

Reinforcement Learning-Based Code Reasoning Optimization for Small Language Models

Gaurika Rau

Department of Computer Science, University of Central Florida, Orlando, FL, USA.
rau1989@ucf.edu

Rian Boreland

Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS, USA.
rian.boreland344@ku.edu

Chantelle R. Ternan

Department of Computer Science, George Mason University, Fairfax, VA, USA.
chantelle.ternan723@gmu.edu

Lisanne van Ginneke

Department of Computer Science, Binghamton University, Binghamton, NY, USA.
vanginnekelisanne@binghamton.edu

Abstract

Small language models are becoming a practical alternative to very large models for code reasoning, especially in settings where latency, privacy, cost, and local deployment matter. However, their limited parameter budgets make multi-step code reasoning fragile, because errors in intermediate reasoning paths can cascade into incorrect programs. This paper examines reinforcement learning-based optimization as a system-level strategy for improving code reasoning in small language models. It argues that the central challenge is not merely reward maximization but the design of a socio-technical infrastructure that filters reasoning trajectories, aligns model behavior with software engineering constraints, and remains stable across deployment conditions. The paper analyzes architectural trade-offs between policy optimization, reward modeling, retrieval, and verification. It discusses data governance, environmental sustainability, robustness, fairness, and policy implications. It also compares small-model optimization with repository-level code assistance, program-aided reasoning, and tool-augmented workflows. The analysis suggests that reinforcement learning can improve small-model code reasoning only when paired with careful trajectory filtering, transparent evaluation, and operational safeguards. Without such safeguards, optimization may reward superficial correctness, increase hidden technical debt, or disadvantage users in under-resourced environments. The paper concludes that sustainable code reasoning optimization requires an integrated view of model architecture, reward design, deployment infrastructure, and governance.

Keywords

small language models; reinforcement learning; code reasoning; alignment; software engineering; deployment governance; fairness; sustainability; model optimization.

1. Introduction

The rapid growth of language models has created a paradox for software engineering. Larger models often demonstrate stronger code generation and reasoning, but their computational demands, privacy risks, and operational costs limit deployment in many organizations. Small language models offer lower latency, easier local hosting, and greater control over data flows. Yet they frequently struggle with code reasoning tasks that require multi-step planning, dependency tracking, and error correction. Reinforcement learning has therefore emerged as an attractive method for improving small-model behavior without proportionally increasing parameter count [1]. By optimizing model outputs against task-specific rewards, reinforcement learning can encourage code generation strategies that are more reliable than those obtained through supervised fine-tuning alone [2]. The challenge is that code reasoning is not a simple sequence prediction problem. It involves structural constraints, execution semantics, repository context, and human review practices. A small model optimized only for passing visible tests may learn shortcuts that fail under real maintenance conditions.

This paper positions reinforcement learning-based code reasoning optimization as a systems problem rather than a narrow modeling problem. The relevant system includes the policy model, the reward model, the data pipeline, the execution sandbox, the retrieval layer, the deployment environment, and the governance regime. Each component changes the incentives that shape model behavior. For example, proximal policy optimization and related methods provide stable update mechanisms for language model alignment [3], while human feedback alignment has shown that reward models can shape instruction following and safety [4]. Chain-of-thought prompting has demonstrated that explicit intermediate reasoning can improve performance on complex tasks [5], but it also increases the number of tokens that must be generated and evaluated. Code benchmarks have revealed that functional correctness remains difficult even for models trained on large code corpora [6]. Small models thus face a tighter trade-off between reasoning depth, inference cost, and reliability.

The central argument of this paper is that reinforcement learning can materially improve code reasoning in small language models, but only if the optimization pipeline treats reasoning trajectories as governed artifacts rather than disposable text. Reasoning paths must be filtered, scored, audited, and connected to downstream software engineering practices. Otherwise, optimization may produce brittle improvements that satisfy benchmark metrics while undermining maintainability, fairness, and trust. The paper develops this argument through an analysis of architecture, deployment, sustainability, robustness, and policy. It emphasizes that small models are not merely scaled-down versions of large models. Their constraints create distinct design opportunities, especially when reinforcement learning is combined with retrieval, execution feedback, and local verification.

2. Foundations of Reinforcement Learning and Code Reasoning

Reinforcement learning provides a formal framework for sequential decision making under uncertainty [1]. In language model alignment, the sequence of tokens can be treated as a trajectory, and the reward signal can reflect correctness, style, safety, or user preference. Deep reinforcement learning demonstrated that high-dimensional inputs can be mapped to effective actions through neural approximation [2]. Proximal policy optimization later became widely used because it balances sample efficiency with stable policy updates [3]. In code reasoning, the action space is not only the next token but also the choice of reasoning strategy, the decision to invoke a tool, the selection of retrieved context, and the willingness to revise an earlier step. A small model that can learn when to simplify, when to verify, and when to request additional context may outperform a larger model that reasons indiscriminately.

Program synthesis research has long explored how models can generate code from natural language intent [7]. Early large-scale code models showed that training on public code repositories improves generation quality, but they also exposed limitations in generalization and correctness [6]. CodeGen extended this direction by treating program synthesis as a multi-turn interaction problem [8]. InCoder introduced infilling capabilities that support code completion and synthesis within existing files [9]. Code Llama demonstrated that open foundation models can be specialized for code tasks and deployed in more controlled environments [10]. CodeRL combined pretrained code models with deep reinforcement learning to improve program synthesis through execution feedback [11]. These developments establish a foundation for reinforcement learning-based code reasoning, but they also reveal that reward design is as important as model architecture.

A recent investigation of scalable reasoning path filtering and alignment for small parameter large language models [12] illustrates the specific tension that motivates this paper. Small models can generate many possible reasoning paths, but only a subset are useful for learning. If all paths are used for alignment, noisy trajectories can reinforce incorrect habits. If too few paths are used, the model may overfit to narrow success cases. The optimization problem therefore involves selection, weighting, and sequencing of reasoning traces. The system must decide which trajectories are educationally valuable, which are merely lucky, and which are dangerous because they produce correct outputs through invalid reasoning. This distinction is especially important in code, where a program may pass a small test suite while violating security, performance, or maintainability requirements.

3. Small Language Models and the Code Reasoning Bottleneck

Small language models face a structural bottleneck in code reasoning because they must compress syntax, semantics, library knowledge, and planning ability into fewer parameters. They often lack the broad world knowledge that helps large models recover from ambiguity. They may also have weaker instruction-following after quantization or distillation. Tree of thoughts and related deliberate search methods show that reasoning quality can improve when models explore multiple candidate paths [13]. Program-aided language models show that offloading computation to external tools can reduce reasoning errors [14]. Repository-level completion systems show that retrieval and iterative generation can compensate for limited parametric memory [15]. For small models, these techniques are not optional enhancements; they are often necessary scaffolds.

The bottleneck is not only parameter count but also context handling. Code reasoning requires the model to track variable definitions, function calls, data types, error handling, and side effects across files. A small model may excel at local completion but fail when asked to modify a complex repository. Reinforcement learning can help by rewarding successful repository-level edits, but the reward must be computed over long horizons. If the reward is delayed until final execution, credit assignment becomes difficult. If intermediate rewards are too dense, the model may optimize for superficial signals such as formatting or comment generation. The design challenge is to create reward structures that align with human software engineering judgment without requiring unrealistic amounts of expert annotation.

Another bottleneck is the evaluator. Small models are often deployed in environments with limited test coverage. They may be used by students, small teams, or public-sector organizations that cannot maintain extensive continuous integration. In such settings, reinforcement learning may inadvertently optimize for the available tests rather than for general correctness. A model that learns to pass hidden tests may still produce fragile code.

This risk is heightened when the reward model is trained on data that reflects the practices of well-resourced technology companies but not the constraints of regulated industries, legacy systems, or educational contexts. Fairness in code reasoning therefore includes fairness in evaluation infrastructure.

4. Reinforcement Learning-Based Optimization Architecture

A reinforcement learning architecture for small-model code reasoning must coordinate several layers. The policy model generates candidate reasoning paths and code edits. The reward model evaluates these outputs using tests, static analysis, human feedback, or learned preferences. The trajectory filter selects which experiences enter the training buffer. The execution sandbox provides safe feedback. The retrieval layer supplies repository context and documentation. The deployment layer controls inference, caching, and monitoring. The governance layer defines acceptable use, audit requirements, and escalation procedures. This architecture resembles the broader machine learning systems view in which hidden technical debt accumulates across data dependencies, configuration, and monitoring [19]. For code reasoning, technical debt can appear as brittle reward models, stale retrieval indexes, or test suites that no longer reflect production behavior.

The transformer architecture remains the dominant substrate for these models [20]. Its attention mechanism supports long-range dependencies, but small models have limited capacity to attend to every relevant token. Reinforcement learning can teach the model to allocate attention-like behavior through prompting and trajectory selection, but it cannot remove the underlying resource constraint. The policy update must therefore be conservative. Aggressive optimization can cause the model to forget general coding ability while specializing in a narrow reward signal. Proximal policy optimization and related methods help stabilize updates [3], but stability at the optimization level does not guarantee stability at the system level. A model may remain stable during training yet become unreliable when the retrieval index changes, the execution environment updates, or users shift to a new programming language.

The architecture must also decide where verification occurs. In one design, the model generates code and the sandbox executes tests before reward assignment. In another, the model generates reasoning steps and a verifier checks intermediate claims. In a third, a human reviewer provides feedback on selected trajectories. Each design has trade-offs. Execution-based verification is objective but narrow. Static analysis is broader but may produce false positives. Human feedback is rich but expensive and inconsistent. For small models, hybrid verification is often more practical than relying on a single signal. The optimization objective should reward not only final correctness but also evidence of good reasoning, such as explicit assumptions, testable intermediate claims, and appropriate uncertainty. However, these qualities are difficult to measure and can be gamed if the reward model is not carefully governed.

5. Data, Reward Modeling, and Reasoning Path Filtering

Reasoning path filtering is one of the most consequential design choices in reinforcement learning for code reasoning. A small model may produce dozens of candidate paths for a single problem. Some paths are correct and concise. Some are correct but inefficient. Some are incorrect but superficially plausible. Some are dangerous because they exploit test gaps or produce insecure code. If the training pipeline treats all successful outputs as equally valuable, it may reinforce undesirable reasoning patterns. If it discards all failed outputs, it loses

information about common errors and recovery strategies. A more sophisticated filter can classify trajectories by correctness, efficiency, security, readability, and generalization. This classification supports curriculum learning, where the model first learns simple patterns and later encounters harder cases.

Reward modeling for code reasoning must confront the difference between human preference and functional correctness. Human feedback alignment has been successful in instruction following [4], but code quality is often evaluated through execution, review, and maintenance. A reward model trained on human preferences may favor code that looks familiar rather than code that is robust. It may also inherit biases from the reviewers, such as preferences for certain libraries or coding styles. For small models, these biases can become amplified because the model has less capacity to represent alternative solutions. A balanced reward model should incorporate multiple signals, including test outcomes, static analysis warnings, complexity metrics, and expert review. Yet adding signals increases the complexity of reward aggregation and makes it harder to explain why a particular trajectory was selected.

Chain-of-thought reasoning provides a useful source of intermediate supervision [5], but it also creates a privacy and security surface. Reasoning traces may contain sensitive information from the repository, internal documentation, or user prompts. If these traces are stored for reinforcement learning, they become part of the training infrastructure and must be governed accordingly. CodeRL shows that execution feedback can improve program synthesis [11], but execution traces may also reveal proprietary logic. A small model deployed locally may reduce data exposure, but local training pipelines still require access controls, retention limits, and audit logs. Reasoning path filtering should therefore be understood as both a machine learning operation and a data governance operation.

6. Deployment, Infrastructure, and Sustainability Trade-Offs

Deployment conditions shape what counts as a good optimization outcome. A small model running on a developer laptop has different constraints from a model running in a shared cloud service. The laptop model must minimize memory and battery use. The cloud model must balance throughput, latency, and cost across many users. A public-sector deployment may require explainability and procurement compliance. An educational deployment may prioritize pedagogical feedback over raw productivity. Reinforcement learning can optimize for each context, but separate reward models increase operational complexity. If the same model is deployed across contexts, the reward function must be general enough to avoid penalizing legitimate variation. This generality is difficult to achieve with small models because they have less capacity to represent context-specific norms.

Sustainability is a central but often overlooked consideration. Training and fine-tuning language models consume energy, water, and hardware resources. Small models are generally more sustainable than large models, but reinforcement learning can increase total compute because it requires repeated sampling and evaluation. If the reward model itself is large, the sustainability advantage of the policy model may shrink. A system-level approach should measure the full lifecycle cost, including data collection, annotation, reward model training, policy optimization, deployment, and retirement. It should also consider rebound effects, where efficiency gains lead to more usage rather than lower total impact. For code reasoning, the most sustainable optimization may be one that improves sample efficiency, reduces unnecessary generation, and uses lightweight verifiers instead of large reward models.

Infrastructure choices also affect robustness. A model that depends on a remote execution sandbox may fail when network connectivity is poor. A model that relies on a vector database may produce stale answers if the index is not updated. A model that uses cached reasoning paths may become inconsistent when the repository changes. These are not merely engineering details; they determine whether reinforcement learning improvements persist in practice. The hidden technical debt framework warns that machine learning systems accumulate maintenance burdens in data dependencies, configuration, and monitoring [19]. Code reasoning systems are especially vulnerable because they combine software artifacts, model weights, retrieval indexes, and execution environments. A deployment that does not monitor these interactions can experience silent degradation even if the model parameters remain unchanged.

7. Robustness, Fairness, and Governance

Robustness in code reasoning requires more than passing tests. A robust model should handle ambiguous requirements, incomplete specifications, and adversarial inputs. It should recognize when it lacks sufficient context. It should avoid generating code that introduces security vulnerabilities or violates licensing constraints. Reinforcement learning can reward robustness if the training environment includes diverse and challenging scenarios, but it can also produce overconfidence if the reward model consistently favors fluent outputs. Small models are particularly susceptible because they may not have the capacity to represent uncertainty well. Governance mechanisms should therefore include red teaming, adversarial evaluation, and human review for high-stakes deployments.

Fairness is a multidimensional concern. Small models can democratize access to code assistance by running on inexpensive hardware and supporting local languages or domain-specific programming environments. However, reinforcement learning may optimize for the dominant languages, frameworks, and coding styles represented in the training data. This can disadvantage developers who work with underrepresented languages, legacy systems, or non-Western software ecosystems. The dangers of large-scale language models include the reproduction of harmful biases and the concentration of power in a few institutions [16]. Foundation model research has similarly highlighted the risks of broad deployment without adequate accountability [17]. For small code models, fairness also includes access to evaluation infrastructure. A model optimized on well-funded test suites may perform poorly for users who cannot provide equivalent feedback.

Governance should make these trade-offs visible. Model cards can document intended use, limitations, evaluation data, and performance across groups [18]. For code reasoning, documentation should include the programming languages covered, the types of repositories used in training, the reward signals employed, and the known failure modes. It should also describe how reasoning paths are filtered and whether human feedback was used. Without such transparency, users cannot know whether a model's confidence reflects genuine competence or merely optimization against a narrow reward. Governance is not a bureaucratic addition to reinforcement learning; it is a condition for trustworthy deployment. It determines which trajectories are considered acceptable, how disputes are resolved, and who bears responsibility when generated code causes harm.

8. Case Illustrations and Cross-Domain Comparisons

Consider a small model deployed within a university programming course. The model is fine-tuned with reinforcement learning to generate explanations and code for introductory

assignments. The reward model values correct output, clear reasoning, and pedagogical hints. If the reward model overemphasizes correct output, students may receive complete solutions instead of learning support. If it overemphasizes hints, students may struggle to complete assignments. The optimization must therefore balance educational goals with functional correctness. Reasoning path filtering can help by selecting explanations that show productive struggle rather than polished but opaque answers. This case illustrates that code reasoning optimization is always situated in a social context.

Now consider a small model used in a regulated financial institution. The model assists with internal scripts that process sensitive data. Deployment must be local or within a controlled cloud environment. The reward model cannot rely on public code execution because the code may touch production data. Verification must use synthetic data, static analysis, and human approval. Reinforcement learning can still improve the model, but the trajectory filter must exclude any path that accesses restricted resources. The model must also produce audit logs that explain why a particular code suggestion was made. In this setting, the value of a small model lies not only in performance but in controllability. A larger model may be more capable, but it may be unacceptable because its data flows cannot be governed.

Cross-domain comparison with repository-level code completion shows similar patterns. RepoCoder demonstrates that iterative retrieval and generation can improve completion by incorporating repository context [15]. Program-aided language models show that external computation can reduce reasoning errors [14]. CodeGen and InCoder show that code-specific pretraining supports synthesis and infilling [8][9]. Code Llama shows that open models can be adapted for code tasks [10]. These approaches complement reinforcement learning. A small model optimized with reinforcement learning can learn when to retrieve, when to call a tool, and when to ask for clarification. However, the reward model must value appropriate tool use rather than rewarding tool calls indiscriminately. The system-level lesson is that optimization should reinforce good workflow behavior, not just final answers.

9. Evaluation, Metrics, and Operational Validation

Evaluation of reinforcement learning-based code reasoning must go beyond pass rates. It should measure correctness, efficiency, security, readability, and generalization. It should also measure the cost of achieving those outcomes, including inference latency, energy use, and human review time. Small models may achieve acceptable correctness on narrow benchmarks but fail on repository-level tasks. They may perform well on function-level synthesis but poorly on debugging or refactoring. Evaluation should therefore include multiple task types and deployment scenarios. It should also include negative testing, where the model is given incomplete or contradictory requirements. The goal is not to prove that the model is universally competent but to understand its operating envelope.

Operational validation should examine the interaction between the policy model, reward model, and environment. A model may improve during training but degrade when the retrieval index changes. A reward model may become stale as programming languages evolve. A trajectory filter may exclude underrepresented but valid solutions. These risks require continuous monitoring, versioning, and rollback procedures. Model cards can provide a baseline for transparency [18], but they must be updated as the system changes. The hidden technical debt perspective reminds us that machine learning systems are not static artifacts; they are evolving infrastructures with dependencies and feedback loops [19]. For code reasoning, this means that evaluation must be integrated into the software development lifecycle rather than treated as a one-time benchmark.

Metrics should also account for fairness. Performance should be reported across programming languages, repository sizes, and user groups. If a small model performs well only on popular frameworks, that limitation should be documented. If reinforcement learning improves average performance but widens disparities, the optimization design should be revisited. Fairness-aware reward models may need to include penalties for harmful disparities, but such penalties must be carefully designed to avoid unintended consequences. The evaluation process should involve domain experts, including software engineers, security analysts, and educators. Their feedback can reveal failure modes that automated tests miss. In this sense, evaluation is not merely technical; it is a governance activity that shapes what the system is allowed to optimize.

10. Policy Implications and Future Directions

Policy for small-model code reasoning should encourage transparency, accountability, and equitable access. Regulators may require documentation of training data, reward signals, and evaluation results for models used in critical software. Procurement rules may favor local deployment and open standards to reduce dependency on a few vendors. Educational institutions may need guidelines for using code assistants in assessment. Open-source communities may need licenses that clarify how model-generated code can be used. These policy questions are not separate from reinforcement learning; they define the incentives that shape optimization. If policy rewards short-term productivity without considering maintenance and security, developers may deploy models that create long-term costs.

Future research should explore hybrid reward models that combine execution, static analysis, human feedback, and formal verification. It should investigate trajectory filtering methods that preserve diverse reasoning strategies while removing harmful noise. It should study how small models can learn from large models without inheriting their biases or unsustainable compute costs. It should also examine the environmental impact of reinforcement learning pipelines and develop metrics for sustainable optimization. The required reference on scalable reasoning path filtering and alignment for small parameter models [12] points toward this direction, but much work remains. The field needs longitudinal studies that track deployed systems over time, not just benchmark improvements.

Another important direction is participatory design. Developers, students, auditors, and affected communities should have a voice in defining what counts as good code reasoning. Reinforcement learning optimizes what is measured, and measurement is a social choice. If only speed and pass rates are measured, the model will optimize for those values. If maintainability, accessibility, and security are included, the optimization landscape changes. Small models are well suited to participatory approaches because they can be hosted locally and adapted to specific communities. However, local adaptation also creates fragmentation and governance challenges. Future work should develop frameworks for shared evaluation and coordinated improvement without centralizing control.

11. Conclusion

Reinforcement learning-based code reasoning optimization for small language models is a promising but demanding systems endeavor. Small models offer important advantages in cost, privacy, latency, and local control, but their limited capacity makes reasoning quality fragile. Reinforcement learning can improve their behavior by rewarding correct, efficient, and well-reasoned code. Yet the success of such optimization depends on trajectory filtering, reward design, verification infrastructure, and governance. If these elements are neglected,

optimization may reward superficial correctness, increase hidden technical debt, and create unfair outcomes. The paper has argued for an integrated approach that treats reasoning paths as governed artifacts, evaluates models across diverse contexts, and considers sustainability and fairness as first-class design constraints. Small language models for code reasoning should not be judged only by benchmark performance. They should be evaluated by their ability to support trustworthy, maintainable, and equitable software practice over time.

References

1. Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.). MIT Press.
2. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529-533. <https://doi.org/10.1038/nature14236>
3. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv*. <https://arxiv.org/abs/1707.06347>
4. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., 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.
5. 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.
6. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv*. <https://arxiv.org/abs/2107.03374>
7. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021). Program synthesis with large language models. *arXiv*. <https://arxiv.org/abs/2108.07732>
8. 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. *International Conference on Learning Representations*.
9. Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W., Zettlemoyer, L., & Lewis, M. (2023). InCoder: A generative model for code infilling and synthesis. *International Conference on Learning Representations*.
10. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozlov, A., Evtimov, I., Bitton, J., Bhatt, M., Canton-Ferrer, C., Grangier, D., Schwab, D., Esposito, D., Singh, S., ... Synnaeve, G. (2023). Code Llama: Open foundation models for code. *arXiv*. <https://arxiv.org/abs/2308.12950>

11. 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.
12. 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.
13. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023). Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 11809-11822.
14. Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., & Neubig, G. (2023). PAL: Program-aided language models. *International Conference on Machine Learning*, 202, 10764-10799.
15. Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.-G., & Chen, W. (2023). RepoCoder: Repository-level code completion through iterative retrieval and generation. *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2471-2484.
16. 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.
17. 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*. <https://arxiv.org/abs/2108.07258>
18. Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I. D., & Gebru, T. (2019). Model cards for model reporting. *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 220-229.
19. 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.
20. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 5998-6008.