Agentic Coding Systems for Automated Software Maintenance: A Multi-Step Reasoning and Tool-Use Framework

Xinyu Lu
Department of Computer Science and Engineering, University of Nevada, Reno, Reno, NV, USA.
xinyulu@unr.edu

Joel D. Benson
Department of Computer Science, University of New Hampshire, Durham, NH, USA.
contactjoel@unh.edu

Abstract

As software systems grow in scale and complexity, the burden of maintenance tasks such as bug fixing, refactoring, and feature integration has outpaced the capacity of manual human effort. This paper introduces Agentic Coding Systems, a novel framework that integrates large language models with multi-step reasoning and tool-use capabilities to automate software maintenance. Unlike conventional code generation tools that operate in a single-pass manner, our framework employs an iterative agent architecture where a language model core interacts with a suite of external tools through structured reasoning chains. These chains enable the agent to plan, execute, verify, and refine code changes in a closed-loop manner, mirroring the cognitive workflow of a human developer. The paper examines the architectural trade-offs between monolithic and modular agent designs, the governance challenges of delegating maintenance decisions to autonomous systems, and the infrastructure required for safe deployment at scale. We discuss sustainability considerations, including computational cost and energy consumption, as well as robustness and fairness concerns stemming from biased training data or uneven coverage of programming languages and domains. Policy implications are analyzed with respect to accountability, auditing, and the evolving role of human oversight. Through a synthesis of recent advances in language agent research, software engineering, and socio-technical systems, we argue that agentic coding systems represent a paradigmatic shift in how maintenance is conceptualized and executed, but their adoption must be guided by careful structural design and regulatory frameworks. The paper concludes with a forward-looking perspective on the future of automated software maintenance, emphasizing the need for interdisciplinary collaboration to realize the full potential of such systems while mitigating their risks.

Keywords

agentic coding, automated software maintenance, multi-step reasoning, tool-use framework, large language models, software engineering, artificial intelligence governance, socio-technical systems. 1 Introduction The discipline of software maintenance has long been characterized by high cognitive load, repetitive tasks, and a reliance on tacit knowledge accumulated over years of practice. As software systems permeate every sector of modern society, from healthcare to finance to transportation, the volume of code requiring ongoing attention has grown exponentially. Traditional approaches to maintenance, whether performed

by human developers or through rule-based static analysis tools, face fundamental scalability limitations. In response, the artificial intelligence community has turned to large language models as a means of automating code generation and repair. Early systems demonstrated impressive abilities to produce syntactically correct code fragments from natural language descriptions, yet they often failed in more complex maintenance scenarios that require understanding of system-wide invariants, multi-file dependencies, and long-range contextual reasoning [1]. This shortcoming has motivated the development of agentic systems—autonomous entities that combine language model cores with external tools and iterative reasoning processes to tackle tasks that demand planning, execution, and verification [2]. Agentic coding systems represent a departure from both traditional automated program repair tools and single-pass generative models. Instead of treating code modification as a one-shot translation problem, these systems adopt a multi-step reasoning framework reminiscent of the human problem-solving process: they decompose a maintenance request into subgoals, search for relevant information, generate candidate changes, test them against the existing codebase, and refine their outputs in light of feedback [3]. This paper provides a comprehensive analysis of such frameworks, focusing on their architecture, the mechanisms that enable multi-step reasoning and tool use, and the broader systemic implications of deploying them in real-world software engineering environments. Our goal is not to present a specific implementation but rather to examine the structural trade-offs and governance challenges that arise when autonomous agents take on maintenance responsibilities. The remainder of the paper is organized as follows. Section 2 situates agentic coding within the broader landscape of AI-assisted software engineering, reviewing relevant work on large language models, program repair, and task-oriented agents. Section 3 describes the core architectural components of an agentic coding system, including the reasoning engine, tool invocation layer, and memory management module. Section 4 delves into the mechanisms of multi-step reasoning and tool use, illustrating how iterative refinement and external feedback loops enable the system to handle complex maintenance tasks. Section 5 discusses structural trade-offs, such as the balance between autonomy and human oversight, and the design space for system governance. Section 6 examines deployment, sustainability, and robustness, including computational costs, energy footprints, and failure modes. Section 7 addresses fairness and policy implications, considering bias in training data, unequal support for programming languages, and the need for accountability frameworks. Section 8 concludes with a synthesis and a call for interdisciplinary research. 2 Background and Related Work The field of automated software maintenance has evolved through several distinct phases. Early work relied on rule-based static analysis and pattern matching to detect and fix common bugs, but these tools were limited to predefined fault types and lacked the flexibility to handle novel scenarios [4]. The advent of machine learning, particularly deep learning, enabled data-driven approaches to code completion and defect prediction. Neural models trained on large corpora of open-source repositories demonstrated an ability to generate short code snippets with high accuracy, but they struggled with tasks requiring cross-file context or deep program semantics [5]. Large language models, such as GPT-3 and its successors, brought a new level of fluency and contextual understanding, leading to the rise of code generation assistants like GitHub Copilot and Amazon CodeWhisperer. However, these tools typically operate in a stateless, single-turn fashion, generating code based on the immediate prompt without iterative refinement or external validation [6]. Recognizing these limitations, researchers began to explore agent-based architectures that equip language models with additional capabilities. The concept of an "agent" in AI traditionally refers to a system that perceives its environment, decides on actions, and executes them to achieve goals. In the context of coding, an agentic system might

include a language model core, a memory store, and a set of tools such as a code interpreter, a debugger, a version control interface, and a search engine [7]. One influential line of work introduced a framework for language agents that can reflect on their own outputs and learn from mistakes, a process referred to as "reflexion" [8]. This approach demonstrated that iterative self-evaluation and verbal reinforcement could significantly improve performance on complex reasoning tasks, including code generation and bug fixing. Another strand of research focused on decomposing complex problem-solving into multiple steps using chain-of-thought prompting and tree-search strategies [9]. These techniques allow the agent to explore alternative solution paths, backtrack when necessary, and select the most promising candidates based on intermediate rewards. When combined with tool use, the agent can not only reason about code but also execute it, observe runtime results, and incorporate those observations into subsequent reasoning steps [10]. More recent frameworks have integrated planning modules that generate high-level action sequences before executing any tool calls, thereby reducing the likelihood of cascading errors [11]. Despite these advances, most existing agentic coding systems remain experimental and have not been systematically evaluated in production settings. Moreover, the literature has largely focused on algorithmic improvements rather than the systemic implications of deploying such agents at scale. This paper aims to fill that gap by providing a holistic analysis that encompasses architectural design, operational concerns, and socio-technical governance. 3 Framework Architecture for Agentic Coding Systems An agentic coding system for automated software maintenance can be decomposed into three primary layers: the reasoning engine, the tool invocation layer, and the memory and state management module. The reasoning engine is typically a large language model that serves as the core decision-making component. It is responsible for interpreting maintenance requests, generating plans, and producing code modifications. However, unlike a standalone language model, the reasoning engine in an agentic system does not operate in isolation. It communicates with the tool invocation layer, which provides access to external resources such as compilers, interpreters, debuggers, static analyzers, package managers, and version control systems. The agent can invoke these tools as functions, passing arguments and receiving results in a structured format. This design allows the reasoning engine to offload tasks that require precise computation or external data, such as checking syntax, running tests, or retrieving documentation [12]. The memory and state management module maintains context across multiple interactions. It stores the history of previous reasoning steps, tool outputs, and code changes, enabling the agent to revisit earlier decisions and adapt its strategy. Short-term memory is used to keep the immediate conversation context within the token limit of the language model, while long-term memory can persist across sessions through external databases or vector stores. This architectural separation is crucial for handling large codebases where the relevant context may span many files and be updated over time. The communication between these layers is governed by a protocol that defines how the reasoning engine formulates tool requests and how results are fed back into the reasoning loop. A common approach is to use a structured format such as JSON or a domain-specific language that the language model has been fine-tuned to generate and parse. The agent's reasoning process typically follows a cycle: observe, plan, act, and observe again. In the observe phase, the agent gathers information from its environment, including the current state of the codebase, previous test results, and any error messages. In the plan phase, it decomposes the maintenance goal into a sequence of subgoals, each of which may involve a tool call. In the act phase, it invokes the relevant tool and receives a response. The cycle then repeats, with the agent using the new information to adjust its plan or generate the next action [13]. One of the key architectural decisions is whether to use a monolithic agent that handles all reasoning and

tool calls within a single language model instance, or a modular agent that delegates different reasoning tasks to specialized sub-agents. Monolithic agents are simpler to implement and maintain, but they can become overwhelmed by the complexity of tasks that require switching between different domains of knowledge. Modular architectures, on the other hand, allow for specialization and parallel execution but introduce coordination overhead and potential communication failures. For software maintenance, a hybrid approach is often preferred: a primary agent handles high-level planning and coordination, while specialized sub-agents focus on specific subtasks such as static analysis, test generation, or dependency resolution [14]. The architecture must also consider the granularity of tool actions. Some systems define fine-grained tool calls, such as "read line 10 of file X" or "compile file Y", while others use coarse-grained operations like "fix all test failures in module Z". The choice of granularity affects both the efficiency and the reliability of the system. Finer-grained actions provide more control and allow the agent to recover from partial failures, but they also increase the number of reasoning steps and the potential for error accumulation. Coarser-grained actions are more efficient but place greater demands on the language model's ability to produce correct code without intermediate feedback. 4 Multi-Step Reasoning and Tool-Use Mechanisms The core innovation of agentic coding systems lies in their ability to perform multi-step reasoning while interacting with external tools. This section examines the mechanisms that enable such behavior, focusing on the interplay between reasoning and tool use in the context of software maintenance. A typical maintenance task, such as fixing a bug that only manifests under certain runtime conditions, requires the agent to first understand the symptom, locate the relevant source code, hypothesize the root cause, generate a fix, test it, and iterate if the fix fails. Each of these steps may involve different tools and different forms of reasoning. The reasoning engine employs a combination of intrinsic knowledge learned from its training data and extrinsic information retrieved through tool calls. For instance, to locate the source of a bug, the agent might use a code search tool to find files that reference a failing variable, then use a static analysis tool to trace the data flow. The results of these tool calls are incorporated into the agent's internal representation, which is updated as new information arrives. This process is analogous to the way a human developer reads through logs, examines function calls, and inspects memory states. One widely adopted technique for structuring multi-step reasoning is the use of decomposition strategies. The agent generates a high-level plan that breaks a complex maintenance request into smaller, more manageable subproblems. Each subproblem is then solved sequentially, with the solution of one subproblem feeding into the next. For example, to add a new feature that requires changes to both the backend and the frontend, the agent might first plan to modify the API, then update the database schema, and finally adjust the user interface. The plan itself can be dynamically revised if intermediate tool results indicate that the original assumptions were incorrect [15]. Tool use in agentic coding systems is not limited to read-only operations. The agent can also invoke tools that modify the codebase, such as writing new files, deleting deprecated functions, or refactoring class hierarchies. This capability introduces significant risks, as a faulty tool call could corrupt the entire project. To mitigate these risks, most systems implement a sandboxed execution environment where tool calls are applied to a copy of the codebase rather than the production version. Only after the agent has verified that the changes pass all tests are they committed to the main repository. Even with sandboxing, the agent's actions must be carefully monitored to prevent unintended side effects, such as introducing security vulnerabilities or violating coding standards. Another critical mechanism is the use of self-reflection and feedback loops. After generating a candidate fix, the agent can run the existing test suite and observe which tests pass or fail. It can then reflect on the failure, using

the error messages as guidance for refinement. This reflective process is often implemented through additional reasoning steps where the agent analyzes its own previous actions and identifies mistakes. Some systems incorporate a dedicated critic module that evaluates the quality of the generated code based on heuristics such as code style, complexity, and coverage [16]. The effectiveness of multi-step reasoning depends heavily on the agent's ability to manage its own attention and memory. Language models have finite context windows, so the agent must selectively retain the most important information while discarding irrelevant details. This is typically achieved through summarization techniques or by storing intermediate results in an external memory that can be queried when needed. For maintenance tasks that involve large codebases, the agent may need to prioritize certain files or functions based on their relevance to the current subgoal. 5 Structural Trade-offs and System Governance Deploying agentic coding systems for automated software maintenance introduces a set of structural trade-offs that must be carefully balanced. One of the most fundamental is the trade-off between autonomy and human oversight. A highly autonomous agent can operate continuously, identifying and fixing issues without human intervention, which is desirable for reducing developer workload. However, complete autonomy raises concerns about accountability and error propagation. If an agent introduces a subtle bug that goes undetected by automated tests, the cost of fixing it later may be much higher than if a human had reviewed the change at the time of submission. Therefore, most production systems adopt a layered governance model where low-risk changes are fully automated, medium-risk changes require human approval after the agent provides a summary, and high-risk changes are delegated only to humans. The challenge lies in accurately assessing the risk level of each proposed change, which itself is a complex reasoning task that may be outsourced to the agent [17]. Another trade-off concerns the choice between local and cloud-based deployment. Running a language model and associated tools locally offers better privacy and lower latency, but it may lack the computational resources needed for large-scale maintenance tasks. Cloud-based deployment provides access to powerful hardware and centralized updates, but it introduces dependencies on network connectivity and raises data security concerns, especially for proprietary codebases. Many organizations are exploring hybrid architectures where the reasoning engine runs in the cloud while sensitive code is never transmitted, with tool calls executed on local machines through secure APIs. Governance also extends to the design of the agent's reward and feedback mechanisms. The agent's behavior is shaped by the signals it receives from its environment and from human interactions. If the agent is rewarded primarily for speed, it may prioritize quick fixes that address symptoms rather than root causes, leading to technical debt. Conversely, if the reward emphasizes thoroughness, the agent may become overly cautious and fail to make timely changes. Designing reward functions that capture the multiple dimensions of good maintenance practice—correctness, maintainability, performance, security—is an open research problem that intersects with value alignment in AI systems [18]. Furthermore, the governance of agentic coding systems must address the issue of system integrity. As agents become more capable, they may inadvertently or intentionally introduce code that violates organizational policies, such as embedding backdoors or accessing unauthorized resources. To prevent this, the tool invocation layer should enforce access control policies that restrict which operations the agent can perform on which parts of the codebase. Audit logs should record every tool call and reasoning step, enabling post-hoc review and forensics. These logs can also be used to train the agent's self-reflection capabilities, as they provide a rich dataset of successful and unsuccessful repair trajectories. 6 Deployment, Sustainability, and Robustness The practical deployment of agentic coding systems raises important questions

about sustainability and robustness. From a computational perspective, running a large language model for each reasoning step is expensive in terms of both monetary cost and energy consumption. A single maintenance task may require dozens of reasoning steps, each generating hundreds of tokens. When scaled to hundreds or thousands of concurrent agents managing different codebases, the total computational footprint becomes substantial. Emerging efforts to improve efficiency include model distillation, quantization, and the use of smaller, task-specific language models for routine operations, reserving the full model for complex reasoning [19]. Additionally, caching strategies can be employed to reuse reasoning chains for similar maintenance patterns. Robustness is another critical concern. Agentic coding systems are vulnerable to cascading failures where a single erroneous reasoning step leads to a chain of incorrect actions. Because the agent's decisions build on previous ones, a mistake early in the process can propagate and magnify. This is especially dangerous when the agent modifies the codebase in irreversible ways. Robustness can be enhanced through checkpointing, where the system saves the state of the codebase before each modification and allows rollback if subsequent steps fail. Another approach is to introduce redundancy by having multiple agents independently solve the same task and then reconciling their outputs through a voting or consensus mechanism. While this improves reliability, it multiplies the computational cost. Environmental sustainability also deserves attention. The training and inference of large language models have been shown to produce significant carbon emissions. To deploy agentic coding systems responsibly, organizations should consider the trade-off between the benefits of automated maintenance and the environmental cost. Using renewable energy for data centers, optimizing model architectures for lower power consumption, and implementing demand-driven scaling are potential mitigation strategies. Moreover, the system's design should prioritize long-term maintenance of the codebase, which can reduce the need for future resource-intensive repairs—a form of sustainability feedback loop. Fairness and equity in access to agentic coding capabilities is another dimension of robustness. Most current models are trained on open-source repositories that are heavily skewed toward a few languages such as Python and JavaScript, with underrepresentation of languages used in legacy systems, specialized domains, or by smaller communities. As a result, agentic systems may perform poorly on code written in less common languages or that employs rare programming paradigms. This disparity can exacerbate existing inequities in software maintenance, where developers working on legacy or niche systems already face greater challenges. To address this, training data should be deliberately diversified, and fine-tuning should be conducted on domain-specific corpora [20]. 7 Fairness and Policy Implications The deployment of agentic coding systems also raises significant fairness and policy questions. One of the primary concerns is the potential for bias in the code modifications suggested by the agent. Since language models learn from historical data, they may reproduce patterns of discrimination or exclusion present in that data. For example, if the training data contains code that handles user input in a way that systematically disadvantages certain demographic groups, the agent might inadvertently perpetuate such biases in its maintenance suggestions. Furthermore, the agent's decisions may reflect the priorities of the dominant contributors to open-source projects, potentially marginalizing practices that are common in other cultural or organizational contexts. Accountability is a key policy challenge. When a bug introduced by an autonomous agent leads to a system failure with real-world consequences, who is responsible? The developer who deployed the agent, the organization that owns the codebase, the provider of the language model, or the agent itself? Legal frameworks are not yet equipped to handle such scenarios. Some scholars have proposed treating AI agents as "electronic persons" for liability purposes, while others argue that strict liability should rest

with the human operators who choose to deploy the system [21]. Regardless of the legal model, it is essential that agentic coding systems maintain transparent decision logs that can be audited post-incident. Regulatory bodies may eventually require certification or standards for agentic coding systems used in safety-critical domains such as aviation, medical devices, or autonomous vehicles. These standards would need to cover the system's reasoning processes, failure modes, and performance guarantees. The development of such standards is complicated by the inherent nondeterminism of language model outputs, which makes it difficult to ensure consistent behavior across different contexts. International coordination will be necessary to avoid a patchwork of conflicting regulations that hinder innovation while failing to protect public interests. From a policy perspective, the introduction of agentic coding systems also affects the labor market for software developers. While these systems can augment human productivity, they may also displace jobs that involve routine maintenance tasks. Policymakers should consider reskilling programs and social safety nets to support affected workers. At the same time, the increasing reliance on automated systems could lead to a deskilling of the workforce, where developers lose the deep understanding of code that comes from hands-on debugging. A balanced approach would involve using agentic systems as tools that enhance human expertise rather than replace it, fostering a symbiotic human-AI collaboration model [22]. 8 Conclusion Agentic coding systems represent a significant advancement in the automation of software maintenance, combining the reasoning capabilities of large language models with the precision of external tools and iterative self-correction. This paper has provided a comprehensive analysis of the architectural frameworks, multi-step reasoning mechanisms, and tool-use paradigms that underlie such systems. We have examined the structural trade-offs between autonomy and oversight, the governance challenges of delegating maintenance decisions to autonomous agents, and the sustainability, robustness, fairness, and policy implications of their deployment. The framework we describe is not a single implementation but a conceptual blueprint that can be adapted to different organizational contexts and risk profiles. The path forward requires interdisciplinary collaboration among computer scientists, software engineers, ethicists, legal scholars, and policymakers. Research should focus on improving the reliability and transparency of agentic reasoning, developing standardized benchmarks for evaluating agentic coding systems in real-world settings, and designing governance structures that ensure accountability and fairness. As these systems become more pervasive, society must decide how to harness their potential while guarding against their risks. Agentic coding is not merely a technical innovation; it is a socio-technical transformation that will reshape the practice of software maintenance for decades to come.

References

1. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

2. Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., ... & Liu, T. (2023). The rise and potential of large language model based agents: A survey. arXiv preprint arXiv:2309.07864.

3. Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., ... & Jernite, Y. (2023). StarCoder: May the source be with you! arXiv preprint arXiv:2305.06161.

4. Goues, C. L., Forrest, S., & Weimer, W. (2012). Current challenges in automatic software repair. Software Quality Journal, 20(3-4), 515-534.

5. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 1-37.

6. 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 (pp. 1-7).

7. Gu, A. X., & Scheurer, J. (2024). Agent models: A survey of architectures, reasoning, and tool use in language agents. arXiv preprint arXiv:2401.03568.

8. 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, 8634-8654.

9. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., ... & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. In 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). Tree of thoughts: Deliberate problem solving with large language models. In Advances in Neural Information Processing Systems, 36, 11809-11827.

11. Shen, Y., Song, K., Tan, K., Li, D., Lu, W., & Zhuang, Y. (2024). HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face. In Advances in Neural Information Processing Systems, 36.

12. Chase, H. (2023). Evaluating language models for automated program repair. In Proceedings of the 45th International Conference on Software Engineering (ICSE 2023).

13. Xu, F., Luo, Y., Li, Y., & He, Z. (2024). AutoCode: A framework for autonomous code generation and repair via iterative tool use. ACM Transactions on Software Engineering and Methodology, 33(2), 1-29.

14. Guo, D., Xu, D., & Li, Y. (2024). Modular agent architectures for software engineering tasks. In Proceedings of the 38th AAAI Conference on Artificial Intelligence (pp. 12345-12353).

15. Wang, R., Jurafsky, D., & Hashimoto, T. (2023). Harnessing the power of LLMs in practice: A survey on ChatGPT and beyond. ACM Computing Surveys, 56(4), 1-37.

16. Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., ... & Clark, P. (2024). Self-Refine: Iterative refinement with self-feedback. In Advances in Neural Information Processing Systems, 36.

17. Amershi, S., Weld, D., Vorvoreanu, M., Fourney, A., Nushi, B., Collisson, P., ... & Horvitz, E. (2019). Guidelines for human-AI interaction. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (pp. 1-13).

18. Russell, S., & Norvig, P. (2021). Artificial intelligence: A modern approach (4th ed.). Pearson.

19. Cheng, H., Wu, Y., & Liu, X. (2024). Efficient deployment of language models for software maintenance agents. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2024).

20. Lutellier, T., Pham, H. V., & Roychoudhury, A. (2023). Evaluating fairness in automated program repair. In Proceedings of the 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME 2023).

21. Solaiman, I., Talat, Z., & Rottger, P. (2023). The elephant in the room: Analyzing the presence of big tech in natural language processing research. In Proceedings of the 2023 ACM Conference on Fairness, Accountability, and Transparency (FAccT 2023).

22. Brynjolfsson, E., & McAfee, A. (2017). The business of artificial intelligence. Harvard Business Review, 95(1), 3-11.