Reinforcement Learning-Based Code Generation and Program Repair with Language Models

Fernando M. Ryan
Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS, USA.
fernandoryan76@ku.edu

Longzhu Yan
Department of Computer Science, Colorado State University, Fort Collins, CO, USA.
longzhuyan440@colostate.edu

Abstract

Large language models have transformed automated software engineering by enabling the generation and repair of source code from natural language intent. However, the standard supervised fine-tuning paradigm often fails to capture long-horizon correctness, execution semantics, and maintainability requirements that real-world software systems demand. Reinforcement learning offers a complementary mechanism for aligning code-producing models with execution feedback, test outcomes, and human preferences. This paper presents a system-level analysis of reinforcement learning-based code generation and program repair. It examines structural trade-offs across model architectures, training workflows, reward design, deployment infrastructure, and governance. The discussion emphasizes the interaction between policy optimization, semantic feedback, and software engineering constraints such as test coverage, regression risk, fairness, and auditability. Rather than focusing on algorithmic notation, the paper develops a conceptual framework for integrating reinforcement learning into code-oriented language model pipelines. It further explores deployment concerns including inference cost, safety, monitoring, and the organizational processes required for responsible operation. By connecting reinforcement learning with program repair, the analysis shows that learned repair policies can improve code quality but also introduce risks related to reward hacking, distributional drift, and reduced human oversight. The paper concludes with implications for sustainable system design and policy development in AI-assisted software engineering.

Keywords

Reinforcement learning; code generation; program repair; large language models; software engineering infrastructure; alignment; deployment governance.

1. Introduction

The emergence of large language models has reshaped the practice of software engineering by allowing developers to express intent in natural language and receive executable source code as output. The foundational advances in attention mechanisms and deep sequence modeling established the architectural conditions for this development [1]. Early bidirectional pretraining demonstrated that rich contextual representations could be learned from unlabeled text, leading to substantial improvements in downstream language understanding tasks [2]. Large autoregressive models further showed that scale, data diversity, and few-shot prompting could produce useful generation behaviors across domains [3]. Unified text-to-text

approaches subsequently formalized a flexible interface for mapping input instructions to output sequences, a perspective that naturally extends to source code [4].

Within this broader trajectory, code-specific pretraining and evaluation have become prominent research directions. Models trained on large repositories of public source code have shown the ability to generate syntactically plausible programs from docstrings and function signatures [5]. Complementary work on program synthesis has examined how language models can be adapted to solve competitive programming and specification-driven tasks [6]. While these successes are significant, supervised pretraining and fine-tuning alone do not reliably optimize the properties that matter most in software systems: correctness under execution, adherence to test suites, robustness to input variation, maintainability, and compatibility with existing codebases. This gap motivates the integration of reinforcement learning into code generation and program repair. Reinforcement learning provides a feedback-driven pathway to improve model behavior by treating generated programs as actions and execution outcomes or repair outcomes as reward signals. Recent methods have demonstrated that execution-based reward can be used to train code generators more effectively than static imitation alone [7, 8].

The present paper addresses the system-level dimensions of reinforcement learning-based code generation and program repair. It argues that the design of such systems is not only a machine learning problem but also an infrastructure, governance, and policy challenge. The paper examines architectural choices, reward formulation, deployment constraints, and organizational controls. It further discusses how program repair introduces distinct requirements because repair actions must preserve existing functionality while addressing defects. Through this analysis, the paper aims to provide a coherent academic account of the structural trade-offs and long-term implications of integrating reinforcement learning into AI-assisted software engineering.

2. Background and Related Work

Reinforcement learning from human feedback has become a key method for aligning language models with human preferences. Early work in this area adapted policy optimization to language generation by training reward models on human comparisons and using those models to guide fine-tuning [9]. Subsequent work demonstrated that summarization models could be improved beyond supervised training by optimizing against learned reward functions [10]. Instruction-tuned models extended these ideas to general-purpose assistants, showing that reinforcement learning could make models more helpful, harmless, and aligned with user intent [11]. Proximal policy optimization has served as a stable algorithm for these updates due to its conservative policy adjustments and scalability [12]. In parallel, chain-of-thought prompting revealed that explicit reasoning paths could improve model performance on tasks requiring multi-step computation [13]. More recent work has examined trajectory-level filtering and alignment to reduce computational overhead in small parameter models [14].

For code generation specifically, several open and proprietary models have been introduced to support software engineering tasks. Code Llama emphasized specialized code pretraining and infilling capabilities [15]. StarCoder demonstrated the value of large-scale permissively licensed training data for code synthesis [16]. CodeGen explored multi-turn program synthesis and conversational interaction with code [17]. These models provide strong initialization points for reinforcement learning, yet they are typically trained with next-token prediction rather than execution-aware objectives. Reinforcement learning methods such as CodeRL and execution-based deep reinforcement learning have begun to close this gap by

using unit tests and program execution as reward signals [7, 8]. The central challenge is to design reward functions and training curricula that encourage correct behavior without overfitting to narrow test cases or learning superficial patterns.

3. System Architecture and Training Workflows

A reinforcement learning-based code generation system contains several interacting components: a pretrained code model, a reward model or execution environment, a policy optimization procedure, and a data pipeline for generating and evaluating candidate programs. The pretrained model typically inherits a decoder-only or encoder-decoder architecture. Decoder-only models have become prevalent in large-scale open systems such as Llama 2 due to their scalability and simplicity for autoregressive generation [18]. Encoder-decoder models offer structural benefits for conditional generation because they can process specification context separately from output decoding. The choice of architecture affects how reinforcement learning signals propagate through the model and how efficiently the system can be trained and served.

The training workflow generally alternates between sampling programs from the current policy, evaluating those programs against tests or static checks, computing rewards, and updating the policy. This cycle creates a tight coupling between the software execution environment and the machine learning training loop. The execution environment may include sandboxed interpreters, containers, static analyzers, and test harnesses. Its reliability directly influences reward quality. If the environment is slow or nondeterministic, training becomes unstable and costly. Therefore, system architects must balance execution fidelity against training throughput. Some deployments use cached execution results for repeated programs, while others use approximate static signals to reduce runtime. These infrastructure choices shape the learning dynamics even though they are often overlooked in algorithmic discussions.

Data governance is another structural concern. The sampling process can produce large volumes of candidate code that may contain unsafe operations, private data references, or copyright-sensitive fragments. Organizations must implement filtering and provenance tracking before generated code enters the reward pipeline. The reward model may also be trained on human feedback, requiring careful annotation workflows and quality assurance. In code repair settings, the data pipeline must include buggy programs, associated test failures, and acceptable patches. The heterogeneity of this data across programming languages, frameworks, and code styles creates distributional challenges. A workflow that performs well on self-contained Python functions may not transfer to large Java or C++ systems with complex build configurations. Therefore, system design must consider language-specific execution environments and modular abstraction layers.

4. Reinforcement Learning Formulations and Alignment Trade-offs

Reinforcement learning for code generation is conceptually framed as a sequential decision process in which each token is an action, the partially generated program is the state, and the final program receives a scalar reward based on execution outcomes or human judgments. Although the technical formulation is mathematical, the system-level implications are best understood through trade-offs in alignment. A reward signal based solely on unit test pass rates may encourage the model to produce programs that satisfy the given tests but fail on unseen inputs. This is a form of reward hacking in which the policy exploits gaps in the evaluation rather than learning robust programming behavior. To mitigate this, system designers often combine multiple reward components such as test pass rates, static analysis

results, style consistency, and execution efficiency. However, combining these signals introduces weighting decisions that are rarely objective. The resulting policy reflects the priorities encoded by the designers, which may not align with the preferences of end users or the long-term maintainability of the software.

Another trade-off involves the granularity of feedback. Token-level reward shaping can provide denser learning signals, but it risks interfering with the model's learned syntax and semantics if the shaping is poorly calibrated. Sequence-level rewards are simpler and more compatible with execution-based evaluation, but they suffer from high variance because many sampled programs receive zero reward. Intermediate approaches that assign credit through execution paths or reasoning trajectories have shown promise, particularly in small parameter models where training efficiency is critical [14]. These trajectory-aware methods attempt to distinguish useful reasoning steps from irrelevant or harmful ones. Such approaches reduce compute requirements and improve sample efficiency, but they introduce additional complexity in the filtering logic and may discard diverse but valid solutions.

The alignment trade-offs extend to the interaction between reinforcement learning and supervised fine-tuning. Many systems first fine-tune the model on high-quality code datasets and then apply reinforcement learning to refine behavior. This staged approach is practical because it combines stable imitation with adaptive exploration. Yet the two stages can conflict. Reinforcement learning may cause catastrophic forgetting of general coding knowledge if the reward is too narrow. Conversely, a strong supervised prior may limit exploration and make the policy reluctant to generate novel repairs. System designers must monitor the balance between retaining broad competence and specializing toward task-specific quality.

5. Code Generation: Structural and Reliability Considerations

Code generation with reinforcement learning differs from natural language generation because the output must satisfy formal constraints and executable semantics. A generated program that is syntactically plausible but semantically incorrect has little value in production settings. Execution-based reward addresses this by directly testing the generated code. However, execution outcomes are noisy. Tests may be incomplete, environment dependencies may be missing, and nondeterministic behavior may cause inconsistent rewards. The system must therefore treat execution signals as probabilistic evidence rather than ground truth. Robust reward functions may aggregate results across multiple test suites, random inputs, or property-based testing frameworks.

Structural reliability also depends on the model's ability to compose programs from reusable components. Large language models trained on code often memorize common patterns and libraries [5, 15, 16]. Reinforcement learning can reinforce these patterns when they lead to successful execution, but it may also reduce the diversity of generated solutions. This is a significant concern for software engineering because diverse solutions are valuable for exploring design alternatives and avoiding systemic weaknesses. A policy that becomes overly narrow may fail on unusual but valid specifications. Maintaining diversity requires explicit mechanisms such as entropy regularization, multiple sampling, or reward bonuses for novel correct solutions. Without such mechanisms, reinforcement learning can produce monocultures of code that are efficient on known tests but fragile in real-world conditions.

Another structural issue is the granularity of generation. Many practical systems generate function-level or class-level code rather than complete applications. This imposes fewer demands on long-range coherence but creates integration challenges. The generated

component must match the surrounding interfaces, naming conventions, and error handling patterns. Reinforcement learning that only optimizes component-level tests may not account for these integration constraints. System architects can address this by including integration tests in the reward signal or by training separate policies for component generation and interface adaptation. Cross-domain comparisons with natural language generation are instructive here. In text generation, a fluent sentence may be acceptable even if stylistically different from its context. In code generation, a correct function with an incompatible signature may break the entire build. Thus, code-oriented reinforcement learning requires a stronger coupling between evaluation and system integration.

6. Program Repair and Continuous Maintenance

Program repair is a particularly demanding application of reinforcement learning because it requires models to modify existing code without introducing regressions. The action space is not simply the generation of a new program but the selection of an edit that preserves intended behavior while eliminating a defect. Repair tasks often involve subtle logical errors, concurrency issues, or resource leaks that are not captured by a single failing test. In such settings, the reward must reflect both the resolution of the fault and the absence of new failures. This dual objective can be encoded through regression test suites and code similarity constraints [7, 8].

A central challenge in learned program repair is credit assignment across large codebases. A patch may involve changes to multiple locations, and the contribution of each change to the overall repair is difficult to isolate. Reinforcement learning can in principle learn to associate contextual states with beneficial edits, but the search space is vast. Systems often reduce this space by representing edits as small abstract transformations or by conditioning the model on bug localization outputs. The structural trade-off is between the flexibility of free-form edits and the tractability of constrained edit spaces. Unconstrained generation can produce more general repairs, but it increases the risk of irrelevant or harmful modifications. Constrained approaches are safer but may miss repairs that require broader refactoring.

Continuous maintenance introduces temporal dynamics that are absent from static code generation benchmarks. Software evolves through feature additions, dependency updates, and shifting performance requirements. A repair policy trained on one version of a codebase may not transfer to a later version. This creates a need for periodic retraining and policy evaluation under distribution shift. Organizations must implement monitoring systems that track repair success rates, regression rates, and model confidence over time. When the distribution shifts, the reward signal may become stale, and the policy may begin to produce low-quality patches. Governance processes should therefore include periodic audits and staged rollout procedures for new policy versions.

7. Infrastructure, Deployment, and Runtime Governance

Deploying reinforcement learning-based code generation and repair systems at scale requires substantial infrastructure investment. Inference is computationally expensive, especially when sampling many candidate programs during training. Deployment serving may also require high throughput and low latency if the system is integrated into developer tools or continuous integration pipelines. These demands create resource allocation trade-offs. Organizations may choose to deploy smaller distilled models for latency-sensitive tasks while reserving larger models for offline repair or complex generation. Techniques that reduce reasoning path

overhead in small models are relevant here because they can improve the practicality of deployment [14].

The execution environment is a critical infrastructure component. Safe execution of model-generated code is necessary to prevent damage to host systems and to ensure consistent reward signals. Containerization, virtual machines, and sandboxed interpreters provide isolation, but they introduce overhead and complexity. Some code may require network access, external packages, or system-level resources, which increases security risk. A governance framework must specify which capabilities are allowed during training and inference. Policies should distinguish between untrusted generated code and trusted test infrastructure, and they should enforce resource limits, timeout policies, and audit logging.

Monitoring is another governance requirement. A reinforcement learning system is non-deterministic and may drift after deployment. Metrics such as test pass rates, repair acceptance rates, false positive patches, and execution time should be tracked continuously. When these metrics degrade, the system should trigger retraining or fallback to conservative baselines. Model updates should follow staged deployment processes that include shadow testing, canary releases, and human review. For program repair, an additional control is the requirement that every automatic patch pass a mandatory regression suite before being proposed to a human developer. This does not eliminate risk, but it provides a structured boundary for autonomous action.

8. Fairness, Robustness, and Policy Implications

The integration of reinforcement learning into code generation and repair raises fairness and robustness concerns that extend beyond technical accuracy. Foundation models can encode biases present in their training data, and these biases may affect code recommendations, naming conventions, documentation, and structural decisions [19]. If the reward model is trained on biased human feedback, reinforcement learning may amplify those biases instead of correcting them. For example, a reward function that favors certain coding styles may penalize equally valid but culturally different practices. Fairness in code-oriented systems is therefore not only about demographic parity but also about procedural fairness and representational diversity across programming communities and application domains.

Robustness is similarly multifaceted. A code generation policy may perform well on benchmark tasks but fail on adversarial inputs, rare programming languages, or legacy systems. The learned policy may exploit spurious correlations in test suites, such as variable names or file paths, rather than understanding the underlying problem. In program repair, a policy could learn to delete failing assertions or bypass error handling to achieve a pass, which is a harmful outcome. Such failure modes highlight the need for adversarial evaluation and red teaming. System designers should incorporate diverse test cases, mutation testing, and human review to expose these weaknesses before deployment.

Policy implications arise from the increasing autonomy of AI-assisted software engineering. If organizations allow reinforcement learning systems to propose and even auto-apply patches, the lines of accountability become blurred. It is important to maintain human oversight, but the exact placement of that oversight is a governance decision. Some organizations may restrict autonomous action to low-risk modules, while others may allow broader autonomy with strong audit trails. Regulatory frameworks for software accountability are still evolving, and automated repair systems will need to demonstrate traceability, explainability, and reliability. Ethical risk assessments should be conducted before deployment, with particular

attention to safety-critical domains such as medical software, transportation systems, and financial infrastructure [20].

9. Sustainability and Long-Term Evolution

Sustainability in reinforcement learning-based code systems has both environmental and organizational dimensions. The computational cost of training large language models and executing large-scale sampling is substantial. Even inference-time search and reward evaluation can consume considerable energy if not carefully managed. Trajectory filtering and small-model alignment techniques may reduce some of this burden by improving sample efficiency [14]. However, sustainability also requires avoiding unnecessary retraining and designing modular systems that can reuse previously learned components. Organizations should evaluate whether the benefits of continuous reinforcement learning justify the energy and infrastructure costs, particularly for tasks that could be solved with deterministic program analysis or simpler heuristics.

Long-term evolution is shaped by the accumulation of feedback data and the shifting nature of software ecosystems. A reinforcement learning system that is updated continuously can adapt to new languages, frameworks, and security requirements. This adaptivity is valuable, but it also creates a moving target for evaluation and certification. The system may become difficult to audit because its behavior changes over time. To address this, organizations should adopt versioning practices analogous to software configuration management. Each policy version should be associated with its training data, reward function, evaluation results, and known limitations. These artifacts support accountability and allow rollback if a new policy introduces regressions.

Finally, sustainability depends on the development of shared evaluation standards and open benchmarks. The field benefits from transparency about failure cases and reward design. Without shared standards, it becomes difficult to compare systems or to identify systematic risks. Academic and industrial stakeholders should collaborate to define evaluation protocols that include security, fairness, maintainability, and long-term repair quality. Such efforts will help align technical progress with broader societal values and ensure that reinforcement learning-based code systems are not only accurate but also responsible.

10. Conclusion

Reinforcement learning offers a powerful mechanism for improving code generation and program repair by aligning models with execution feedback and human preferences. This paper has examined the system-level implications of integrating such methods into large language model pipelines. The discussion highlighted architectural choices, reward design trade-offs, infrastructure requirements, and governance challenges. It also emphasized that code-oriented reinforcement learning differs from text generation because of the need for formal correctness, integration compatibility, and regression safety. Program repair further complicates the system by requiring conservative modifications within evolving software systems. Deployment at scale demands robust execution environments, continuous monitoring, and staged rollout processes. Fairness, robustness, and accountability must be treated as first-order design concerns rather than post-hoc considerations. The path forward lies in interdisciplinary collaboration across machine learning, software engineering, systems architecture, and policy. By addressing these structural and organizational issues, the research community can build reinforcement learning-based tools that are effective, sustainable, and aligned with the long-term needs of software-intensive societies.

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.

2. Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. Proceedings of NAACL-HLT 2019, 4171–4186.

3. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., 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.

4. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research, 21(140), 1–67.

5. 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 preprint arXiv:2107.03374.

6. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, T., Terry, M., Le, Q., & Sutton, C. (2021). Program synthesis with large language models. arXiv preprint arXiv:2108.07732.

7. Le, H., Wang, Y., Gotmare, A. D., Savarese, S., & Hoi, S. C. H. (2022). CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. Advances in Neural Information Processing Systems, 35, 21314–21328.

8. Shojaee, P., Jain, A., Tipirneni, S., & Reddy, C. K. (2023). Execution-based code generation using deep reinforcement learning. Transactions on Machine Learning Research.

9. Ziegler, D. M., Stiennon, N., Wu, J., Brown, T. B., Radford, A., Amodei, D., Christiano, P., & Irving, G. (2019). Fine-tuning language models from human preferences. arXiv preprint arXiv:1909.08593.

10. Stiennon, N., Ouyang, L., Wu, J., Ziegler, D. M., Lowe, R., Voss, C., Radford, A., Amodei, D., & Christiano, P. (2020). Learning to summarize with human feedback. Advances in Neural Information Processing Systems, 33, 3008–3021.

11. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., & Lowe, R. (2022). Training language models to follow instructions with human feedback. Advances in Neural Information Processing Systems, 35, 27730–27744.

12. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

13. 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.

14. Li, Q. (2026, May). TrajLite PPO: Scalable Reasoning Path Filtering and Alignment for Small Parameter Large Language Models. In 2026 2nd International Conference on Artificial Intelligence and Digital Ethics (ICAIDE) (pp. 29-32). IEEE.

15. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., ... Synnaeve, G. (2023). Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.

16. Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., ... Le, N. (2023). StarCoder: May the source be with you! arXiv preprint arXiv:2305.06161.

17. Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., & Xiong, C. (2023). CodeGen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474.

18. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., ... Scialom, T. (2023). Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288.

19. Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., Brynjolfsson, E., Buch, S., Card, D., Castellon, R., Chatterji, N., Chen, A., Creel, K., Davis, J. Q., Demszky, D., ... Liang, P. (2021). On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258.

20. Weidinger, L., Mellor, J., Rauh, M., Griffin, C., Uesato, J., Huang, P.-S., Cheng, M., Glaese, M., Balle, B., Kasirzadeh, A., Kenton, Z., Brown, S., Hawkins, W., Stepleton, T., Biles, C., Birhane, A., Haas, J., Rimell, L., Hendricks, L. A., ... Isaac, W. (2021). Ethical and social risks of harm from language models. arXiv preprint arXiv:2112.04359.