World Journal of Advanced Research and Reviews
eISSN: 2581-9615 CODEN (USA): WJARAI
Cross Ref DOI: 10.30574/wjarr
Journal homepage: https://wjarr.com/

(REVIEW ARTICLE)

Check for updates

Agentic workflows for end-to-end software engineering automation

Harsh Verma *

Palo Alto Networks, Artificial Intelligence, United States.

World Journal of Advanced Research and Reviews, 2025, 25(03), 2555-2574

Publication history: Received on 11 February 2025; revised on 26 March 2025; accepted on 29 March 2025

Article DOI: https://doi.org/10.30574/wjarr.2025.25.3.0887

Abstract

With the rise of large language models (LLMs) and independent AI systems, software engineering is undergoing a transformational shift. This conceptual paper theorizes agentic workflows systems, where an AI agent or agents proactively perceive, plan, act and reflect throughout the entire software development lifecycle (SDLC); its implications for end to end software engineering automation are also discussed. This paper constructs a theory for agentic SE systems viewed from four different angles: architectural configuration, reasoning capability, SDLC coverage, and evaluation validity, referring to 30 basic and contemporary papers from the past 25 years since 2000. It is then supported with quantitative evidence from benchmark studies: top agentic systems can now solve up to 43% of real-world GitHub issues on SWE-bench Lite, with 85.9% Pass@1 on Human-Eval and 55.8% less time spent on completing a developer task in controlled experiments. There are, however, significant theoretical tensions that are not resolved: autonomy and oversight, benchmark performance and validity in the real world, and the capability of the system and ethical responsibility. Finally, the paper outlines a research agenda that focuses on the specification level of agentic SE systems, on their self-verification, and on their governance.

Keywords: Agentic Workflows; Software Engineering Automation; Large Language Models; Multi-Agent Systems; Automated Program Repair; Code Generation; LLM Agents; SDLC Automation

1. Introduction

Software engineering is in crisis, not in the sense of overruns and failures (which are of course ongoing), but in the sense of a fundamental mismatch between the scales of systems the world needs to build and the minds of the people responsible for building them. As many as 50% of the working hours of developers are spent on error fixing and debugging instead of value creation (Zheng et al., 2023). As of 2023, the total value of software testing worldwide is over $40 billion and the industry is projected to grow at around 14% per year (Grand View Research, 2023), a testament to the vast amount of resources allocated to quality assurance alone. At the same time, the amount of open-source code, enterprise code base and the amount of software dependent infrastructure is growing at a rate greater than human engineering capacity to keep pace.

Amidst this, AI has become a compelling potential solution and is gaining traction. The turning point is the release of Codex (Chen et al., 2021), a large language model specifically fine-tuned on code which succeeds in able to produce syntactically and semantically correct code from natural language descriptions with meaningful reliability, for the first time allowing a general-purpose neural model to do so. By 2023, Copilot, a software designed for developers by GitHub, a service based on Codex, suggested AI code to more than 200,000 developers 420 million times a day on average, which, by then, was more than just a research experiment; it was an integral part of most engineers' day-to-day engineering workflows (GitHub, 2023).

* Corresponding author: Harsh Verma

However, while powerful code completion, it can only solve a part of the software engineering problem. Writing code is one among dozens of tasks that make up the entirety of SDLC: writing requirements, formalizing requirements, making designs, maintaining dependencies, creating tests, localizing defects, repairing defects, keeping documentation, performing code review, coordinating amongst teams, etc. While having to do either one of those things automated is useful, a system which could automate their integration, which could perceive a high level goal, decompose it into different lifecycle phases and generate and verify the artifacts at each phase and recover from any failures would be transformative.

The promise of agentic workflows for software engineering is that AI systems will not just respond to specific prompts or inputs but will serve as it's own agents, working independently and as part of a chain, and repeating and correcting along the entire software development lifecycle. MetaGPT (Hong et al., 2024), ChatDev (Qian et al., 2024), SWE-agent (Yang et al., 2024) and Agentless (Xia et al., 2024) are recent instantiations that show that multi-step, multi-agent AI pipelines can overcome genuine problems on GitHub, offer to write executable software from natural language specifications, and work towards end to end test driven development with minimal human input.

Even though agentic SE has seen a great deal of empirical development, its theory is still somewhat fragmented. Much of the previous work is more system specific, more attuned to benchmarking, and more empirical; there continue to be some fundamental questions: What are the properties that tell an excellent agentic SE system apart? How do the interactions between reasoning, tool-use and memory come into play and do they support or hinder end-to-end automation? So what's the theory limit for autonomous software engineering and where does the human have to be? What evaluation schemes are suitable for full-life cycle agentic system?

In this paper we make conceptual and theoretical contributions to such questions. It does not introduce any original experiments, but combines, reformulates and extends the current knowledge to create a coherent theory of agentic workflows for software engineering automation. The paper has four main contributions: First, a formal definition and taxonomy of agentic SE workflows is provided; second, a multi-dimensional theoretical framework is presented to analyze and witness agentic systems; third, quantitative evidence from the field is synthesized and used to substantiate and stress-test the framework; and fourth, theoretical tensions are identified and a research agenda is formulated.

2. Background and Conceptual Foundations

2.1. The Software Engineering Automation Continuum

Automation in software engineering has been around for some time. Eventually, in 1950s compilers did the work of converting human-readable language to machine instructions. In the 70's static analysis tools were invented and they automated the process of identifying defects. Over the 1990s and 2000's more and more of software engineering – starting in the 1990s with model driven engineering, and with the test generation frameworks and Continuous Integration pipelines – became automated. The current moment has three unique properties: (1) language understanding to comprehend NLS, (2) code generation with production-quality fidelity and (3) autonomous multi-step planning to orchestrate multi-phase workflows.

Myers-Benn and Rajinikanth (2013) present a sample of the vast literature around automated program repair, one of the earlier SE tasks to gain attention from AI, highlighting the search-based approaches (GenProg, Le Goues et al., 2012), machine-learning variants, and consistent theoretical conflicts concerning the correctness and generalizability of patches (Monperrus, 2018). Importantly, this is a longitudinal perspective: the moving of SE automation from search-based to learning-based to LLM-based is not just a technical progression, but a conceptual one as each stage comes with their own theoretical assumptions of what software correctness should be, what developers ought to be doing, and the scope of what automation can achieve.

2.2. Large Language Models as a Technological Substrate

The theoretical importance of LLMs in software engineering is the ability to use the LLM to observe both natural language and code, both of which represent the human intent and the machine's behavior, respectively, and which could be connected with each other through LLMs. They also found that scaling alone (with parameters in GPT-3 totaling 175 billion) could also lead to emergent capabilities like in-context learning where models can do new things in their immediate context with just a few examples, without updating their parameters. The implications were significant for software engineering: such a general language model, untuned for any specific task, would be able to reason about code tasks with the right prompting.

This possibility was turned into reality by Chen et al. (2021) who used Codex to achieve a performance of 28.8% Pass@1 on HumanEval, a score that was a bit low compared to modern day metrics, but proved that LLMs had the ability to produce functionally correct code from natural language descriptions at a non-trivial rate. The following developments have been explosive: On HumanEval, without any scaffolding, GPT-4 reached around 67% of the Pass@1 rate, and the agentic system created by MetaGPT and based on GPT-4 reached 85.9% with iterative feedback and multi-agent verification loops (Hong et al., 2024). To enable applications in code search, documentation, and defect detection, which are bidirectional code understanding tasks, Feng et al. (2020) expanded LLMs to support bidirectional code understanding with CodeBERT.

2.3. Agents from Language Models

A language model that generates code when prompted by a prompt is not in any meaningful theoretical sense, an agent. The agency that is evolved by Wooldridge and Jennings (1995) and later in AI requires autonomy, reactivity, proactivity and social ability. LLM-as-agent was built on two major advancements: first, AIs that can interact with the outside world, and second, AIs that can plan and iteratively perform multiple steps to achieve a goal.

The first of these innovations was formalized by Yao et al. (2023) as ReAct, a prompting framework that fuses thought (Thought), action (Act) and observation (Observation) into each generation loop while affecting the environment. In software engineering, it entails that an agent can think about a bug, write a patch, run the patch on a test suite, see the failure, reason about it, and repeat the cycle, similar to the way that skilled humans work. Wei et al. (2022) laid the theoretical foundation for the reasoning aspect of chain-of-thought prompting as it was shown to enhance performance in challenging tasks by up to 57% on multi-step arithmetic tasks whose underlying structure can be transferred to coding tasks in code debugging.

This was expanded upon by Shinn et al. (2023) with Reflexion, a verbal reinforcement learning (VRL) approach to which the agents generate natural language evaluations of their previous actions and leverage those natural language feedbacks to guide the actions they will take next. On coding tasks, Reflexion attained a 22% improvement compared to vanilla ReAct, underscoring the importance of self-reflection for iterative agent improvement, particularly when it involves tasks not amenable to gradient-based learning. Recently, Madaan et al. (2023) extended this to a general approach, named Self-Refine, that shows that iterative LLM self feedback and revision helps boost quality for a range of tasks on their own being several rounds of feedback and revision leading to a performance improvement by 5–15% on code generation tasks in controlled ablations.

2.4. The Multi-Agent Turn

Even very capable single-agent systems, however capable, have basic limitations when used to solve complex multi-phase software engineering tasks. Even with large models, the context window restricts an agent's ability to reason simultaneously over a large amount of information. Cognitive division of labor among people is hard to hardcode into one agent, because this is a characteristic of high-performing human engineering teams. Propagation of errors is more difficult to detect or correct if an independent review agent is not involved (and) error propagation across phases" (a bad requirement leading to a bad design leading to faulty code).

These theoretical observations led to the multi-agent SE turn. Building on this, Hong et al. (2024) proposed MetaGPT, which ent'abbed the coding habit standard procedures (SOPs) as constraints for inter-agent communication, dispatching different kinds of coding agents with specialized roles Product Manager, Architect, Engineer, QA Engineer to separate LLM instances. In ChatDev, agents communicate through a structured chat, proposed by Qian et al. (2024). To address the challenge of flexible composition of agent networks in multi-agent conversation, Wu et al. (2023) suggested AutoGen, an universal framework. In another world, Park et al. (2023) showed that generative agents with memory, reflection, and planning skills could exhibit emergent social behaviours, which has implications for more complex agent interaction in software development settings.

3. Theoretical Framework for Agentic SE Workflows

3.1. Formal Definition

An agentic software engineering workflow is a computational process with one or more AI agents equipped with language understanding, reasoning, planning, memory and tool-use capabilities: a goal oriented multi-step process that takes place in one or more phases of the software engineering lifecycle, and that creates verifiable software products, that can self-correct upon receiving feedback from the environment and/or feedback from other agents.

There are a number of important theoretical properties of this definition. First it is goal-directed: it is not just for a snapshot of input and output, but for the achievement of a desired effect (a running program, a fixed bug, a passing test suite). Third, involves multiple inputs and outputs: the system may need to plan and execute a series of steps, sending and receiving multiple pieces of information throughout the process. Third, it acts autonomously: No human intervention is necessary for any step of the system though the intervention of humans may be allowed at checkpoints. Finally, it is verifiable its work products (code, tests, documentation) can be assessed against objectively based criteria. Fifth, it's self-correcting, meaning the system can tell and react to failures in a way that is non-deterministic automation pipelines cannot.

3.2. The Four Analytical Dimensions

We suggest that agentic SE systems may be compared using four dimensions each representatively addressing a conceptually relevant facet of the system's design and its capability. These dimensions and their main elements are summarized in Table 1.

Table 1 Four analytical dimensions of agentic SE workflows with sub-components and representative instantiations.

S/N	Dimension	Sub-components	Key Design Questions	Representative Systems
1	Architectural Configuration	Agent count; role specialization; communication topology; memory architecture	How many agents? What roles? How do they coordinate?	MetaGPT, ChatDev, AutoGen, SWE-agent
2	Reasoning Capability	Planning depth; reflection mechanisms; tool use; decomposition strategy	How does the system plan, self-correct, and ground reasoning in action?	ReAct, Reflexion, AlphaCodium, Self-Refine
3	SDLC Coverage	Phases addressed; phase coherence; artifact handoff quality; end-to-end integration	Which lifecycle phases are covered and how well are outputs integrated?	MetaGPT (full), Agentless (repair), CodaMOSA (testing)
4	Evaluation Validity	Benchmark contamination; task simplicity; metric adequacy; ecological validity	Do evaluation metrics measure real-world capability? Is the benchmark contaminated?	SWE-bench, HumanEval, SWE-bench+, SWE-bench Pro

- Dimension 1: Architectural Configuration deals with the organization of agents, as single agents that run in a loop, multiple functionally specialized agents that run parallel and/or serially, and hierarchically organized networks of agents. The number of roles in a system - and therefore how specialized each role can be - as well as the ability to detect and correct errors is based on the architectural configuration of that system.
- Dimension 2: Reasoning Capability relates to the planning, task decomposition, hypothesis generation and self-correction of agents. This dimension includes reasoning systems used (chain of thought, ReAct, Reflexion, tree search), the range of iterative refinement allowed, as well as the extent to which agents are able to reason about their own uncertainty and limitations.
- Dimension 3: SDLC Coverage is focused on what phases of the software development lifecycle can the system support, what is the level of integration across the various phases. Earlier, automation of each phase was sufficient for full automation – also the outputs from the preceding phases must be valid inputs for the succeeding phases – we call this phase coherence.
- Dimension 4: Evaluation Validity relates to whether the measure/metrics and benchmarks satisfying the "capable of" claims are valid. This dimension is often overlooked in both theoretical and empirical research and yet it can be of far-reaching significance for the interpretation of the results.

3.3. Theoretical Propositions

For this conceptual position, four theoretical propositions are drawn up which serve to structure the following analysis:

- Proposition 1 (Architectural Complementarity): This specialization of agents' roles results in an architecture that is complementary to a single agent architecture for a complex, multi-phase SE task, and therefore, will be more effective.
- Proposition 2 (Reasoning Depth Effect): Systems which perform iterative self-reflection and multi-step planning will benefit from single-pass systems of generation, and the benefit will be greater for the more complex system.
- Phase Coherence is an essential but easily over-looked design property as discussed in Proposition 3 (Phase Coherence Imperative): The performance of an end to end SE task over the system will be limited by the poorest phase-to-phase handoff in the pipeline.
- Proposition 4 (Benchmark Validity Gap): The current code generation/repair benchmarks unintentionally cherry-picked and simplify real-world tasks and have defective evaluation metrics which overestimate the agentic SE performance from the perspective of data contamination.

4. Architectural Dimension: How Agentic SE Systems Are Organized

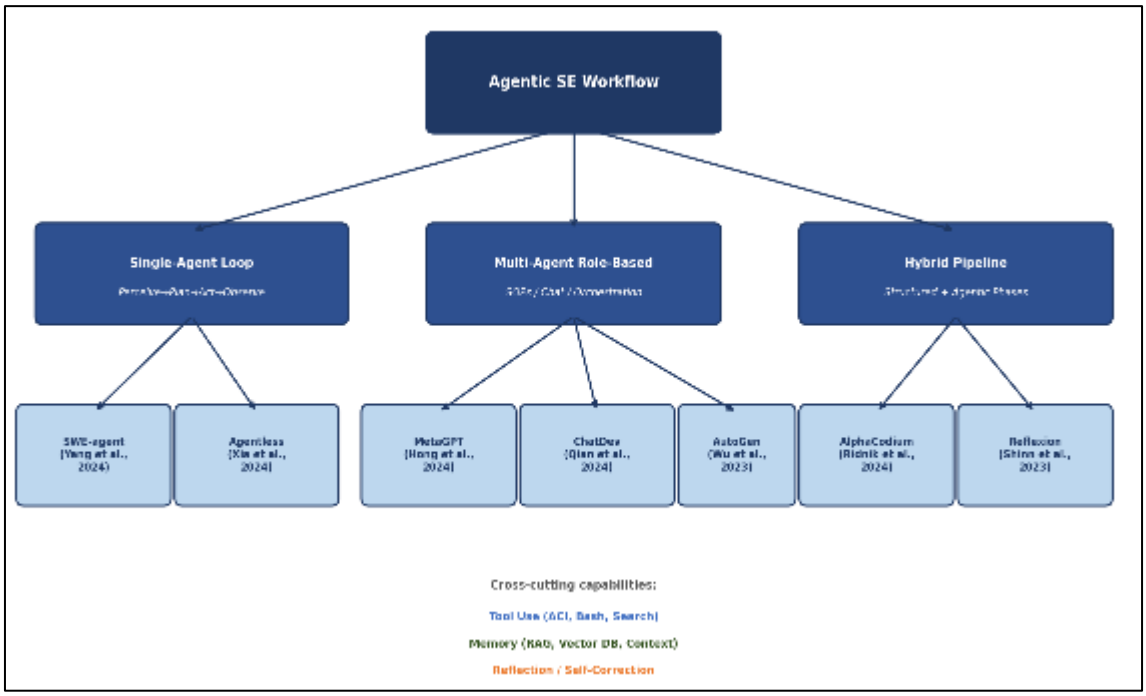

Figure 1 Taxonomy of Agentic SE Workflow Architectures.

4.1. Single-Agent Architectures

The most basic kind of agentic SE is the single-agent loop: one LLM instance is used to perceive a task description, create a plan, take actions (whether it's for code generation, test running, or web search), or observe the results, and then repeat the process. At its best such architecture can be realized in SWE-agent (Yang et al., 2024) where a single LLM is combined with a very well-designed Agent-Computer Interface (ACI), supporting file editing, code running, repository browsing, and access to Bash shell in a container environment. The theoretical impact of the ACI design is significant: The authors show that the quality of the interface between agent, and computing environment, is almost as critical as the actual capabilities of the underlying LLM, and that ACI improvements can lead to gains of +3.6% on SWE-bench compared to baseline agent configurations.

An equally significant theoretical contrast is also agentless (Xia et al., 2024), a very light-weight pipeline that avoids generating an autonomous agent by breaking down repository-level correction of bugs into fault localization, file-level correction, and patch selection. Agentless outperforms the other non-agentic methods on SWE-bench Lite, with 27.3% resolution, competitive with fully agentic approaches, but at much lower computational cost, indicating that task

decomposition/semantic pipelines provide a significant amount of utility of autonomous agency for well-defined repair tasks. But Agentless doesn't flatter itself regarding its capabilities - it obviously fails at tasks that involve creative exploration, multi-phase coherence and dynamic replanning in response to unanticipated environments.

4.2. Multi-Agent Role-Based Architectures

This way of operating a system, parallel distributing of cognitive tasks between specialized agents is called a multi-agent architecture. MetaGPT (Hong et al., 2024) is the most well-theorized multi-agent SE system, which encodes SOPs (Software Engineering SOPs) structured workflows that defines how Engineering roles interacts as constraints on inter-agents messages passing. The theoretical argument is that SOPs limit the possible set of behaviors the agents can perform, and make each agent concentrate on a different part of the problem space, helping to avoid the problem of passing unvalidated artifacts between phases.

Empirically, these consequences of using the MetaGPT approach are remarkable. MetaGPT gets nearly double the score (3.9/5) compared to general purpose agent frameworks (AutoGPT: 1.0/5, LangChain: 1.0/5, AgentVerse: 1.0/5) lacking SE specific SOPs (Hong et al., 2024) and it achieves highly executable software from natural language software specifications on SoftwareDev benchmark (ChatDev: 2.1/5). HumanEval ablation experiments found that the combination of executable feedback (in which the QA agents execute the generated code and send back the error messages) and the presence of feedback loops led to a 4.2 percentage point increase in Pass@1.

Instead of using SOPs as a vision of multi-agent, ChatDev (Qian et al., 2024) adopts another vision of multi-agent: the communicative dialogue between role-playing agents is used to co-construct software artifacts. The seven-agents system (CEO, CTO, CPO, Programmer, Reviewer, Tester, and Designer) of ChatDev simulates the development process experienced in an organization, featuring requirement analysis, design review, code implementation, and testing stages to develop software. The conceptual contribution of ChatDev lies in the proof that structured natural language dialogue, as opposed to the computation feedback, can be used for multi-agent SE to coordinate the agents.

To address these issues, Wu et al. (2023) suggested a general purpose multi-agent conversation framework, AutoGen, which allows for flexible agent network, where agents can have any specified role, and can communicate via flexible conversation patterns. Unlike MetaGPT or ChatDev, which encode SE-specific workflows, AutoGen offers architectural generality, namely, priDmitives that conversational agents, tool-use agents, and human-proxy agents can be composed from. This simplicity, however, is sacrificed for SE-specific optimizations that render MetaGPT and ChatDev's performance superior with structured SE benchmarks.

4.3. Memory and Context Management

The cross-context problem in agentic SE systems is a critical issue in architecture. Major projects (SE tasks) are routinely performed at a repository level, with code bases of millions of lines exceeding the current context window of even the largest of the LLMs. This presents a fundamental architectural tension, because information to carry out a task properly might not be contained within the context window of the model and thus you need to select which information to retrieve but they need to understand the task in order to select the information to retrieve.

To address this challenge, Lewis et al. (2020) introduced a principled method called retrieval-augmented generation or RAG which incorporates the retrieved chunks of documents (from an external knowledge store) into LLM generation. On-demand retrieval of relevant code files, documentation and historical issues are possible with the application of RAG to SE. Context management is supported with SWE-agent's ACI's file viewer that lets agents read only the parts of the files that are relevant, thus reducing overall file load times, and with Agentless, which limits file loading by using BM25 retrieval to search through files and determine which ones are most relevant before attempting repair.

The theoretical implication is that the architecture of memory is not only an engineering concern, but one of fundamental importance for the capability of agents: rich and well-structured architectures of memory support richer and more complex tasks, but at a fundamental level the richness and complexity of memory architectures introduce failure modes into agent systems. By the time the misretrieved file or the wrong dependency actually gets to the end of a multi-step pipeline, it can be hard to see and address.

4.4. Tool Use and Agent-Computer Interfaces

The interaction between an agent and its computational environment is an under-theorized but over empirical issue. An agent capable of producing only text is locked in a "closed loop of language" an agent capable of executing code, running test suites, accessing databases, and searching the Web and manipulating file systems is locked in a "open loop

of langauge" from which it can reason from the observable world. What Yang et al. (2024) show explicitly is that the design elements of the SWE-agent ACI namely, commands for searching and viewing code when navigating, operationally lifting edit commands to a per-line level of granularity, and a structured bash interface for interacting with the agent--improve performance more collectively than by switching between an underlying LLM used of similar competence.

In another setting, Wang et al. (2023) showed how to augment an embodied agent (named Voyager) in Minecraft with a progressive skill development, which was stored in an external code library and retrieved/composed for new tasks. Voyager's theoretical importance in the context of SE lies on its use of skill accumulation to help agents learn, build, store and retrieve verified code solutions to problems, thus forming a repository of reusable competencies, a mechanism that can be directly applied to an agentic SE system working on a sequence of development tasks.

5. Reasoning Dimension: Planning, Reflection, and Self-Correction

5.1. Chain-of-Thought Reasoning in Software Engineering

In fact, chain-of-thought (CoT) prompting, which advocates for prompting an LLM to generate intermediate reasoning steps to boost its performance on complex tasks, directly and impactfully applies to software engineering. Multi-step cognitive processes are intrinsic to code generation, debugging, and test synthesis: we do not go straight from a problem statement to a solution, but we reason about problem decomposition, choice of algorithms, identification of edge cases, and the approach to implementation. Explicitly prompting the model to generate this reasoning pathway in its output as opposed to an implicit capability demonstrated through the in-context model – was found to boost performance by up to 57% on complex reasoning tasks; see Wei et al., 2022, for more details.

Previously, CoT-based techniques in the SE context have been used in program synthesis (PS), in which a sequence of step-by-step algorithmic reasoning is performed prior to writing program code; fault localization (FL), where step-by-step algorithmic reasoning is used to create and consider hypotheses about potential causes of bugs before automatically creating patches; and test synthesis (TS), in which algorithmic reasoning about a program's expected behavior occurs first before test case generation. The theoretical justification is that an explicit form of reasoning was crafted that will now be examined for consistency and correctness by the model, before making a leap to a final artefact (in direct generation, this cognitive error-checking step was missing).

5.2. ReAct: Grounding Reasoning in Action

The major disadvantage of pure chain-of-thought is there is no way for it to anchor reasoning in real world, and it is confined to the model's context window. An LLM reasoning about a bug can hallucinate the results of a test or the value of a variable; for a ReAct agent to actually run the test or print the value of the variable. Yao et al. (2023) define this as the Thought-Action-Observation loop, where the agent produces a reasoning trace (Thought), performs a selected action (Act), and uses the outcome of the action (Observation) for the reasoning for the next action.

This architecture has far-reaching theory implications for SE. Software engineering is, more than anything else, an empirical field – its programs are not correct or incorrect in theory, but in practice. Not being able to run programs and see how they work makes reasoning about correctness severely restricted. The ReAct framework offers a principled way to make linguistic reasoning concrete in the domain of computation, availed by the generate-execute-debug cycle in both human expert debugging and the best current SE agents.

5.3. Reflection and Iterative Self-Improvement

Shinn et al. (2023) proposed that if LLMs are playing in competitive environments, they can perform better by learning from their own mistakes, without requiring gradient updates, by using a mechanism called "Reflexion. Following every incompletion of a task, a Reflexion agent generates a verbal reflection (analyses of the task's failure and success in natural language) and prepends it to its context (language memory) before its next attempt. As shown on coding tasks, Reflexion is able to take Pass@1 to 78.6% from 67.0% on HumanEval (17.3% improvement) and to 60.7% from 37.7% on more challenging LeetCode-style problems (61.0% improvement), showing that self-reflection is powerful and theoretically elegant mechanism for iterative improvement.

Beyond numbers is the performance of Reflexion as part of theory. Traditional machine learning optimizes model performance by gradient based update of model parameters on a labeled dataset, which demands a lot of data, computing, and offline training infrastructure. All of this is accomplished by in-context learning alone, by leveraging the

model's own output as a source of self-supervision. This advocates a theoretically way forward to finding agents that learn during deployment, adjusting to the codebase and development environment that they are dealing with.

Madaan et al. (2023) adapted this mechanism in a generalization, Self-Refine, using only multi-step iterative feedback, defined as each step containing a critique of the previous output and a new output reflecting it, increasing performance on various tasks. Results of code generation experiments demonstrated that Self-Refine obtained 5–15% higher pass rates than single-pass code generation, with larger improvements on more difficult tasks – in line with the theory that iterations will be most beneficial if the task is so difficult that it is unreliable to generate correctly on a single pass.

5.4. Planning and Task Decomposition

In complex tasks involving SE, not only do they need to consider the reasoning behind performing actions but they also need to do planning for long time horizons. A multi-phase task, that involves elicit requirements, design, write code, execute code, and interpret results to determine what went wrong, is not a single generate-execute-observe loop but will need decomposition of the task into subparts, sequencing of those parts, and dynamic replanning as results of subtask execution reveal that the initial plan is insufficient.

Ridnik et al. (2024) provide an approach to code generation in which they plan in a flow-wise manner: they have a problem reflection phase, a public test reasoning phase, a solution iteration phase, and a code generation phase with private test feedback. For competitive programming benchmarks, AlphaCoding3 performs 8% points it on average, showing that structured planning workflows continue to outperform (even with the same large model) single-shot generation. The theoretical significance goes to the planning structure, that is, the manner in which a problem is broken down and the order in which the parts of the problem are solved, as an independent contributing factor of the performance difference besides the model capability.

6. SDLC Coverage Dimension: Agentic Systems Across the Development Lifecycle

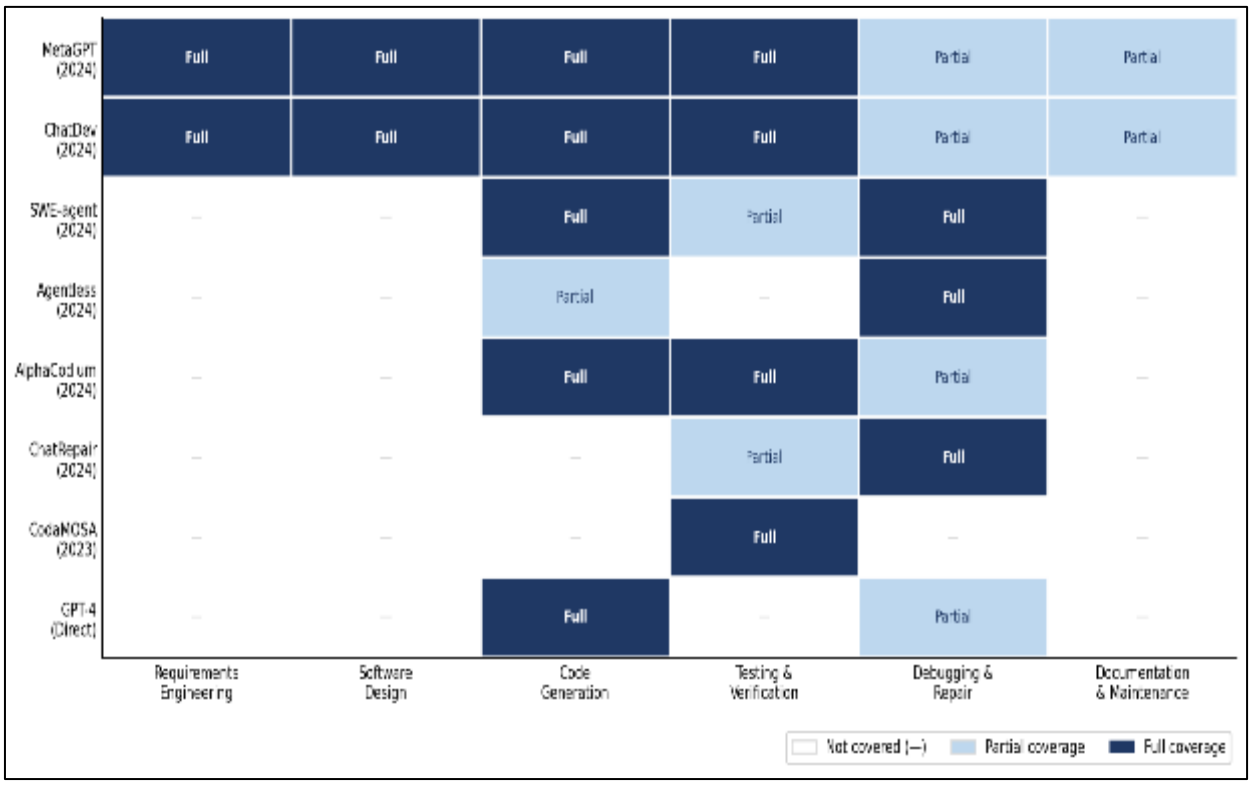

Figure 2 SDLC Phase Coverage Matrix for Major Agentic SE Systems. Full coverage indicates the system explicitly addresses the phase with verifiable output artifacts; Partial indicates limited or implicit support. MetaGPT and ChatDev achieve the broadest lifecycle coverage; most other systems are specialized to one or two phases

6.1. Requirements Engineering and Specification

Requirements engineering, that is, eliciting, formalizing and verifying stakeholder needs, is the least explored scope of SE automation with agents. That the requirements are theoretical is a real problem: there are requirements that sit between the human intention and the formal specification, and these are exactly the sort of requirements that are defined to be ambiguous, incomplete and inconsistentthe attributes that formal methods are supposed to remove. An agentic requirements engineering system has to traverse this boundary, defusing ambiguities through dialogue, finding incompleteness through formal analysis and finding inconsistencies in constraint satisfaction while at the same time being faithful to the human intention that drives the requirement.

The current agentic systems only provide implicit support for requirements engineering. With PRD (Product Requirements Document), MetaGPT's Product Manager agent converts natural language task descriptions into structured requirements documents, and ChatDev's CEO and CPO agents are structured to co-construct requirements using dialogue. These approaches show that LLMs can support the basic requirements formalisation for well-scoped software tasks; however, neither of them has yet tackedled the industrial requirements formalisation task fully, especially elicitation of latent requirements, addressing conflicting stakeholder requirements and formal verifying consistency of requirements.

6.2. Software Design and Architecture Generation

The structure of a system: how components are organized and what their interfaces are, and how responsibilities are assigned is highly impacted by the quality of the requirements and also highly impacts the quality of the resulting code. Agentic systems which derive architecture from requirements may be prone to passing ambiguities found in requirements on to structural flaws that are costly to amend "downstream.

Based on the PRD created by the Product Manager, MetaGPT's Architect agent will create system designs in the form of data structures, APIs and call flows. Structured output formats JSON encoded data flows and Python class hierarchies serve as contract between the architecture phase and the implementation phase and are the theoretical interest of this phase in MetaGPT. It is a kind of phase coherence: the Architect is not providing a prose description of the design; instead, the Architect is providing a machine-readable specification that restricts how the Engineer can implement the design.

6.3. Code Generation and Synthesis

The most well studied aspect of agentic SE is code generation, where it can be compared precisely quantitatively between systems with the help of the extensive benchmark literature. In Table 2, we report the performance of various representative systems on HumanEval, MBPP and SWE-bench.

Table 2 Benchmark performance comparison of representative agentic and non-agentic code generation systems.

System	Base Model	HumanEval Pass@1 (%)	MBPP Pass@1 (%)	SWE-bench Lite Resolved (%)	Year
Codex (baseline)	GPT-3 (175B)	28.8			2021
GPT-3.5 (direct)	GPT-3.5	48.1	~45	~0.5	2023
MetaGPT (GPT-3.5)	GPT-3.5	62.8	74.4		2024
GPT-4 (direct)	GPT-4	67.0	~52	~2.0	2023
AlphaCodium	GPT-4	~75.0 (+8 pp)			2024
MetaGPT (GPT-4)	GPT-4	85.9	87.7		2024
SWE-agent	Claude 3.5 Sonnet			~20.0	2024
Agentless	GPT-4o			27.3	2024
Top leaderboard (SWE-bench Verified)	Various			75.2	2025

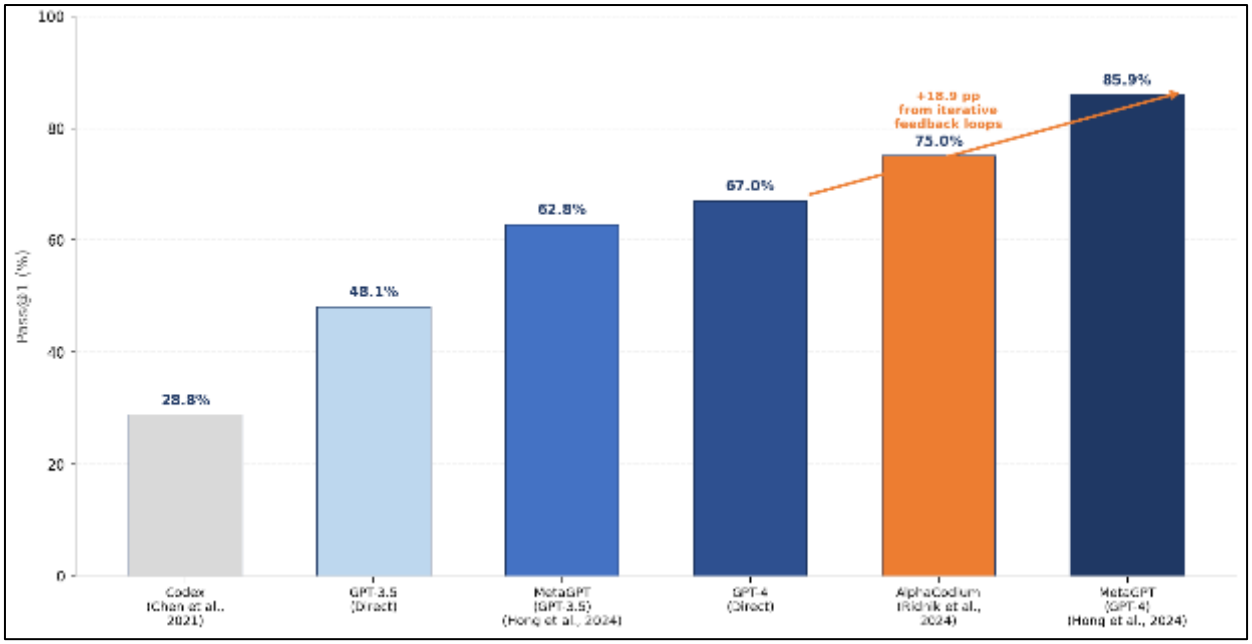

Figure 3 HumanEval Pass@1 Scores Across Agentic and Non-Agentic Systems (2021–2024). The orange annotation highlights the 18.9 percentage point gain attributable to iterative executable feedback loops in MetaGPT versus direct GPT-4 prompting. Data sourced from Chen et al. (2021), OpenAI (2023), Hong et al. (2024), and Ridnik et al. (2024).

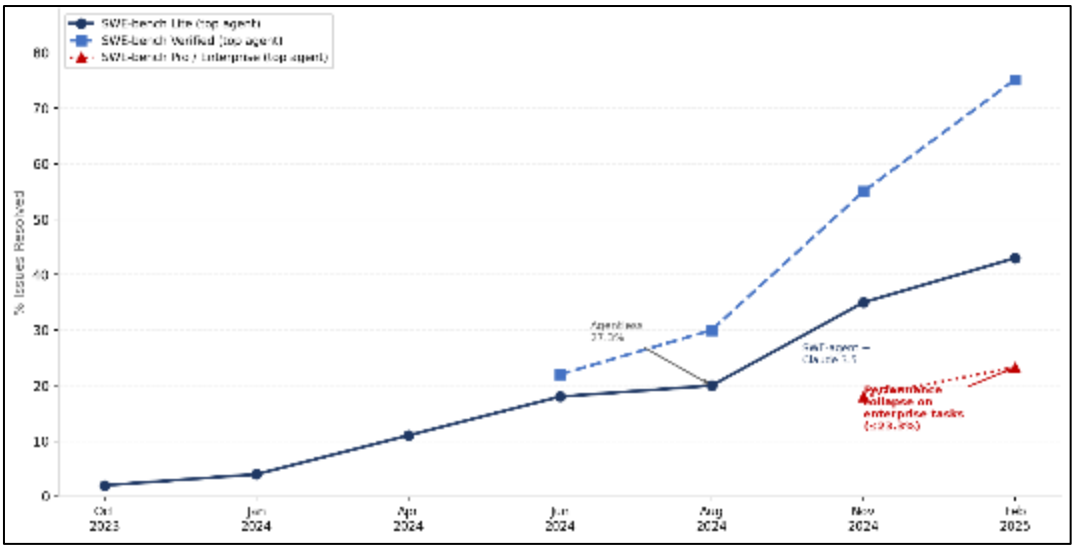

Figure 4 SWE-bench Resolution Rate Over Time (October 2023 – February 2025). Lines represent the top-performing system on each benchmark variant at each time point. The red line illustrates the enterprise performance gap: top agents reaching 75.2% on SWE-bench Verified collapse to ≤23.3% on SWE-bench Pro, directly supporting Proposition 4 (Benchmark Validity Gap).

The data of table 2 show the following patterns, all of theoretical interest. For one, agentic systems that are equipped with iterative feedback loops outperform single-pass generations by a long shot MetaGPT achieves 85.9% Pass@1 on HumanEval, surpassing GPT-4 by nearly 19 points. Secondly, the difference between single-pass and agentic is wider on harder benchmarks: the higher the complexity of the tasks (MBPP), the more the iteration adds. Thirdly, performance on the competitive programming benchmarks (CodeContests and APPS) is significantly worse than on the function-level benchmarks, corroborating that planning depth cannot only affect the quality of code generated, but also performance on complex tasks.

The trend of improvement on SWE-bench the most ecologically valid benchmark to date based on real GitHub issues in real repositories is remarkable. Approximately 2% of the issues were addressed during best system at the time of

benchmark launch (October 2023). By mid-2024, top systems achieved 20% on SWE-bench and 43% on SWE-bench Lite (Yang et al., 2024). In early 2025, the accuracy of the median entry on the SWE-bench (where a human expert verified a set of 500 tasks), is 46.9%, and the highest score is 75.2%. The highest median accuracy for entries in the leaderboard, across a human validated subset of 500 tasks (SWE-bench Verified), is achieved at 46.9% by early 2025. But in a contamination controlled benchmark running enterprise codebases (SWE-bench Pro) performance breaks down to less than 23.3% as stated in Proposition 4 (Benchmark Validity Gap).

6.4. Automated Testing and Verification

Agentic SE is in principle intrinsically centered on testing, which essentially generates external feedback as necessary for self-correcting iterations. If a code generation agent is unable to check the output with the test cases, it is in open loop because it cannot see if it generates correct or incorrect code. Because the two are combined in a single agentic pipeline, the combination of test generation and code generation amounts to a closed loop that, in theory, can proceed, generate code, generate tests, run tests, identify failures, modify code, and do it again.

By leveraging LLMs for test generation, Lemieux et al. (2023) showed that the LLM could break through the coverage plateaus traditional automated tools reached. CodaMOSA (LLM-based extension of coverage stagnated tools) improved the branch coverage relative to EvoSuite in 33 out of 50 benchmark modules, with an average increase of 11.6 percentage points in branch coverage. This is interesting, because it can be used to define some kind of semantic context for test generation that is not possible with coverage-based approaches: LLMs have some understanding of program semantics and may also be able to generate semantically meaningful edge cases that are structurally hard for random or evolutionary approaches.

LLM-made tests have distinguishing quality profile compared to a human-made test. Yuan et al. (2023) reported 47% to 56% test pass rates by ChatGPT-generated tests for various Java programs, showing that its test pass rates are higher for simple, well-documented functions and lower for functions with implicit behaviors that are harder. The failure modes are informational in that they tend to test for what should happen rather than what will happen, tests that pass with the buggy code and only fail after a fix; the opposite of test-driven development.

6.5. Automated Program Repair and Debugging

The automated program repair (APR) subdomain of SE has the longest history, with a deep theoretical and empirical foundation that can help understand the evolution of search-based to learning-based to LLM-driven program repair automation. Table 3 offers a comparison of sample APR systems against typical measures.

Table 3 Comparative analysis of automated program repair systems by approach, benchmark, and fix rate.

System	Approach	Benchmark	Bugs Fixed / Total	Fix Rate (%)	Year
GenProg	Search-based (genetic programming)	Defects4J	55 / 224	24.6	2012
CoCoNuT	Neural seq2seq ensemble	Defects4J	45 / 395	11.4	2020
DEAR	Pre-trained LM (CodeBERT)	Defects4J	56 / 395	14.2	2022
ChatRepair	LLM iterative (ChatGPT)	D4J + QuixBugs + HumanEval-J	334 / 581	57.5	2024
ContrastRepair	LLM + contrastive test pairs	D4J + QuixBugs + HumanEval-J	360 / 581	62.0	2024
ChatGPT (zero-shot)	LLM zero-shot prompting	EvalGPTFix (151 bugs)	109 / 151	72.2	2023
SWE-agent + GPT-4 (SWE-bench+)	Agentic (contamination-controlled)	SWE-bench+	~0.55% resolved	0.55	2024

Change of APR from Search to LLM isn't just quantitative, it's qualitative. Search-based systems (GenProg, Angelix) are based on syntactic transformations of the buggy programs, exploring a space of edits spanned by mutation operators.

Learning-based systems (CoCoNuT, DEAR) learn to convert buggy to fixed code traces from prior repair practices. LLM systems may be used to reason about the meaning of a program, to infer the intent of the programmer from documentation and commit messages, and to produce patches that need conceptual knowledge, instead of syntactic changes or transformations.

When handling 581 bugs from Defects4J, QuixBugs and HumanEval-Java, ConGres removed 360 (7.8%) bugs, outperforming the current state-of-the-art five test generation tools: including ChatRepair (Ye et al., 2024), which repaired 334 (7.1%). Perhaps most significantly, ContrastRepair also cut down on API calls by 20.91% versus ChatRepair, showing that more structured reasoning frameworks can enable more effective models to work more efficiently on a computational level – a key observation for deploying at scale.

As significant as the progress made by LLM-based APR, are the boundaries of such approaches. The SWE-Agent + GPT-4 is evaluated on a contamination-controlled subset of SWE-bench (called SWE-bench+), where only novel tasks (similar and equally difficult ones as the tasks that might be seen in the training data) are evaluated, Xin Jian Xia et al. (2024) shows dropping the resolution rate from ~4% to 0.55%, a reduction 86%. This finding directly confirms, through empirical data, Proposition 4, namely that the performance of current benchmark tasks is significantly higher than the capability of agentic APR systems on truly novel tasks.

6.6. Generation and maintenance of documentation

Documentation (the explanatory text that accompanies code and gives help to developer to understand, integrate and maintain software) is a stage of SDLC that is always in practice too less important (theoretically might be important for software sustainability). Khan and Uddin (2022) showed that with current LLMs, GPT-3 can generate meaningful API docs from source code and quality of the generated code achieved by GPT-3 is moderately in between 71% and the human written one, which made the APIdocs generation practically attainable as an agentic SE task.

Nam et al. (2024) performed the most thorough study of the use of LLM as an assistant in assisting code understanding, and showed that LLM-generated code explanations significantly boosted code understanding by developers on complex code segments scoring an average 23.6% higher in the comprehension test compared to simply reading the code. The interaction could go either way: LLMs can not only generate documentation but also act as interactive documentation systems, answering devs' questions on-demand through unfamiliar code, which partially solves the issue of documentation maintenance, and activities, by changing from static documentation to dynamic, query-based documentation supporting.

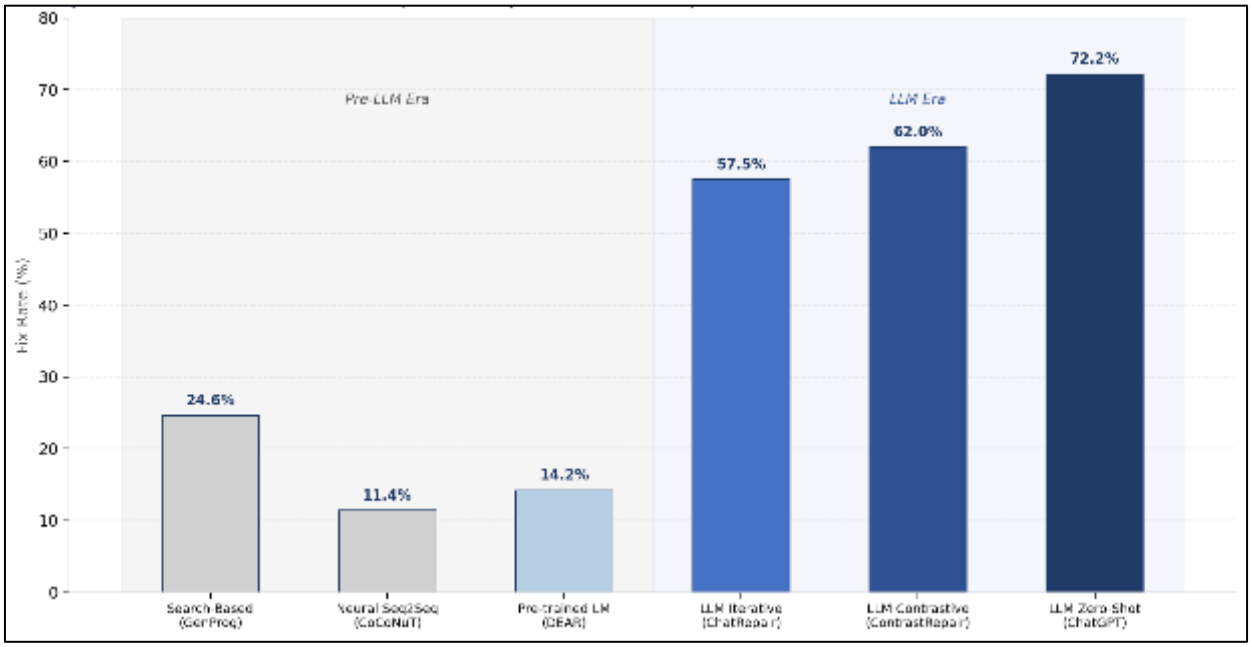

Figure 5 Automated Program Repair Fix Rates by Approach and Era. Grey shading marks the pre-LLM era (search-based and neural); blue shading marks the LLM era. Note that benchmark compositions differ across eras, making direct numerical comparison indicative rather than definitive. Data from Monperrus (2018), Lutellier et al. (2020), Li et al. (2022), and Ye et al. (2024).

7. Evaluation Dimension: Benchmarks, Measures, and Validity.

7.1. The Benchmark Landscape

The assessment of the correctness of an agentic SE system is complicated by the wide range of tasks the systems deal with, the also multi-step nature of their functioning, and the fact that correctness of complex software systems is not well-defined. While suites of benchmarks are forming the field and it is possible to compare between them, it is worth making explicit what is assumed in each benchmark as theoretical (hypothetical) definitions of SE capability. Table 4 contains a tabular comparison of various important SE benchmarks.

Table 4 Comparison of major software engineering benchmarks by task type, scale, metric, and key limitations.

Benchmark	Tasks (N)	Source	Primary Metric	Key Limitation	Contamination Risk
HumanEval	164	Hand-crafted (Chen et al., 2021)	Pass@k	Short, self-contained tasks; limited diversity	High widely reproduced online
MBPP	500	Crowd-sourced Python	Pass@1	Simpler tasks; limited to function-level	High
SWE-bench	2,294	Real GitHub issues (12 repos)	% Resolved	Complex evaluation; resource-intensive; public repos	Medium
SWE-bench Lite	300	Subset of SWE-bench	% Resolved	Smaller N; selection may not be representative	Medium
SWE-bench Verified	500	Human-validated subset	% Resolved	Reduces evaluation noise; still publicly available	Medium
SWE-bench Pro	N/A (private)	Enterprise codebases	% Resolved	Not publicly available; narrow accessibility	Low private
SWE-bench+	N/A	Contamination-controlled novel tasks	% Resolved	Small N; high construction cost	Low novel tasks only

HumanEval is a dataset of 164 well-designed unit test Python programs to solve problems where Pass@k represents the probability that a minimum of one out of k generated programs pass all unit tests. The limitations behind HumanEval are well-known: tasks are relatively short (median 7 lines of solution code), tasks are stand-alone with no external dependency, and tasks are easily encountered in LLM training data. Nguyen and Nadi (2022) determined that on HumanEval, GitHub Copilot's suggestions were accurate 57% of the time; markedly higher on easier tasks, but lower on more complex, contextually richer tasks, indicating performance of HumanEval is an overestimate of its real-world capability.

Theoretically, there are several important advances in benchmark design that SWE-bench (Jimenez et al., 2023) represents: it contains 2294 tasks taken from real-world git repositories and bug reports (repository issues and pull requests) on the internet written by real-world humans, in which systems must is asked to navigate the repositories and generate patch files that pass the repositories' own test suite. Such ecological validity, however, comes at a price: Evaluation takes the form of complex containerized execution of 12 Python repositories each having different dependency configurations running the full SWE-bench benchmark. Working sub-sets such as SWE-bench Verified (500 human-validated tasks) and SWE-bench Lite (300 tasks with clean test configurations) are also offered, offering lower amount of evaluations with good level of ecological validity and at the same time diminishing the effort of evaluation.

7.2. The Benchmark Validity Gap: Theoretical Analysis

Based on several lines of evidence, multiple lines of evidence support Proposition 4, which is a lack of systematic theoretical discussion of how current benchmarks systematically overestimate performance on agentic SE in the real world.

The most obvious validity threat is data contamination. LLMs are built on huge collections of text found online, such as GitHub repositories and Stack Overflow discussions, or even coding tutorial websites the very same sources one can find in many benchmarks. If a model during training has encountered the answers to the problems in HumanEval, at test time it is not exhibiting reasoning capability to generate the same answer; rather, memorization. This was made to be tested empirically by Liu et al., (2023): When problem description was changed to be the same meaning with different surface forms, ChatGPT's code correctness rate was significantly decreased in solving programming problems, which proved that surface-level memorization does play an important role in an LLM's benchmark performance.

Task simplification is another validity threat. Most frequently used human-eval, MBPP benchmarks are short, independent functions that can be comprehended and programmed on their own. Rarely do real software engineering problems have these qualities: built on a large body of code, dependent on other things, subject to a set of simultaneous constraints, and having a variety of changes that must be made in many files. The lack of performance improvement on the 39 of 49 agent skills on SWE-Skills-Bench, where the average gain on the successful skills is 1.2% (Zhang et al., 2024), indicates that agents trained or optimized on simpler benchmarks do not learn the skills needed to perform more challenging real-world tasks.

The third threat to validity is metric inadequacy. Pass@k does not measure the correctness of the code generatedthough it does measure whether it passes a given test suitewhether it satisfies edge conditions not designed by and tested against, maintainability, security, or the presumed correctness of the code as intended by its designer versus that expected by the given test suite. In fact, Barke et al. (2023) offer a theoretically grounded qualitative examination of how programmers actually interact with Copilot, finding patterns of interaction such as ‘acceleration’ (using suggestions directly) and ‘exploration' (taking suggestions as starting points) that appear to be missed by the pass-rate metrics, but are evidently important for understanding the value and risk profile of AI code generation in the real world.

7.3. Human-Centric Evaluation

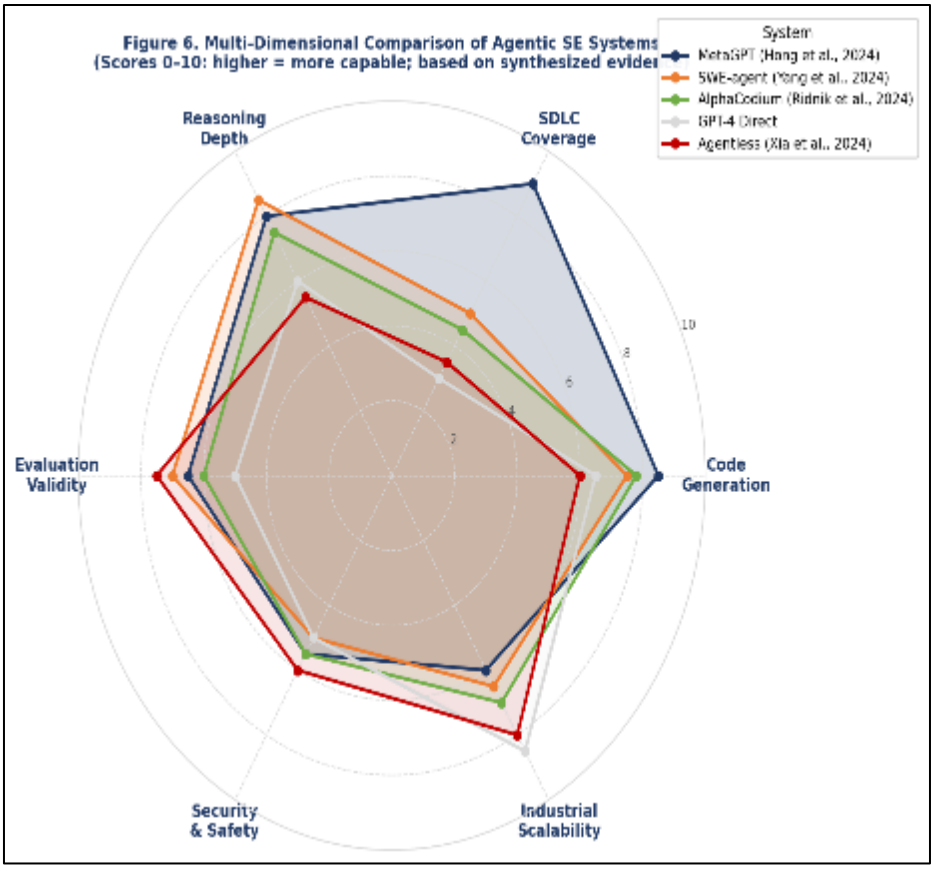

Figure 6 Multi-Dimensional Comparison of Major Agentic SE Systems (Scores 0–10). Dimensions represent synthesized capability ratings based on evidence from benchmark studies reviewed in this paper. MetaGPT leads on SDLC coverage and code generation; Agentless and GPT-4 Direct score higher on industrial scalability due to lower computational overhead.

The true evaluation of agentic SE systems in the environmental ecosystem is the measurement of how they impact human developer productivity, quality and experience in realistic development contexts. Peng et al. (2023) designed the most thorough study to date: a controlled experiment, in which 95 professional developers were randomly allocated to either use GitHub Copilot or not during the development of an HTTP server in JavaScript. Completion times for users of Copilot were 55.8% lower than the control group ($p = 0.0017$, 95% CI: [21%, 89%]), which is the strongest causal estimate for both with regard to the productivity effect of LLM-assisted code generation and the effect size.

A very important theoretical addition to the productivity studies is a qualitative exploration of developer expectations and experience with Copilot by Vaithilingam et al. (2022), which uncovers a discrepancy between the two. Developers were surprised to see Copilot sometimes produce contextually running but slightly inaccurate code for new and domain-specific use cases that necessitated checking the output and that often impacted the gains in time, versus boilerplate creation and syntax completion.

Aoryne acquired the information from a large-scale survey conducted by GitHub, involving 2000 developers, which revealed that 88 per cent of the respondents said that they felt more productive with Copilot however self-reported productivity is a theoretically problematic measure as it can include exogeneity outside of the productivity measure itself, such as personal effort, satisfaction with the tools used, and novelty effects. A more behavior-focused metric, the Microsoft-Accenture study 2023 finds 67% weekly usage for enterprise Copilot users, but again, this can't be used as the sole measure to determine valueit also looks to preference and habit as drivers of usage.

8. Challenges and Theoretical Tensions

8.1. The Autonomy-Oversight Tension

The two theory poles of agentic SE are autonomy and oversight. It is precisely the autonomy of these systems which gives them value: If the system has to have a human approve to make each step, then it offers little leverage over engineering done with a human. However, things will go wrong if there's no oversight, for example, an agent may autonomously implement a feature that is mistaken for a requirement, or autonomously add a security leak during bug resolution, which may be difficult to be detected or to be reverted and expensive to do so.

It is not just a practical or theoretical problem of how to have autonomous systems perform without oversight, it's a question that has to do with the mentalities involved when autonomous systems are required to act on their own. Formal verification the mathematical demonstration that a program is correct with respect to a program specification provides a theoretical answer: If the agent can ensure by mathematical means that its output is correct, then it does not need to control the output of the program. However, in realistic software domain, formal verification of the software is computation intractable even for very narrow domains, and its narrowness has to be verified by human beings. One partial solution that has emerged is the use of LLMs to produce proof sketches or formal specifications, which human engineers then review to complete the formalization and verification this is the nascent area of "AI-assisted formal verification.The nascent area of "AI-assisted formal verification," which produces proof sketches or formal specifications that human engineers can then edit to finish the formalization and verification process, represents a promising partial solution, though it does not truly resolve the basic tension.

8.2. Semantic Correctness and Hallucination

A known deficiency of even the most advanced language models is often referred to as LLM hallucination: the generation of fluent but factually incorrect content, and, which is especially relevant for the SE context, content that is not up to date with latest facts and figures. An LLM which produces a natural language text that hallucinate an historical fact, is an inconvenience; an LLM which produces code that hallucinate a library API, is a run-time error and an LLM which produces a security patch which hallucinate the non-existence of a class of vulnerabilities, is a security breach.

Liu et al. evaluated ChatGPT-generated code directly in terms of syntactic correctness, the rate of which is high, and semantic correctness, which is the rate of the code running correctly to solve the intended problems and of which is significantly lower, especially for the tasks that demand understanding of the program's semantics, concurrency, or security properties. The problem in theory is that LLMs are not learning the semantics of computation, but only the distribution of code text and they write code that looks similar to correct code without necessarily understanding what the code does.

Agentic architectures partially correct for hallucination by means of the execute-and-observe loop: an agent running its generated code and receiving a test failure will have observable evidence of wrongness, and be able to revise. In addition

to this, however, execution-based error detection fails to detect errors that result in output failures for inputs not included in the test suite, logic errors that result in slightly wrong output in the correct range, and errors due to security flaws that are triggered by adversarial inputs.

8.3 Security and Safety in Agentic SE (particularly for private providers)

Agentic SE systems that can interact with the real world through reading, writing and executing code can also be a source of security concerns that isn't evident with read-only LLM assistants. Theoretically, a crucial attack vector for agentswho may take text from malicious or otherwise dubious sources, as part of their action cycleis a type of manipulation known as prompt injection, which involves inserting harmful or otherwise questionable content into the text like comments in code, in commit messages, or elsewhere. In principle, an agent refactoring across the entire repository could be maliciously designed to redirect the agent, and thus be used to attack the codebase on which it is supposed to work.

In the context of security, Deng et al. (2023) examine the dual-functionality of agentic LLMs by analyzing PentestGPT, a penetration testing agent with a LLM to direct and complete security evaluations. The theoretical contribution is the fact that LLM agents can ingest and apply domain expertise (in this instance, offensive security knowledge) in ways that are directly applicable to the dual use context: the capabilities of PentestGPT that are useful for legitimate security testing can also be useful for unauthorized attacks using LLM agents.

8.3. Intellectual Property and Ethical Issues

There are potentially different theory and practice problems relative to the status of intellectual property of the code generated by LLM. LLMs are trained on code licensed under several open-source licenses, but more than enough legal and theoretical questions remain about what would constitute derivative work of code, similar in form or function to that used in training, and whether "churning" copyleft-licensed code repositories to train a model that produces similar code without attribution runs afoul of those licenses.

Theoretically, there is also a significant accountability deficit in agentic SE beyond IP. Human engineer makes a bug, engineer is responsible. If an agentic system causes harm due to a bug that it introduces, responsibility would be shared by the developer of the system, the supplier of the model, the entity to which the system was deployed, and the engineers who are users of the agent's result a distribution of responsibility that current legal and organizational structures have not yet adapted. One of the key open threats at the intersection of AI ethics and software engineering is the theoretical challenge of designing an adequate framework of accountability for SE using AI agents.

8.4. Scalability and industrial deployment

Agentic SE systems are expensive in terms of computation. For the frontier models, the cost of thousands of tokens for each iteration of code generation, code execution and code repair for a single SWE-bench task can range into the dollars. For an enterprise codebase where hundreds to thousands of people commit code every day and continuous integration pipelines run, the computational costs of fully agentic SE might be too high, assuming no efficiencies can be gained.

This cost challenge is not just economic, but theoretical, as it represents a basic trade-off between the level of reasoning that can be achieved by an agent and the amount of computing power they have available. One potential theoretical answer is agentless, another one has already been made (Xia et al., 2024) showing exactly that structured pipelines (without the agents) can also out-compete their fully agentic counterparts for well-specified tasks. The implication is that an optimum strategy for automation could be a hybrid one – an exploratory, open-ended scenario requiring depth of reasoning could be an agentic approach; a routine, well-specified scenario could be a pipeline approach.

9. Discussion

9.1. The Review of the Theoretical Propositions

With the evidence collected in this paper we may now turn to the four theoretical propositions presented in Section 3. Proposition 1 (Architectural Complementarity) is well supported by empirical evidence – the multi-agent SOP-constrained architecture of MetaGPT makes large jumps over single-agent and general purpose over structured SE benchmarks (3.9 vs. 1.0-2.1 on SoftwareDev executability). But, this does need some qualification: the difference between multi-agent architectures and otherwise is most evident for tasks that do indeed involve role-differentiated expertise or outputs that must be coherent with respect to the phases. Single agent architectures are competitive with respect to simple, self-contained coding tasks.

This claim has been backed up by multiple studies: The relative improvement of using Proposition 2 (Reasoning Depth Effect) over vanilla ReAct is 17.3% as reported by Reflexion, while AlphaCodium achieves an 8 point improvement using this over direct prompting and MetaGPT achieves 4.2 point improvement using the same approach as executable feedback. The corollary of the proposition, i.e., the harder the task, the greater the advantage, is confirmed by the trend of larger gains on more challenging benchmarks (LeetCode vs. HumanEval, SWE-bench vs. MBPP).

The MetaGPT architecture is theoretically supportive of the Prophetesis of Proposition 3 (Phase Coherence Imperative) which is not measured directly in the studies currently identified. None of the available metric measures the quality of any phase-to-phase handoffs in multi-phase agentic pipelines; there is only present focus on final products. This is something of a methodological weakness and something future research should deal with.

Proposition 4 (Benchmark Validity Gap) is the most substantiated on an empirical level by the following: A set of synthetic, contamination-controlled data in SWE-bench+, and that the floor of SWE-Agents' performance is in the enterprise benchmark SWE-bench Pro: 23.3%, as well as the known role of results driven by memorization in similar data for HumanEval.

9.2. Research Agenda for Agentic SE

From the theoretical and empirical findings presented in this paper, we suggest four areas that need to be studied within SE about agents.

First of all, specification inference and intent alignment. In agentic SE, the basic problem is that what the software should do (the requirements) are not always completely specified. Future research needs theoretical frameworks and practical mechanisms for agents to represent developer intentions, to clarify and verify the intentions, and to resolve the ambiguity in task requirements with structured dialogue, formal constraint satisfaction, and analogy with similar previous tasks.

Secondly, verification of artifacts created by AI models with AI. The more agentic SE systems are put in charge of the development pipeline, the more reliable verification of their deliverables will be similarly required. LLM agents should be able to formally verify themselves and each other, theoretically and practically, both inputs and outputs. It is, of course, a theoretical and practical possibility that there be a 'verification agent' that could be used to pinpoint defects in the output offered by a 'generation agent'.

Third, contamination-resistant evaluation. The validity gap for the benchmarks outlined in this paper cannot be remediated with the development of new benchmarks; any benchmark based on publicly accessible data is subject to contamination risks with growing LLMs and training data. Future research should aim to create contamination-resistant methodologies such as the dynamic construction of benchmarks (at evaluation time), assessment by trusted third parties at held-out benchmarks and also assessment on proprietary enterprise codebases.

Fourthly, autonomous SE governance frameworks. Agentic SE systems have governance missing because of accountability issues, IP issues, and security issues. Future work needs to include developing theoretical models of accountability distribution in human-agent/organization systems, as well as reassessing and reevaluating existing laws and regulations to fit the new challenges of liabilities related to AI-generated code and designing organizational governance structures that would allow for the benefits of the agentic SE automation while mitigating the risks.

9.3. Implications for Practice

This paper provides practitioners with some practical implications. Second, agentic SE tools need to be deployed in a staged fashion – implementing them to handle better defined, contained tasks (such as documentation generation, unit test generation, simple bug fix etc.) where the ability to monitor the outcomes is easy, then progressively extending their applicability to more complex, multi-phase activities where more elements can affect the end result. Secondly, evaluation metrics should be based on tasks and real-world application: try to avoid relying on HumanEval performance as a surrogate measure of the real world coding performance in this particular domain. Third, phase interfaces should be human-supervised: Individual phases may be made conformable but phase boundaries should not – the handoffs between phases need to be verified by human supervision to detect coherence failures which individual-phase evaluation will fail to find.

10. Conclusion

This paper has created a conceptual and theoretical relationship of agentic workflows when it comes to end to end automation in software engineering practice, one of the most important and fastest growing areas where artificial intelligence and software engineering occur. Based on 30 works written from a foundational perspective up until the present, it has introduced and proposed a formal definition of agentic SE workflows, a four-dimensional analytical framework and four theoretical propositions that organize the empirical evidence in the literature.

It is both encouraging and sobering. Despite being relatively new, agentic SE systems have made significant strides and shown impressive performance in various recent tasks, ranging from solving 43% of real GitHub issues on SWE-bench Lite to synthesizing executable software from natural language descriptions, whose executability is evaluated to be nearly twice that of previous methods; agentic SE systems have also demonstrated improved speed over human performance in controlled experiments (55.8%) and have singlehandedly fixed hundreds of bugs on standardized repair benchmarks. The working principles behind such successes iterative self-reflection, multi-agent specialization, use of tools to mediate reasoning, and structured planning workflows are clearly identified and can be used as a principled basis for further improvements.

However the difference between the ideal and the actual performance capability is very significant and of theoretical interest. At the same time, the 86% drop in performance on contamination-controlled benchmarks, the almost zero resolution rate on enterprise-grade long-horizon tasks, and the reported failure modeshallucination, phase coherence, and security consistently point towards the fact that current agentic SE systems are more of powerful assistants than autonomous engineers. True end-to-end software systems engineering automation does not exist; the path between the status quo and true automation involves specification inference, reliable self-verification of the implementation, contamination protection during its evaluation, and governance suitable for the demands of autonomous systems and accountable decision making.

Most importantly, perhaps, in this paper it is argued that the theoretical underpinnings of agentic SE should progress hand-in-hand with its engineering advances. There are currently insufficient theories of phase coherence, agent accountability, specification inference, and benchmark validity that could inform the design of better systems, as well as the evaluation of claims concerning agents' capabilities. These theoretical deficiencies will be critical to the responsible realisation of the transformative potential of agentic SE automation.

Compliance with ethical standards

Disclosure of conflict of interest

No conflict of interest to be disclosed.

References

[1] Barke, S., James, M. B., & Polikarpova, N. (2023). Grounded Copilot: How programmers interact with code-generating models. Proceedings of the ACM on Programming Languages, 7(OOPSLA1), Article 78. https://doi.org/10.1145/3586030

[2] Verma, H. (2019). Secure Real-Time Heterogeneous IoT Data Management System. Available at SSRN 6573879.

[3] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., … Amodei, D. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877–1901. https://arxiv.org/abs/2005.14165

[4] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). Evaluating large language models trained on code. arXiv. https://arxiv.org/abs/2107.03374

[5] Deng, G., Liu, Y., Mayoral-Vilches, V., Liu, P., Li, Y., Xu, Y., Zhang, T., Liu, Y., Pinzger, M., & Rass, S. (2023). PentestGPT: An LLM-empowered automatic penetration testing tool. arXiv.

[6] Islam, M. S., Verma, H., Khan, L., & Kantarcioglu, M. (2019, December). Secure real-time heterogeneous iot data management system. In 2019 first IEEE international conference on trust, privacy and security in intelligent systems and applications (TPS-ISA) (pp. 228-235). IEEE.

[7] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In Findings of the Association for Computational Linguistics: EMNLP 2020 (pp. 1536–1547). Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[8] 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. In The Twelfth International Conference on Learning Representations (ICLR 2024). https://arxiv.org/abs/2308.00352

[9] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Narasimhan, K., & Press, O. (2023). SWE-bench: Can language models resolve real-world GitHub issues? arXiv. https://arxiv.org/abs/2310.06770

[10] Khan, J. Y., & Uddin, G. (2022). Automatic code documentation generation using GPT-3. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022) (Article 1). ACM. https://doi.org/10.1145/3551349.3559548

[11] Lemieux, C., Inala, J. P., Lahiri, S. K., & Sen, S. (2023). CodaMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE 2023) (pp. 919–931). IEEE. https://doi.org/10.1109/ICSE48619.2023.00085

[12] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, 33, 9459–9474. https://arxiv.org/abs/2005.11401

[13] Li, Y., Wang, S., & Nguyen, T. N. (2022). DEAR: A novel deep learning-based approach for automated program repair. In Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE 2022) (pp. 511–523). IEEE. https://doi.org/10.1145/3510003.3510177

[14] 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. Advances in Neural Information Processing Systems, 36, 21558–21572. https://arxiv.org/abs/2305.01210

[15] Lutellier, T., Pham, H. V., Pang, L., Li, Y., Wei, M., & Tan, L. (2020). CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020) (pp. 101–114). ACM. https://doi.org/10.1145/3395363.3397369

[16] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., & Clark, P. (2023). Self-Refine: Iterative refinement with self-feedback. Advances in Neural Information Processing Systems, 36, 46534–46594. https://arxiv.org/abs/2303.17651

[17] Mashhadi, E., & Hemmati, H. (2021). Applying CodeBERT for automated program repair of Java simple bugs. In Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR 2021) (pp. 505–509). IEEE. https://doi.org/10.1109/MSR52588.2021.00063

[18] Monperrus, M. (2018). Automatic software repair: A bibliography. ACM Computing Surveys, 51(1), Article 17. https://doi.org/10.1145/3105906

[19] Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., & Myers, B. (2024). Using an LLM to help with code understanding. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024) (pp. 1–13). ACM. https://doi.org/10.1145/3597503.3639187

[20] Nguyen, N., & Nadi, S. (2022). An empirical evaluation of GitHub Copilot's code suggestions. In Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories (MSR 2022) (pp. 1–5). ACM. https://doi.org/10.1145/3524842.3528470

[21] Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. In Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST 2023) (pp. 1–22). ACM. https://doi.org/10.1145/3586183.3606763

[22] 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. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL 2024) (pp. 15174–15186). Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.acl-long.810

[23] Ridnik, T., Kredo, D., & Friedman, I. (2024). Code generation with AlphaCodium: From prompt engineering to flow engineering. arXiv. https://arxiv.org/abs/2401.08500

[24] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems, 36, 8634–8652. https://arxiv.org/abs/2303.11366

[25] Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In CHI Conference on Human Factors in Computing Systems Extended Abstracts (CHI EA 2022) (pp. 1–7). ACM. https://doi.org/10.1145/3491101.3519665

[26] Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., & Anandkumar, A. (2023). Voyager: An open-ended embodied agent with large language models. arXiv. https://arxiv.org/abs/2305.16291

[27] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35, 24824–24837. https://arxiv.org/abs/2201.11903

[28] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., & Wang, C. (2023). AutoGen: Enabling next-generation LLM applications via multi-agent conversation. arXiv. https://arxiv.org/abs/2308.08155

[29] Xia, C. S., Deng, Y., Dunn, S., & Zhang, L. (2024). Agentless: Demystifying LLM-based software engineering agents. arXiv. https://arxiv.org/abs/2407.01489

[30] 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. arXiv. https://arxiv.org/abs/2405.15793

[31] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. In The Eleventh International Conference on Learning Representations (ICLR 2023). https://arxiv.org/abs/2210.03629