SPECTRUM OF ENGINEERING SCIENCES
ISSN (E) 3007-3138 (P) 3007-312X VOLUME 4, ISSUE 3, 2026

LLM BASED AUTONOMOUS AGENTS FOR AUTOMATED SOFTWARE DEVELOPMENT, TESTING, DEBUGGING, AND MAINTENANCE

Muhammad Usman[*1], Hammad Tariq[2], Jahangir khan[3], Gullelala Jadoon[4]

[*1]*University of Mainsail,*
[2]*Riphah International University Department: Software Engineering*
[3]*Department of Computer Science, National University of Modern language Islamabad*
[4]*Department of Information Technology, University of Haripur*

[*1]mr.muhammadusman2015@gmail.com, [2]hammadworkshops@gmail.com, [3]jkhan335953@gmail.com
[4]gullelala@uoh.edu.pk

DOI: https://doi.org/10.5281/zenodo.23559840

Keywords
autonomous agents, large language models, software engineering, automated program repair, multi-agent systems, code generation, software maintenance, evaluation benchmarks

Article History
Received: 01 January 2026
Accepted: 16 March 2026
Published: 31 March 2026

Copyright @Author
Corresponding Author: *
Muhammad Usman

Abstract

The emergence of Large Language Model (LLM)-based autonomous agents represents a paradigm shift in software engineering, transitioning from passive code completion tools to active, environment-grounded cyber entities capable of orchestrating complex development workflows. This systematic review examines the architectural foundations, lifecycle applications, evaluation methodologies, and systemic trade-offs of LLM-based autonomous agents across automated software development, testing, debugging, and maintenance. We deconstruct core agent architectures comprising reasoning engines, memory structures, tool interfaces, and verification controllers that enable autonomous repository-scale operations. Our analysis reveals that multi-agent collaboration topologies introduce a fundamental trade-off between coordination dividends derived from task decomposability and context isolation and coordination taxes from handoff losses, state desynchronization, and token inflation. Empirical evaluations across benchmarks including SWE-bench Verified, SWE-bench Lite, and SWE-EVO demonstrate that while state-of-the-art models achieve resolution rates exceeding 70% on isolated bug-fixing tasks, performance dramatically declines to 18-25% on long-horizon software evolution scenarios. Notably, minimalist, scaffolded frameworks achieve competitive performance at significantly lower operational costs, challenging assumptions about required architectural complexity. We identify critical challenges including context window constraints, verification reliability, and state synchronization in extended execution trajectories. Our synthesis suggests that future advances depend on integrating neuro-symbolic verification frameworks, optimizing agent communication topologies, and developing rigorous benchmarks that better capture real-world software evolution demands. This review provides researchers and practitioners with a comprehensive framework for understanding the capabilities, limitations, and design principles of LLM-based autonomous agents in software engineering.

1. INTRODUCTION

1.1 Evolution of AI in Software Engineering

Software engineering has undergone a paradigm shift in its underlying methods for automated code synthesis, program repair, and system evolution. Early applications of Large Language Models (LLMs) in software engineering focused primarily on localized, function-level code completions or isolated single-file tasks. Systems operating under traditional Retrieval-Augmented Generation (RAG) paradigms functioned passively, relying on localized text prompts to retrieve relevant snippets and predict syntax sequences (Ghai et al., 2024). While these early generation models proved effective for standard auto-completion and short syntax generation, they lacked awareness of full repository structures, operational project dependencies, execution environments, and compiler feedback loops. Consequently, when tasked with resolving real-world, repository-scale software bugs, traditional RAG baselines performed poorly, resolving fewer than 2% of GitHub issues in benchmark evaluations (Izadi et al., 2022).

The transition from static code generation models to LLM-based autonomous agents marks a fundamental shift from code synthesis as passive text generation to software engineering as tool-mediated, environment-grounded action (Zhang et al., 2025). Modern software engineering agents couple core foundational reasoning engines with execution harnesses, allowing them to autonomously navigate complex directory trees, parse abstract syntax trees (ASTs), execute terminal commands, analyze build logs, run regression test suites, and iteratively modify multi-file codebases (Ning et al., 2026). This shift to active agency has enabled automated systems to achieve resolution rates exceeding 30% on repository-level benchmarks such as SWE-bench Lite and over 70% on curated subsets like SWE-bench Verified (Teo et al., 2025).

1.2 Paradigm Shift: Passive Chatbots to Autonomous Cyber Entities

The evolution of LLMs from passive conversational assistants into autonomous cyber entities represents a fundamental change in software automation. Modern autonomous agents do not merely answer natural language queries; they independently organize and execute complex technical workflows across extended operational horizons. By integrating dynamic perception, multi-step goal planning, structured memory retrieval, and tool-use capabilities, these systems complete multi-stage software engineering tasks with minimal human intervention (Wang et al., 2025).

In modern development environments, this autonomous operational capacity spans the entire software engineering lifecycle. Autonomous agents actively manage continuous integration and deployment (CI/CD) workflows, perform complex multi-file refactoring, reproduce reported defects by synthesizing standalone test cases, and conduct automated, interactive code reviews. At the same time, this agentic autonomy presents dual-use security risks (Joshi, 2025). Cyber entities equipped with execution environments and network command toolkits can execute attack strategies including automated target scouting, vulnerability discovery, credential harvesting, and exploit synthesis thereby altering both offensive and defensive cybersecurity dynamics (Xu et al., 2025).

1.3 Scope and Structure of the Review

This review provides a systematic examination of LLM-based autonomous agents across the software development lifecycle. The paper synthesizes empirical findings, system architectures, and theoretical foundations across four primary domains: automated software development, testing, debugging, and software maintenance. Section 2 deconstructs the architectural foundations, memory structures, tool interfaces, and collaboration topologies that govern agent behavior. Section 3 systematically examines lifecycle applications, ranging from multi-agent enterprise simulations to formal SMT-driven legacy code migration (He et al., 2025). Section 4 evaluates empirical performance across benchmarks like SWE-bench Verified and SWE-EVO, while dissecting critical data integrity issues such as solution leakage and weak test evaluation. Section 5 synthesizes key theoretical trade-offs,

including the balance between agent coordination dividends and communication overhead. Section 6 outlines major technical challenges and future research directions, and Section 7 presents the concluding remarks (Ma et al., 2025).

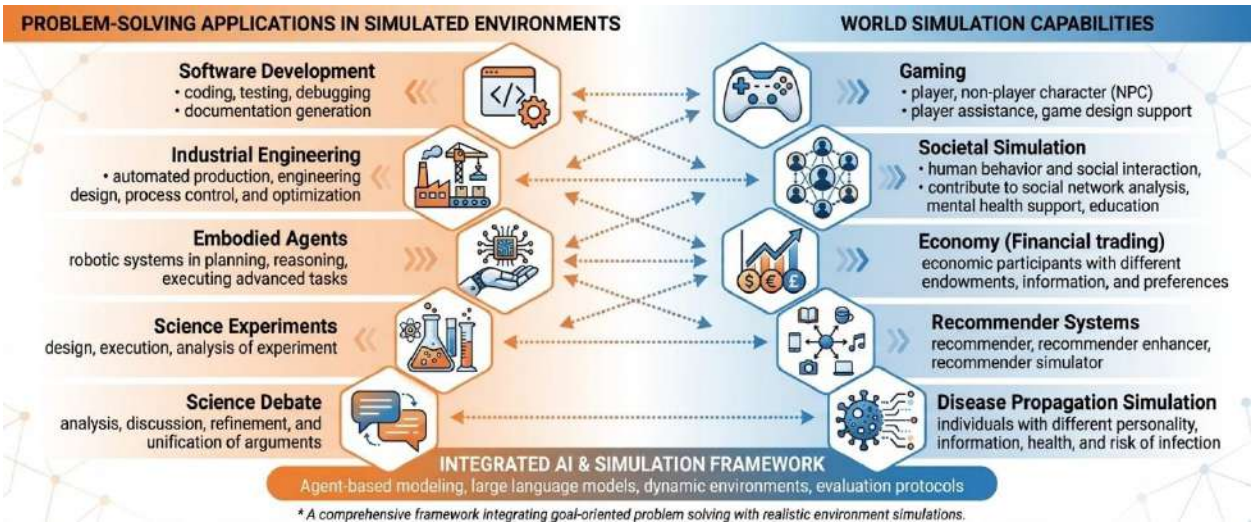

Figure 2: Comprehensive Framework Integrating Problem-Solving Applications and World Simulation Capabilities

2. Architectural Foundations and Taxonomies of SE Agents

2.1 Core Architectural Components

An LLM-based software engineering agent combines an underlying foundational language model with four core operational pillars: reasoning and planning engines, memory architectures, tool interfaces, and verification feedback controllers (Liu et al., 2024).

The reasoning and planning engine serves as the central control unit, decomposing abstract software requirements or bug reports into structured operational steps. Modern agents employ iterative reasoning loops such as Reasoning and Acting (ReAct), Chain-of-Thought (CoT), or Tree-of-Thought (ToT) to plan and adapt their actions. In complex dynamic codebases, control architectures also integrate self-adaptive control loops inspired by the Monitor-Analyze-Plan-Execute over Knowledge (MAPE-K) model, allowing agents to respond dynamically to execution errors and system state changes (Bruegge & Dutoit, 2010).

Memory architectures provide the storage mechanisms required to maintain context across complex project repositories and long execution trajectories. Agent memory is divided into short-term working memory and long-term externalized memory. Short-term memory maintains immediate command execution outputs, local code windows, and recent interaction traces within the LLM's active context window (Cacciatori, 2008). Long-term memory utilizes vector indexes, hierarchical repository summaries, or externalized evidence schemas (such as TrajSpec) to store and retrieve historical interaction trajectories, system dependency graphs, and project conventions (Hu et al., 2025).

Tool interfaces allow agents to perceive and alter their software execution environments. These tools fall into three functional categories: data tools, action tools, and orchestration tools. Data tools handle passive information gathering and code discovery through file readers, Abstract Syntax Tree (AST) parsers, grep utilities, port scanners, and vulnerability enumerators (Yang et al., 2024). Action tools manipulate the system environment directly via command-line shell execution, file editing mechanisms, automated test runners, build systems, and network payload dispatchers. Orchestration tools manage workflow control, sequencing sub-actions, handling retries,

and dispatching sub-tasks to specialized sub-agents (Shotts, 2026).

Verification and execution controllers establish closed-loop feedback mechanisms that evaluate agent actions. Instead of relying solely on unvalidated model generations, verification controllers capture compiler messages, runtime exception traces, unit test execution states, and automated theorem prover outputs. This execution feedback is routed back to the reasoning engine, enabling the agent to self-correct invalid syntax or broken patches before finalizing changes (Liu et al., 2025).

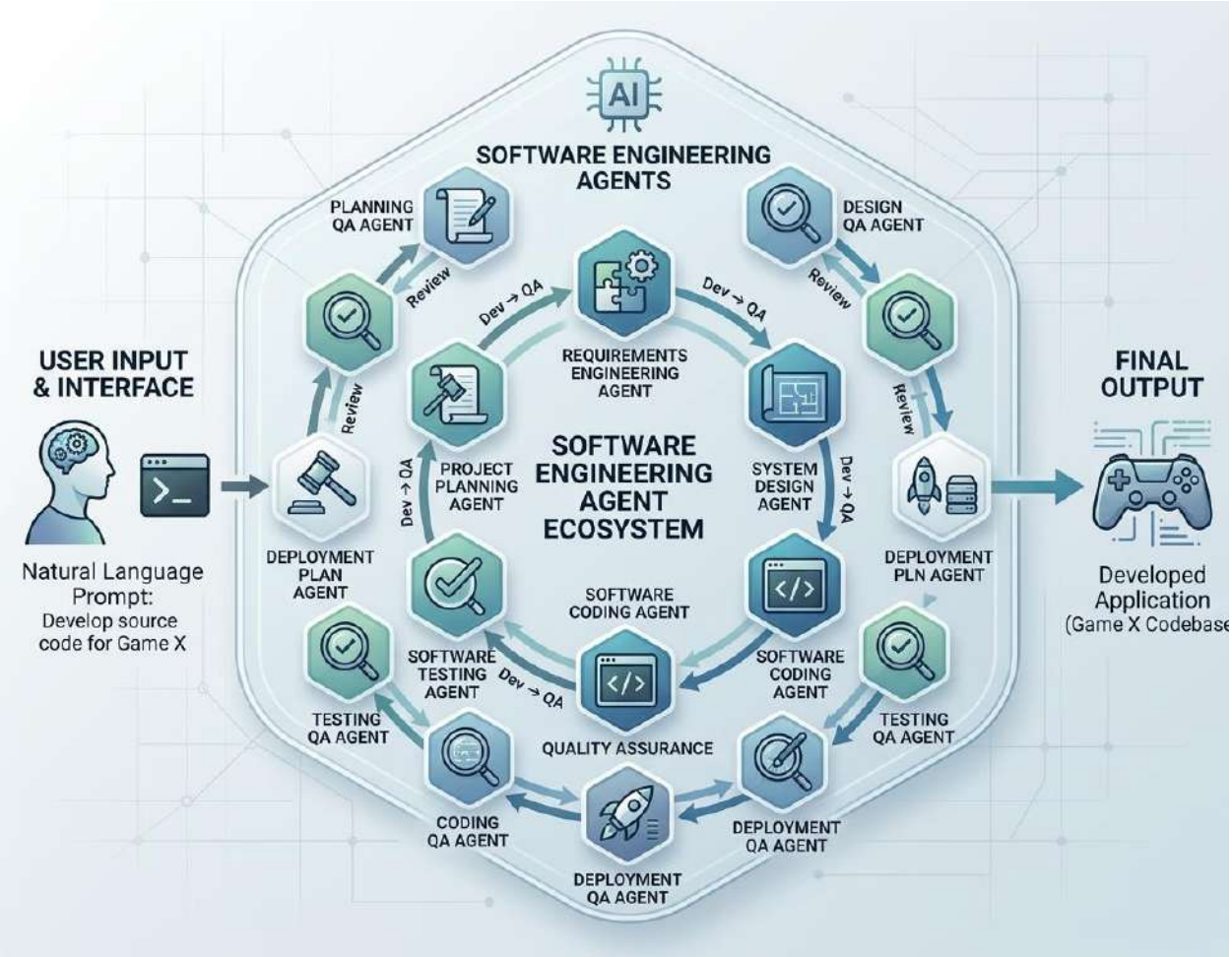

Figure 2: Architectural Topology of a Multi-Agent Quality Assurance and Development Ecosystem

2.2 Collaboration Paradigms and Communication Topologies

The architectural design of software engineering agents spans a spectrum from highly structured multi-agent organizations to streamlined single-agent pipelines.

Multi-agent systems (MAS) frame software engineering as a collaborative organizational process (Lin et al., 2025). Persona-based multi-agent frameworks, such as ChatDev and MetaGPT, simulate entire software development teams by assigning agents specialized functional roles, such as Chief Executive Officer, Product Manager, System Architect, Programmer, and Quality Assurance Engineer. Communication within these systems occurs either through conversational chat chains or structured artifact

exchanges governed by Standard Operating Procedures (SOPs) (Tseng et al., 2024).

Modular task-graph architectures, such as MASAI and AgentCoder, replace social role-playing with specialized, goal-driven sub-agents. Instead of engaging in open-ended conversations, specialized sub-agents such as Issue Reproducers, Edit Localizers, Patch Fixers, and Test Designers interact through explicit message pathways, receiving constrained sub-objectives and localized context windows (Kwon & Zhang, 2025).

Scaffolded single-agent and minimalist architectures such as SWE-agent, OpenHands, AutoCodeRover, and Agentless prioritize direct tool interaction through optimized Agent-Computer Interfaces (ACIs) or deterministic multi-stage pipelines. These approaches demonstrate that strong search, code editing, and command execution capabilities can match or exceed the performance of complex conversational agent networks (Rombaut, 2026).

2.3 Execution Control and Autonomy Taxonomies

Systemic autonomy in software engineering agents can be classified into three operational levels based on execution control flow:

1. **Emergent Autonomy**: The agent is provided with an open-ended goal and a set of execution tools. The operational trajectory, tool selection, step ordering, retries, and exit conditions are determined dynamically by the agent based on environmental feedback and internal reasoning. Systems like interactive, terminal-based SWE-agent setups instantiate this fully agentic pattern (Karuppuchamy, 2026).
2. **Scaffolded Execution**: A human-designed workflow defines the global operational stages (such as fault localization → patch generation → test verification) and the permitted transitions between them. The agent exercises local autonomy within each stage, selecting specific tool calls and local strategies. Frameworks such as MASAI and PANTHER rely on scaffolded execution (Martinez & Franch, 2025).
3. **Fixed Execution**: The control flow follows a hardcoded sequence with deterministic branching logic. The system lacks dynamic replanning capabilities and executes a fixed set of actions, similar to traditional Automated Program Repair (APR) pipelines. Agentless operates under a fixed two-phase localization and repair workflow (Borghoff et al., 2025).

Table 1: Comparative Taxonomy of Prominent LLM-Based Software Engineering Agent Architectures

Agent Framework	Topology / Paradigm	Control Autonomy	Primary Operational Strategy	Target Task Scope	Key Benchmark Performance	Financial & Token Cost Profile
ChatDev	Virtual Software Organization / Multi-Agent	Emergent Conversational	Role-based conversational chat chains across design, coding, and testing phases	Small, standalone, end-to-end applications	85.9% Pass@1 (HumanEval)	Low financial cost (<1.00/app), high interaction turns
MetaGPT	SOP Assembly Line / Multi-Agent	Scaffolded Workflow	Exchange of structured artifacts (PRDs, architecture diagrams) via SOPs	Requirement-to-code enterprise software synthesis	85.9% Pass@1 (HumanEval), 87.7% Pass@1 (MBPP)	High context efficiency via structured document handoffs

SWE-agent [cite: 3, 5, 13]	Single Agent + Custom ACI	Emergent Autonomy	Interactive bash terminal control via specialized file view/edit tools	Repository-level GitHub issue resolution	12.5% (SWE-bench Full), 22.92% (SWE-EVO w/ GPT-5.2)	High token consumption due to long interaction histories
AutoCodeRover	Two-Stage Sub-Agent Pipeline	Scaffolded Execution	AST-guided code search paired with context-aware patch generation	Repository bug fixing and program repair	30.67% (SWE-bench Lite)	Moderate cost (0.50–1.00 per issue)
Agentless	Minimalist Two-Phase Pipeline	Fixed Execution	Rigid hierarchical file/function ranking followed by patch synthesis	Repository bug fixing without tool-use loops	27.33% (SWE-bench Lite)	Extremely cost-effective (0.34 per issue average)
MASAI	Decoupled Task-Graph Sub-Agents	Scaffolded Execution	Goal-constrained sub-agents (Localizer, Fixer, Reproducer) using ReAct/CoT	End-to-end repository issue resolution	33.30% (SWE-bench Lite)	High efficiency due to localized sub-agent contexts
PANTHER	Hierarchical Multi-Stage Agent	Scaffolded Execution	Hierarchical fault localization paired with dual-patch validation	Repository issue resolution and program repair	32.00% (SWE-bench Lite)	Highly efficient (0.56 / ~39.8k tokens per issue)

3. Lifecycle Applications in Software Engineering

3.1 Automated Requirement Engineering and System Synthesis

In the initial stages of the software engineering lifecycle, agents translate high-level, ambiguous user requirements into structured software architectures and executable codebases. Role-based multi-agent frameworks handle requirement engineering by simulating software product teams (Akbar et al., 2025). In MetaGPT, a product manager agent parses user prompts to synthesize Product Requirement Documents (PRDs), an architect agent converts PRDs into Unified Modeling Language (UML) class structures and system interface specifications, and engineer agents implement the designated code modules (Hong et al., 2024).

ChatDev structures software synthesis around communicative seminars. By organizing agents into sequential stages design, coding, testing, and documentation ChatDev can generate small, executable software applications in under seven minutes at operational costs below one dollar.

However, empirical studies indicate that while role-based multi-agent frameworks excel at generating standalone applications or solving function-level benchmarks like HumanEval and MBPP, they struggle when applied directly to large, highly coupled pre-existing codebases (Qian et al., 2024).

3.2 Code Generation, Testing, and Test-Driven Refinement

Autonomous agents advance code generation by operationalizing test-driven development (TDD) within automated, closed-loop execution environments. Frameworks like AgentCoder establish a three-agent collaboration loop comprising a programmer agent, a test designer agent, and a test executor agent (Sapkota et al., 2025). The test designer agent generates functional unit tests from natural language specifications, the test executor agent runs these tests against code produced by the programmer agent, and runtime error logs are fed back to the programmer agent for iterative code refinement (Guan, 2025)

In user interface and web application development, agents automate user behavior simulation and dynamic front-end testing. Systems like AgentA/B parse web Document Object Model (DOM) trees and visual interfaces, using LLM agents to execute simulated user navigation workflows across alternative interface designs. This enables automated A/B testing, user path validation, and interface optimization prior to software deployment (Holter et al., 2026).

3.3 Program Repair, Fault Localization, and Automated Debugging

Automated Program Repair (APR) on repository-scale codebases represents a primary benchmark application for software engineering agents. Resolving bug reports requires navigating complex file trees, isolating root causes, generating valid code modifications, and ensuring changes do not break existing functionality (Puvvadi et al., 2025). The general automated program repair workflow follows a four-phase operational pipeline:

1. **Issue Reproduction**: The agent parses an ambiguous natural language issue report and writes a dedicated, standalone reproduction test case. The agent verifies the test fails on the unpatched codebase, establishing a clear baseline (FAIL_TO_PASS state) (Rombaut, 2026).
2. **Fault Localization**: The agent navigates the codebase to identify the precise files, classes, or function lines causing the failure. Strategies include AST traversals, interactive ReAct command execution, or hierarchical LLM file-ranking passes (Rafi et al., 2025).
3. **Patch Generation**: The agent extracts relevant code context surrounding the localized fault and generates candidate patches designed to fix the underlying defect.
4. **Patch Validation**: The agent executes the reproduction test case to confirm the bug is resolved. It then runs the project's broader regression test suite to ensure existing features remain unaffected (PASS_TO_PASS state) (Yang et al., 2026).

Fault localization remains a critical bottleneck in repository-scale repair. Frameworks address localization using distinct strategies: AutoCodeRover constructs targeted AST search queries to retrieve class and method signatures; MASAI uses an Edit Localizer sub-agent running interactive ReAct loops with concise representations, achieving a 75% localization pass rate on SWE-bench Lite; and Agentless and PANTHER use hierarchical prompt ranking to identify target files and line blocks without open-ended tool loops (Pan et al., 2025). Dedicated reproduction sub-agents, such as MASAI's Issue Reproducer, significantly boost downstream repair success by converting vague bug reports into concrete, runnable test cases (Ceka et al., 2025).

3.4 Software Maintenance and Legacy Code Migration

Software maintenance extends beyond single-bug repairs to include long-horizon software evolution, library version updates, and legacy code translation. Modern software systems evolve through continuous modifications spanning many files and project commits (Xiao et al., 2026).

A major challenge in AI-driven legacy code migration (such as translating legacy SAS or COBOL pipelines into modern Python or

PySpark) is the occurrence of silent semantic migration errors. In these instances, the generated code executes without throwing runtime errors or syntax exceptions, yet produces incorrect numeric or dataflow results (Kumari, 2026). Hybrid frameworks address this by combining LLM translation capabilities with formal Satisfiability Modulo Theories (SMT) and Z3 theorem solvers:

- **Constraint-Driven Adversarial Input Synthesis (CDAIS)**: Encodes divergence conditions between legacy and target language semantics into Z3 formal constraints. CDAIS synthesizes minimal adversarial test datasets that expose subtle translation bugs in a single trial, providing coverage certificates scoped to structural data shapes (Fazelnia et al., 2024).
- **Migration Invariant Synthesis (MIS)**: Evaluates candidate behavioral invariants against corpora of verified translation pairs, establishing empirically grounded specifications to validate generated code without requiring legacy runtime environments (Robillard et al., 2012).

Table 2: Functional Mapping of Autonomous Agent Paradigms Across Software Engineering Lifecycle Phases

Lifecycle Phase	Dominant Agent Mechanisms	Key Technical Challenges	Representative Frameworks	Primary Verification Metrics
Requirement & Architecture [cite: 1, 5, 17]	Multi-role collaborative discussions, SOP artifact generation, UML synthesis	Ambiguous user specifications, information loss across role handoffs	MetaGPT, ChatDev	Pass@1 Code Executability, PRD-to-Code Completeness
Implementation & Refinement [cite: 2, 3, 23]	Closed-loop programmer-tester feedback, dynamic web DOM interaction	Context window limits, code hallucination, state synchronization	AgentCoder, AgentA/B	Unit Test Pass Rate, Simulated User Task Completion
Fault Localization & Debugging [cite: 3, 7, 8, 22]	AST code search, ReAct sub-agent localizers, standalone issue reproduction	Search space explosion, vague bug descriptions, context clutter	AutoCodeRover, MASAI, Agentless, PANTHER	Localization Recall@K, FAIL_TO_PASS Test Conversion Rate
Software Evolution & Maintenance [cite: 3, 14]	Long-horizon multi-file planning, regression test harness execution	Context drift across turns, cumulative error propagation	OpenHands, SWE-agent, SWE-EVO Harness	Multi-file Patch Resolution Rate, PASS_TO_PASS Preservation
Legacy Code Migration [cite: 12]	SMT input synthesis, migration invariant checking, formal verification	Silent semantic migration errors, execution-passing incorrect dataflows	CDAIS + MIS (Z3 Solver integration)	Input Equivalence Rate, Invariant Oracle Pass Rate

4. Evaluation Methodologies, Benchmarks, and Data Integrity

4.1 Repository-Level Benchmarks

Evaluating software engineering agents has transitioned from short, function-level code completion benchmarks (such as HumanEval, MBPP, APPS, and ClassEval) to full, repository-scale software benchmarks (Guo et al., 2025).

Standard repository-scale benchmarks evaluate agents within executable development environments:

- **SWE-bench**: Mined from 12 popular open-source Python repositories, requiring agents to resolve actual GitHub issues by generating patches validated against project test suites (Zhang et al., 2026).
- **SWE-bench Lite & Verified**: Curated subsets designed to improve evaluation efficiency and quality. SWE-bench Lite focuses on 300 bug-fixing tasks. SWE-bench Verified contains 500 human-validated issues screened for problem clarity and test suite strength (Badertdinov et al., 2026).
- **SWE-EVO**: A benchmark designed to evaluate long-horizon software evolution. Built from release transitions in mature open-source repositories, SWE-EVO tasks require multi-step edits spanning an average of 20.9 files, 610.5 modified lines, and test suites averaging 874 tests per task instance (Lam et al., 2026).
- **Specialized Domain Benchmarks**: Emerging benchmark suites assess agent performance across multilingual codebases (Multi-SWE-bench, SWE-PolyBench), cloud infrastructure code (SWE-InfraBench), visual interface domains (Visual SWE-bench), and freelance software tasks (SWE-Lancer) (Hofman et al., 2026).

4.2 Empirical Performance Analysis Across Models and Frameworks

Empirical evaluations demonstrate clear performance differences across foundational models and operational execution harnesses. Leading reasoning models reach resolution rates above 70% on isolated bug-fixing benchmarks like SWE-bench Verified, but experience substantial performance drops when evaluated on long-horizon evolution tasks like SWE-EVO (Doumanas et al., 2026).

Table 3 presents comparative performance data for leading foundational models evaluated within standard agent harnesses (OpenHands and SWE-agent) across SWE-bench Verified and SWE-EVO.

Table 3: Empirical Performance Comparison Across Standard and Long-Horizon Software Benchmarks

Model Family	Underlying Foundation Model	SWE-bench Verified Pass Rate (%)	SWE-EVO Pass Rate (OpenHands Harness) (%)	SWE-EVO Pass Rate (SWE-agent Harness) (%)	SWE-EVO Patch Apply Rate (%)
OpenAI	GPT-5.4		25.00	25.00	97.92
OpenAI	GPT-5.2	72.80	18.75	22.92	97.92
OpenAI	GPT-5-08-07	65.00	18.75	20.83	100.00
OpenAI	GPT-4o (2024-11)	21.62	6.25	6.25	97.92
DeepSeek	DeepSeek-v3p2	70.00	20.83	23.40	95.83
DeepSeek	DeepSeek-R1	57.60	10.42	8.33	100.00
Zhipu AI	GLM-5	72.80	8.33	37.50	97.92
Moonshot AI	Kimi-k2p5	70.80	22.92	25.00	97.92
Qwen	Qwen3-Coder-480B	55.40	14.58	14.58	97.92

The performance drop observed when transitioning from SWE-bench Verified (e.g., 72.80% for GPT-5.2) to SWE-EVO (18.75%–22.92%) illustrates that current agent architectures struggle with sustained, multi-file

reasoning across long operational horizons (Le et al., 2025).

5. Emerging Insights, Synthesis, and Systemic Trade-offs

5.1 The Agent Coordination Dividend vs. Coordination Tax

The deployment of multi-agent collaboration topologies introduces a tradeoff between task decomposition gains and communication overhead. This balance can be analyzed through the *Agent Coordination Dividend* (Iqbal et al., 2022):

Net System Performance
= Dividend (Decomposability, Specialization, Context Isolation)
− Tax (Handoff Loss, State Desync, Token Inflation)

The Agent Coordination Dividend represents benefits gained from task decomposability, specialized role prompts, parallel trajectory exploration, and context isolation. Restricting sub-agents to specific context windows prevents context clutter and keeps input tokens focused on local sub-tasks (Riedl, 2026).

Conversely, the Agent Coordination Tax represents the systemic costs incurred by multi-agent coordination. Handoff loss occurs when natural language summaries drop critical technical details between sub-agents. State desynchronization occurs when sub-agents develop conflicting representations of the codebase (Cheng et al., 2026). In addition, correlated errors can cascade across communication paths, compounding initial mistakes. Controlled studies show that while multi-agent structures improve performance on highly parallelizable tasks, adding conversational agents to sequential reasoning workflows degrades performance by 39% to 70% while increasing token consumption up to 15-fold (Reid et al., 2025).

5.2 The Simplicity Imperative: Autonomous Tool-Driven Agents vs. Minimal Scaffolding

A key finding from recent evaluations is the competitive performance of minimalist, scaffolded frameworks compared to complex role-playing topologies.

Frameworks like Agentless demonstrate that complex multi-agent conversations, dynamic replanning, and open-ended tool environments are not strictly necessary for strong benchmark performance. By using a simple two-phase workflow (hierarchical file/function localization followed by direct patch generation), Agentless achieved a 27.33% pass rate on SWE-bench Lite at an average cost of $0.34 per issue, outperforming more complex open-source agent setups (Emde et al., 2026). Similarly, systems like Mini SWE-agent and PANTHER show that streamlined agent-computer interaction loops reduce operational failure surfaces. Architectural complexity should therefore be added incrementally, justified by measurable performance gains rather than structural novelties (Yang et al., 2024).

5.3 Long-Horizon Trajectories and the Generalization Gap

Empirical data highlights a significant generalization gap when agents move from short-horizon program repair to long-horizon software evolution. State-of-the-art models that achieve high pass rates on single-issue repairs experience performance drops on multi-commit benchmarks like SWE-EVO (Jiang et al., 2026).

This performance gap stems from compounding errors across extended execution trajectories. In long-horizon tasks spanning dozens of files and hundreds of code edits, early errors in fault localization or architectural assumptions propagate throughout subsequent steps. Without multi-file verification mechanisms, agents struggle to maintain structural consistency across long interaction histories (Sinha et al., 2026).

6. Critical Challenges and Future Research Directions

6.1 State Synchronization and Context Budget Optimization

As execution trajectories lengthen, agents face severe context window constraints and information clutter. Retaining complete terminal execution logs, raw file diffs, and long conversational turns degrades model reasoning efficiency.

Future research must focus on dynamic state synchronization and context compaction mechanisms. Key approaches include lazy code

representations that keep file context concise, structured trajectory ledgers, and evidence frameworks (such as TrajSpec) that prune raw interaction logs into structured repositories of task evidence (Tang et al., 2025).

6.2 Rigorous Verification and Formal Semantic Guarantees

Given the prevalence of weak test suites and plausible patches in standard benchmarks, relying solely on basic test execution is insufficient for reliable code synthesis. Agents require stronger verification mechanisms to guarantee functional correctness (Yu et al., 2026).

A key research direction is the integration of neuro-symbolic verification frameworks. Coupling LLM reasoning with automated theorem provers, Satisfiability Modulo Theories (SMT) solvers (such as Z3), static analysis engines, and dynamic test enhancement tools can provide formal semantic guarantees for generated patches and translated codebases (Beg et al., 2025).

7. Conclusion

This review has systematically examined the transformative role of LLM-based autonomous agents across the software engineering lifecycle, from automated requirements engineering to legacy code migration. The transition from passive, function-level code generation to active, repository-scale autonomous agents marks a fundamental paradigm shift in software automation. Modern agents integrate reasoning engines, memory architectures, tool interfaces, and verification controllers, enabling them to navigate complex codebases, execute commands, run test suites, and iteratively modify multi-file systems with minimal human intervention. Our analysis reveals several key insights. First, while multi-agent collaboration frameworks offer benefits through task decomposability, specialized role prompting, and context isolation, they incur significant coordination taxes including handoff loss, state desynchronization, and token inflation that can degrade performance by 39-70% in sequential tasks. Second, empirical evidence demonstrates that minimalist, scaffolded architectures such as Agentless and PANTHER achieve competitive benchmark performance at substantially lower operational costs, challenging assumptions about the necessity of complex agent topologies. Third, the significant performance gap between short-horizon program repair (70%+ on SWE-bench Verified) and long-horizon software evolution (18-25% on SWE-EVO) highlights fundamental limitations in current agents' ability to maintain structural consistency across extended execution trajectories. Critical challenges persist that require urgent research attention. Context window constraints and information clutter severely impact reasoning efficiency in long-horizon tasks, necessitating dynamic state synchronization and intelligent context compaction. The reliance on weak test suites for verification produces inflated success rates and cannot guarantee semantic correctness, demanding integration of neuro-symbolic verification frameworks such as SMT solvers and theorem provers. Additionally, the dual-use nature of autonomous agents presents security risks that must be addressed through governed deployment strategies. Looking forward, the evolution of software engineering agents will likely follow three trajectories: enhanced reasoning capabilities through advanced planning algorithms, robust verification through formal methods integration, and efficient coordination through optimized communication protocols. The field must develop more rigorous benchmarks that capture real-world software complexity, establish standardized evaluation protocols to ensure reproducibility, and create governance frameworks that balance automation benefits with security considerations. As LLM-based autonomous agents continue to evolve, they promise to fundamentally reshape software engineering practice, augmenting human developers' capabilities while raising important questions about autonomy, trust, and the future of software creation.

8. REFERENCES

Zhang, G., Geng, H., Yu, X., Yin, Z., Zhang, Z., Tan, Z., ... & Bai, L. (2025). The landscape of agentic reinforcement learning for llms: A survey. arXiv preprint arXiv:2509.02547.

Ning, X., Tieu, K., Fu, D., Wei, T., Li, Z., Bei, Y., ... & He, J. (2026). Code as agent harness. arXiv preprint arXiv:2605.18747.

Teo, R., Lee, A., & Lim, S. (2025). CODING REVOLUTION: HOW AI AGENTS ARE TAKING OVER SOFTWARE REPOSITORY MAINTENANCE. Journal of Computer Science Advancements, 3(6), 360-375.

Wang, Y., Guo, S., Pan, Y., Su, Z., Chen, F., Luan, T. H., ... & Niyato, D. (2025). Internet of agents: Fundamentals, applications, and challenges. IEEE Transactions on Cognitive Communications and Networking.

Joshi, S. (2025). A review of generative AI and DevOps pipelines: CI/CD, agentic automation, MLOps integration, and large language models. CD, Agentic Automation, MLOps Integration, and Large Language Models (June 01, 2025).

Xu, M., Fan, J., Huang, X., Zhou, C., Kang, J., Niyato, D., ... & Lam, K. Y. (2025). Forewarned is forearmed: A survey on large language model-based agents in autonomous cyberattacks. arXiv preprint arXiv:2505.12786.

He, J., Treude, C., & Lo, D. (2025). Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. ACM Transactions on Software Engineering and Methodology, 34(5), 1-30.

Ma, Z., Zhang, T., Liu, J., Zhang, W., Luo, M., Zhang, S., & Chen, K. (2026). Rethinking verification for llm code generation: From generation to testing. Advances in Neural Information Processing Systems, 38, 137549-137582.

Liu, J., Wang, K., Chen, Y., Peng, X., Chen, Z., Zhang, L., & Lou, Y. (2024). Large language model-based agents for software engineering: A survey. ACM Transactions on Software Engineering and Methodology.

Bruegge, B., & Dutoit, A. H. (2010). Object Oriented Software Engineering: Using UML Patterns and Java. Prentice Hall.

Cacciatori, E. (2008). Memory objects in project environments: Storing, retrieving and adapting learning in project-based firms. Research Policy, 37(9), 1591-1601.

Hu, Y., Liu, S., Yue, Y., Zhang, G., Liu, B., Zhu, F., ... & Yan, S. (2025). Memory in the age of ai agents. arXiv preprint arXiv:2512.13564.

Yang, J., Jimenez, C., 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, 50528-50652.

Shotts, W. (2026). The Linux command line: a complete introduction. No Starch Press.

Liu, Y., Liang, Z., Chen, Z., Chen, T., Hu, M., Dong, W., ... & Mu, Y. (2025). Hycodepolicy: Hybrid language controllers for multimodal monitoring and decision in embodied agents. arXiv preprint arXiv:2508.02629.

Lin, Y. C., Chen, K. C., Li, Z. Y., Wu, T. H., Wu, T. H., Chen, K. Y., ... & Chen, Y. N. (2025, November). Creativity in LLM-based multi-agent systems: A survey. In Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (pp. 27572-27595).

Tseng, Y. M., Huang, Y. C., Hsiao, T. Y., Chen, W. L., Huang, C. W., Meng, Y., & Chen, Y. N. (2024, November). Two tales of persona in llms: A survey of role-playing and personalization. In Findings of the Association for Computational Linguistics: EMNLP 2024 (pp. 16612-16631).

Rombaut, B. (2026). Inside the scaffold: A source-code taxonomy of coding agent architectures. arXiv preprint arXiv:2604.03515.

Martinez, M., & Franch, X. (2025). Dissecting the SWE-bench leaderboards: Profiling submitters and architectures of LLM-and agent-based repair systems. arXiv preprint arXiv:2506.17208.

Borghoff, U. M., Bottoni, P., & Pareschi, R. (2025). Beyond prompt chaining: The tb-cspn architecture for agentic ai. Future Internet, 17(8), 363.

Akbar, M. A., Khan, A. A., Hamza, M., Ghaffar, A., & Hajikhani, A. (2025). Agentic AI in software engineering: Practitioner perspectives across the software development life cycle. Software Engineering: Practitioner Perspectives Across the Software Development Life Cycle (September 16, 2025).

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., ... & Schmidhuber, J. (2024, May). MetaGPT: Meta programming for a multi-agent collaborative framework. In International Conference on Learning Representations (Vol. 2024, pp. 23247-23275).

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., ... & Sun, M. (2024, August). Chatdev: Communicative agents for software development. In Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers) (pp. 15174-15186).

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.

Holter, S., Koh, E., Dogan, M. D., & Chan, G. Y. Y. (2026). Uxcascade: Scalable usability testing with simulated user agents. arXiv preprint arXiv:2601.15777.

Puvvadi, M., Arava, S. K., Santoria, A., Chennupati, S. S. P., & Puvvadi, H. V. (2025, March). Coding agents: A comprehensive survey of automated bug fixing systems and benchmarks. In 2025 IEEE 14th International Conference on Communication Systems and Network Technologies (CSNT) (pp. 680-686). IEEE.

Rombaut, B. (2026). Inside the scaffold: A source-code taxonomy of coding agent architectures. arXiv preprint arXiv:2604.03515.

Rafi, M. N., Kim, D. J., Chen, T. H., & Wang, S. (2024). A multi-agent approach to fault localization via graph-based retrieval and reflexion. arXiv preprint arXiv:2409.13642.

Yang, Y., Li, C., Han, Z., Li, R., Xu, K., Li, Q., ... & Luo, B. (2026). Patch generation in APR: A survey from the perspectives of utilizing LLMs and using APR-specific information. ACM Transactions on Software Engineering and Methodology, 35(8), 1-62.

Pan, Y., Chen, Z., Lu, S., Chu, Z., Li, X., Li, H., ... & Ye, H. (2025). Prometheus: Towards long-horizon codebase navigation for repository-level problem solving. arXiv preprint arXiv:2507.19942.

Ceka, I., Mitchell, H., Pujar, S., Buratti, L., Ramji, S., Yang, J., ... & Ray, B. (2025). Understanding automated program repair agents through the lens of traceability: An empirical study. arXiv preprint arXiv:2506.08311.

Xiao, C., Jiao, Z., Wang, S., Wang, W., Zhao, B., Wei, H., ... & Qu, L. (2026). Socratic-SWE: Self-Evolving Coding Agents via Trace-Derived Agent Skills. arXiv preprint arXiv:2606.07412.

Robillard, M. P., Bodden, E., Kawrykow, D., Mezini, M., & Ratchford, T. (2012). Automated API property inference techniques. IEEE Transactions on Software Engineering, 39(5), 613-637.

Guo, J., Huang, S., Li, M., Huang, D., Chen, X., Zhang, R., ... & Lam, K. Y. (2025). A comprehensive survey on benchmarks and solutions in software engineering of llm-empowered agentic system. arXiv preprint arXiv:2510.09721.

Zhang, L., He, S., Zhang, C., Kang, Y., Li, B., Xie, C., ... & Zhang, D. (2026). Swe-bench goes live!. Advances in Neural Information Processing Systems, 38.

Badertdinov, I., Golubev, A., Nekrashevich, M., Shevtsov, A., Karasik, S., Andriushchenko, A., ... & Yangel, B. (2026). Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. Advances in Neural Information Processing Systems, 38.

Lam, M. H., Wang, C., Liu, H., Xiao, J., Li, H. S., Huang, J. T., ... & Lyu, M. R. (2026). SWE-Chain: Benchmarking Coding Agents on Chained Release-Level Package Upgrades. arXiv preprint arXiv:2605.14415.

Hofman, O., Brokman, J., Rachmil, O., Bose, S., Pahuja, V., Shimizu, T., ... & Vainshtein, R. (2026, March). Maps: A multilingual benchmark for agent performance and security. In Findings of the Association for Computational Linguistics: EACL 2026 (pp. 821-845).

Doumanas, D., Soularidis, A., & Kotis, K. (2026). Causal reasoning and large language models for military decision-making: Rethinking the command structures in the era of generative AI. AI, 7(1), 14.

Le, T., Thai, M. V., Manh, D. N., Nhat, H. P., & Bui, N. D. (2025). SWE-EVO: Benchmarking coding agents in long-horizon software evolution scenarios. arXiv preprint arXiv:2512.18470.

Iqbal, S., Costales, R., & Sha, F. (2022). Alma: Hierarchical learning for composite multi-agent tasks. Advances in neural information processing systems, 35, 7155-7166.

Cheng, G., Zhang, Y., Lin, Z., Zhao, X., Liu, S., Yin, J., & Deng, S. (2026). Is Collaboration Worth It? A Decision-Oriented Survey in Multi-Agent Systems.

Reid, A., O'Callaghan, S., Carroll, L., & Caetano, T. (2025). Risk analysis techniques for governed LLM-based multi-agent systems. arXiv preprint arXiv:2508.05687.

Emde, C., Rubinstein, A., Goel, A., Heakl, A., Yun, S., Oh, S. J., & Gubri, M. (2026, July). Maseval: Extending multi-agent evaluation from models to systems. In Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations) (pp. 345-356).

Yang, J., Jimenez, C., 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, 50528-50652.

Jiang, M., Fu, D., Shi, J., Zeng, J., Si, W., Li, K., ... & Liu, P. (2026). davinci-agency: Unlocking long-horizon agency data-efficiently. arXiv preprint arXiv:2602.02619.

Sinha, A., Arun, A., Goel, S., Staab, S., & Geiping, J. (2026, April). The illusion of diminishing returns: Measuring long horizon execution in llms. In International Conference on Learning Representations (Vol. 2026, pp. 35184-35211).

Tang, Q., Xiang, H., Yu, L., Yu, B., Lu, Y., Han, X., ... & Lin, J. (2025). Beyond turn limits: Training deep search agents with dynamic context window. arXiv preprint arXiv:2510.08276.

Yu, B., Cao, Y., Zhang, Y., Lin, L., Xu, J., Zhong, Z., ... & Briand, L. (2026). Swe-abs: Adversarial benchmark strengthening exposes inflated success rates on test-based benchmark. arXiv preprint arXiv:2603.00520.

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.

Izadi, M., Gismondi, R., & Gousios, G. (2022, May). Codefill: Multi-token code completion by jointly learning from structure and naming sequences. In Proceedings of the 44th international conference on software engineering (pp. 401-412).

Ghai, A. S., Rawat, V., Gupta, V. K., & Ghai, K. (2024, August). Artificial intelligence in system and software engineering for auto code generation. In 2024 International Conference on Electrical Electronics and Computing Technologies (ICEECT) (Vol. 1, pp. 1-5). IEEE.

Fazelnia, M., Mirakhorli, M., & Bagheri, H. (2024, October). Translation titans, reasoning challenges: Satisfiability-aided language models for detecting conflicting requirements. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (pp. 2294-2298).

Kwon, D. C., & Zhang, X. (2025, September). CP-AgentNet: Autonomous and explainable communication protocol design using generative agents. In 2025 IEEE 33rd International Conference on Network Protocols (ICNP) (pp. 1-12). IEEE.

Guan, T. (2025, December). A multi-agent coding assistant for cloud-native development: From requirements to deployable microservices. In Proceedings of the 2025 6th International Conference on Computer Science and Management Technology (pp. 1285-1290).

Karuppuchamy, S. (2026, February). Governed Agentic AI for Software Platforms: A Reference Architecture for Safe Autonomy at Scale. In SoutheastCon 2026 (pp. 1-6). IEEE.

Kumari, A. (2026, March). A Systematic Literature Survey on Agentic AI Frameworks for Legacy Software Modernization. In 2026 IEEE Madhya Pradesh Section Conference (MPCON) (pp. 1054-1059). IEEE.

Riedl, C. (2026, April). Emergent coordination in multi-agent language models. In International Conference on Learning Representations (Vol. 2026, pp. 120776-120799).