Review Article

Enhancing Code LLMs with Reinforcement Learning in Code Generation: A Survey

Yangfan He[1], Tianyu Shi[2], Yuchen Li[3], Qiuwu Chen[3]

1. University of Minnesota - Twin Cities, United States; 2. University of Toronto, Canada; 3. AIGCode

With the rapid evolution of large language models (LLM), reinforcement learning (RL) has emerged as a pivotal technique for code generation and optimization in various domains. This paper presents a systematic survey of the application of RL in code optimization and generation, highlighting its role in enhancing compiler optimization, resource allocation, and the development of frameworks and tools. Subsequent sections first delve into the intricate processes of compiler optimization, where RL algorithms are leveraged to improve efficiency and resource utilization. The discussion then progresses to the function of RL in resource allocation, emphasizing register allocation and system optimization. We also explore the burgeoning role of frameworks and tools in code generation, examining how RL can be integrated to bolster their capabilities. This survey aims to serve as a comprehensive resource for researchers and practitioners interested in harnessing the power of RL to advance code generation and optimization techniques.

Corresponding author: Tianyu Shi, tianyu.s@outlook.com

1. Introduction

As software systems grow more complex with tighter development timelines, manual code development and optimization become impractical to a certain extent. Code generation and optimization from natural language (NL) have thus become essential for boosting software development efficiency[1][2]. Meanwhile, advances in natural language processing (NLP), particularly in large language models (LLMs), have opened new possibilities for code generation. Compiler optimizations are essential in enhancing software performance and reducing resource consumption. Conventional compiler optimization relies on techniques like autotuning[3], while deep learning

approaches for optimized compiler sequences[4] struggle with generalization. Although large language models (LLMs) improve code generation and optimization[5], they often produce biased or inconsistent outputs[6][7] and require time-consuming pre-training with code-specific models like Code T5[8] and Code T5+[9]. In Figure 1, we present a schematic diagram of memory management, wherein the main controller selects specific strategies based on the constraints. Subsequently, it interacts with relevant hardware components such as the compiler and configures the registers, thereby achieving an overall optimization effect.

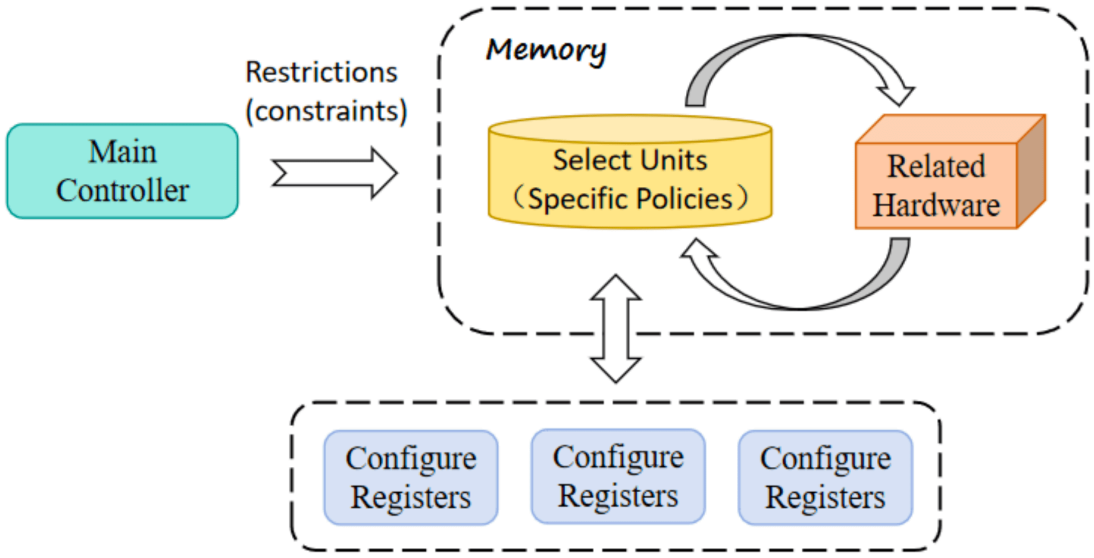

Figure 1. The controller is in control of the resources.

Current Code LLM research is more focused on the pre-training of code-related corpora. Reinforcement learning (RL), as a method that can learn the optimal strategy in complex environments, provides a new approach to code generation[10], and optimization[11] in Code LLMs. It allows label-free input-output pairs and leverages existing knowledge and refining strategies through trial and error. The advantages of using RL in code optimization and compiler enhancement lie in its capacity to reduce reliance on pre-trained models and to enable large language models (LLMs) to adapt more flexibly to evolving environmental conditions. Figure 2 illustrates related work on reinforcement learning applied to compiler optimization and code generation.

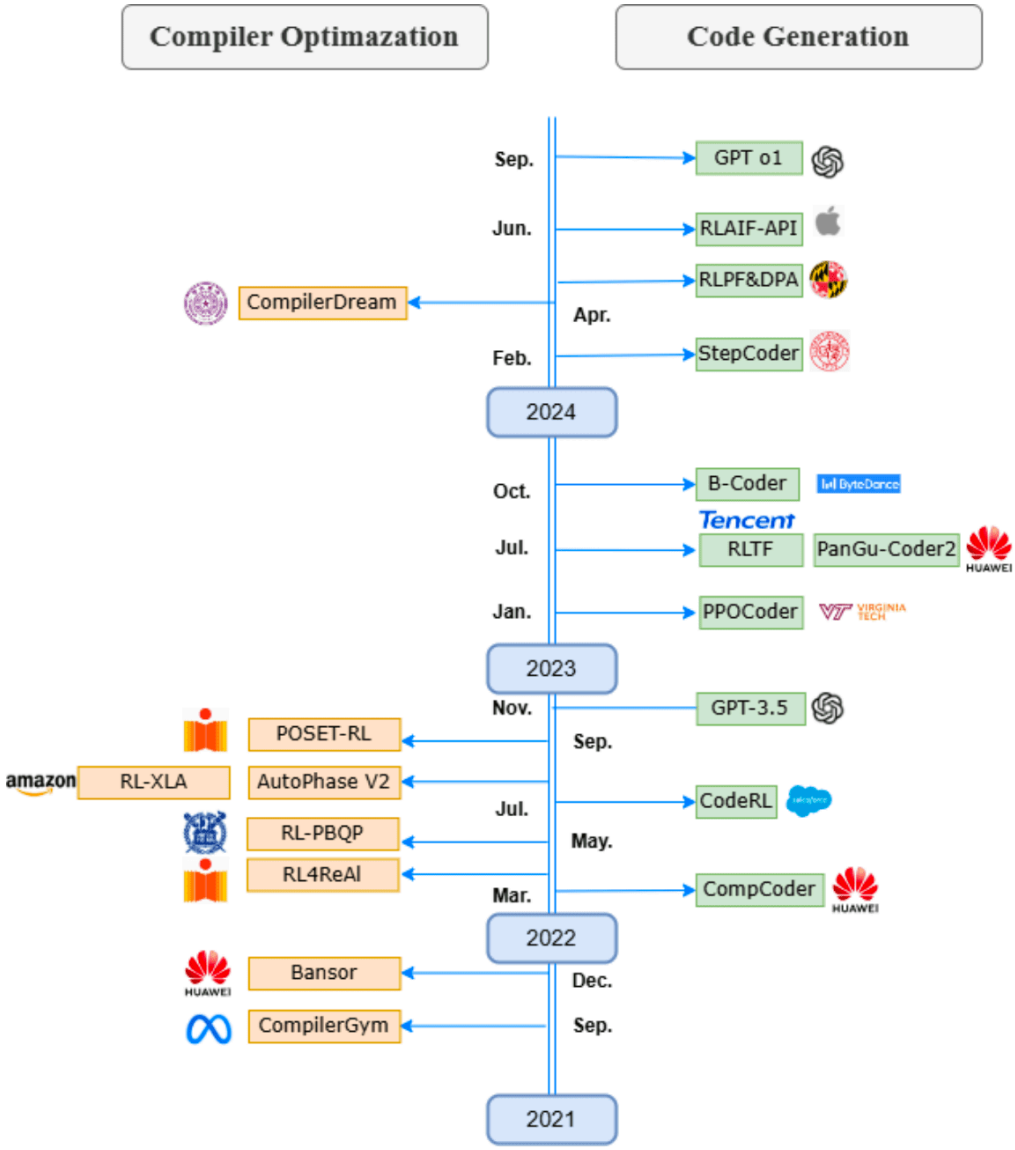

Figure 2. The diagram illustrates the development of code optimization and generation from 2021 to 2024, covering key projects such as CompilerDream, the GPT series, and other important contributions from various organizations.

Given the great potential for reinforcement learning applications in the most important aspects of improving software performance and efficiency, this paper explores how reinforcement learning can be successfully applied and provides a broader overview of RL-related issues to encourage more researchers to benefit from advances in RL.

doi.org/10.32388/8G8TB2

2. Background and Fundamentals

2.1. Code Generation: Concepts and Evolution

Code generation is a fundamental task of Code LLMs, which is essential for automatic programming tasks by generating executable code from natural language descriptions[12]. These descriptions usually contain statements of programming problems and sometimes include information about the programming context, such as function signatures or assertions, and also a formal input and output. The generated code is then executed by a compiler or interpreter and verified by unit tests to ensure that the generated code meets the requirements and works correctly.

Although LLMs have made significant progress in code generation, there are still some challenges in the quality of the code. Studies have shown that LLM can produce a shorter but more complex code when dealing with complex problems, which is a discrepancy compared to standard solutions[13].

As the LLM's context learning capabilities advance, sample code can be introduced into the code generation process to enhance the generation or to control the code format. These examples consist of a fixed set of example pairs that contain several pairs of example inputs and corresponding output codes. By including these examples, the model can refer to similar pairs of inputs and outputs during the generation process, thus improving the accuracy and consistency of the generated code. Decoding strategies commonly used in code generation include two main categories: deterministic strategies and sampling strategies. Deterministic strategies include greedy search and bundle search, which seek to generate the optimal solutions. In contrast, sampling strategies employ methods such as temperature sampling, Top-K sampling, and Top-P (kernel) sampling to introduce variety and flexibility for a wide range of possible code solutions. These different decoding strategies provide a variety of implementations for code generation and adapt to different application requirements.

2.2. Reinforcement Learning in Code LLMs

Reinforcement learning (RL) is a technique that determines the best strategies by receiving reward signals from its environment interactions[14]. Its goal is to discover an optimal policy parameter θ that maximizes the sum of rewards through ongoing engagement with the environment[15]. The distribution $\pi_\theta(a|s)$ indicates the probability of choosing action a given state s. Direct computation of cumulative rewards is challenging due to the environment being frequently unknown or only partially observable. To address this, value-based and policy-based methods are used to approximate

cumulative rewards or gradients, facilitating iterative updates of θ. Within the realm of reinforcement learning applied to Code LLMs, policy-based methods, such as the PPO method and the Actor-Critic framework, are particularly significant. The Actor-Critic architecture is widely employed in reinforcement learning by combining an action executor (actor) and an evaluator (critic)[16]. The actor's responsibility is to execute actions following the present policy, while the critic assesses the actions' values and provides feedback to enhance the actor's strategy.

PPO enhances the strategic model by training a value function and incorporating token-wise KL penalties into the rewards to balance the updates to the strategy and prevent excessive optimization of the reward model[17]. The value function is frequently distinct and comparable in size to the policy model, which can result in significant computational and memory requirements. Moreover, within reinforcement learning (RL), the value function serves as a baseline for computing advantages and reducing variance. However, in the context of large language models (LLMs), the reward model generally computes the reward only for the final token, which can complicate the training of the value function for individual tokens.

Some research has introduced DPO[18] and GRPO[19] approaches to address the problems discussed. Direct preference Optimization (DPO) fundamentally focuses on directly optimizing policies to match human preferences, avoiding the requirement of policy enhancement via reinforcement learning (RL) [18]. This method eliminates the reliance on reward models in traditional RL, instead opting to fit an implicit reward model using a simple classification objective, from which the optimal policy can be articulated. GRPO takes a different approach by removing the extra value function and directly incorporating the KL divergence between training and reference policies into the loss function[19]. It uses the average reward from different solutions to the same problem as a baseline, streamlining the PPO training process, and mitigating the risk of excessively optimizing model rewards.

In summary, reinforcement learning (RL) improves Code LLMs by tuning policy parameters to increase rewards. It employs methods such as PPO, DPO, and GRPO to enhance strategies via environmental interactions and produce code that aligns with human preferences.

2.3. RL-based Fine-tuning Algorithms in LLMs

Reinforcement Learning from Human Feedback (RLHF) has become a crucial algorithmic strategy. Using feedback from humans, RLHF fine-tunes large language models, aligning their outputs with human preferences or specific task objectives. This approach is especially significant in complex tasks,

such as program synthesis, where traditional supervised learning struggles to grasp subtle performance indicators. In RLHF, models are trained to favor actions that receive the most favorable evaluation from humans, allowing greater precision in controlling language and code generation.

In the context of code generation, Reinforcement Learning from Human Feedback (RLHF) encounters specific hurdles, as achieving functional correctness—a pivotal aim in programming—cannot be reliably accomplished with token-based similarity metrics such as BLEU or ROUGE, which are typically utilized in translation and summarization tasks. In code generation, token similarity does not consistently correlate with correctness or functionality. Consequently, it is crucial to employ reward signals that directly assess program correctness. Unit test signals offer a potent solution here: they provide a concrete measure of functionality, as programs that pass unit tests can be deemed functionally correct. The feedback system of the reinforcement learning (RL) framework is depicted in Figure 3, through which various elements are utilized to evaluate the action and provide feedback to optimize the agent behavior with diverse operational strategies. Users can also select preferred results to influence the outcomes, enabling RL to exhibit flexibility and adaptability in dynamic environments. By exploiting these unit test outcomes as reward signals, RL-based methods can bring code generation models more aligned with desired outputs, thereby narrowing the gap between generated code and actual functional requirements.

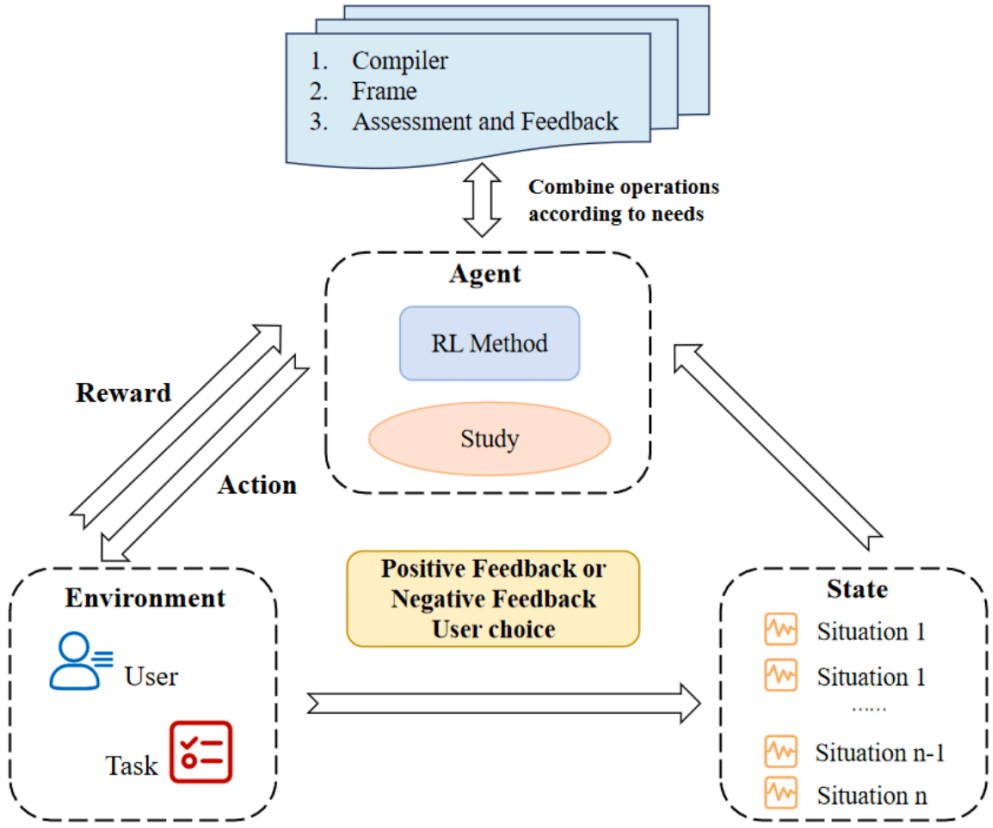

Figure 3. Principles of reinforcement learning (RL) code generation.

Rather importantly, reinforcement learning (RL)-based fine-tuning methods are crucial in code generation tasks. These methods often employ execution-guided synthesis techniques to refine the strategy of the code model by detailed tuning, which ensures that the generated code is both correct and functionally meets expectations. For instance, this process might involve conducting real-time functional tests on code produced by the model, then adjusting the model's behavior according to the outcomes, thereby enhancing the model's capability to handle intricate programming tasks.

3. Frameworks in Code Generation and Optimization

3.1. Theoretical Basis of Code Generation and Optimization

Generating code entails transforming a natural language description into source code. Given a natural language input detailed as a sequence $x = [x_1, \ldots, x_{|x|}]$, a language model (LM) p_{LM} is used to predict the next token sequentially. At each time step t, the LM calculates the probability distribution for the

doi.org/10.32388/8G8TB2

following token, considering all previous tokens, represented as $p_{LM}(x_t \mid x_{1:t-1})$. The probability of creating a program y consisting of the token sequence $y = [x_{|x|+1}, \ldots, x_{|x|+|y|}]$ is computed as the product of these conditional probabilities:

$$P(y \mid x) = \prod_{t=|x|+1}^{|x|+|y|} p_{LM}(x_t \mid x_{1:t}) \tag{1}$$

Within the framework of few-shot learning utilizing large language models (LLMs), the generation process frequently relies on a predetermined ensemble of m exemplars, represented by $\{\langle x_i, y_i \rangle\}_{i \leq m}$. As a result, the code generation via LLM can be described as:

$$P_{LM}(y \mid x) = P(y \mid x, \{\langle x_i, y_i \rangle\}_{i \leq m}) \tag{2}$$

Optimizing code involves substituting equivalent code to enhance efficiency in terms of time and space. Local optimization focuses on regions with high time complexity to boost code performance, whereas global optimization considers the overall code structure and its execution. With technological progress, optimization also occurs during the compilation phase, including intermediate code optimization, which refines code structure, and object code optimization, which transforms intermediate code into effective machine code. Additionally, dynamic optimization happens during the program's runtime. Optimizing algorithms and data structures markedly diminishes computational and spatial complexities. The equation to determine the optimal model parameters, θ^*, is given by:

$$\theta^* = \text{argmax}_\theta P(y_{\text{best}} \mid x; \theta) \tag{3}$$

In this context, θ^* stands for the set of parameters that optimizes the likelihood of producing the best candidate program y_{best} from the input program x using the model parameters θ.

3.2. Pre-training and Post-training

3.2.1. Construction of Datasets

At the outset of constructing datasets for large language models (LLMs) oriented towards code, it is crucial to define the objectives and requirements of the dataset clearly. For such models, the core purpose of the dataset is to enhance the model's capabilities in code generation, comprehension, and task execution. Therefore, collecting code data that encompasses rich algorithmic logic, adheres to good programming practices, and aligns with practical application scenarios becomes particularly important. Figure 4 illustrates the training process for code language models (Code LLM), proceeding

through data preprocessing, model pre-training, fine-tuning, post-training, and other operations for the applications in code generation.

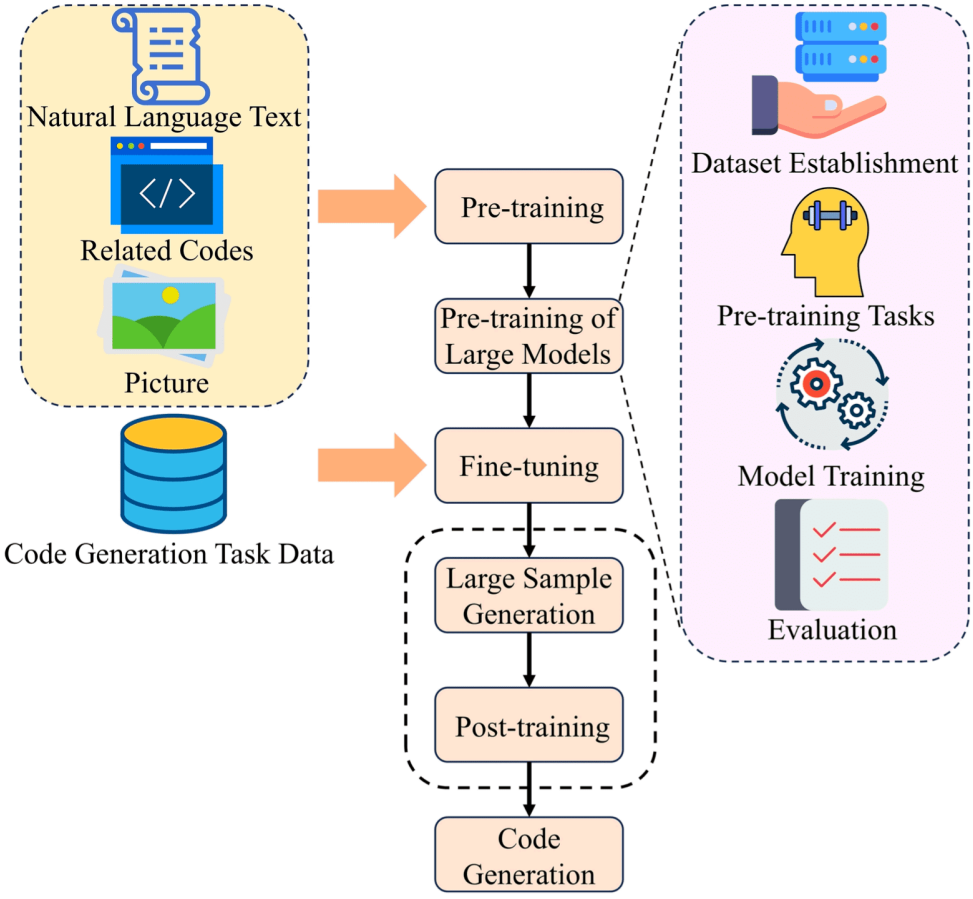

Figure 4. Flowchart of training a code language model (Code LLM).

To obtain high-quality data sources, GitHub, a widely recognized platform for code sharing and collaboration, serves as an ideal original data source. This platform hosts a vast array of open-source projects, providing us with abundant code data. When selecting data, emphasis should be placed on choosing code files that are logically complete, standardly formatted, and clearly commented. Additionally, to further diversify the dataset, the Common Crawl dataset, a large-scale web scraping dataset, is often utilized. By parsing code-related content from it, a more varied set of code data can be obtained.

However, raw data often contains unrecognized or duplicated content, making data preprocessing an indispensable step. Initially, a rigorous data cleaning process should be implemented to remove non-

informative content such as pure hexadecimal code and overly short code snippets, with filtering based on generic attributes like file size and number of lines. Heuristic filtering rules are applied for more refined cleaning tailored to the data characteristics. Given the high degree of repetition in GitHub source code, file-level deduplication strategies, including exact and fuzzy deduplication, are typically employed to effectively reduce data redundancy.

Based on data cleaning, further quality optimization is conducted on the code files. Firstly, code files are checked for their ability to run independently, ensuring they do not depend on external files or libraries. Secondly, code files with unclear logic or disordered structure are eliminated, ensuring the code adheres to standard formatting norms and is converted into a format suitable for LLM training.

To further enhance model performance, during the data refinement stage, not only the cleaned raw data is used, but also algorithm corpora and synthetic data are introduced. These high-quality data rewrites aid the model in better memorizing and embedding knowledge. While preserving the integrity of the original data distribution, downsampling is applied to data from certain high-resource programming languages to improve data utilization efficiency. Simultaneously, to bolster the model's instruction comprehension capabilities, large-scale instruction data is synthesized. Language sampling, task specification modules, and other means are employed to ensure the diversity and specificity of the instruction data. Throughout the construction process, a combination of manual inspection and perplexity (PPL)-based evaluation methods is used to comprehensively assess the data's quality and learnability[20].

On the other hand, addressing the limitations of traditional datasets, researchers have proposed the RL4RS framework, a novel system evaluation framework. This framework encompasses multiple dimensions such as environmental simulation evaluation, environmental evaluation, counterfactual policy evaluation, and opportunity test set construction, aiming to provide a more comprehensive assessment of RL algorithm performance. Unlike most related research, RL4RS evaluates policies directly on raw data, avoiding the impact of environment model generalization errors. By introducing counterfactual policy evaluation algorithms and real-world datasets, RL4RS significantly enhances its alignment with the real world[21].

As dataset size continues to expand, the time and labor required to obtain high-quality human feedback become technological bottlenecks. To address this issue, researchers have introduced ULTRAFEEDBACK[22], a large-scale, high-quality, and diversified AI feedback dataset. This dataset broadens the scope and depth of instructions and responses, covering a wider range of user-assistant

interaction scenarios while mitigating symbolic bias to improve AI feedback reliability. However, large language models face numerous challenges when implementing Reinforcement Learning from Human Feedback (RLHF)[23], such as introducing additional PTX losses and studying the distribution shifts between Supervised Fine-Tuning (SFT) and RLHF. To tackle these challenges, SFT data can be used as alternative data for fine-tuning, enhancing transparency from pretraining to SFT and fostering a better understanding of model changes. Furthermore, resource considerations are also crucial in large language model training. High-performance GPUs are used for training, and a combination of human and AI approaches is employed to improve testing consistency.

In recent years, the application of reinforcement learning algorithms to datasets has matured[24]. However, the trial-and-error learning and reward mechanisms of reinforcement learning rely on interactions between the agent and the environment, which are often time-consuming and costly. The emergence of offline reinforcement learning algorithms offers the possibility of data-driven approaches without requiring expensive real-world exploration, instead relying on large pre-collected datasets. Offline reinforcement learning methods can provide effective initialization for online fine-tuning, but current benchmark tasks for evaluating offline reinforcement learning algorithms are relatively simple and approaching performance saturation. The advent of the D5RL dataset provides a new perspective for evaluating offline reinforcement learning. It offers both offline and online fine-tuning evaluations and designs specific pretraining and fine-tuning for certain tasks. By simulating accessible evaluation environments and providing a level of realism that reflects real-world system attributes, D5RL lays a new foundation for the development of offline reinforcement learning datasets[25]. Figure 5 demonstrates the process of constructing a dataset, ranging from data collection and preprocessing to data optimization and augmentation, with automation and efficiency fully achieved in the process.

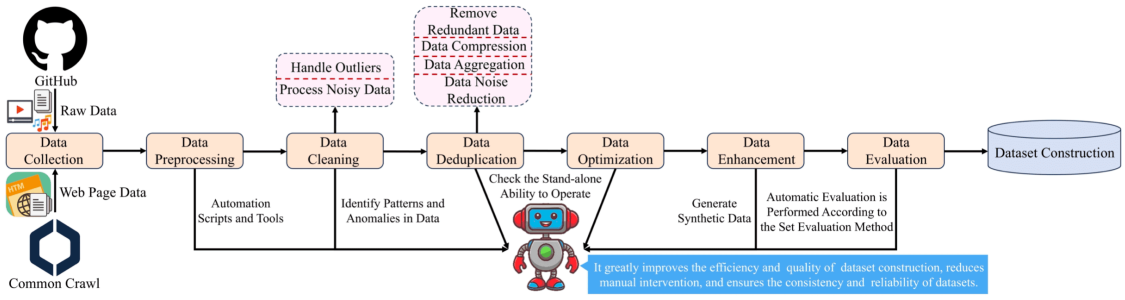

Figure 5. The dataset building process.

doi.org/10.32388/8G8TB2

3.2.2. *Pre-training*

During the pre-training for Code LLMs, datasets often comprise a large-scale code corpus along with a smaller fraction of data sources like mathematics, text, etc., or involve fine-tuning a code corpus on top of general-purpose LLMs. Code LLM architectures, similar to generic LLMs, can be categorized into three types: encoder-only, decoder-only, and encoder-decoder models. Encoder-only models, such as CodeBERT, are generally effective for code understanding tasks, including type prediction, code retrieval, and code clones detection[26]. Decoder-only models, like StarCoder, excel in generative tasks such as code generation, translation, and summarization[27]. Encoder-decoder models, such as CodeT5, are capable of addressing both code understanding and generation tasks, although they are not necessarily superior to their encoder-only or decoder-only counterparts[28]. Additionally, some Code LLMs, like Qwen2.5-Coder, leverage synthetic data in their training procedures, thereby exhibiting higher comprehension abilities[29].

Within the field of code generation using LLMs, the design of current models generally belongs to one of the two main types: encoder-decoder architectures, like CodeT5, CodeT5+, and CodeRL[30][28][31][32]; or decoder-only architectures, such as Codex, StarCoder, CodeLlama, and CodeGemma[33][27][34][35].

3.3. *Post-training*

The searching phase in post-training is essential for refining Code LLMs by systematically exploring and optimizing solutions within large and complex code spaces. This stage involves generating candidate solutions using pre-trained models through probabilistic decoding methods such as beam search, Top-k sampling, nucleus sampling, or deterministic strategies like greedy search, ensuring a balance between diversity and high-confidence outputs. To evaluate these candidates, functional correctness is prioritized using metrics like pass@k and unit tests, which directly assess program functionality and complement traditional token similarity measures. Reinforcement learning techniques, such as Proximal Policy Optimization (PPO), are often employed to iteratively refine candidates by leveraging execution feedback and semantic evaluations, thereby optimizing policies to align with syntactic and functional criteria. Given the computational intensity of searching, strategies like model quantization, caching, and parallelized execution pipelines are employed to enhance scalability and efficiency. Overall, this phase bridges the gap between pre-trained model capabilities

and real-world requirements, ensuring that the generated code is both high-quality and functional for practical applications.

OpenCoder significantly improves its proficiency in theoretical computer science and practical programming tasks through a two-phase instruction fine-tuning process, overcoming the limitations of models concentrating on just one field. DeepSeekMath uses Online Rejection Sampling Fine-tuning (Online RFT) which differs from conventional methods by leveraging outputs from a live policy model for fine-tuning, rather than a supervised approach. Similarly, Group Relative Policy Optimization (GRPO), a variation of Proximal Policy Optimization (PPO), enhances policies by evaluating relative rewards from multiple outputs for the same problem, instead of depending on a single value function. These methods strive to enhance the model's problem-solving abilities through more dynamic and context-sensitive learning processes. Using a framework known as CORGI (Controlled Generation with RL for Guided Interaction), certain researchers have enabled models to obtain immediate textual feedback over several cycles. This is accomplished by simulating interactive sessions with an automated critique system, prompting the models to modify their responses to adhere to predefined constraints derived from the feedback. Qwen2.5-Coder series utilizes a comprehensive strategy to improve model performance. It involves creating instruction-tuning datasets by identifying multilingual programming code and generating instructions from GitHub. A combined method of coarse-to-fine tuning and mixed tuning strategy then incorporates both low- and high-quality instruction samples, enhancing the model's ability to respond to commands. Furthermore, data decontamination is performed to reduce test set leakage effects on evaluation accuracy. Together, these approaches enhance the robustness and effectiveness of the Qwen2.5-Coder series for coding tasks.

In the post-training phase, preference feedback-based learning methods, especially PPO (Proximal Policy Optimization) and DPO (Direct Preference Optimization), have become the key techniques to improve the performance of language models. PPO, as an online reinforcement learning algorithm in the post-training phase, includes reward model training and policy model optimization. First, PPO uses the preference data to train a reward model, which evaluates the quality of the responses generated by the model and becomes the objective function for subsequent policy optimization. Next, PPO leverages the reward model to score the responses generated by the strategy model and further uses this score to optimize the strategy model. This optimization process ensures training stability by introducing a KL penalty term that prevents the model's policy distribution from deviating too much from the initial policy. The advantage of PPO is that its ability to train with online data helps the model to remain

exploratory and adaptable in real-world applications, especially for tasks that require complex reasoning and coding capabilities. However, the online training approach of PPO brings higher computational cost and engineering complexity. In contrast, DPO, as an offline reinforcement learning approach, eliminates the step of training reward models by optimizing policy models directly on preference data, which simplifies the training process and effectively reduces the demand for computational and engineering resources. The optimization process of DPO improves the performance of the policy model by increasing the log-likelihood difference between the selected and rejected responses while ensuring that the model does not overly deviate from the initial strategy. DPO is computationally efficient and is particularly suitable for resource-constrained environments, although it may not be as good as PPO in terms of model adaptability and exploration capabilities.

Comparing PPO and DPO, each has its own strengths and limitations in the post-training phase. PPO shows stronger potential and typically performs better on multiple tasks, especially on tasks involving complex reasoning and coding capabilities. However, DPO is ideal for resource-constrained environments due to its more streamlined training process and lower computational cost. Overall, PPO and DPO provide two different paths for language model optimization in the post-training phase, with PPO providing stronger performance on complex tasks, while DPO meets more stringent resource constraints by improving computational efficiency. As language modeling technology continues to evolve, the selection of an appropriate post-training method will depend on specific application requirements, resource conditions, and performance goals.

4. Related Applications of Reinforcement Learning

4.1. Enhancing Code Language Models with Reinforcement Learning

The integration of reinforcement learning (RL) into code language models (Code LLMs) offers a significant enhancement in accurate and efficient code generation through interactive feedback mechanisms. RL frameworks effectively handle vital aspects such as unit tests and functional correctness by employing real-time evaluation and iterative improvement methods that traditional supervised fine-tuning often overlooks. CodeRL[10] is an example that combines pre-trained language models with deep RL to refine code generation processes. In this system, the code model acts as an actor network, while a critic network evaluates the functional correctness of the generated code, using feedback from unit tests as reward signals. This interactive training cycle continuously enhances the syntactic and semantic precision of the model, producing code that is both accurate and robust

particularly in complex programming settings. Further advancements in RL-based code generation are demonstrated by PPOCoder[36] and PanGu-Coder2[37]. PPOCoder merges CodeT5 with Proximal Policy Optimization (PPO), improving stability and reliability by restricting policy updates and using execution feedback to optimize the code's structure and function. A reward function assesses the alignment between the code's Abstract Syntax Tree (AST) and the ground truth, enabling PPOCoder to generate highly precise, functionally correct code. On the other hand, PanGu-Coder2 uses Ranking Reinforcement from Human Feedback (RRHF) to directly integrate human preferences into the code generation process. This framework employs ranking-based reinforcement to emphasize outputs that satisfy human expectations, considerably boosting the relevance and quality of code generated for complex tasks. Collectively, these frameworks demonstrate how RL-enhanced Code LLMs can dynamically evolve to achieve excellence in code functionality, accuracy, and alignment with human standards.

4.2. The Impact of Reinforcement Learning on End-to-end Software Development

Reinforcement learning (RL) is revolutionizing software engineering, contributing to enhancements across various phases from design to deployment[38]. By learning optimal actions through environmental interactions, RL facilitates automation in areas such as code suggestion and generation, accelerating development and minimizing human error. In the realm of software testing, RL streamlines test case generation, determines execution orders, and optimizes processes, thus enhancing test quality and coverage. For network control, RL also advances traffic management and control strategies, which are essential for distributed systems[39].

As RL technology advances, its role in comprehensive software development—spanning code creation, testing, and resource management—will become increasingly prominent, establishing it as an indispensable tool for software engineers. GitHub Copilot, utilizing OpenAI's Codex model, employs Reinforcement Learning from Human Feedback (RLHF) to improve functions such as code completion, generation, refactoring, and documentation. This strategy, in conjunction with extensive training on large code datasets, enables Copilot to deliver real-time coding support in popular IDEs like Visual Studio and JetBrains with substantially boosted developing efficiency. Similarly, Zhipu AI's CodeGeeX, leveraging the ChatGLM model with RLHF, supports code generation, translation, and completion across various languages in IDEs including VS Code and IntelliJ. Huawei's CodeArts Snap, using PanGu-Coder2 and Reinforcement Learning from Human Reverse Feedback (RRHF), enhances code

generation, debugging, and test generation with contextually tailored code recommendations. These models, optimized through RLHF or RRHF, exhibit effective end-to-end applications by aligning code generation with actual developer needs in IDE settings.

5. Metrics and Benchmarks

5.1. Metrics

Identifying efficient and reliable automatic evaluation metrics for code generation has been a significant challenge[40]. Initially, drawing inspiration from machine translation and text summarization, many efforts relied on metrics that evaluated token matching. Notable examples include BLEU[41], ROUGE[42], and METEOR[43]. Nevertheless, these methods typically struggle to accurately assess the syntactic and functional accuracy of the code, as well as its semantic attributes. Furthermore, these metrics are not tailored for various programming languages and specific compilers, which also diminishes their practicality. To mitigate these limitations in token-matching based metrics, CodeBLEU[40] was developed. It combines syntactic and semantic elements from Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs) with conventional BLEU scores, thereby enhancing the evaluation precision for code generation. However, CodeBLEU still fails to fully resolve the issues related to execution errors and discrepancies in execution results.

Given these obstacles, execution-based metrics have become increasingly important for assessing code generation. Notable methodologies include execution accuracy[44], pass@t[45], n@k[46], and pass@k[47]. When reinforcement learning (RL) is applied to enhance the code generated by large language models (LLM), execution-based metrics, especially pass@k[47], have shown greater significance. The estimation of pass@k is described as follows:

$$\text{pass@k} := \mathbb{E}_{\text{problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{4}$$

Here, c represents the number of successful tests among the generated n codes, with a larger n resulting in a more precise estimate. It assesses the likelihood that at least one of the created code samples k passes all unit tests. Such metrics are crucial in establishing the functional correctness of the generated code by examining its execution performance, serving as a key evaluation tool for modern Code LLMs. For an evaluation of the code generated by Code LLMs enhanced with reinforcement learning, see Table 1. However, these execution-focused evaluation techniques depend heavily on the

integrity of unit tests and are confined to estimating the quality of executable code. In scenarios where unit tests are not suitable, token matching metrics are frequently employed as an alternative evaluation approach.

In summary, choosing the right metrics to assess the quality of code generated by Code LLMs is essential. While current methods, such as token-matching and execution-based approaches, are well established, there remains a shortage of metrics to effectively evaluate the generated code's security and efficiency. More sophisticated metrics are necessary to evaluate the models.

Model	Size	Pass@1				Pass@5				Pass@1000			
		Intro	Inter	Comp	All	Intro	Inter	Comp	All	Intro	Inter	Comp	All
Base Models													
Codex	12B	4.14	0.14	0.02	0.92	9.65	0.51	0.09	2.25	25.02	3.70	3.23	7.87
AlphaCode	1B	-	-	-	-	-	-	-	-	17.67	5.24	7.06	8.09
GPT-3	175B	0.20	0.03	0.00	0.06	2.70	0.73	0.00	1.02	-	-	-	-
GPT-2	0.1B	1.00	0.33	0.00	0.40	3.60	1.03	0.00	1.34	-	-	-	-
GPT-2	1.5B	1.30	0.70	0.00	0.68	5.50	1.03	0.00	1.58	27.90	9.27	8.80	12.32
GPT-Neo	2.7B	3.90	0.57	0.00	1.12	5.50	0.80	0.00	1.58	27.90	9.83	11.40	13.76
GPT-J	6B	5.60	1.00	0.50	1.82	9.20	1.73	1.00	3.08	35.20	13.15	13.51	17.63
CodeT5†	60M	1.40	0.67	0.00	0.68	2.60	0.87	0.10	1.06	-	-	-	-
CodeT5†	220M	2.50	0.73	0.00	0.94	3.30	1.10	0.10	1.34	-	-	-	-
CodeT5†	770M	3.60	0.90	0.20	1.30	4.30	1.37	0.20	1.72	-	-	-	-
Reinforcement Learning-based Models													
CodeRL	770M	6.20	1.50	0.30	2.20	9.39	1.90	0.42	3.10	35.30	13.33	13.60	17.78
PPOCoder	770M	5.20	1.00	0.50	1.74	9.10	2.50	1.20	3.56	35.20	13.35	13.90	17.77
RLTF	770M	4.16	0.97	0.20	1.45	10.12	2.65	0.82	3.78	38.30	15.13	15.90	19.92
$\mathcal{B}$-Coder	$\leq$ 770M/stage[3]	6.70	1.50	0.30	2.30	10.40	2.63	0.70	3.80	37.00	13.67	12.60	18.12
Multi Reinforcement Learning- based Models													
CodeRL + CodeT5	770M	4.90	1.06	0.5	1.71	8.60	2.64	1.0	3.51	36.10	12.65	13.48	17.50
PPOCoder + CodeT5	770M	5.20	1.00	0.5	1.74	9.10	2.50	1.20	3.56	35.20	13.35	13.90	17.77

Table 1. Performance of Base, Reinforcement Learning-based, and Multi Reinforcement Learning-based Models on APPS benchmark

doi.org/10.32388/8G8TB2

5.2. Benchmarks

To thoroughly evaluate the performance of large language models (LLMs) in code generation, researchers have recently developed numerous high-caliber benchmarks. Building upon foundational studies, several variations of the HumanEval dataset have been introduced, alongside additional benchmarks designed to assess the code generation abilities of LLMs in a wider scope. Benchmarks used for testing, often involving reinforcement learning-related Code LLMs, typically encompass the following elements.

HumanEval features 164 selected Python programming tasks, each including a function signature, a descriptive docstring, an implementation, and several unit tests[48]. HumanEval+ expands on the original HumanEval benchmark by increasing the number of test cases by 80-fold. This expanded testing capability allows HumanEval+ to detect a significant amount of previously unnoticed flawed code produced by LLMs[49]. MBPP is a collection of around 974 beginner-level Python coding tasks sourced from public contributions. Each task offers an English description, a code solution, and three automated test cases. MBPP+ enhances MBPP by removing poorly designed problems and fixing flawed solutions. It also boosts the test capacity by a factor of 35 to improve coverage[50].

In the realm of competitions, the APPS benchmark contains 10,000 Python problems spanning three levels of difficulty: introductory, interview, and competition. Each problem includes a description in English, a correct Python solution, and corresponding test cases defined by inputs and outputs or function names, if available. The APPS+ dataset consists of 7,456 entries. It improves upon the original APPS dataset by eliminating defective entries, standardizing input and output formats, and ensuring quality and coherence through unit tests and manual review. Each entry includes a problem description, a standard solution, a function name, unit tests, and initial code. LiveCodeBench provides a comprehensive and uncontaminated benchmark designed to assess a wide range of coding skills in LLMs, such as code creation, self-repair, execution, and test output prediction[51]. It continuously gathers new coding challenges from competitions on three renowned platforms: LeetCode, AtCoder, and CodeForces. The dataset's latest update includes 713 problems released between May 2023 and September 2024.

6. Prospects for Future Development

Through our survey, it is evident that reinforcement learning (RL) emerges as a transform strategy for enhancing large language code models (Code LLMs) in the domain of code generation with significant advancements in performance. However, there are still several challenges that require addressing:

Creating High-Quality Code Datasets

The success of Code LLMs is greatly influenced by the diversity and quality of the code datasets used for pretraining and fine-tuning. At present, there is a shortage of comprehensive, high-quality datasets that cover a wide array of programming tasks, styles, and languages. This shortfall impedes the ability of LLMs to generalize to new programming tasks, various coding settings, and real-world software development. Advanced data acquisition methods, including automated mining of code repositories, sophisticated filtering techniques, and code data synthesis, could facilitate the development of more enriched datasets. Although RL algorithms often excel at specific tasks, their challenge lies in adapting to new tasks or environments, which restricts their versatility and applicability.

Formulating Comprehensive Benchmarks and Metrics for Code Generation in Code LLMs

Current benchmarks, such as HumanEval, may not comprehensively evaluate the array of coding skills required in practical software development. Additionally, many of the evaluation metrics currently in use prioritize syntactic accuracy or functional performance, overlooking critical aspects such as code efficiency and security. Creating benchmarks that replicate the complexities of real-world software development could provide a more accurate evaluation of the coding ability of LLMs.

Enhancing Support for Low-Level and Domain-Specific Programming Languages

LLMs are mainly trained on popular high-level languages, resulting in limited support for low-level and domain-specific languages like assembly and lean. This underrepresentation curtails the use of LLMs in specialized domains and systems programming. Progressing research in transfer learning and meta-learning could allow LLMs to apply knowledge from widely-used languages to improve their performance on lesser-known ones.

Minimizing Computational Costs

RL algorithms, especially those involving extensive state spaces or intricate decision-making processes, typically require significant computational power, such as high-performance GPUs and substantial memory. Such demands can be prohibitive in environments with limited resources. Exploring more efficient RL methods and refining resource utilization can help mitigate these computational needs. Confronting these challenges is essential for fully realizing the potential of RL-augmented LLMs in code generation and enhancing their capabilities across varied programming domains.

7. Conclusion

In this paper, we discuss current reinforcement learning (RL) approaches to code generation and optimization, and analyze various RL-based strategies in different generation and optimization directions. We examine the commonalities and differences of these methods, arguing that RL holds significant promise for code generation and optimization, potentially marking a major shift in the field. Our goal is to help researchers gain a comprehensive understanding of the possible directions and the core challenges, and to inspire future advancements and progresses in this evolving field.

8. Limitations

In this comprehensive survey, we have examined the application of reinforcement learning in code generation and optimization, analyzing a range of methods and techniques. However, due to space limitations, we have not provided a comprehensive analysis of all aspects under discussion. Firstly, we did not provide a detailed account of the datasets employed for model training, which are of paramount importance for the model's generalization and performance. Further research could examine the influence of disparate datasets on model performance and the construction of more diverse and representative datasets to improve generalization. Secondly, the scalability and generalization of reinforcement learning models in the context of large-scale codebases and across multiple projects were not discussed. In practical applications, models must be capable of handling codebases of varying scales and complexities, necessitating good scalability and adaptability. Further research could concentrate on improving the scalability of the models and the transfer of learning between disparate projects and programming languages. Lastly, a detailed comparison of the training and inference times of various algorithms was not provided. In the context of software development, the efficiency of the

algorithm is of paramount importance. Consequently, future studies could assess the time complexity of different reinforcement learning algorithms during the training and inference phases to optimize these times.

References

1. ^Zhu Q, Luo X, Liu F, Gao C, Che W (2022). "A Survey on Natural Language Processing for Programmin g". arXiv. **abs/2212.05773**: 1690-1704.
2. ^Allamanis M, Barr ET, Devanbu P, Sutton C (2018). "A survey of machine learning for big code and natu ralness". ACM Comput. Surv.. **51** (4): Article 81, 37 pages. doi:10.1145/3212695.
3. ^Basu P, Hall M, Khan M, Maindola S, Muralidharan S, Ramalingam S, Rivera A, Shantharam M, Venkat A (2013). "Towards making autotuning mainstream". Int. J. High Perform. Comput. Appl.. **27** (4): 379–39 3. doi:10.1177/1094342013493644.
4. ^Li M, Liu Y, Liu X, Sun Q, You X, Yang H, Luan Z, Qian D (2020). "The Deep Learning Compiler: A Compr ehensive Survey". IEEE Transactions on Parallel and Distributed Systems. **32**: 708-727. S2CID 21106966 6.
5. ^Cummins C, Seeker V, Grubisic D, Elhoushi M, Liang Y, Rozière B, Gehring J, Gloeckle F, Hazelwood KM, Synnaeve G, Leather H (2023). "Large language models for compiler optimization". ArXiv. **abs/2309.070 62**. S2CID 261705851.
6. ^Wang S, Li Z, Qian H, Yang C, Wang Z, Shang M, Kumar V, Tan S, Ray B, Bhatia P, Nallapati R, Ramanat han MK, Roth D, Xiang B. ReCode: Robustness evaluation of code generation models. In: Proceedings of t he 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Toront o, Canada: Association for Computational Linguistics; 2023. p. 13818-13843. doi:10.18653/v1/2023.acl-lo ng.773. Available from: https://aclanthology.org/2023.acl-long.773.
7. ^Barke S, James MB, Polikarpova N (2023). "Grounded Copilot: How Programmers Interact with Code-G enerating Models". Proc. ACM Program. Lang.. **7** (OOPSLA1): Article 78, 27 pages. doi:10.1145/3586030.
8. ^Wang Y, Wang W, Joty S, Hoi SCH. CodeT5: Identifier-aware unified pre-trained encoder-decoder mod els for code understanding and generation. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. Online and Punta Cana, Dominican Republic: Association for Computati onal Linguistics; 2021. p. 8696-8708. doi:10.18653/v1/2021.emnlp-main.685. Available from: https://acl anthology.org/2021.emnlp-main.685.

9. ^Wang Y, Le H, Gotmare A, Bui N, Li J, Hoi S (2023). "CodeT5+: Open code large language models for code understanding and generation". Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. Singapore: Association for Computational Linguistics. pp. 1069–1088. doi:10.18653/v1/2023.emnlp-main.68.
10. a, bLe H, Wang Y, Gotmare AD, Savarese S, Hoi SCH (2022). "Coderl: Mastering code generation through pretrained models and deep reinforcement learning". Advances in Neural Information Processing Systems. 35: 21314–21328.
11. ^Bendib N, Aouadj IN, Baghdadi R (2024). "A Reinforcement Learning Environment for Automatic Code Optimization in the MLIR Compiler". https://api.semanticscholar.org/CorpusID:272694311.
12. ^Jiang J, Wang F, Shen J, Kim S, Kim S (2024). "A Survey on Large Language Models for Code Generation". arXiv preprint arXiv:2406.00515.
13. ^Dou S, Jia H, Wu S, Zheng H, Zhou W, Wu M, Chai M, Fan J, Huang C, Tao Y, Liu Y, Zhou E, Zhang M, Zhou Y, Wu YZ, Zheng R, Wen Mb, Weng R, Wang J, Cai X, Gui T, Qiu X, Zhang Q, Huang X (2024). "What's Wrong with Your Code Generated by Large Language Models? An Extensive Study". ArXiv. **abs/2407.06153**. S2CID 271050610.
14. ^Fujimoto S, Meger D, Precup D. Off-policy deep reinforcement learning without exploration. In: Chaudhuri K, Salakhutdinov R, editors. Proceedings of the 36th International Conference on Machine Learning. Proceedings of Machine Learning Research. 2019 Jun 09-15;97:2052-2062. PMLR. Available from: https://proceedings.mlr.press/v97/fujimoto19a.html.
15. ^Sutton RS (2018). "Reinforcement learning: An introduction". A Bradford Book.
16. ^Konda VR, Tsitsiklis JN (1999). "Actor-Critic Algorithms". In: Neural Information Processing Systems.
17. ^Ouyang L, Wu J, Jiang X, Almeida D, Wainwright CL, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A, Schulman J, Hilton J, Kelton F, Miller LE, Simens M, Askell A, Welinder P, Christiano PF, Leike J, Lowe RJ (2022). "Training language models to follow instructions with human feedback". ArXiv. **abs/2203.02155**. S2CID 246426909.
18. a, bRafailov R, Sharma A, Mitchell E, Ermon S, Manning CD, Finn C (2024). "Direct Preference Optimization: Your Language Model is Secretly a Reward Model". Preprint, arXiv:2305.18290. Available from: https://arxiv.org/abs/2305.18290.
19. a, bShao Z, Wang P, Zhu Q, Xu R, Song J, Bi X, Zhang H, Zhang M, Li YK, Wu Y, Guo D (2024). "DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models". Preprint, arXiv:2402.03300. Available from: https://arxiv.org/abs/2402.03300.

20. ^Huang S, Cheng T, Liu JK, Hao J, Song L, Xu Y, Yang J, Liu JH, Zhang C, Chai L, et al. OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models. arXiv preprint arXiv:2411.04905. 2024.
21. ^Wang K, Zou Z, Zhao M, Deng Q, Shang Y, Liang Y, Wu R, Shen X, Lyu T, Fan C (2023). "RL4RS: A real-world dataset for reinforcement learning based recommender system". In: Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 2935–2944.
22. ^Cui G, Yuan L, Ding N, Yao G, Zhu W, Ni Y, Xie G, Liu Z, Sun M (2023). "Ultrafeedback: Boosting language models with high-quality feedback".
23. ^Dong H, Xiong W, Pang B, Wang H, Zhao H, Zhou Y, Jiang N, Sahoo D, Xiong C, Zhang T (2024). "Rlhf workflow: From reward modeling to online rlhf". arXiv preprint arXiv:2405.07863.
24. ^Levine S, Kumar A, Tucker G, Fu J (2020). "Offline reinforcement learning: Tutorial, review, and perspectives on open problems". arXiv preprint arXiv:2005.01643.
25. ^Rafailov R, Hatch K, Singh A, Smith L, Kumar A, Kostrikov I, Hansen-Estruch P, Kolev V, Ball P, Wu J, et al. D5rl: Diverse datasets for data-driven deep reinforcement learning. arXiv preprint arXiv:2408.08441. 2024.
26. ^Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, et al. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155. 2020.
27. a, bLi R, Ben Allal L, Zi Y, Muennighoff N, Kocetkov D, Mou C, Marone M, Akiki C, Li J, Chim J, et al. 2023. "Starcoder: may the source be with you!" arXiv preprint arXiv:2305.06161.
28. a, bWang Y, Wang W, Joty S, Hoi SCH (2021). "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation". arXiv preprint arXiv:2109.00859.
29. ^Hui B, Yang J, Cui Z, Yang J, Liu D, Zhang L, Liu T, Zhang J, Yu B, Lu K, et al. (2024). "Qwen2. 5-coder technical report". arXiv preprint arXiv:2409.12186.
30. ^Ye JC, Sung WK (2019). "Understanding geometry of encoder-decoder CNNs". In: International Conference on Machine Learning. PMLR. pp. 7064–7073.
31. ^Wang Y, Le H, Gotmare AD, Bui NDQ, Li J, Hoi SCH (2023). "Codet5+: Open code large language models for code understanding and generation". arXiv preprint arXiv:2305.07922.
32. ^Le H, Wang Y, Gotmare AD, Savarese S, Hoi SCH (2022). "Coderl: Mastering code generation through pretrained models and deep reinforcement learning". Advances in Neural Information Processing Systems. 35: 21314–21328.
33. ^Wu J, Gaur Y, Chen Z, Zhou L, Zhu Y, Wang T, Li J, Liu S, Ren B, Liu L, et al. On decoder-only architecture for speech-to-text and large language model integration. In: 2023 IEEE Automatic Speech Recognition a

nd Understanding Workshop (ASRU). IEEE; 2023. p. 1–8.

34. ^*Roziere B, Gehring J, Gloeckle F, Sootla S, Gat I, Tan XE, Adi Y, Liu J, Remez T, Rapin J, et al. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950. 2023.*

35. ^*Team C, Zhao H, Hui J, Howland J, Nguyen N, Zuo S, Hu A, Choquette-Choo CA, Shen J, Kelley J, et al. (2024). "Codegemma: Open code models based on gemma". arXiv preprint arXiv:2406.11409.*

36. ^*Shojaee P, Jain A, Tipirneni S, Reddy CK (2023). "Execution-based code generation using deep reinforcement learning". arXiv preprint arXiv:2301.13816.*

37. ^*Shen B, Zhang J, Chen T, Zan D, Geng B, Fu A, Zeng M, Yu A, Ji J, Zhao J, et al. (2023). "PanGu-Coder2: Boosting Large Language Models for Code with Ranking Feedback. CoRR abs/2307.14936 (2023)". arXiv preprint arXiv:2307.14936. **10**. Available from: arXiv:2307.14936.*

38. ^*Zhuang P, Xu T, Wang S (2021). "A Reinforcement-Learning-Based Deployment Strategy for GPP Components". Proceedings of the 9th International Conference on Computer and Communications Management. doi:10.1145/3479162.3479188.*

39. ^*Xiao Y, Liu J, Wu J, Ansari N (2021). "Leveraging Deep Reinforcement Learning for Traffic Engineering: A Survey". IEEE Communications Surveys & Tutorials. **23**: 2064–2097. doi:10.1109/comst.2021.3102580.*

40. a, b*Ren S, Guo D, Lu S, Zhou L, Liu S, Tang D, Zhou M, Blanco A, Ma S (2020). "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis". ArXiv. **abs/2009.10297**. S2CID 221836101.*

41. ^*Papineni K, Roukos S, Ward T, Zhu WJ (2002). "Bleu: a Method for Automatic Evaluation of Machine Translation". In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics; p. 311–318. doi:10.3115/1073083.1073135. Available from: https://aclanthology.org/P02-1040.*

42. ^*Lin CY (2004). "ROUGE: A Package for Automatic Evaluation of Summaries". In: Text Summarization Branches Out. Barcelona, Spain: Association for Computational Linguistics; p. 74–81.*

43. ^*Banerjee S, Lavie A (2005). "METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments". In: Goldstein J, Lavie A, Lin CY, Voss C, editors. Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization. Ann Arbor, Michigan: Association for Computational Linguistics. p. 65–72.*

44. ^*Rajkumar N, Li R, Bahdanau D (2022). "Evaluating the Text-to-SQL Capabilities of Large Language Models". ArXiv. **abs/2204.00498**. S2CID 247922681.*

45. ^*Olausson TX, Inala JP, Wang C, Gao J, Solar-Lezama A (2024). "Is Self-Repair a Silver Bullet for Code Generation?". Preprint, arXiv:2306.09896.*

46. ^Li Y, Choi D, Chung J, Kushman N, Schrittwieser J, Leblond R, Eccles T, Keeling J, Gimeno F, Dal Lago A, Hubert T, Choy P, de Masson d'Autume C, Babuschkin I, Chen X, Huang P-S, Welbl J, Gowal S, Cherepanov A, Molloy J, Mankowitz DJ, Sutherland Robson E, Kohli P, de Freitas N, Kavukcuoglu K, Vinyals O (2022). "Competition-level code generation with AlphaCode". Science. **378** (6624): 1092–1097. doi:10.1126/science.abq1158.

47. ^a, ^bChen M, Tworek J, Jun H, Yuan Q, Pondé H, 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, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Petroski Such F, Cummings DW, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss WH, Nichol A, Babuschkin I, Balaji S, Jain S, Carr A, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight MM, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W (2021). "Evaluating Large Language Models Trained on Code". ArXiv. **abs/2107.03374**.

48. ^Zheng Q, Xia X, Zou X, Dong Y, Wang S, Xue Y, Wang Z, Shen L, Wang A, Li Y, et al. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. arXiv preprint arXiv:2303.17568. 2023.

49. ^Liu J, Xia CS, Wang Y, Zhang L (2024). "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation". Advances in Neural Information Processing Systems. **36**.

50. ^Guo W, Yang J, Yang K, Li X, Rao Z, Xu Y, Niu D (2023). "Instruction fusion: advancing prompt evolution through hybridization". arXiv preprint arXiv:2312.15692.

51. ^Jain N, Han K, Gu A, Li WD, Yan F, Zhang T, Wang S, Solar-Lezama A, Sen K, Stoica I (2024). "Livecode bench: Holistic and contamination free evaluation of large language models for code". arXiv preprint arXiv:2403.07974.

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.