing the general finding that tool design is as important as model capability [9].

4) Memory: Agents maintain short-term memory (the interaction history within a task) and, increasingly, episodic memory across attempts. Reflexion showed that storing verbal self-critiques of failed attempts and conditioning subsequent attempts on them raises HumanEval pass@1 to 91%, surpassing the underlying GPT-4 baseline of 67% [2], [8].

5) Verification and Self-Correction: Execution is the agent's ground truth: running reproduction scripts and test suites converts plausible patches into validated ones. AutoDev integrates generation with build, execution, and test actions inside secured Docker containers, reporting 91.5% pass@1 for code generation and 87.8% for test generation on HumanEval when the full action space is available [15].

IV. Multi-Agent Collaboration Systems

A complementary line of work decomposes development across *multiple* role-specialized agents that communicate in natural language.

MetaGPT encodes standardized operating procedures—product manager, architect, engineer, QA—into an assembly line where agents exchange structured artifacts (requirement documents, design diagrams, interface specifications) rather than free-form chat, reaching 85.9% and 87.7% pass@1 on HumanEval and MBPP respectively [16].

ChatDev organizes a virtual software company whose agents converse through a chat chain covering design, coding, and testing; it completes small end-to-end applications in under seven minutes at a cost below one dollar, illustrating the economic potential of agentic decomposition [17].

AutoGen provides a general programming framework for building such systems, in which conversable agents—LLM-backed, tool-backed, or human—solve tasks through automated multi-agent conversation, and has been applied to coding, mathematics, and decision-making workflows [18].

The common rationale is error decorrelation and specialization: a reviewer agent with a testing persona catches defects the coder agent overlooks, at the price of additional token cost and coordination overhead [12].

V. Empirical Methodology

As in evidence-based software engineering, we synthesize published experimental results rather than anecdote, restricting extraction to primary studies with public artifacts so that every figure in Section VI is independently verifiable.

A. Research Questions

- RQ1 (Autonomy): What fraction of real-world repository-level tasks can autonomous agents resolve end-to-end, and how much of the gain is attributable to agentic scaffolding versus the base model?
- RQ2 (Collaboration): How do self-reflective and multi-agent designs affect code-generation quality relative to single-pass prompting?
- RQ3 (Dependability): What do current studies reveal about the reliability, cost, and security of autonomous agents?

B. Study Selection and Extraction Protocol

Inclusion criteria were: (i) peer-reviewed venue or widely replicated public report; (ii) quantitative evaluation on a public benchmark (SWE-bench, SWE-bench Lite, HumanEval, MBPP, Defects4J) with a public harness; (iii) explicit methodology; and (iv) relevance to one RQ. For RQ1 we extracted resolved-rate (% of issues whose hidden tests pass after the agent's patch) from the original papers [4]–[6], [10], [14], keeping the benchmark split fixed (SWE-bench full or Lite) within each comparison. For RQ2 we extracted pass@1 from Reflexion [8], MetaGPT [16], and AutoDev [15] against the single-pass GPT-4 baseline [2]. For RQ3 we extracted cost figures, failure analyses, and security findings [14], [19], [20], [22].

TABLE I
PUBLISHED RESOLUTION RATES OF AUTONOMOUS AGENTS ON REPOSITORY-LEVEL BENCHMARKS

System	Base model	Benchmark	Resolved (%)
RAG baseline [10]	Claude 2	SWE-bench (full)	1.96
SWE-agent [4]	GPT-4	SWE-bench (full)	12.5
SWE-agent [4]	GPT-4	SWE-bench Lite	18.0
AutoCodeRover [5]	GPT-4	SWE-bench Lite	≈19
OpenHands [6]	CodeAct	SWE-bench Lite	26.0
Agentless [14]	GPT-4o	SWE-bench Lite	27.3

C. Replication Protocol

RQ1 results can be reproduced as follows: (1) obtain SWE-bench Lite and its containerized evaluation harness [10]; (2) run the candidate agent on each instance from the repository snapshot at the issue's base commit, with no access to the gold patch or hidden tests; (3) apply the produced patch and execute the benchmark's fail-to-pass and pass-to-pass tests; (4) report the percentage of fully resolved instances and the mean cost per instance. Pinning model versions, temperature, and harness release is essential for comparability.

VI. RESULTS AND ANALYSIS

A. RQ1: End-to-End Issue Resolution

Table I and Fig. 1 summarize the published results. On the full SWE-bench test set, non-agentic retrieval-augmented baselines resolve under 2% of issues—the best, Claude 2 with BM25 retrieval, reaches 1.96% [10]. Wrapping a comparable model in SWE-agent's ACI raises this to 12.5% [4], a more than sixfold improvement attributable purely to scaffolding. On SWE-bench Lite, structure-aware search (AutoCodeRover, ≈19%) [5], generalist sandboxed agents (OpenHands CodeAct, 26.0%) [6], and the deliberately simplified Agentless pipeline (27.3% at an average cost of $0.34 per issue) [14] progressively raised the state of the art during 2024. Two conclusions follow: agentic interfaces, not just larger models, drive resolution gains; yet even the best published systems leave roughly three quarters of curated real-world issues unresolved, so full autonomy on maintenance work remains distant.

B. RQ2: Self-Reflection and Multi-Agent Collaboration

Table II contrasts single-pass generation with agentic designs at function and application granularity. Reflexion's verbal self-feedback loop lifts GPT-4 from 67.0% to 91.0% pass@1 on HumanEval—a 24-point gain obtained without any weight updates, purely through iteration against execution feedback [2], [8]. Role-specialized multi-agent pipelines achieve comparable function-level quality (MetaGPT: 85.9% HumanEval, 87.7% MBPP) while additionally producing coherent multi-file applications with design documents [16]. AutoDev shows that granting a single agent a full DevOps action space (build, execute, test) inside a sandbox yields 91.5% pass@1 for code and 87.8% for tests [15], and ChatDev demonstrates end-to-end application assembly in under seven

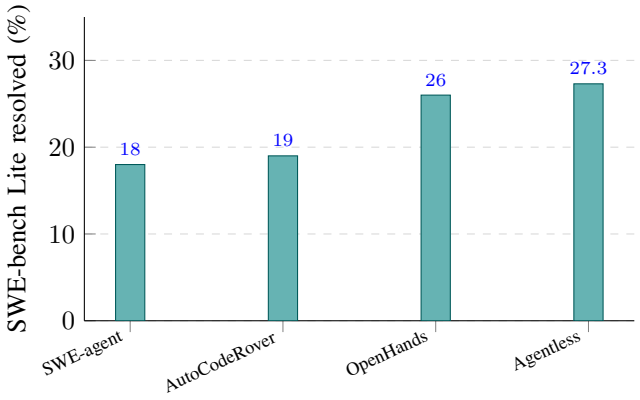

Fig. 1. Resolution rates of representative autonomous agent frameworks on SWE-bench Lite, as reported in the original publications [4]–[6], [14]. For reference, non-agentic retrieval baselines resolve under 2% of the full benchmark [10].

TABLE II
EFFECT OF REFLECTIVE AND MULTI-AGENT DESIGNS ON CODE GENERATION (PASS@1, %)

System	HumanEval	MBPP
GPT-4, single pass [2]	67.0	–
Reflexion (GPT-4) [8]	91.0	77.1
MetaGPT [16]	85.9	87.7
AutoDev (GPT-4) [15]	91.5	–

minutes for less than one dollar [17]. The consistent pattern is that *closing the loop with execution feedback*—whether via one reflective agent or several conversing specialists—is the dominant quality lever.

C. RQ3: Dependability, Cost, and Security

Reliability. Failure analyses on SWE-bench attribute most unresolved instances to incorrect fault localization and to patches that fix the symptom but break adjacent behavior; Agentless further documents that benchmark instances with insufficient problem descriptions inflate apparent failure rates, motivating its filtered subset [14]. In automated program repair, RepairAgent—an agent that autonomously interleaves bug diagnosis, fix synthesis, and validation—repaired 164 bugs in the Defects4J benchmark, including 39 bugs not fixed by prior techniques, at an average cost of cents per bug [19].

Cost and scalability. Agentic loops multiply token consumption relative to single-pass prompting, but published cost figures remain modest: $0.34 per attempted issue for Agentless [14] and sub-dollar end-to-end application builds for ChatDev [17], suggesting that economics will not be the binding constraint.

Security. Autonomy enlarges the attack surface. Greshake et al. demonstrated *indirect prompt injection*: adversarial instructions embedded in data the agent processes (web pages, repository files, issue text) can hijack tool-using LLM applications into exfiltrating data or executing attacker-chosen actions [20]. Combined with prior evidence that LLM-generated code

frequently contains exploitable weaknesses [21] and that AI assistance can increase developer overconfidence [22], this implies that agent actions—especially shell execution and patch merging—must be sandboxed, least-privileged, and human-gated in production.

VII. Discussion: Benefits and Limitations

A. Productivity and Integration into CI/CD

The natural insertion point for agents is the continuous-integration pipeline: triaging failing builds, drafting candidate patches for review, generating regression tests, and updating documentation. Because the CI environment already provides sandboxing and objective pass/fail signals, it supplies exactly the verification substrate that agent reliability depends on [10], [15]. Evidence from assistant-era studies that AI support accelerates well-scoped tasks [3] plausibly compounds when the agent also executes and validates its own output.

B. Decision-Making Reliability

Agents inherit LLM failure modes—hallucinated APIs, brittle long-horizon planning—and add new ones, such as action loops and premature termination [12]. The RQ1 ceiling of $\approx$27% on curated issues argues for a proposer–verifier division of labor: agents propose validated patches; humans retain merge authority on consequential changes.

C. Security and Ethical Considerations

Beyond injection attacks [20], autonomous patching raises accountability questions (who is responsible for an agent-introduced defect?), provenance questions (license compliance of generated code), and workforce questions (redistribution of junior-level tasks). Governance mechanisms—audit logs of every agent action, reproducible execution traces, and organizational policies defining which changes agents may merge autonomously—are prerequisites for responsible deployment.

VIII. Threats to Validity

Construct validity: resolved-rate on SWE-bench measures test-passing patches, not patch quality, maintainability, or security. Internal validity: the synthesized systems use different base models, prompts, and harness versions; we mitigated this by fixing the benchmark split within comparisons and reporting base models explicitly. External validity: SWE-bench covers popular Python repositories with strong test suites; generalization to other languages, weakly tested codebases, and proprietary systems is unestablished, and training-data contamination of public repositories is a recognized concern [10], [14]. Conclusion validity: heterogeneous designs preclude pooled effect estimates; results are therefore reported narratively with full provenance.

IX. Future Directions

Four directions will define the next phase. First, verification-rich autonomy: coupling agents with stronger oracles—property-based tests, static analysis, and formal specifications—so that self-validation extends beyond existing test suites. Second, long-horizon memory and project grounding: persistent, repository-specific memory that accumulates conventions and past decisions across tasks [11]. Third, secure agent infrastructure: principled defenses against indirect prompt injection, capability-based sandboxing, and standardized audit trails [20]. Fourth, benchmarks beyond bug fixing: multilingual, feature-development, and team-level benchmarks that measure what organizations actually need, addressing the gap between SWE-bench Lite and industrial maintenance [12], [13].

X. Conclusion

This paper examined autonomous software engineering agents through a structured synthesis of published experimental evidence. The data show that agentic scaffolding—interfaces, planning, execution feedback, and memory—rather than raw model scale alone, transformed repository-level issue resolution from a near-zero capability into a practically useful one, while reflective and multi-agent designs pushed function-level synthesis above 90% correctness. At the same time, three quarters of curated real-world issues remain unresolved, and autonomy introduces qualitatively new security and accountability risks. Autonomous agents are therefore best understood not as replacements for software engineers but as the next layer of the engineering stack: tireless proposers of validated changes operating under human governance. Organizations that pair agent autonomy with rigorous verification, sandboxing, and clear merge authority stand to capture substantial gains in efficiency, reliability, and scale—realizing intelligent software production systems in which human expertise and artificial intelligence are genuinely complementary.

References

[1] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

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

[3] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of AI on developer productivity: Evidence from GitHub Copilot," *arXiv preprint arXiv:2302.06590*, 2023.

[4] J. Yang *et al.*, "SWE-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024.

[5] Y. Zhang, C. S. Xia, and L. Zhang, "AutoCodeRover: Autonomous program improvement," *arXiv preprint arXiv:2404.05427*, 2024.

[6] X. Wang *et al.*, "OpenHands: An open platform for AI software developers as generalist agents," *arXiv preprint arXiv:2407.16741*, 2024.

[7] S. Yao *et al.*, "ReAct: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2023.

[8] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 8634–8652.

[9] T. Schick *et al.*, "Toolformer: Language models can teach themselves to use tools," *arXiv preprint arXiv:2302.04761*, 2023.

[10] C. E. Jimenez *et al.*, "SWE-bench: Can language models resolve real-world GitHub issues?," in *The Twelfth International Conference on Learning Representations*, 2024.

[11] L. Wang *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, pp. 1–26, 2024.

[12] Z. Liu *et al.*, "Large language models for software engineering: A comprehensive survey," *arXiv preprint arXiv:2402.13854*, 2024.

[13] X. Hou *et al.*, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–41, 2024.

[14] C. S. Xia *et al.*, "Agentless: Demystifying LLM-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024.
[15] M. Tufano *et al.*, "AutoDev: Automated AI-driven development," *arXiv preprint arXiv:2403.08299*, 2024.
[16] S. Hong *et al.*, "MetaGPT: Meta programming for a multi-agent collaborative framework," in *The Twelfth International Conference on Learning Representations*, 2024.
[17] C. Qian *et al.*, "Communicative agents for software development," *arXiv preprint arXiv:2308.00352*, 2023.
[18] Q. Wu *et al.*, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," *arXiv preprint arXiv:2308.08155*, 2023.
[19] S. Bouzenia *et al.*, "RepairAgent: An autonomous, LLM-based agent for program repair," in *Proceedings of the 47th International Conference on Software Engineering*, 2025.
[20] K. Greshake, S. Abdelnabi, S. Mishra, A. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 2023, pp. 79–90.
[21] H. Pearce, B. Ahmad, B. Raye, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 754–768.
[22] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2785–2799.