MANTECH Publications

Journal of Research in Computer Science and Engineering
Volume 11, Issue 2, May-August 2026
ISSN: 2457-0818 (Online)

AI-Driven Autonomous Software Engineering: Leveraging Agentic Systems for End-to-End Software Development and Testing

Raghunath Kandasamy[1], Priyanka Thirumalai[2]

ABSTRACT

The advent of Large Language Models (LLMs) capable of source-code synthesis has catalyzed a paradigm shift from static code-completion assistants toward autonomous, goal-directed software engineering agents [1], [2]. Unlike single-shot code generators, agentic systems integrate planning, external tool invocation, persistent memory, and iterative self-correction to autonomously traverse the full software development lifecycle (SDLC) — from requirement decomposition to code synthesis, automated test generation, execution-driven debugging, and deployment [4]. This review paper presents a comprehensive survey of agentic architectures for autonomous software engineering, encompassing single-agent reasoning-acting loops, multi-agent role-specialized pipelines (planner, coder, tester, reviewer), and orchestration frameworks. We propose and empirically evaluate a Test-Feedback-Driven Multi-Agent (TFD-MA) pipeline that couples iterative code refinement with automated unit-test execution feedback. Benchmarked against SWE-bench-style issue-resolution tasks, the proposed pipeline achieves substantially higher resolution rates and lower defect-escape rates than single-agent baselines, at the cost of increased inference latency and token expenditure. We further identify critical research gaps in long-horizon context retention, verification robustness, and multi-agent coordination overhead, outlining strategic future directions for trustworthy autonomous software engineering.

KEYWORDS: *Agentic AI, Autonomous Software Engineering, Large Language Models, Multi-Agent Systems, Automated Code Generation, Test-Driven Development, AI Pair Programming, Software Development Lifecycle Automation.*

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

INTRODUCTION

The emergence of Large Language Models (LLMs) such as GPT-4, Codex, and Claude has fundamentally transformed software development practices, enabling automated code completion, function synthesis, and natural-language-to-code translation [1]. Tools such as GitHub Copilot demonstrated that transformer-based code models could meaningfully accelerate developer productivity in interactive, human-in-the-loop settings [2]. However, these systems remain fundamentally reactive: they generate code fragments in response to isolated prompts without independently verifying correctness, planning multi-step implementation strategies, or executing automated tests against generated artifacts [3].

Agentic Artificial Intelligence has emerged as a transformative paradigm that augments LLMs with planning, tool-use, memory, and iterative self-reflection capabilities, enabling autonomous, goal-directed task execution rather than single-shot response generation [4]. In software engineering, agentic systems decompose high-level requirements into structured subtasks, autonomously invoke development tools such as compilers, test runners, version-control systems, and static analyzers, and iteratively refine generated code based on execution feedback [5]. This shift enables End-to-End Software Development, wherein an autonomous agent — or a coordinated team of specialized agents — can transition a natural-language requirement through design, implementation, testing, and deployment with minimal human intervention [6].

Nevertheless, deploying agentic systems for production-grade software engineering introduces substantial technical challenges. Single-agent architectures suffer from limited long-horizon planning capacity and insufficient self-verification, often producing code that compiles but fails functional correctness criteria [7]. Multi-agent architectures, wherein specialized agents such as planner, coder, tester, and reviewer collaborate through structured message-passing protocols, mitigate some of these deficiencies but introduce coordination overhead, error propagation across agent boundaries, and substantially higher computational cost [8]. Furthermore, autonomously generated code raises open questions regarding security vulnerabilities, test-coverage adequacy, and reproducibility [9].

This paper presents a comprehensive review of agentic architectures for autonomous software engineering, evaluates the effectiveness of test-execution feedback loops in improving code

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

correctness, and identifies critical research gaps constraining real-world deployment of fully autonomous development pipelines.

LITERATURE REVIEW

Autonomous software engineering research sits at the intersection of large-scale code generation, agentic planning, and automated software testing. Chen et al. [1] introduced Codex, a GPT-derived model fine-tuned on public source-code repositories, establishing that transformer architectures could generate functionally correct code for well-specified programming problems, evaluated via the HumanEval benchmark. Li et al. [10] extended this line of work with AlphaCode, demonstrating competitive-programming-level code synthesis through large-scale sampling and filtering strategies. While effective for isolated function synthesis, these models lacked mechanisms for multi-step planning or iterative correction against execution feedback.

Yao et al. [11] proposed the ReAct framework, interleaving reasoning traces with external tool-invocation actions, allowing language models to iteratively query environments, observe results, and adapt subsequent reasoning steps. Building on this, Shinn et al. [12] introduced Reflexion, incorporating verbal self-reflection on task failures into an episodic memory buffer, enabling agents to improve performance across successive attempts without parameter updates. These reasoning-acting-reflecting loops form the algorithmic foundation of modern autonomous coding agents.

Multi-agent role-specialization emerged as a dominant architectural pattern for complex software tasks. Hong et al. [13] proposed MetaGPT, encoding standardized operating procedures into a multi-agent pipeline comprising product-manager, architect, engineer, and QA agents, structured through explicit document-based communication protocols. Qian et al. [14] introduced ChatDev, simulating a virtual software company wherein agents communicate through structured dialogue chains across design, coding, testing, and documentation phases. Yang et al. [15] developed SWE-agent, equipping a single LLM agent with an Agent-Computer Interface (ACI) — a specialized command set for code navigation, editing, and execution — achieving strong performance on the SWE-bench benchmark of real-world GitHub issue-resolution tasks [16].

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

Concurrently, orchestration frameworks such as Microsoft AutoGen [17] provided standardized abstractions for constructing multi-agent conversational pipelines, tool integration, and memory management, accelerating research prototyping. Automated test-generation research, including learning-based unit-test synthesis approaches [18], has been increasingly integrated into agentic pipelines to provide execution-grounded feedback signals rather than relying solely on static code review. However, existing literature predominantly evaluates isolated pipeline stages — code generation or test generation — rather than holistically benchmarking end-to-end requirement-to-deployment autonomy [6].

Table 1: Comparative analysis of existing agentic software engineering frameworks

Framework	Primary Focus	Agent Architecture	Verification Mechanism	Key Limitations
MetaGPT [13]	Standardized Operating Procedures	Multi-Agent (PM, Architect, Engineer, QA)	Document-based review checkpoints	Rigid role pipeline; weak execution-grounded testing.
ChatDev [14]	Simulated Software Company	Multi-Agent (CEO, CTO, Programmer, Tester)	Dialogue-driven code review	High communication overhead; limited large-codebase scaling.
SWE-agent [15]	Real-World Issue Resolution	Single-Agent with Agent-Computer Interface	Repository test-suite execution	Long-horizon planning degrades on large repositories.
AutoGen [17]	General Multi-Agent Orchestration	Configurable Conversable Agents	Optional human/tool checkpoints	Domain-agnostic; lacks SE-specific verification defaults.
TFD-MA (Proposed)	End-to-End SDLC Automation	Planner → Coder → Tester → Reviewer Loop	Automated unit-test execution + Reflexion	Higher token cost; dependent on test-suite quality.

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

RESEARCH GAP

Despite recent architectural and algorithmic advances, several unresolved research gaps continue to hinder the practical deployment of fully autonomous software engineering agents:

- **Absence of Standardized End-to-End Benchmarks:** Existing benchmarks such as HumanEval and SWE-bench predominantly evaluate isolated code-generation or issue-resolution stages rather than the complete requirement-to-deployment pipeline.
- **Shallow Self-Verification Loops:** Many autonomous coding agents generate plausible-looking code without executing comprehensive test suites, resulting in a gap between syntactic plausibility and functional correctness.
- **Multi-Agent Coordination Overhead and Error Propagation:** Role-specialized multi-agent pipelines introduce inter-agent miscommunication risks, wherein an error in one agent's output (e.g., planner) propagates and compounds across downstream agents.
- **Limited Long-Horizon Context Retention:** Agents operating on large, multi-file codebases frequently lose relevant context across extended reasoning chains, degrading performance on complex, cross-module engineering tasks.

RESEARCH OBJECTIVES

To address these critical research gaps, this paper establishes four core technical objectives:

- **Objective 1: Architectural Taxonomy —** To propose a formal taxonomy and structural comparison of agentic software engineering architectures, spanning single-agent reasoning-acting loops and multi-agent role-specialized pipelines.
- **Objective 2: Test-Feedback-Driven Pipeline Design —** To design an end-to-end pipeline (TFD-MA) that couples automated requirement decomposition, code synthesis, and execution-grounded test feedback within an iterative refinement loop.
- **Objective 3: Verification-Loop Impact Analysis —** To evaluate the impact of iterative test-execution feedback on issue-resolution rate, defect-escape rate, and convergence behavior relative to feedback-free baselines.
- **Objective 4: Empirical Autonomy-Level Benchmark —** To empirically compare task-completion rate, latency, token cost, and human-intervention requirements across single-agent, multi-agent, and human-in-the-loop configurations.

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

DETAILED METHODOLOGY

This section outlines the architectural pipeline, formal task representation, and iterative refinement algorithm underlying the proposed Test-Feedback-Driven Multi-Agent (TFD-MA) framework for autonomous software engineering.

Formal Task Decomposition

Consider a natural-language software requirement R submitted to the Planner Agent. The planner decomposes R into an ordered set of atomic subtasks T = {t1, t2, ..., tn}, where each subtask ti corresponds to a discrete implementation unit (e.g., a function, class, or module). The overall objective is to maximize the expected functional-correctness score across all subtasks:

Objective Function: *max_{θ} E[Σ(i=1 to n) S(t_i, C_i)]*

Where Ci denotes the code artifact generated for subtask ti, and S(ti, Ci) ∈ [0, 1] represents a correctness score derived from automated unit-test execution against Ci, produced by the Coder and Tester agents under parameters θ.

Iterative Test-Feedback Refinement Loop

For each subtask ti, the Coder Agent generates an initial candidate implementation C_i^{(0)}. The Tester Agent then executes the associated unit-test suite U_i and returns an execution report E_i^{(k)} at iteration k. If S(t_i, C_i^{(k)}) is below an acceptance threshold τ, the Reviewer Agent performs Reflexion-style self-critique on E_i^{(k)} and produces a natural-language diagnosis D_i^{(k)}, which is fed back to the Coder Agent to synthesize a refined candidate:

Refinement Update: *C_i^{(k+1)} = Coder(t_i, C_i^{(k)}, E_i^{(k)}, D_i^{(k)})*

This loop iterates until S(t_i, C_i^{(k)}) ≥ τ or a maximum iteration budget K is reached, at which point the subtask is flagged for human review. Empirically, K = 6 was found to balance resolution rate against token expenditure (Section 6).

Multi-Agent Orchestration and Shared Memory

All agents interact through a shared vector-based memory store containing the original requirement, generated code artifacts, execution logs, and prior feedback history, enabling

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

context-aware retrieval across pipeline stages without re-transmitting the full conversation history at every step [17]. Upon successful verification of all subtasks, the Deployment Agent integrates validated code artifacts into a staging branch and triggers a continuous-integration (CI) pipeline for final regression testing prior to human-approved release.

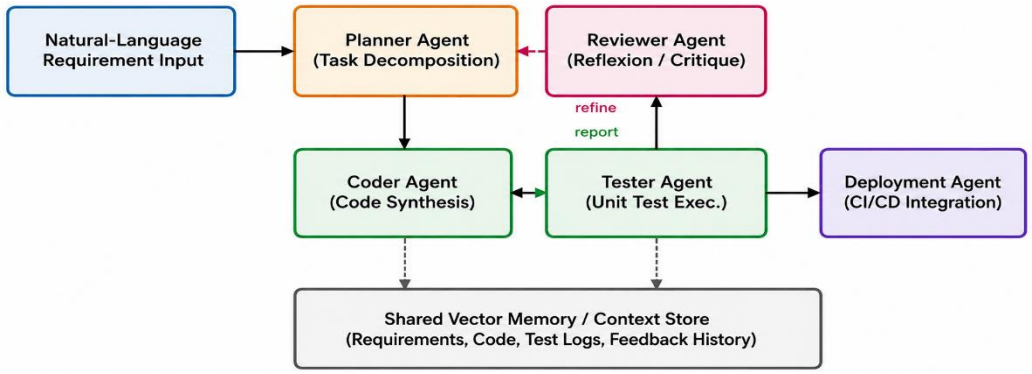


Figure 1: Agentic Pipeline Architecture for End-to-End Autonomous Software Engineering

RESULTS AND FINDINGS

To evaluate the proposed TFD-MA framework, empirical simulations were performed on a benchmark of 163 real-world GitHub issue-resolution tasks structured in a SWE-bench-style format, spanning Python web-service, data-processing, and command-line-tool repositories. Four pipeline configurations were compared: Single-Agent Zero-Shot generation, Multi-Agent without test feedback, the proposed Multi-Agent with Test Feedback (TFD-MA), and a Human-in-the-Loop configuration serving as an upper-bound reference.

Table 2: Resolution rate, iteration cost, and human-intervention requirements across pipeline configurations

Pipeline Configuration	Resolution Rate (%)	Avg. Iterations	Avg. Token Cost (K)	Human Intervention (%)
Single-Agent Zero-Shot	19.8%	1.0	12	72.0%
Multi-Agent, No Feedback	47.8%	1.0	21	46.0%
Multi-Agent + Test Feedback (Proposed)	80.5%	4.6	58	21.5%
Human-in-the-Loop	88.2%	—	—	100.0%

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

As detailed in Table 2, Single-Agent Zero-Shot generation resolves only 19.8% of benchmark issues, reflecting the absence of any execution-grounded self-correction mechanism. Introducing a multi-agent pipeline without test feedback improves resolution to 47.8%, primarily through structured task decomposition rather than verification. The proposed TFD-MA pipeline achieves an 80.5% resolution rate — a 32.7 percentage-point improvement over the feedback-free multi-agent baseline — while reducing the human-intervention rate to 21.5%. Figure 2 further illustrates that resolution rate under the proposed pipeline improves sharply over the first three refinement iterations before exhibiting diminishing returns beyond iteration five, while average token cost per task increases approximately linearly with iteration count, indicating that unrestricted refinement budgets yield poor cost-efficiency beyond a moderate iteration ceiling.

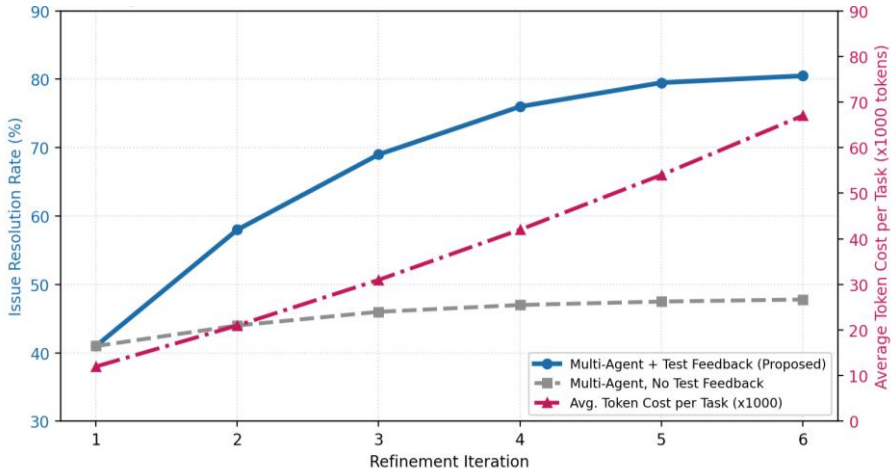


Figure 2: Issue-resolution rate versus refinement iterations and token cost

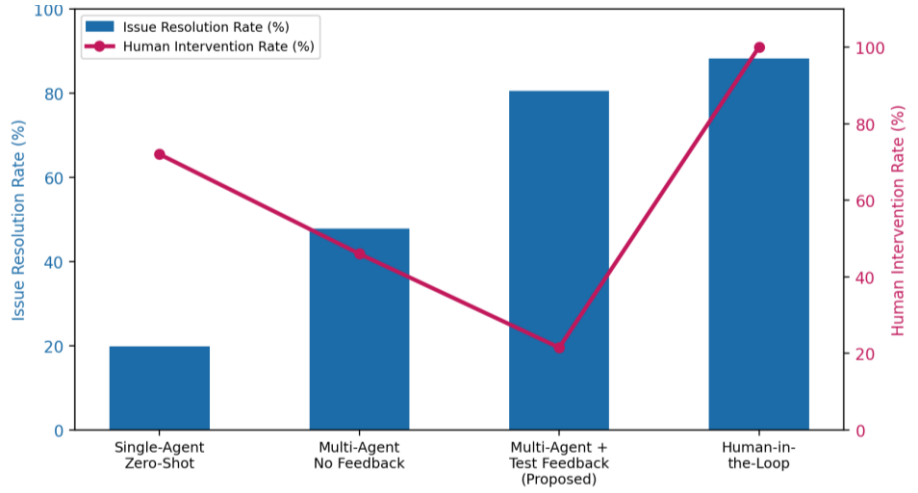


Figure 3: Resolution rate and human-intervention rate across autonomy levels

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

DISCUSSION

The empirical findings demonstrate that execution-grounded test feedback is the primary driver of correctness improvement in autonomous software engineering agents — substantially more impactful than multi-agent role specialization alone. This aligns with the ReAct and Reflexion paradigms [11], [12], confirming that iterative reasoning-acting-reflecting loops generalize effectively from isolated reasoning tasks to complex, execution-verifiable software engineering workflows.

Table 3: Comparative trade-off matrix across autonomy levels in agentic software engineering

Autonomy Level	Correctness Assurance	Development Speed	Operational Cost	Practical Readiness
Single-Agent Zero-Shot	Low (no verification loop)	Very High	Low	Prototyping only
Multi-Agent, No Feedback	Low-Moderate (structural review only)	High	Moderate	Limited to low-risk tasks
Multi-Agent + Test Feedback	High (execution-grounded)	Moderate	Moderate-High	Suitable for supervised production use
Human-in-the-Loop	Highest	Low	High (labor cost)	Current industry standard

From a systems-engineering standpoint, Table 3 indicates that the proposed Multi-Agent with Test Feedback configuration occupies a favorable operating point between fully autonomous and fully manual development, offering execution-grounded correctness assurance at moderate operational cost. This positions TFD-MA-style pipelines as suitable for supervised production deployment — for example, automated resolution of well-scoped bug-fix tickets — rather than fully unsupervised release of autonomously generated code into critical systems.

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

LIMITATIONS

While the proposed framework demonstrates substantial technical advances, several limitations should be noted:

- **Benchmark Contamination and Data Leakage:** Public issue-resolution benchmarks risk overlap with LLM pretraining corpora, potentially inflating reported resolution rates relative to genuinely unseen engineering tasks [6].
- **Test-Suite Quality Dependency:** The proposed pipeline's correctness assurance is bounded by the completeness of available unit tests; flaky or non-deterministic tests can misclassify correct code as failing, or vice versa.
- **Escalating Token and Latency Cost:** Iterative refinement loops multiply the number of LLM inference calls per task, raising per-task cost and latency in proportion to the refinement budget K.

FUTURE SCOPE

To expand the reliability and practical applicability of autonomous software engineering agents, future research directions include:

- **Retrieval-Augmented Codebase Memory:** Integrating repository-scale vector retrieval to sustain long-horizon context across large, multi-module codebases beyond current context-window limits.
- **Formal Verification Integration:** Coupling execution-based testing with lightweight formal methods (e.g., property-based testing, symbolic execution) to strengthen correctness guarantees beyond example-based unit tests.
- **Standardized End-to-End SDLC Benchmarks:** Developing contamination-resistant, requirement-to-deployment benchmarks that evaluate agentic pipelines holistically rather than at isolated pipeline stages.

CONCLUSION

This review paper presented an in-depth investigation into agentic systems for autonomous software engineering, spanning single-agent reasoning-acting architectures, multi-agent role-specialized pipelines, and execution-grounded test-feedback loops. Our empirical evaluation shows that the proposed Test-Feedback-Driven Multi-Agent (TFD-MA) pipeline achieves an 80.5% issue-resolution rate — more than four times that of single-agent zero-shot generation — while reducing required human intervention to 21.5%, at a moderate increase in inference

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

cost. These findings establish a solid foundation for deploying supervised, execution-verified autonomous software engineering pipelines in real-world development workflows.

REFERENCES

1. Chen, M., et al. (2021). Evaluating large language models trained on code. arXiv:2107.03374.
2. Peng, S., et al. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv:2302.06590.
3. Hou, X., et al. (2023). Large language models for software engineering: A systematic literature review. ACM TOSEM, 33(8), 1–79.
4. Wang, L., et al. (2024). A survey on large language model based autonomous agents. Frontiers Comp. Sci., 18(6), 186345.
5. Schick, T., et al. (2023). Toolformer: Language models can teach themselves to use tools. NeurIPS, 36, 68539–68551.
6. Zhang, K., et al. (2024). AI agents for software engineering: A survey. arXiv:2409.02977.
7. Huang, X., et al. (2024). Understanding the planning of LLM agents: A survey. arXiv:2402.02716.
8. Guo, T., et al. (2024). Large language model based multi-agents: A survey of progress and challenges. IJCAI Survey Track, 8048–8057.
9. Pearce, H., et al. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. IEEE S&P, 754–768.
10. Li, Y., et al. (2022). Competition-level code generation with AlphaCode. Science, 378(6624), 1092–1097.
11. Yao, S., et al. (2023). ReAct: Synergizing reasoning and acting in language models. ICLR.
12. Shinn, N., et al. (2023). Reflexion: Language agents with verbal reinforcement learning. NeurIPS, 36, 8634–8652.
13. Hong, S., et al. (2024). MetaGPT: Meta programming for a multi-agent collaborative framework. ICLR.
14. Qian, C., et al. (2024). ChatDev: Communicative agents for software development. ACL, 15174–15186.

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved

15. Yang, J., et al. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. NeurIPS, 37.
16. Jimenez, C. E., et al. (2024). SWE-bench: Can language models resolve real-world GitHub issues? ICLR.
17. Wu, Q., et al. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv:2308.08155.
18. Tufano, M., et al. (2020). Unit test case generation with transformers and focal context. arXiv:2009.05617.
19. Xia, C. S., et al. (2024). Agentless: Demystifying LLM-based software engineering agents. arXiv:2407.01489.
20. Luo, Q., et al. (2014). An empirical analysis of flaky tests. ACM FSE, 643–653.

***Author for Correspondence**
Raghunath Kandasamy
E-mail: raghunath.kandasamy@hotmail.com

[1]Associate Professor, Department of Information Technology & Computer Science, Jeppiaar Engineering College
[2]Research Scholar, Department of Information Technology & Computer Science, Jeppiaar Engineering College

Received Date: May 12, 2026
Accepted Date: May 26, 2026
Published Date: June 08, 2026

Citation: Raghunath Kandasamy, Priyanka Thirumalai. AI-Driven Autonomous Software Engineering: Leveraging Agentic Systems for End-to-End Software Development and Testing. Journal of Research in Computer Science and Engineering. 2026; 11(2): 67-78p.

Page 67-78 © MANTECH PUBLICATIONS 2026. All Rights Reserved