ENHANCING CODE LLMS WITH REINFORCEMENT LEARNING IN CODE GENERATION: A SURVEY

Anonymous authors
Paper under double-blind review

ABSTRACT

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.

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 Zhu et al. (2022); Allamanis et al. (2018). 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 Basu et al. (2013), while deep learning approaches for optimized compiler sequences Li et al. (2020) struggle with generalization. Although large language models (LLMs) improve code generation and optimization Cummins et al. (2023), they often produce biased or inconsistent outputs Wang et al. (2023b); Barke et al. (2023) and require time-consuming pre-training with code-specific models like Code T5 Wang et al. (2021a) and Code T5+ Wang et al. (2023c). 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.

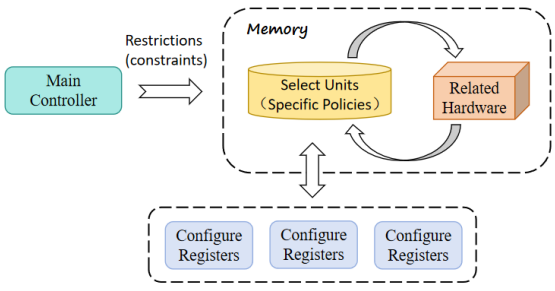

Figure 1: The controller is in control of the resources.

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 Le et al. (2022a), and optimization Bendib et al. (2024) 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.

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.

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 Jiang et al. (2024). 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 Dou et al. (2024).

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 Fujimoto et al. (2019). Its goal is to discover an optimal policy parameter θ that maximizes the sum of rewards through ongoing engagement with the environment Sutton (2018). 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) Konda & Tsitsiklis (1999). 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 Ouyang et al. (2022). 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 Rafailov et al. (2024b) and GRPO Shao et al. (2024) 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) Rafailov et al. (2024b). 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 Shao et al. (2024). 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 2, 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.

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 following token, considering all previous tokens, represented as $p_{LM}(x_t \mid x_{1:t-1})$. The

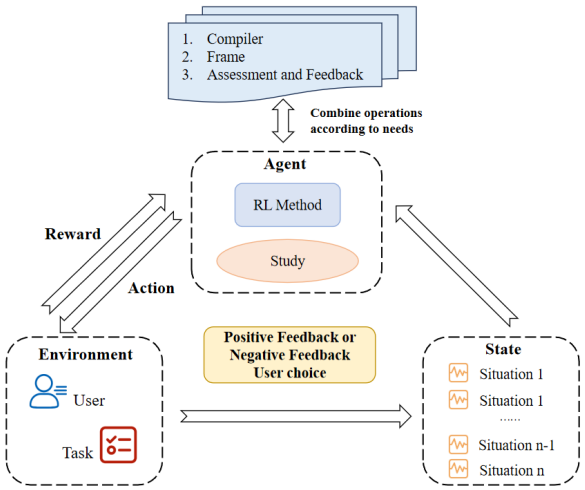

Figure 2: Principles of reinforcement learning (RL) code generation.

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:

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

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:

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

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

Constructing high-quality datasets is foundational for enhancing Code LLMs' generation and comprehension capabilities. As illustrated in Figure 3 and Figure 4, the dataset construction pipeline spans the entire lifecycle from data collection and preprocessing to data optimization, augmentation, and subsequent pre-training and fine-tuning, with increasing automation and efficiency throughout the process. Primary sources such as GitHub and Common Crawl provide abundant raw data, which necessitates rigorous preprocessing to ensure quality. This includes heuristic filtering to remove uninformative content, file-level deduplication to reduce redundancy, and dependency validation to ensure code executability. To further improve robustness, real-world data is often augmented with synthetic instruction data and algorithm corpora. Additionally, strategies such as downsampling high-resource

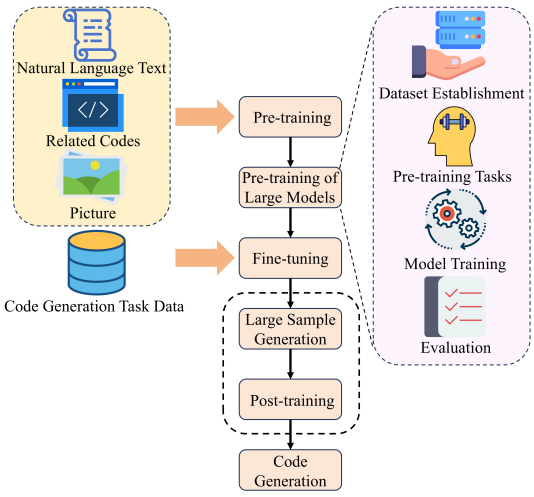

Figure 3: Flowchart of training a code language model (Code LLM).

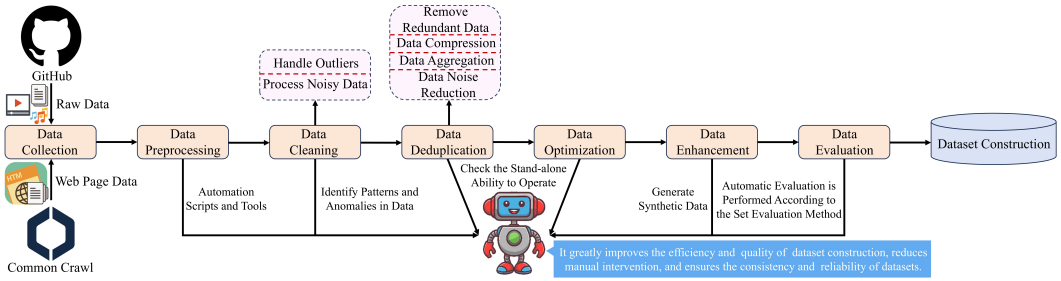

Figure 4: The dataset building process.

languages and perplexity (PPL)-based evaluation are employed to balance data distribution and assess learnability Huang et al. (2024).

In the context of reinforcement learning (RL), specialized datasets and frameworks address specific evaluation and training challenges. The RL4RS framework Wang et al. (2023a) introduces multi-dimensional assessments, such as counterfactual policy evaluation on raw data, to reduce environment simulation bias and better align with real-world systems. To alleviate the bottleneck of human feedback, ULTRAFEEDBACK Cui et al. (2023) provides large-scale and diverse AI-generated feedback, mitigating symbolic bias. Furthermore, to address distribution shifts and training instability in RLHF, researchers increasingly incorporate supervised fine-tuning (SFT) datasets to improve training stability and transparency Dong et al. (2024). Finally, to overcome the high cost of online interaction, the D5RL dataset Rafailov et al. (2024a) enables offline reinforcement learning by providing task environments that support reliable evaluation without expensive real-world exploration.

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 Feng et al. (2020). Decoder-only models, like StarCoder, excel in generative tasks such as code generation, translation, and summarization Li et al. (2023). 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 Wang et al. (2021b). Additionally, some Code LLMs, like Qwen2.5-Coder, leverage

synthetic data in their training procedures, thereby exhibiting higher comprehension abilities Hui et al. (2024).

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 Ye & Sung (2019); Wang et al. (2021b; 2023d); Le et al. (2022b); or decoder-only architectures, such as Codex, StarCoder, CodeLlama, and CodeGemma Wu et al. (2023); Li et al. (2023); Roziere et al. (2023); Team et al. (2024).

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 Le et al. (2022a) 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 Shojaee et al. (2023) and PanGu-Coder2 Shen et al. (2023). 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 Zhuang et al. (2021). 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 Xiao et al. (2021).

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 Ren et al. (2020). Initially, drawing inspiration from machine translation and text summarization, many efforts relied on metrics that evaluated token matching. Notable examples include BLEU Papineni et al. (2002), ROUGE Lin (2004), and METEOR Banerjee & Lavie (2005). 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 Ren et al. (2020) 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 Rajkumar et al. (2022), pass@t Olausson et al. (2024), n@k Li et al. (2022), and pass@k Chen et al. (2021). When reinforcement learning (RL) is applied to enhance the code generated by large language models (LLM), execution-based metrics, especially pass@k Chen et al. (2021), 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.

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

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

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

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 Zheng et al. (2023). 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 Liu et al. (2024). 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 Guo et al. (2023).

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 Jain et al. (2024). 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

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), July 2018. ISSN 0360-0300. doi: 10.1145/3212695. URL https://doi.org/10.1145/3212695.

Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare Voss (eds.), *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pp. 65–72, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics. URL https://aclanthology.org/W05-0909.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. doi: 10.1145/3586030. URL https://doi.org/10.1145/3586030.

Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. Towards making autotuning mainstream. *Int. J. High Perform. Comput. Appl.*, 27(4):379–393, November 2013. ISSN 1094-3420. doi: 10.1177/1094342013493644. URL https://doi.org/10.1177/1094342013493644.

Nazim Bendib, Iheb Nassim Aouadj, and Riyadh Baghdadi. A reinforcement learning environment for automatic code optimization in the mlir compiler, 2024. URL https://api.semanticscholar.org/CorpusID:272694311.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021. URL https://api.semanticscholar.org/CorpusID:235755472.

Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Wei Zhu, Yuan Ni, Guotong Xie, Zhiyuan Liu, and Maosong Sun. Ultrafeedback: Boosting language models with high-quality feedback, 2023.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Kim M. Hazelwood, Gabriel Synnaeve, and Hugh Leather. Large language models for compiler optimization. *ArXiv*, abs/2309.07062, 2023. URL https://api.semanticscholar.org/CorpusID:261705851.

Hanze Dong, Wei Xiong, Bo Pang, Haoxiang Wang, Han Zhao, Yingbo Zhou, Nan Jiang, Doyen Sahoo, Caiming Xiong, and Tong Zhang. Rlhf workflow: From reward modeling to online rlhf. *arXiv preprint arXiv:2405.07863*, 2024.

Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, Yan Liu, Enyu Zhou, Ming Zhang, Yuhao Zhou, Yue Zhong Wu, Rui Zheng, Ming bo Wen, Rongxiang Weng, Jingang Wang, Xunliang Cai, Tao Gui, Xipeng Qiu, Qi Zhang, and Xuanjing Huang. What's wrong with your code generated by large language models? an extensive study. *ArXiv*, abs/2407.06153, 2024. URL https://api.semanticscholar.org/CorpusID:271050610.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th*

International Conference on Machine Learning, volume 97 of *Proceedings of Machine Learning Research*, pp. 2052–2062. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/fujimoto19a.html.

Weidong Guo, Jiuding Yang, Kaitong Yang, Xiangyang Li, Zhuwei Rao, Yu Xu, and Di Niu. Instruction fusion: advancing prompt evolution through hybridization. *arXiv preprint arXiv:2312.15692*, 2023.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In *Neural Information Processing Systems*, 1999. URL https://api.semanticscholar.org/CorpusID:207779694.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022a.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022b.

Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32:708–727, 2020. URL https://api.semanticscholar.org/CorpusID:211069666.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pp. 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://aclanthology.org/W04-1013.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation?, 2024. URL https://arxiv.org/abs/2306.09896.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155, 2022. URL https://api.semanticscholar.org/CorpusID:246426909.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Pierre Isabelle, Eugene Charniak, and Dekang Lin (eds.), *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pp. 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040.

Rafael Rafailov, Kyle Hatch, Anikait Singh, Laura Smith, Aviral Kumar, Ilya Kostrikov, Philippe Hansen-Estruch, Victor Kolev, Philip Ball, Jiajun Wu, et al. D5rl: Diverse datasets for data-driven deep reinforcement learning. *arXiv preprint arXiv:2408.08441*, 2024a.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2024b. URL https://arxiv.org/abs/2305.18290.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. Evaluating the text-to-sql capabilities of large language models. *ArXiv*, abs/2204.00498, 2022. URL https://api.semanticscholar.org/CorpusID:247922681.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv*, abs/2009.10297, 2020. URL https://api.semanticscholar.org/CorpusID:221836101.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. Pangu-coder2: Boosting large language models for code with ranking feedback. corr abs/2307.14936 (2023). *arXiv preprint arXiv:2307.14936*, 10, 2023.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*, 2023.

Richard S Sutton. Reinforcement learning: An introduction. *A Bradford Book*, 2018.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, et al. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*, 2024.

Kai Wang, Zhene Zou, Minghao Zhao, Qilin Deng, Yue Shang, Yile Liang, Runze Wu, Xudong Shen, Tangjie Lyu, and Changjie Fan. 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, 2023a.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. ReCode: Robustness evaluation of code generation models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13818–13843, Toronto, Canada, July 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.773. URL https://aclanthology.org/2023.acl-long.773.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, November 2021a. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021b.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. CodeT5+: Open code large language models for code understanding and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 1069–1088, Singapore, December 2023c. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.68. URL https://aclanthology.org/2023.emnlp-main.68.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023d.

Jian Wu, Yashesh Gaur, Zhuo Chen, Long Zhou, Yimeng Zhu, Tianrui Wang, Jinyu Li, Shujie Liu, Bo Ren, Linquan Liu, et al. On decoder-only architecture for speech-to-text and large language model integration. In *2023 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pp. 1–8. IEEE, 2023.

Yang Xiao, Jun Liu, Jiawei Wu, and N. Ansari. Leveraging deep reinforcement learning for traffic engineering: A survey. *IEEE Communications Surveys & Tutorials*, 23:2064–2097, 2021. doi: 10.1109/comst.2021.3102580.

Jong Chul Ye and Woon Kyoung Sung. Understanding geometry of encoder-decoder cnns. In *International Conference on Machine Learning*, pp. 7064–7073. PMLR, 2019.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*, 2023.

Qingfu Zhu, Xianzhen Luo, Fang Liu, Cuiyun Gao, and Wanxiang Che. A survey on natural language processing for programming, 2022.

Peiwen Zhuang, Tao Xu, and Shan Wang. A reinforcement-learning-based deployment strategy for gpp components. *Proceedings of the 9th International Conference on Computer and Communications Management*, 2021. doi: 10.1145/3479162.3479188.