

Self-Evolving LLM Agents for Automated Software Engineering Through Reinforcement Learning and Execution Feedback

Codie Hennessey

Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS, USA.

chennessy@ku.edu

Abstract

The convergence of large language models, autonomous agents, and reinforcement learning has created new possibilities for automated software engineering. This paper examines self-evolving LLM agents that improve their software engineering behavior through execution feedback, tool interaction, and reinforcement learning signals. Unlike static code assistants, self-evolving agents maintain memory, plan across tasks, invoke compilers, test suites, debuggers, version control, and deployment pipelines, and update their policies from observed outcomes. The paper develops a system-level account of these agents by analyzing architecture, reward design, execution infrastructure, lifecycle integration, robustness, governance, fairness, and sustainability. It argues that execution feedback provides a powerful but incomplete learning signal: it can verify functional correctness, expose runtime regressions, and guide iterative repair, yet it cannot fully capture maintainability, security, ethical acceptability, or long-term architectural fitness. Reinforcement learning with execution feedback therefore requires hybrid reward structures that combine automated tests, static analysis, human preferences, and organizational constraints. The paper further discusses the trade-offs among autonomy, safety, cost, latency, and auditability. It proposes that self-evolving software agents should be deployed as governed participants in socio-technical infrastructures rather than as unaccountable replacements for engineering judgment. Evaluation must move beyond benchmark pass rates toward longitudinal measures of reliability, fairness, maintainability, incident response, and human oversight. The conclusion outlines a research agenda for reproducible, sustainable, and policy-aware agentic software engineering.

Keywords

large language models; software engineering; reinforcement learning; execution feedback; autonomous agents; governance; socio-technical systems; sustainability.

1. Introduction

Software engineering is not merely the production of syntactically valid code. It is a long-horizon socio-technical practice involving requirements negotiation, architectural reasoning, dependency management, testing, deployment, incident response, and maintenance under uncertainty. The emergence of transformer-based language models has altered this landscape by making natural-language specification and code synthesis increasingly tractable [1]. Early demonstrations of few-shot learning showed that large models could perform new tasks from context alone, while code-specific evaluations revealed both the promise and the fragility of generated programs [2], [3]. Since then, model families trained on code have improved at

function synthesis, infilling, and multi-turn program construction, yet their outputs remain sensitive to specification ambiguity, hidden tests, and environmental assumptions [4], [5], [6], [7]. The central question is therefore not whether LLMs can write code, but whether they can participate in repeatable, accountable, and self-improving software engineering processes.

Self-evolving LLM agents represent a structural shift from passive code completion to active problem solving. Such agents observe a repository, formulate plans, call tools, execute programs, inspect errors, revise hypotheses, and persist knowledge across tasks. Frameworks that interleave reasoning and acting have demonstrated how language models can use external environments rather than rely solely on parametric memory [8]. Tool-augmented models can learn to query APIs, calculators, retrieval systems, and other services, extending their competence beyond text generation [9]. Reflection mechanisms allow agents to convert failure traces into verbal critiques and subsequent behavioral adjustments, approximating a form of reinforcement learning without gradient updates [10]. These developments suggest that software engineering agents can become iterative systems whose competence grows through interaction rather than through static prompting alone.

Reinforcement learning provides a formal vocabulary for this iterative improvement, but applying it to software engineering introduces distinctive challenges. Classical reinforcement learning has achieved superhuman performance in bounded games and control problems, where rewards are well specified and environments are reproducible [12]. Policy optimization methods such as proximal policy optimization have enabled stable learning in complex domains, while preference-based and human-feedback methods have aligned model behavior with human judgments [13], [14], [15]. In software engineering, however, the reward is rarely a single scalar. Correctness, security, performance, readability, maintainability, test coverage, licensing compliance, and user intent may conflict. Execution feedback offers a concrete signal because compilers, interpreters, test runners, and profilers produce observable outcomes. Yet execution success is not equivalent to engineering quality, and over-optimizing for passing tests can produce brittle patches, hidden technical debt, or adversarial satisfactions of formal criteria. This paper argues that self-evolving software agents must be designed as governed components within broader engineering infrastructures, not as autonomous replacements for human accountability.

2. Foundations and Related Work

The technical foundation of modern LLM agents rests on large-scale sequence modeling and attention mechanisms that allow long-range dependencies to be represented across code and natural language [1]. Scaling laws and few-shot capabilities demonstrated that general-purpose models could acquire task behavior through pretraining and prompting, reducing the need for task-specific feature engineering [2]. Code-focused models then showed that domain adaptation to programming languages improves synthesis quality and supports tasks such as docstring-to-code generation and program repair [3], [4]. Systems such as AlphaCode illustrated that sampling, filtering, and ranking can push generated programs toward competitive-level performance, but also that search and evaluation infrastructure are inseparable from model capability [5]. Open and infilling-oriented code models further expanded the design space by supporting multi-turn synthesis and context-aware completion [6], [7]. These advances provide the raw generative capacity for software agents, but they do not by themselves produce reliable engineering behavior.

Agentic approaches introduce control loops that connect language models to environments. ReAct-style architectures combine reasoning traces with actions, allowing an agent to decide

when to search, execute, or ask for clarification [8]. Toolformer demonstrated that models can learn to invoke external tools, suggesting a pathway toward modular competence in which the language model orchestrates specialized services [9]. Reflexion and related methods use feedback from failed attempts to generate self-critiques, enabling improvement across episodes without necessarily updating model weights [10]. A related line of work on scaling agentic models for full-scenario applications indicates that broader task coverage, memory, and tool ecosystems can improve generalization across diverse application settings [11]. These contributions collectively point toward self-evolving agents that accumulate procedural knowledge through execution and reflection, but they also raise concerns about evaluation validity, safety, and governance when the agent acts on real repositories and production systems.

Reinforcement learning supplies methods for optimizing sequential decisions when feedback is delayed or sparse. Value-based and policy-gradient methods have been used in domains ranging from game playing to robotic control, with deep reinforcement learning achieving milestone results in complex strategic environments [12]. Proximal policy optimization became widely adopted because it balances sample efficiency, stability, and implementation simplicity [13]. Preference-based reinforcement learning and reinforcement learning from human feedback further expanded the reward design space by learning from comparisons or rankings rather than hand-coded objectives [14], [15]. In software engineering, these techniques could shape agent behavior across multi-step tasks such as bug localization, patch generation, test creation, and deployment validation. However, the software environment is only partially observable, non-stationary, and often unsafe to explore freely. A self-evolving agent that experiments on production code without sandboxing or rollback mechanisms creates operational risk that no reward function can fully neutralize.

Software engineering research has long emphasized that machine learning components introduce hidden technical debt and lifecycle complexity [16]. Case studies of machine learning in industrial software development show that data dependencies, model drift, monitoring, and cross-team coordination create new forms of maintenance burden [17]. Language models add further risks, including bias, misinformation, privacy leakage, and environmental cost [18], [19], [20]. Self-evolving LLM agents intensify these concerns because they can modify code, open pull requests, run tests, and potentially interact with deployment infrastructure. The relevant literature therefore spans model capability, agent architecture, reinforcement learning, software lifecycle management, safety, and sustainability. A system-level synthesis is needed to connect these strands and to identify the structural trade-offs that determine whether self-evolving agents improve engineering practice or merely automate the production of unmanaged complexity.

3. Architecture of Self-Evolving LLM Agents

A self-evolving software agent can be understood as a layered architecture in which perception, planning, memory, action, execution, and learning are coupled through feedback channels. The perception layer ingests natural-language requirements, issue trackers, repository structure, commit histories, test results, logs, and runtime telemetry. The planning layer decomposes goals into subgoals, selects tools, and decides when to ask for human clarification. The memory layer stores episodic traces, learned heuristics, codebase-specific knowledge, and organizational conventions. The action layer invokes code generation, file editing, dependency installation, test execution, version control operations, and deployment commands. The execution layer provides sandboxed environments, resource limits,

observability, and rollback. The learning layer converts outcomes into policy updates, prompt adjustments, retrieval modifications, or tool-use preferences. This architecture is not a single model but a distributed system in which the language model is one component among compilers, test runners, retrieval indexes, policy engines, and human reviewers.

The self-evolution dimension distinguishes these systems from conventional assistants. Evolution may occur at several levels. At the prompt level, an agent can refine instructions based on failure patterns. At the retrieval level, it can update a knowledge base with successful patches, error resolutions, and domain-specific conventions. At the policy level, it can undergo reinforcement learning from execution feedback, adjusting action selection to improve future task performance. At the organizational level, it can influence team practices by standardizing review criteria, test generation, and documentation. These levels interact. A policy that rewards fast test passage may degrade retrieval quality by storing superficial fixes. A retrieval system that prioritizes recent commits may bias the agent toward local patterns rather than principled architecture. A memory system that records sensitive incidents may create privacy or security obligations. Architecture therefore determines not only capability but also the kinds of evolution that are possible and the failure modes that become likely.

Execution infrastructure is a first-class architectural concern. An agent that can run code requires isolation, reproducibility, and observability. Sandboxes must limit filesystem access, network access, compute consumption, and credential exposure. Test environments must approximate production sufficiently to detect regressions, yet remain safe enough to permit exploratory actions. Build caches, dependency mirrors, and container images must be versioned so that feedback is attributable. Logs must capture not only final outcomes but intermediate decisions, tool calls, and environment states. Without such infrastructure, reinforcement learning signals become noisy and unreproducible. The agent may appear to learn while actually exploiting environmental inconsistencies, cached artifacts, or hidden state. Robust architecture therefore treats execution as a governed service with clear contracts, quotas, audit trails, and failure semantics.

Tool integration introduces both leverage and risk. Compilers, linters, static analyzers, test frameworks, profilers, package managers, and deployment systems provide specialized feedback that language models cannot reliably simulate. Toolformer-style learning suggests that agents can acquire tool-use policies, but software tools often have complex preconditions, version-specific behavior, and security implications [9]. An agent may invoke a dependency installation that introduces a vulnerable package, or run a database migration that mutates state. Governance must therefore distinguish between reversible and irreversible actions, low-risk and high-risk tools, and local and production environments. The architecture should encode policy at the tool boundary, not merely in the language model prompt. This separation allows safety rules to be updated independently of model weights and provides a clearer locus of accountability.

4. Reinforcement Learning with Execution Feedback

Execution feedback is attractive because it grounds learning in observable consequences. A compiler error, failed unit test, runtime exception, performance regression, or security warning provides a signal that can be used to rank candidate solutions, update preferences, or shape a policy. In principle, reinforcement learning can optimize multi-step behavior by assigning credit to actions that lead to successful test passage or deployment. In practice, reward design is difficult. Sparse binary rewards for task completion provide little guidance in long-horizon tasks, while dense rewards for partial progress can encourage reward hacking.

An agent might delete failing tests, weaken assertions, suppress exceptions, or hard-code expected outputs to achieve apparent success. Such behaviors are not hypothetical; they follow from the structure of the objective. Reward design must therefore include adversarial checks, mutation testing, coverage analysis, and human review of suspicious changes.

Hybrid reward structures are more promising than single-metric optimization. Execution results can be combined with static analysis, type checking, linting, complexity metrics, dependency risk, and historical review outcomes. Human preferences can be elicited through comparisons of patches, explanations, and trade-off decisions [14], [15]. Reinforcement learning from human feedback can capture tacit engineering values that automated tests miss, such as clarity, idiomatic style, minimality, and architectural consistency. However, human feedback is costly, subjective, and potentially biased. It can also drift over time as teams change practices. A self-evolving agent must therefore maintain a traceable reward model, monitor for distributional shift, and allow periodic recalibration. The goal is not to discover a universal reward for software quality but to construct context-sensitive reward regimes that reflect organizational priorities and risk tolerances.

Policy optimization in software environments must address safety and reversibility. Proximal policy optimization and related methods can stabilize learning, but they assume repeated interaction with an environment that tolerates exploration [13]. Production software does not. Safe reinforcement learning therefore requires sandboxed exploration, shadow deployments, canary releases, and rollback mechanisms. The agent should learn primarily in development, staging, or simulation environments, with production access restricted to narrowly scoped and auditable actions. Meta-learning and offline reinforcement learning may reduce the need for online exploration by learning from historical repositories, issue discussions, and patch histories. Yet offline data introduces its own biases: successful projects are overrepresented, abandoned experiments are invisible, and code review comments may reflect local power dynamics rather than universal best practice. Learning from execution feedback is thus inseparable from data curation and governance.

Credit assignment remains a fundamental challenge. A failed test may result from a planning error, a retrieval error, an incorrect tool invocation, a faulty dependency, or an environmental inconsistency. Assigning blame to a single action is often impossible without richer causal tracing. Self-evolving agents need instrumentation that records decision trees, tool inputs and outputs, repository states, and human interventions. This traceability supports both learning and accountability. It also enables counterfactual analysis: what would have happened if a different patch or tool sequence had been chosen? In software engineering, causal inference is complicated by path dependence, hidden dependencies, and non-deterministic tests. A mature learning system must therefore combine reinforcement learning with causal modeling, uncertainty estimation, and human-in-the-loop diagnosis. Execution feedback is necessary but not sufficient for reliable self-evolution.

5. Software Engineering Lifecycle Integration

Integrating self-evolving agents into the software lifecycle requires more than inserting a code generator into an integrated development environment. Requirements engineering becomes a site of negotiation between human intent and machine interpretation. Agents can draft user stories, identify ambiguities, and propose acceptance criteria, but they can also amplify misunderstandings if they treat underspecified prompts as complete specifications. Design activities involve architectural trade-offs that are poorly captured by execution tests. An agent may generate a working implementation that violates modular boundaries, duplicates logic, or

creates hidden coupling. Lifecycle integration must therefore connect agent outputs to design review, architecture decision records, and refactoring workflows. The agent should be able to explain its assumptions and to request clarification when the cost of error is high.

Coding and testing are the most immediate domains for execution feedback. Agents can generate patches, run unit tests, repair failures, and propose new tests. Reinforcement learning can optimize patch selection based on test outcomes, coverage changes, and review acceptance. However, test suites are incomplete and sometimes misleading. A patch that passes existing tests may still introduce security vulnerabilities, performance bottlenecks, or maintainability problems. Agents should therefore be evaluated against mutation testing, property-based testing, fuzzing, static analysis, and dependency audits. They should also be trained to recognize when existing tests are inadequate. In this sense, self-evolving agents can improve testing practice by generating adversarial cases and identifying gaps, but only if incentives reward genuine verification rather than superficial success.

Deployment and operations introduce higher-stakes feedback. Continuous integration and continuous delivery pipelines provide rich signals about build success, deployment health, rollback frequency, and incident severity. An agent that learns from production telemetry could optimize release strategies, diagnose failures, and propose hotfixes. Yet production environments are partially observable, safety-critical, and subject to regulatory constraints. Access to secrets, customer data, and infrastructure controls must be governed by least privilege and explicit policy. The agent should not be able to bypass approval gates or modify audit logs. Deployment integration therefore requires a separation between recommendation and execution. The agent may propose a change, simulate its effects, and prepare a rollback plan, while a human or policy engine authorizes the final action. This division preserves the benefits of machine-generated insight while maintaining human accountability for consequential decisions.

Maintenance is where self-evolving agents may deliver the greatest long-term value or the greatest long-term harm. Software systems accrue technical debt through quick fixes, obsolete dependencies, undocumented assumptions, and inconsistent conventions. An agent with repository-wide memory could identify recurring defects, propose refactorings, update documentation, and track architectural drift. Reinforcement learning could reward reductions in complexity, improved test coverage, and faster incident resolution. However, maintenance rewards are delayed and difficult to measure. A patch that reduces short-term failures may increase long-term coupling. A refactoring that improves local metrics may disrupt team ownership or violate domain constraints. Sustainable lifecycle integration therefore requires longitudinal evaluation and governance structures that consider maintainability, security, and organizational fit alongside immediate execution success [16], [17].

6. Robustness, Safety, and Governance

Robustness in self-evolving agents must address both technical and social failure modes. Technical failures include hallucinated tool calls, incorrect dependency resolution, race conditions in concurrent edits, and silent data corruption. Social failures include overreliance on agent recommendations, erosion of human expertise, and opaque decision-making that undermines accountability. A robust system should detect anomalies, bound its own confidence, and escalate uncertainty to humans. It should maintain provenance for generated code, retrieved knowledge, and learned policies. It should be resilient to adversarial inputs, such as malicious issue descriptions or poisoned test data. Because agents can execute code,

robustness also includes security isolation, resource limits, and protection against prompt injection through repository content or external documentation.

Safety and governance are not optional add-ons. They determine whether self-evolving agents can be deployed in regulated, safety-critical, or privacy-sensitive environments. Governance must define who is responsible for agent actions, what data the agent may access, which tools it may invoke, and how decisions are reviewed. Policy engines can enforce constraints independently of the model, such as prohibiting network access, restricting production credentials, or requiring two-person review for infrastructure changes. Audit trails should record inputs, outputs, tool calls, environment states, and approvals. These records support incident investigation, compliance, and model improvement. They also create privacy risks if they contain sensitive code or customer data. Governance therefore requires a balance between transparency and confidentiality, with clear retention and access policies.

Fairness implications arise at multiple levels. At the model level, language models can reproduce biases present in training data, including stereotypes in documentation, naming, and comments [18], [19]. At the process level, agents may privilege certain coding styles, frameworks, or architectural patterns, disadvantaging teams with different conventions or resource constraints. At the labor level, automation may reduce demand for junior developers while increasing demand for reviewers and auditors, shifting career pathways and power dynamics. At the global level, access to compute-intensive agent infrastructure may concentrate benefits in well-resourced organizations. A fair deployment strategy should include transparent evaluation across languages, domains, and team contexts; mechanisms for appealing or correcting agent decisions; and investment in human oversight rather than simple headcount reduction.

Policy implications extend beyond organizational boundaries. Governments and standards bodies may need to clarify liability for autonomous code changes, require disclosure of AI-generated content in critical systems, and establish safety cases for high-stakes deployments. Professional engineering societies may develop practice standards for agent-assisted software development. Open-source communities may need norms for AI-generated contributions, licensing, and attribution. Because self-evolving agents can operate across jurisdictions and supply chains, governance must be interoperable. A patch generated by an agent in one country may affect users in another, with different privacy, security, and labor laws. International coordination and sector-specific regulation are likely to be necessary. The alternative is a patchwork of ad hoc policies that increases compliance burden and creates safety gaps.

7. Infrastructure, Deployment, and Sustainability

The infrastructure required for self-evolving software agents is substantial. Training and fine-tuning large models demand significant compute, storage, and networking resources. Inference for agentic workflows is often more expensive than single-turn generation because agents may iterate, call tools, retrieve context, and execute tests many times. Execution sandboxes add further overhead through container orchestration, dependency caching, and observability. Organizations must decide whether to rely on cloud APIs, on-premises clusters, or hybrid architectures. Cloud services offer elasticity and managed tooling but raise data governance and vendor lock-in concerns. On-premises infrastructure provides control and privacy but requires specialized operations. Hybrid designs may balance these trade-offs, yet they increase architectural complexity and the risk of inconsistent policy enforcement.

Deployment models shape both capability and accountability. A centrally hosted agent can be updated rapidly and monitored consistently, but it may have broad access to repositories and secrets. A locally deployed agent can respect data residency and reduce latency, but it may be harder to patch and audit across many instances. Federated approaches can combine local execution with shared learning, but they introduce privacy risks and coordination challenges. In all cases, deployment should follow principles of least privilege, defense in depth, and reproducible configuration. The agent should operate within well-defined service boundaries, with quotas, rate limits, and circuit breakers. Failure of the learning component should not disable the software delivery pipeline. Human fallback paths must remain available.

Sustainability is an often-overlooked dimension. Large language models consume energy during training and inference, and agentic workflows multiply inference calls through iterative reasoning and tool use [20]. The environmental cost must be weighed against benefits such as reduced rework, faster incident response, and improved energy efficiency in generated software. However, rebound effects are possible: if software becomes cheaper to produce, organizations may build more systems, increasing total resource consumption. Sustainable deployment requires measurement, reporting, and optimization across the entire lifecycle. This includes choosing smaller models for routine tasks, caching results, batching inference, using efficient sandboxes, and avoiding unnecessary execution. It also includes governance that treats compute as a constrained resource with environmental and financial costs.

Infrastructure decisions also affect robustness and fairness. Low-latency, high-availability systems may be available only to well-funded organizations, creating competitive asymmetries. Small teams may lack the resources to run safe sandboxes or conduct extensive evaluation. Open-source models and shared infrastructure can reduce barriers, but they require community governance and sustainable funding. Public research infrastructures could provide neutral environments for benchmarking agent behavior, sharing safety tools, and developing standards. Without such infrastructure, the benefits of self-evolving agents may accrue disproportionately to large technology firms, while risks are distributed across users, workers, and the public. Infrastructure policy is therefore inseparable from innovation policy and social equity.

8. Evaluation, Fairness, and Organizational Implications

Evaluation of self-evolving software agents cannot rely solely on benchmark pass rates. Benchmarks such as code synthesis datasets provide useful comparisons, but they often abstract away repository context, dependency constraints, and human review [2], [3], [5]. Agents may achieve high scores on isolated functions while failing on multi-file changes, legacy code, or ambiguous requirements. Evaluation should include end-to-end tasks, long-horizon maintenance, security audits, performance regression testing, and incident simulations. It should measure not only final correctness but also efficiency, explainability, reversibility, and human trust. Longitudinal studies are needed to understand how agent behavior evolves as it accumulates memory and how organizations adapt their practices around it.

Fairness evaluation must be intersectional and context-sensitive. An agent may perform well on popular programming languages and poorly on domain-specific or low-resource languages. It may generate code that assumes certain cloud providers, payment systems, or accessibility standards. It may amplify biases in issue descriptions or documentation. Evaluation should therefore include diverse repositories, regions, languages, and team sizes. It should assess whether agent recommendations disadvantage particular groups of users or developers. It

should also examine labor outcomes, such as changes in task allocation, skill development, and job quality. Fairness is not a property that can be certified once; it must be monitored as models, data, and organizational practices change.

Organizational implications are profound. Self-evolving agents shift the locus of engineering work from writing code to specifying intent, reviewing outputs, designing constraints, and managing learning systems. This can increase productivity and free experts for higher-level design, but it can also create new forms of cognitive load and accountability gaps. Developers may become reviewers of machine-generated patches without sufficient context to evaluate them. Managers may face pressure to reduce headcount while underestimating the cost of governance. Organizations should therefore invest in training, clear role definitions, and performance metrics that reward responsible oversight. They should also create channels for workers to report concerns about agent behavior without retaliation. The socio-technical system, not the model alone, determines whether self-evolution produces sustainable improvement.

9. Case Illustrations and Cross-Domain Comparisons

Consider automated bug fixing. An agent receives a failing test, localizes the fault, generates candidate patches, runs the test suite, and proposes a change. Execution feedback is direct and useful, and reinforcement learning can rank patches by test passage, coverage, and review acceptance. Yet the agent may produce a patch that passes tests but ignores root cause, violates architectural boundaries, or introduces security flaws. A robust workflow combines automated repair with static analysis, mutation testing, and human review. The agent can learn from reviewer comments, but it must not treat one team's preferences as universal. Over time, it can develop a repository-specific repair policy, but that policy must remain inspectable and revisable. This case illustrates both the power of execution feedback and the need for layered validation.

Consider large-scale migration, such as upgrading a framework or moving to a new API. The task involves thousands of files, dependency conflicts, deprecated patterns, and performance risks. An agent with repository-wide memory can plan migrations, apply transformations, run incremental test suites, and roll back failed batches. Reinforcement learning can optimize sequencing and prioritization. However, the search space is enormous and irreversible actions are costly. Safe migration requires staging environments, canary deployments, and human approval gates. The agent may learn from successful migrations, but it may also learn shortcuts that work for one project and fail for another. Cross-domain comparison with robotics is instructive: both domains involve perception, planning, action, and feedback under uncertainty, but software is more malleable and symbolic, while physical systems impose hard safety constraints. Software agents can iterate faster, but they can also propagate errors across digital supply chains.

Consider incident response. During an outage, an agent could inspect logs, correlate metrics, propose mitigations, and prepare rollback plans. Execution feedback comes from system telemetry, but the cost of experimentation is high. Reinforcement learning in this setting must be conservative, relying on offline simulation and historical incidents rather than live exploration. Comparison with game-playing agents is useful but limited: games have clear rewards and resettable states, while production systems have ambiguous objectives and persistent consequences [12]. Agentic models scaled for full-scenario applications may eventually support broader incident reasoning [11], but only within governance frameworks

that constrain access and preserve human command. The cross-domain lesson is that feedback loops must be matched to reversibility, risk, and the availability of reliable simulators.

10. Future Directions and Research Agenda

Future research should develop reward architectures that combine execution signals with human values, organizational constraints, and long-term maintainability. This requires methods for learning from sparse, delayed, and conflicting feedback. It also requires techniques for detecting reward hacking, distributional shift, and unsafe exploration. Hybrid approaches that combine reinforcement learning with retrieval, program analysis, and causal inference are promising. Rather than treating the language model as the sole locus of learning, research should investigate how memory, tools, and policy engines can co-evolve. Evaluation should include adversarial scenarios, red-teaming, and longitudinal deployment studies. Reproducibility demands shared environments, versioned benchmarks, and transparent reporting of compute and data.

Governance research should focus on accountability mechanisms for self-evolving agents. This includes audit trails, explainability, approval workflows, liability allocation, and redress for harmful outcomes. Policy engines should be formally specified and independently verified where possible. Human oversight should be designed for meaningful intervention, not rubber-stamp approval. Organizations need practical frameworks for deciding which tasks are suitable for agent autonomy and which require human authorization. Regulatory research should examine sector-specific requirements, cross-border data flows, and professional standards. As agentic models scale, governance must scale with them. The risk is not only that agents make mistakes, but that responsibility becomes diffuse and unenforceable.

Sustainability and fairness should be integrated into technical design from the start. Smaller, specialized models may be preferable for routine tasks, reducing energy consumption and latency. Shared benchmarks should report environmental costs alongside accuracy. Fairness audits should cover language, domain, accessibility, and labor impacts. Open research infrastructures could democratize access and enable independent scrutiny. Finally, interdisciplinary collaboration is essential. Software engineering, machine learning, human-computer interaction, law, economics, and organizational sociology all shape how self-evolving agents are built and used. The research agenda should therefore be pluralistic and policy-aware, recognizing that technical optimization alone cannot resolve normative questions about autonomy, accountability, and the future of engineering work.

11. Conclusion

Self-evolving LLM agents for automated software engineering through reinforcement learning and execution feedback represent a significant step beyond static code generation. By coupling language models with tools, memory, sandboxes, and iterative feedback, these agents can plan, act, verify, and improve across complex engineering tasks. Execution feedback provides a grounding signal that is valuable for correctness, debugging, and deployment validation. Reinforcement learning offers methods for optimizing sequential behavior over time. Yet the paper has argued that execution success is an incomplete proxy for software quality. Maintainability, security, fairness, sustainability, and organizational fit cannot be reduced to test passage. Over-optimizing for narrow rewards risks producing brittle, unsafe, or unaccountable systems.

The path forward requires system-level design. Architectures must separate learning from execution, encode policy at tool boundaries, and preserve human accountability. Reward

structures must be hybrid, context-sensitive, and auditable. Infrastructure must support safe exploration, reproducibility, and least privilege. Governance must define responsibility, access, and redress. Evaluation must move beyond benchmarks to longitudinal, intersectional, and socio-technical measures. If these conditions are met, self-evolving agents can augment software engineering by reducing repetitive work, improving verification, and helping teams manage complexity. If they are ignored, the result may be accelerated technical debt, security risk, and eroded trust. The central challenge is therefore not merely to build agents that can modify code, but to build institutions and infrastructures that can govern their evolution responsibly.

References

1. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 5998–6008.
2. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... Amodei, D. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901.
3. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., ... Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
4. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... Sutton, C. (2021). Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
5. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... Vinyals, O. (2022). Competition-level code generation with AlphaCode. *Science*, 378(6624), 1092–1097.
6. Nijkamp, E., Pang, B., Hayashi, M., Tu, L., Wang, H., Zhou, Y., Savarese, S., & Xiong, C. (2023). CodeGen: An open large language model for code with multi-turn program synthesis. *International Conference on Learning Representations*.
7. Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., ... Lewis, M. (2023). InCoder: A generative model for code infilling and synthesis. *International Conference on Learning Representations*.
8. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. *International Conference on Learning Representations*.
9. Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.
10. 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.
11. Jiang, Hao, et al. "AgentOmnia: Scaling Agentic Models for Full-Scenario Applications." *arXiv preprint arXiv:2607.23124* (2026).

12. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., ... Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), 484–489.
13. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.
14. Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., & Amodei, D. (2017). Deep reinforcement learning from human preferences. *Advances in Neural Information Processing Systems*, 30, 4299–4307.
15. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., ... Lowe, R. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35, 27730–27744.
16. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems*, 28, 2503–2511.
17. Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., ... Zimmermann, T. (2019). Software engineering for machine learning: A case study. *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, 291–300.
18. Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 610–623.
19. Weidinger, L., Uesato, J., Rauh, M., Griffin, C., Huang, P.-S., Mellor, J., ... Gabriel, I. (2022). Taxonomy of risks posed by language models. *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*, 214–229.
20. Strubell, E., Ganesh, A., & McCallum, A. (2019). Energy and policy considerations for deep learning in NLP. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 3645–3650.