Dijlah Journal of Engineering Science (DJES) Vol. 3, No. 3, August, 2026, pp. 161-170
ISSN: Printed: **3078-9656, Online: 3078-9664**, paper ID: 194

Demystifying LLM-Based Software Engineering Agents: A Review of Capabilities, Benchmarks, and Failure Modes

Saif Safaa Shakir [1], Hasan Fadil Qasim [2], Rana M. Hasan [3]
[1] College of Computer Science and Information Technology, University of Al-Qadisiyah, Iraq,
[2] College of Agriculture , University of Misan, Alamara, Misan, Iraq
[3] College of Computer Science and Mathematics, University of Mosul, Mosul, Iraq

Article Info

Article history:

Received: April 19, 2026
Revised: May,30,2026
Accepted: Aug,.15,2026

Keywords:

LLM agents
Automated software engineering
SWE-bench
code generation
failure modes
multi-agent systems
program repair
benchmark evaluation

ABSTRACT

The combination of LLMs and autonomous agent architectures has led to assertive but exaggerated claims about how they can revolutionise the automation of software engineering. This review critically reviews the journey of LLM-based software engineering agents from 2015 to 2025, combining insights from verified more than 20 primary studies to investigate true capabilities, benchmark outputs and documented failures.

Our comparative studies are original, being done in a total of six architectural classes from a simple pipeline system to a complex multi-agent collaborative framework. We provide a quantitative comparison of resolution rate, cost, and failure taxonomy. An important conclusion is the Agentless Paradox: structurally simpler approaches match and outperform autonomous agents on the dominant SWE-bench benchmark, with resolution rates of leading systems reaching 75% on SWE-bench Verified (2025) versus <5% at the benchmark's 2023 launch. We categorize six modes of failures: hallucination, context overflow, test overfitting, over-engineering, inter-agent communication errors, and security boundary violations. We then critically assess how well the state-of-the-art architectural solutions address each of them. Further research is required in benchmarking ecological validity, adversarial robustness, and safety auditing. These gaps are priority issues.

Corresponding Author:

Hasan Fadil Qasim
College of Agriculture , University of Misan, Alamara, Misan, Iraq
Email: hasan.fadhil@uomisan.edu.iq

1. INTRODUCTION

Since quite some time software engineering is trying to automate software engineering. Tools across the ages and across the generations have attempted to "off-load" the cognitive binding that they impose on the programmer. The initial compiler from the FORTRAN company translated an assembly code but delegated it to an automated one. Next came the expert systems which would apply rules from the specific domain to synthesize code. Large language models have changed the different conditions under which this takes place. Unlike all the previous approaches, which depend on explicit symbolic rules, LLMs do not. LLMs are shown to learn programming knowledge implicitly from being shown billions of tokens of natural language and code, and apply that knowledge flexibly to novel problems [1].

Due to such flexibility, a new class of system has emerged: the autonomous software engineering agent. An agent like this may be given a natural language problem statement, search through a codebase of thousands of files, identify a bug or missing feature, devise a patch that fixes it, run tests to validate the patch and improve its patch iteratively without human intervention [2]. If agents of this type are reliable, they will have far-reaching practical consequences. For instance, they might speed up the software development cycle and reduce maintenance costs significantly.

Serious methodological questions remain despite the rapid development of these alleged capabilities. According to benchmark scores, synthetic or contrived tasks are simple problems. These problems do not happen in messy, ambiguous and contextual environments. In addition, the sophisticated nature of current agent architectures such as chains of tools calls, multi-agent debates, iterative self-refinement loops may obscure and perhaps not address the limitations of current LLMs [7]. This review adopts a critical perspective – we summarise true progress while inspecting weakness modes and limitations of methods, and the evidence for various architectural decisions which are typically adopted without justification.

The remainder of this paper is organized as follows. Section 2 presents the review methodology. Section 3 discusses the evolution of large language models in software engineering. Section 4 analyzes different agent architectures. Section 5 evaluates benchmark performance across various approaches. Section 6 introduces a taxonomy of common failure modes. Section 7 examines the agentless paradigm and its implications. Section 8 discusses open challenges and future research directions. Finally, Section 9 concludes the paper.

2. Methodology

This review follows a structured approach to identify, select, and analyze relevant literature on LLM-based software engineering agents published between 2015 and 2025. A total of more than 20 primary studies were selected based on their relevance, citation impact, and experimental contribution to the field.

The selection criteria included:

(i) empirical evaluation on established benchmarks such as SWE-bench,

(ii) clear architectural contributions (e.g., single-agent, multi-agent, or pipeline-based systems),

and (iii) availability of quantitative performance metrics.

The analysis focuses on three main dimensions:

(1) architectural design and system complexity,

(2) benchmark performance and evaluation methodologies,

and (3) failure modes and limitations.

Rather than conducting a purely descriptive survey, this review adopts a critical perspective, emphasizing comparative analysis, reproducibility concerns, and practical deployment implications.

2.1. A Decade of LLM Evolution for Software Engineering (2015–2025)

2.1.1. Pre-Transformer Code Models (2015–2019)

The majority of code intelligence large-scale research carried out during this duration was restricted to probabilistic n-gram language models and recurrent neural networks trained on token sequences. Raychev et al. [8] explain that statistical models operating on abstract syntax tree paths can predict variables naming and sequences of API calls. These predictions are sufficiently good for use in practical implementations of auto-completion. When it comes to certain semantically-complex tasks like cross-file reasoning and intent understanding, these systems manage fine tune for very specific narrow and syntactically regular patterns. An architectural design limitation was implicit memory lack across arbitrary code distances.

Pointer networks and attention models began to improve code completion. Nonetheless, their fragility (exact syntactic context; semantic ambiguity) led to these models being integrated mostly as IDE plug-ins rather than development assistants more generally. The technique that does not require programming for code generation may be more useful than programming it.

2.1.2. Transformer Models and Large-Scale Pre-Training (2020–2022)

It was already known that the original transformer architecture proposed in Vaswani et al. (2017) scaled very well, and experiments scaling the size and data of the model found that the quality of the code generator was more about the size than the architecture for the task. Researchers systematically investigate code language models after noticing the surprising capabilities of GPT-3. This language model showed some zero-shot and few-shot capability. Codex is the first example of code generation using an LLM that employed over 159 GB of Python code from public GitHub repositories as training data. The HumanEval benchmark introduced a standardized reproducible evaluation framework focused on functional correctness using unit tests, setting the first benchmark in the field.

2.1.3. Instruction Tuning and Reasoning Frameworks (2022–2023)

With instruction tuning and reinforcement learning from human feedback, or RLHF, large language models went from text-completion engines to interactive reasoning systems that can follow subtle instructions expressed in natural language. For software engineering tasks, this change was meaningful, as it allowed LLMs to receive natural language descriptions of problems and produce structured repair plans, rather than simply predicting the next token.
The ability to code solutions to complex tasks with many steps can be improved via prompt engineering [8]. The ReAct setup [10] extended this notion in interactive situations via interleaving natural language reasoning traces with environment actions that can be performed, which is the basis of all subsequent autonomous SE agents. The advancements mentioned earlier created the theoretical and technical frameworks for agent-based systems, which will be prevalent in 2023.

2.1.4. Autonomous Agents and End-to-End SE Pipelines (2023–2025)

In late 2023, the release of the SWE-bench [11] inspired renewed examination of autonomous software repair at the repository scale. Using real GitHub issues as benchmark problems exposes algorithms to the real challenges of legacy code and non-trivial test suites in multi-file repositories. This brings benchmarks of laboratory code-generation algorithms and actual software engineering closer together. The agent systems that were the first to be deployed on the SWE-bench solved under 5% of the tasks. By early 2025, the best methods are able to solve over 75% on the verified set [12]. We have seen a 15x gain in less than two years.
At about this time, appeared multi-agent collaborative frameworks like MetaGPT [13], ChatDev [14], which can generate software systems from high-level natural language specifications. The proposal of Agentless [7] indicates that an agentless (1) non-agentic and structured (e.g., sequential) pipeline could outperform a complicated architecture with autonomous agent(s). There is a tendency in our field towards over-engineering architecture.

2.2. Agent Architectures: Classification and Critical Analysis

Contemporary software engineering systems employing large language models (LLMs) can be grouped into six architectural categories, which reflect distinct beliefs on the role of autonomy, specialization, and structured interaction in effective software automation. Table 1 gives a comparative analysis of these categories.

Table 1. Comparative Classification of LLM-Based SE Agent Architectures

Architecture	Representative Systems	Autonomy Level	Strengths	Key Limitations
Single-Agent ReAct	SWE-agent [5]	High	Interface-aware tool use; adaptive exploration	Error accumulation over long trajectories; no specialization
Structured Pipeline (Agentless)	Agentless [7], SWE-Fixer	Low	Interpretable; cost-efficient; low hallucination	Rigid workflow; struggles with multi-file issues
Multi-Agent Collaborative	MetaGPT [13], ChatDev [14]	Medium–High	Role specialization; cross-validation; reduced hallucination	High token cost; coordination overhead; role conflicts
AST-Augmented Agent	AutoCodeRover [15]	Medium	Semantic code navigation; structured localization	Depends on parse quality; language-specific
MCTS-Enhanced Agent	SWE-Search [21]	High	Iterative tree search; adaptive feedback	Very high computation cost; slow convergence
Open Platform	OpenHands [6]	Configurable	Flexible; supports diverse configurations	Reproducibility challenges; security surface

2.2.1. Single-Agent ReAct Systems

The ReAct framework for software engineering was developed by the SWE-agent [5], Agent-Computer Interfaces (ACIs) based specialized command-line interfaces provide file viewing, line editing, and test execution. Yang et al. found that calculated design is a significant factor in performance variance, regardless of the model used. According to a significant discovery, an agent using ACI tools that resolves at 12.5% level on SWE-bench outperformed the same model having normal shell access. The need for interface engineering as a first class research issue rather than as an afterthought of implementation is suggested by this finding.

The design of the ReAct system that uses a single agent is inherently flawed because an early mis-localization decision in the course of a trajectory percolates through the repair and validation steps while lacking any structural means to detect failures or recovery from upstream.

As single-agent systems demonstrate substantial variance over multiple resolutions of the same problem, any single-run benchmark evaluation can be quite misleading. It is this gap that introduces fragility in architecture.

2.2.2. Structured Pipeline Approaches and the Agentless Paradigm

Agentless [7] proposed a structured three-stage pipeline hierarchical fault localization, multi-patch sampling with LLM-based ranking, and syntax filtering that intentionally avoids autonomous tool selection and planning. The work of Agentless, whose characterisation has been fashioned as a certain kind of agent, is not technical but diagnostic in nature. By exhibiting competitive performance without being an agent, it shows that the main source of value in many agents is structured interaction.

This discovery has direct consequences for evaluation methodology. When complex agent systems outperform Agentless by a small margin on SWE-bench, we must think hard whether this can be attributed to behaviour in the agent involving planning, tool choice and self-reflection. The cost difference multi-agent systems cost far more at Agentless you pay $0.34 of values is an open question. If the gain from the increased architectural complexity on offers value that commensurate with – cost?.

2.2.3. Multi-Agent Systems: Coordination Benefits and Costs

Multi-agent frameworks overcome single-agent limitations through role specialization and cross-agent validation. MetaGPT made the software engineering roles (product manager, architect, engineer, tester) into structured agent interactions through standardized documents. This process mitigates hallucination due to redundant verification. However, it also has the downside of coordination overhead. Moreover, errors can cascade downstream when agents generate flawed specifications that are implemented up to specification by subsequent agents.

When multiple agents are instantiated from one and the same underlying LLM, their outputs will not differ too much. Such problems must be taken into account in any critical analysis of multi-agent systems. When many agents make the same mistake, we call it a correlated error. We think that debate will correct these. But it actually reinforces them. Nobody is in opposition to the use of advanced models, using other base models and different prompting strategies could lead to different error correlations.

3. Benchmark Analysis and Critical Evaluation

3.1. The SWE-bench Ecosystem

The leading evaluation standard in the field is SWE-bench[11], making its design choices worthy of careful critical assessment. Because the benchmark is used with actual GitHub issues, it has ecological validity that artificial benchmarks lack. Yet, various methodological issues deserve note. To begin with, the original benchmark contains issues that cannot be solved through a static examination of the repository alone, requiring external knowledge (e.g., Kubernetes expertise), human clarification, or environmental configuration. SWE-bench Verified [12] relies on expert human annotation of solvability which provides a more reliable evaluation signal.

Second, the lack of explicit scientific explanation reduces its generalizability. The evaluation of Multi-SWE-bench [18] was made to six languages, it found that Python was not a reliable predictor of performance in statically-typed Java or Go as the failure modes are different. Another reason is that benchmark saturation is becoming a problem. Systems can get more than 75% on SWE-bench Verified. This means that the benchmark's discriminative power at the high end is diminishing. More challenging variants like SWE-bench Pro are being developed in response to the increasing efficiency.

3.2. Comparative Benchmark Properties

Table 2. Comparative Analysis of Major SE Benchmarks (2021–2025)

Benchmark	Year	Scale	Languages	Ecological Validity	Main Limitation	Ref.
HumanEval	2021	164 problems	Python only	Low (isolated functions)	No multi-file or repo context	[4]
MBPP	2021	374 problems	Python only	Low	Trivial complexity	—
AgentBench	2023	8 environments	Multiple	Medium	Dated; GPT-4 only in original	[19]
SWE-bench	2023	2,294 issues	Python only	High (real GitHub)	Annotation noise; solvability issues	[11]
SWE-bench Lite	2024	300 issues	Python only	High	Subset; may not represent full distribution	[11]
SWE-bench Verified	2024	500 issues	Python only	Very High	Still Python-only; labor-intensive annotation	[12]
Multi-SWE-bench	2025	1,632 issues	6 languages	High	New; limited baseline data	[18]
TheAgentCompany	2024	175 tasks	Multiple	Very High (professional tasks)	Small scale; specialized environment	[20]
SWE-bench Pro	2025	1,865 problems	123 languages	Very High	Evaluation infrastructure overhead	—

3.3. Metrics and Their Limitations

The field has settled on “resolution rate” (pass@1 on SWE-bench) as the main metric, hiding important distinctions. A patch gets marked as successful even if it resolves a test case but does not properly fix an associated bug, a phenomenon called test overfitting. Patches that introduce new bugs while resolving the original bug will get credit too. The industry seldom reports more costly metrics like semantic correctness, test coverage improvement, and non-regression, because calculating those at scale is expensive.

Cost per resolved issue has become an important secondary metric in practice. The measured range between systems from $0.34 (Agentless) to $10+ (some multi-agent configurations) differs by more than an order of magnitude for systems achieving similar resolution rates, a difference notably not highlighted in benchmark leaderboard presentations. Cost efficiency and accuracy should be treated as first-class metrics in future benchmark designs.

4. Quantitative Performance Comparison

Table 3 examines performance metrics of racepapers and benchmark leaderboards based on representative systems. The SWE-bench Verified reports resolution rates when available while systems tested before the Verified release use SWE-bench Lite as a proxy. All cost estimates are rounded and based on published or estimated per-issue costs.

Table 3. Quantitative Performance of Representative LLM-Based SE Systems

System	Base Model	SWE-bench Verified (%)	SWE-bench Lite (%)	Approx. Cost/Issue	Architecture Type	Year	Ref.
RAG Baseline	GPT-4	—	3.8%	< $0.10	Non-agentic retrieval	2023	[11]
SWE-agent v1	GPT-4	—	12.5%	~$1.50	Single-Agent ReAct	2024	[5]
AutoCodeRover	GPT-4o	—	19.0%	~$0.65	AST-augmented agent	2024	[15]
Agentless 1.0	GPT-4o	—	27.3%	$0.34	Structured pipeline	2024	[7]
Agentless 1.5	Claude 3.5 Sonnet	50.8%	40.7%	~$0.45	Structured pipeline	2024	[7]
CodeR	GPT-4o	—	28.3%	~$1.20	Multi-agent task graph	2024	[16]
SWE-search	GPT-4o	—	23.0%	~$4.00	MCTS-enhanced agent	2024	[21]
OpenHands+Claude	Claude 3.5	53.0%	—	~$1.80	Open platform agent	2025	[6]
SWE-bench Pro SOTA	Various	~75%+	—	Variable	Ensemble/advanced	2025	[12]

4.1. Analytical Observations from Performance Data

According to the recorded comparison in Table 3, there are three patterns. At the outset, it is clear that no positive relationship between the two exists. Agentless 1.0, one of the simplest systems of SWE-bench Lite, outperformed stronger SWE-search, which uses Monte Carlo tree search, by 4.3 percentage points – for about 12 times less cost. The idea that complex design can create performance is undermined by this.

Selecting the basis model plays a significant role in outcome In the Agentless framework, the transition from GPT-4o to Claude 3.5 Sonnet saw resolution rates improve from 27.3% to 40.7% (i.e., relative improvement of 49%). Hence, the improvement is a property of the model, not architecture. According to this finding, on the other hand, further improvement of the base models may reduce the comparative advantage of complex agent architectures over simple pipelines in practice.

The existing literature does not sufficiently report the third aspect, cost efficiency. The systems with similar resolution rates are priced between $0.34 and $4.00+ per resolved issue. There is a 10-fold variation in costs.

In manufacturing situations where automated repair is used on thousands of defects, this difference is economically decisive. When reporting on benchmarks, it will soon be important to consider not just accuracy numbers but also cost efficiency metrics to compare systems fairly.

5. Failure Mode Taxonomy and Critical Analysis

A principled taxonomy of failure modes is essential both for diagnosing current system limitations and for guiding architectural improvements. We find six main failure types, based on evidence from primary evaluation studies, ablation experiments, and the literature's trajectory analyses. This taxonomy is summarized in Table 4.

Table 4. Taxonomy of Failure Modes in LLM-Based SE Agents

Failure Mode	Description	Prevalence	Architectural Mitigation	Mitigation Effectiveness
Hallucination	Fabricated APIs, incorrect function signatures, non-existent modules	Very High	Multi-agent cross-validation; RAG grounding	Partial — correlated errors persist
Context Overflow	Attention degradation over long codebases; 'lost in the middle'	High	Hierarchical retrieval; AST-based navigation	Partial — deep cross-file dependencies unresolved
Test Overfitting	Patches pass test suite without fixing root cause	High	Human-verified benchmarks; semantic correctness checks	Incomplete — difficult to detect automatically
Over-Engineering	Unnecessary actions; architectural complexity for simple bugs	Medium	Structured pipelines; turn limits	Effective for known patterns
Inter-Agent Errors	Correlated mistakes in multi-agent systems; flawed spec propagation	Medium	Different model diversity; debate mechanisms	Limited if same base model
Security Boundary Violations	Unauthorized file/system access in sandboxed environments	Low–Medium	Containerization; permission controls	Effective with proper implementation

5.1. Hallucination in Code Generation

Hallucination is different in coding setting than in natural language generation. Instead of creating sentences that don't make sense, code hallucination makes code that can't function even though it looks like it could such as incorrect method signatures. According to Liu et al. [17] a significant amount of ChatGPT-generated code that passes simple unit tests all have undetected semantic bugs on edge cases. Similar evaluations on SWE-bench confirmed this finding where patches that pass automated tests are later rejected by humans.

The occurrence of apparent issues may be caught by a separate “critic” agent as a partial mitigation. There’s not enough independence if all agents are instantiation from the same base model with correlated output distributions. The technique of retrieval-augmented generation (RAG), which grounds code generation in retrieved relevant code snippets, provides a more principled approach to hallucination reduction, but it features its own failure modes if retrieval is poor.

5.2. Context Window Limitations

In reality, source code repositories contain tens of thousands of files and millions of lines of code, several orders of magnitude greater than the current LLM context windows. Even models that use 128K tokens and contexts face important limitations: (i) relevant context is usually spread between many files that don’t have an obvious linear ordering; (ii) the attention mechanism has non-uniform focus, where information in the middle of a long context receives disproportionately less processing; and (iii) overlapping full context windows perform worse on reasoning than sparser context windows.

Hierarchical retrieval strategies are a way of retrieving repositories, files, and then functions to zoom into relevant code by limiting the context progressively. Both Agentless and AutoCodeRover utilize variants of this mechanism. These strategies assume the relevant code can be identified using localization signals (eg, file names, function signatures, semantic similarity). This assumption breaks down for crosscutting concerns where relevant logic is spread across structurally different places.

5.3. Test Overfitting and Patch Validity

When test overfitting happens during automated program repair, the agent produces a patch that satisfies the test oracle but does not fix the underlying bug. This can happen in three different ways. First, the agent finds the

testing logic and creates code that satisfies that logic while not fixing the behavior being tested. Second, the patch hardcodes output values for test inputs but leaves the general case broken. Third, the patch modifies code near the test instead of the core problematic logic. All three methods generate patches that, while presenting as resolved according to benchmark metrics, actually fail to fulfill the intent of the issue.

The SWE-bench Verified [12] is one option, which adds human verification of issues solvable, and ground-truth patch quality, but cannot fix semantic overfitting that fools humans. Open problem: develop automated verification of semantic correctness possibly using formal verification tools or differential testing.

5.4. Over-Engineering Tendencies

Autonomous agents with access to a broader array of tools typically use more tools and take more actions than necessary to solve problems. Upon analyzing the SWE-agent trajectories, it is revealed that a large fraction of the turns are spent on exploratory actions with no new information, such as re-reading already-seen files, running tests that passed before, and creating new patches for resolved subproblems. According to a study conducted by Xia et al. [7], Agentless brings the overhead to the quantifiable level, whereby the resolution rates achieved by Agentless using a three-phase deterministic workflow are on par with Agentless due to the lack of any exploratory overhead.

6. Critical Discussion and Open Research Challenges

6.1. The Ecological Validity Problem

Currently prominent benchmarks evaluate agents using tasks that originate from open-source Python repositories that heavily represent LLM pre-training data. This introduces a potential concern of data leakage, whereby agents may rely on memorized code patterns instead of true generalization. Multi-SWE-bench [18] partially mitigates this issue by including six different languages. However, the issue of temporal leakage (where issues created before the cutoff date of the model can be considered “seen” during training) remains neglected in practice and seldom flagged as an issue in the published evaluations.

Real enterprise software engineering has additional complexities. These are missing from current benchmarks. We have closed-source codebases and proprietary frameworks. These also have security-sensitive code review requirements. Long-running refactoring tasks take place over multiple sessions. Lastly, we have to coordinate with human stakeholders. TheAgentCompany [20] offers advancement towards a more realistic professional evaluation, but its 175-task scale is insufficient for the statistical reliability of performance characterization.

6.2. Measurement and Reproducibility

The field is experiencing a moderate reproducibility crisis. Benchmark leaderboards showcase high-performing agent systems, most of them are closed-source, and their published results cannot be independently verified. Results from a single run with no confidence intervals due to temperature sampling no longer mean much as the resolution rate has a high run-to-run variance. Benchmark contamination instances in which agents are fine-tuned or prompted with benchmark-specific data is hard to detect and is rarely disclosed. All benchmark submissions should be required to disclose inference parameters, random seeds, and training data provenance according to the field.

6.3. Promising Research Directions

The following research directions are identified as high-priority based on current gap analysis:

Priority Research Directions for LLM-Based SE Agents

- Hybrid formal-LLM verification: combining LLM patch generation with model checking for correctness guarantees
- Adversarial benchmark design: issues crafted to expose specific failure modes rather than measuring average-case performance
- Uncertainty-aware agents: mechanisms for agents to express and act on confidence levels in their decisions
- Cross-session memory: persistent agent memory enabling multi-session refactoring tasks spanning thousands of files
- Security-first agent design: formal security policies integrated into agent architectures rather than applied post-hoc
- Multi-stakeholder evaluation: assessment of agent behavior in collaborative human-agent workflows

6.4. The Simplicity–Complexity Trade-off Revisited

This review, exploring how design complexity ought to inform production outcomes, is a recurring theme. In the field, there is an increase in complexity in agent architectures such as multi-agent debates, MCTS-guided exploration and hierarchical planning. However, the motivation for these designs tends to be theoretical rather than empirical and less benefit seeen. The Agentless paradigm [7] provides an important corrective, but its lesson should not be misinterpreted as a general argument against agentic strategies. The optimum degree of architectural complexity is probably task-dependent: complex agents can afford genuine advantages on long-horizon tasks requiring adaptive decision-making, while structured pipelines outperform for the well-defined localization-and-repair tasks that constitute most current benchmarks.

It is suggested that future work develops principled criteria to predict when agentic complexity is called for For example, research could use measures of task complexity (issue length, number of files relevant, degree of underspecification) and agent capability profiles (context utilization, tool selection accuracy, self-correction rate). Such an approach would facilitate evidence-based architectural choice, replacing the current norm of a default maximum complexity.

7. CONCLUSION

This review analyzes the evolution of LLM-based software engineering agents over the last ten years using critical examination, various primary empirical studies, benchmark studies, and architectural analysis. According to SWE-bench Verified, resolution rates of the field have gone from below 5% in 2023 to over 75% in early 2025, the fastest capability improvement across any established software engineering benchmark in the history of the field.

Our analysis reveals three main conclusions. The architectural complexity does not predict performance in any systematic way. The Agentless paradigm shows that structured, non-autonomous pipelines can match or exceed performance of complex agent systems at a far lower cost and with much better interpretability. The second main conclusion is that the quality of the base model is the main performance driver. That is, boosts to the underlying LLM explain most resolution rate gains through system generations. Another point to consider is that existing tests are underestimating the challenges in the real world such as ecological validity, temporal leakage and limiting language diversity. Therefore, the published performance figures are probably overestimating deployment readiness.

This review identifies six failure modes hallucination, context overflow, test overfitting, over-engineering, inter-agent correlated errors, and security boundary violations each of which calls for specific architectural and methodological responses. At present, no approach addresses sufficiently all six, moreover the interrelation between failure modes is poorly characterized. Advancements in these areas along with the use of more ecologically valid benchmarks and stricter reproducibility standards will be vital to make LLM-based software engineering move from research demonstration to production capability.

REFERENCES

[1] Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W.X., Wei, Z., & Wen, J. (2024). A survey on large language model based autonomous agents. Frontiers of Computer Science, 18(6), 186345. https://doi.org/10.1007/s11704-024-40231-1

[2] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. ACM Transactions on Software Engineering and Methodology, 33(8), 1–79. https://doi.org/10.1145/3695988

[3] [Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., & Amodei, D. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877–1901.

[4] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., & Brockman, G. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. DOI: 10.48550/arXiv.2107.03374

[5] Yang, J., Jimenez, C.E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. Advances in Neural Information Processing Systems (NeurIPS 2024). arXiv:2405.15793.

[6] Wang, X., Li, B., Song, Y., Xu, F.F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., & Singh, J. (2024). OpenHands: An open platform for AI software developers as generalist agents. arXiv preprint arXiv:2407.16741.

[7] Xia, C.S., Deng, Y., Dunn, S., & Zhang, L. (2024). Agentless: Demystifying LLM-based software engineering agents. arXiv preprint arXiv:2407.01489. DOI: 10.48550/arXiv.2407.01489

[8] Raychev, V., Vechev, M., & Yahav, E. (2014). Code completion with statistical language models. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014), 419–428. https://doi.org/10.1145/2594291.2594321

[9] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q.V., & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35, 24824–24837.

[10] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. International Conference on Learning Representations (ICLR 2023). arXiv:2210.03629.

[11] Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? International Conference on Learning Representations (ICLR 2024). arXiv:2310.06770.

[12] OpenAI. (2024). Introducing SWE-bench Verified. OpenAI Technical Blog. https://openai.com/index/introducing-swe-bench-verified/

[13] Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S.K.S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., & Schmidhuber, J. (2024). MetaGPT: Meta programming for a multi-agent collaborative framework. International Conference on Learning Representations (ICLR 2024). arXiv:2308.00352.

[14] Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z., & Sun, M. (2024). ChatDev: Communicative agents for software development. Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL 2024). arXiv:2307.07924.

[15] Zhang, Y., Ruan, H., Fan, Z., & Roychoudhury, A. (2024). AutoCodeRover: Autonomous program improvement. Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024), 1592–1604. https://doi.org/10.1145/3650212.3680384

[16] Chen, D., Lin, S., Zeng, M., Zan, D., Wang, J.G., Cheshkov, A., Sun, J., Yu, H., Dong, G., & Aliev, A. (2024). CodeR: Issue resolving with multi-agent and task graphs. arXiv preprint arXiv:2406.01304.

[17] Liu, J., Xia, C.S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. Conference on Neural Information Processing Systems (NeurIPS 2023). https://openreview.net/forum?id=1qvx610Cu7

[18] Zan, D., Deng, Y., Zhang, H., Shao, Y., Liao, W., Chen, X., Liu, Y., Chen, B., Zhang, B., Wang, P., et al. (2025). Multi-SWE-bench: A multilingual benchmark for issue resolving. arXiv preprint arXiv:2504.02605.

[19] Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., & Tang, J. (2023). AgentBench: Evaluating LLMs as agents. International Conference on Learning Representations (ICLR 2024). arXiv:2308.03688.

[20] Xu, F.F., Wang, Y., Tran, H.H., Pan, J., Li, X., Udeshi, K., Song, Y., Li, B., Xue, Y., Sun, Y., Tang, M., Yu, K., Bisk, Y., Dietz, L., Fried, D., Shi, P., Neubig, G., & Hockenmaier, J. (2024). TheAgentCompany: Benchmarking LLM agents on consequential real world tasks. arXiv preprint arXiv:2412.14161.

[21] Antoniades, A., Örwall, A., Zhang, K., Xie, Y., Goyal, A., & Wang, W. (2024). SWE-search: Enhancing software agents with Monte Carlo tree search and iterative refinement. arXiv preprint arXiv:2410.20285.

[22] Xia, C.S. & Zhang, L. (2023). Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023), 819–831. https://doi.org/10.1145/3597926.3598099