A Survey on Large Language Models for Code Generation

JUYONG JIANG and FAN WANG, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China
JIASI SHEN, The Hong Kong University of Science and Technology, Hong Kong, China
SUNGJU KIM, NAVER Cloud, Seoul, South Korea
SUNGHUN KIM, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China

Large Language Models (LLMs) have garnered remarkable advancements across diverse code-related tasks, known as Code LLMs, particularly in code generation that generates source code with LLM from natural language descriptions. This burgeoning field has captured significant interest from both academic researchers and industry professionals due to its practical significance in software development, e.g., *GitHub Copilot*. Despite the active exploration of LLMs for a variety of code tasks, either from the perspective of Natural Language Processing (NLP) or Software Engineering (SE) or both, there is a noticeable absence of a comprehensive and up-to-date literature review dedicated to LLM for code generation. In this survey, we aim to bridge this gap by providing a systematic literature review that serves as a valuable reference for researchers investigating the cutting-edge progress in LLMs for code generation. We introduce a taxonomy to categorize and discuss the recent developments in LLMs for code generation, covering aspects such as data curation, latest advances, performance evaluation, ethical implications, environmental impact, and real-world applications. In addition, we present a historical overview of the evolution of LLMs for code generation and provide a quantitative and qualitative comparative analysis of experimental results of code LLMs, sourced from their original papers to ensure a fair comparison on the HumanEval, MBPP, and BigCodeBench benchmarks, across various levels of difficulty and types of programming tasks, to highlight the progressive enhancements in LLM capabilities for code generation. We identify critical challenges and promising opportunities regarding the gap between academia and practical development. Furthermore, we have established a dedicated resource GitHub page (https://github.com/juyongjiang/CodeLLMSurvey) to continuously document and disseminate the most recent advances in the field.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Software and its engineering → Software development techniques**; • **Computing methodologies → Artificial intelligence**;

Additional Key Words and Phrases: Large Language Models, Code Large Language Models, Code Generation

Juyong Jiang and Fan Wang contributed equally to this research.
This work is supported in part by Alibaba Innovative Research grant.
Authors' Contact Information: Juyong Jiang, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China; e-mail: jjiang472@connect.hkust-gz.edu.cn; Fan Wang, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China; e-mail: fwang380@connect.hkust-gz.edu.cn; Jiasi Shen (corresponding author), The Hong Kong University of Science and Technology, Hong Kong, China; e-mail: sjs@cse.ust.hk; Sungju Kim (corresponding author), NAVER Cloud, Seoul, South Korea; e-mail: sungju.kim@navercorp.com; Sunghun Kim (corresponding author), The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China; e-mail: hunkim@cse.ust.hk.

ACM 1557-7392/2026/1-ART58
https://doi.org/10.1145/3747588

ACM Reference format:
Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* 35, 2, Article 58 (January 2026), 72 pages.
https://doi.org/10.1145/3747588

1 Introduction

The advent of **Large Language Models (LLMs)** such as ChatGPT[1] [194] has profoundly transformed the landscape of automated code-related tasks [49], including code completion [87, 169, 268, 280], code translation [53, 135, 242], and code repair [75, 126, 193, 202, 289, 308]. A particularly intriguing application of LLMs is code generation, a task that involves producing source code from **Natural Language (NL)** descriptions. Despite varying definitions across studies [52, 218, 235, 267], for the main scope of this survey, we focus on the code generation task and adopt a consistent definition of code generation as the **Natural-Language-to-Code (NL2Code)** task [17, 18, 305]. To enhance clarity, the differentiation between code generation and other code-related tasks, along with a more nuanced definition, is summarized in Table 1. This area has garnered substantial interest from both academia and industry, as evidenced by the development of tools like GitHub Copilot[2] [49], CodeGeeX[3] [320], and Amazon CodeWhisperer,[4] which leverage groundbreaking code LLMs to facilitate software development.

Initial investigations into code generation primarily utilized heuristic rules or expert systems, such as probabilistic grammar-based frameworks [11, 63, 119, 125, 286] and specialized language models [65, 83, 117]. These early techniques were typically rigid and difficult to scale. However, the introduction of Transformer-based LLMs has shifted the paradigm, establishing them as the preferred method due to their superior proficiency and versatility. One remarkable aspect of LLMs is their capability to follow instructions [57, 185, 198, 273, 287], enabling even novice programmers to write code by simply articulating their requirements. This emergent ability has democratized coding, making it accessible to a broader audience [305]. The performance of LLMs on code generation tasks has seen remarkable improvements, as illustrated by the HumanEval leaderboard,[5] which showcases the evolution from PaLM 8B [55] of 3.6% to LDB [324] of 95.1% on `pass@1` metrics. As can be seen, the HumanEval benchmark [49] has been established as a *de facto* standard for evaluating the coding proficiency of LLMs [49].

To offer a comprehensive chronological evolution, we present an overview of the development of LLMs for code generation, as illustrated in Figure 1. The landscape of LLMs for code generation is characterized by a spectrum of models, with certain models like ChatGPT [198], GPT4 [6], LLaMA [249, 250], and Claude 3 [15] serving general-purpose tasks, while others such as StarCoder [146, 168], Code Llama [224], DeepSeek-Coder [88], and CodeGemma [60] are tailored specifically for code-centric tasks. The convergence of code generation with the latest LLM advancements is pivotal, especially when **Programming Languages (PLs)** can be considered as distinct dialects of multilingual NL [17, 320]. These models are not only tested against **Software Engineering (SE)** requirements but also propel the advancement of LLMs into practical production [315].

While recent surveys have shed light on code LLMs from the lenses of **Natural Language Processing (NLP)**, SE, or a combination of both disciplines [74, 101, 172, 305, 315, 323], they have often encompassed a broad range of code-related tasks. There remains a dearth of literature

[1]https://chat.openai.com.
[2]https://github.com/features/copilot.
[3]https://codegeex.cn/en-US.
[4]https://aws.amazon.com/codewhisperer.
[5]https://paperswithcode.com/sota/code-generation-on-humaneval.

Table 1. The Applications of Code LLMs in Various Code-Related Understanding and Generation Tasks

Type	I-O	Task	Definition
Understanding	C-L	Code Classification	Classify code snippets based on functionality, purpose, or attributes to aid in organization and analysis.
		Bug Detection	Predict, detect, and diagnose defects, bugs, or vulnerabilities in code to ensure functionality and security.
		Clone Detection	Identifying duplicate or similar code snippets in software to enhance maintainability, reduce redundancy, and check plagiarism.
		Exception Type Prediction	Predict different exception types in code to manage and handle exceptions effectively.
	C-C	Code-to-Code Retrieval	Retrieve relevant code snippets based on a given code query for reuse or analysis.
	NL-C	Code Search	Find relevant code snippets based on NL queries to facilitate coding and development tasks.
Generation	C-C	Code Completion	Predict and suggest the next portion of code, given contextual information from the prefix (and suffix), while typing to enhance development speed and accuracy.
		Code Translation	Translate the code from one PL to another while preserving functionality and logic.
		Code Repair	Identify and fix bugs in code by generating the correct version to improve functionality and reliability.
		Mutant Generation	Generate modified versions of code to test and evaluate the effectiveness of testing strategies.
		Test Generation	Generate test cases to validate code functionality, performance, and robustness.
	C-NL	Code Summarization	Generate concise textual descriptions or explanations of code to enhance understanding and documentation.
	NL-C	Code Generation	Generate source code from NL descriptions to streamline development and reduce manual coding efforts.

The I-O column indicates the type of input and output for each task, where C, NL, and L represent code, natural language, and label, respectively. Note that the detailed definitions of each task align with the descriptions in [8, 17, 18, 192, 305]. The scope of this survey focuses on code generation, which is highlighted in yellow.

specifically and comprehensively reviewing advanced topics in code generation, such as meticulous data curation, instruction tuning, alignment with feedback, prompting techniques, the development of autonomous coding agents, **Retrieval-Augmented Code Generation (RACG)**, LLM-as-a-Judge for code generation, among others. A notably pertinent study [17, 305] also concentrates on LLMs for text-to-code generation (NL2Code), yet it primarily examines models released from 2020 to 2022. Consequently, this noticeable temporal gap has resulted in an absence of up-to-date literature reviews that contemplate the latest advancements, including models like CodeQwen [246], WizardCoder [171], CodeFusion [238], and PPOCoder [235], as well as the comprehensive exploration of the advanced topics previously mentioned.

Recognizing the need for a dedicated and up-to-date literature review, this survey endeavors to bridge that gap. We provide a systematic review that will serve as a foundational reference for researchers quickly exploring the latest progress in LLMs for code generation. A taxonomy is introduced to categorize and examine recent advancements, encompassing data curation [171, 266, 276], advanced topics [46, 52, 104, 139, 162, 169, 185, 188, 203, 236, 307], evaluation methods [49, 95, 123, 331], and practical applications [49, 320]. Furthermore, we pinpoint critical challenges and identify promising opportunities to bridge the research-practicality divide. Therefore, this survey allows NLP and SE researchers to acquire a thorough understanding of LLM for code generation, highlighting cutting-edge directions and current hurdles and prospects.

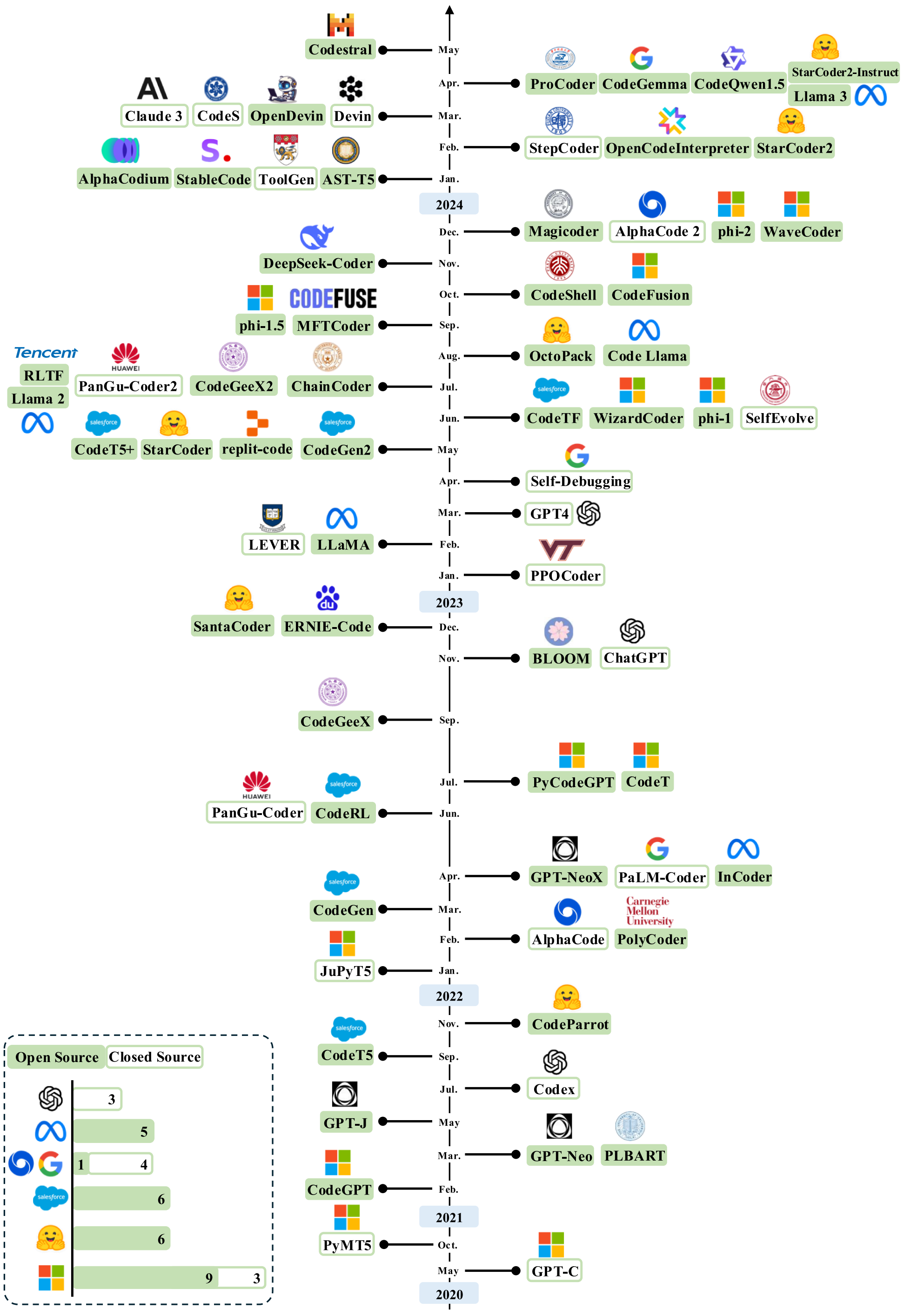


Fig. 1. A chronological overview of LLMs for code generation in recent years. The timeline was established mainly according to the release date. The models with publicly available model checkpoints are highlighted in green color.

The remainder of the survey is organized following the structure outlined in our taxonomy in Figure 6. In Section 2, we introduce the preliminaries of LLM with Transformer architecture and formulate the task of LLM for code generation. In Section 3, we detail the systematic methodologies employed in conducting literature reviews. Then, in Section 4, we propose a taxonomy, categorizing the complete process of LLMs in code generation. Section 5 delves into the specifics of LLMs for code generation within this taxonomy framework. In Section 6, we underscore the critical challenges and promising opportunities for bridging the research-practical gap and conclude this work in Section 7.

2 Background

2.1 LLMs

The effectiveness of LLMs is fundamentally attributed to their substantial quantity of model parameters, large-scale and diversified datasets, and the immense computational power utilized during training [97, 127]. Generally, scaling up language models consistently results in enhanced performance and sample efficiency across a broad array of downstream tasks [273, 318]. However, with the expansion of the model size to a certain extent (e.g., GPT-3 [34] with 175B-parameters and PaLM [55] with 540B), LLMs have exhibited an unpredictable phenomenon known as emergent abilities,[6] including instruction following [198], in-context learning [70], and step-by-step reasoning [105, 274], which are absent in smaller models but apparent in larger ones [273].

Adhering to the same architectures of the Transformer [254] in LLMs, general-purpose LLMs are pre-trained primarily on large-scale text data, incorporating a smaller amount of code and math data to enhance logical reasoning capabilities, whereas code LLMs are specifically pre-trained (or continually pre-trained on general LLMs) using large-scale unlabeled code corpora with a smaller portion of text (and math) data. Additionally, some code LLMs, such as Qwen2.5-Coder [109], incorporate synthetic data in their training processes, a practice that is attracting increasing attention from both industry and academia. Analogous to LLMs, Code LLMs can also be classified into three architectural categories: encoder-only models, decoder-only models, and encoder-decoder models. For encoder-only models, such as CodeBERT [76], they are typically suitable for code comprehension tasks including type prediction, code retrieval, and clone detection. For decoder-only models, such as StarCoder [34], they predominantly excel in generation tasks, such as code generation, code translation, and code summarization. Encoder-decoder models, such as CodeT5 [269], can accommodate both code understanding and generation tasks but do not necessarily outperform encoder-only or decoder-only models. The overall architectures of the different Code LLMs for code generation are depicted in Figure 2.

In the following subsection, we will delineate the key modules of the Transformer layers in Code LLMs.

*2.1.1 **Multi-Head Self-Attention (MHSA)** Modules.* Each Transformer layer incorporates a MHSA mechanism to discern the inherent semantic relationships within a sequence of tokens across h distinct latent representation spaces. Formally, the MHSA employed by the Transformer can be formulated as follows:

$$\mathbf{h}^{(l)} = \text{MultiHeadSelfAttn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}\left\{\text{Head}_i\right\}_{i=1}^{h} \mathbf{W}^{\mathbf{O}}, \tag{1}$$

$$\text{Head}_i = \text{Attention}(\underbrace{\mathbf{H}^{(l-1)}\mathbf{W}_i^{\mathbf{Q}}}_{\mathbf{Q}}, \underbrace{\mathbf{H}^{(l-1)}\mathbf{W}_i^{\mathbf{K}}}_{\mathbf{K}}, \underbrace{\mathbf{H}^{(l-1)}\mathbf{W}_i^{\mathbf{V}}}_{\mathbf{V}}), \tag{2}$$

[6]It should be noted that an LLM is not necessarily superior to a smaller language model, and emergent abilities may not manifest in all LLMs [318].

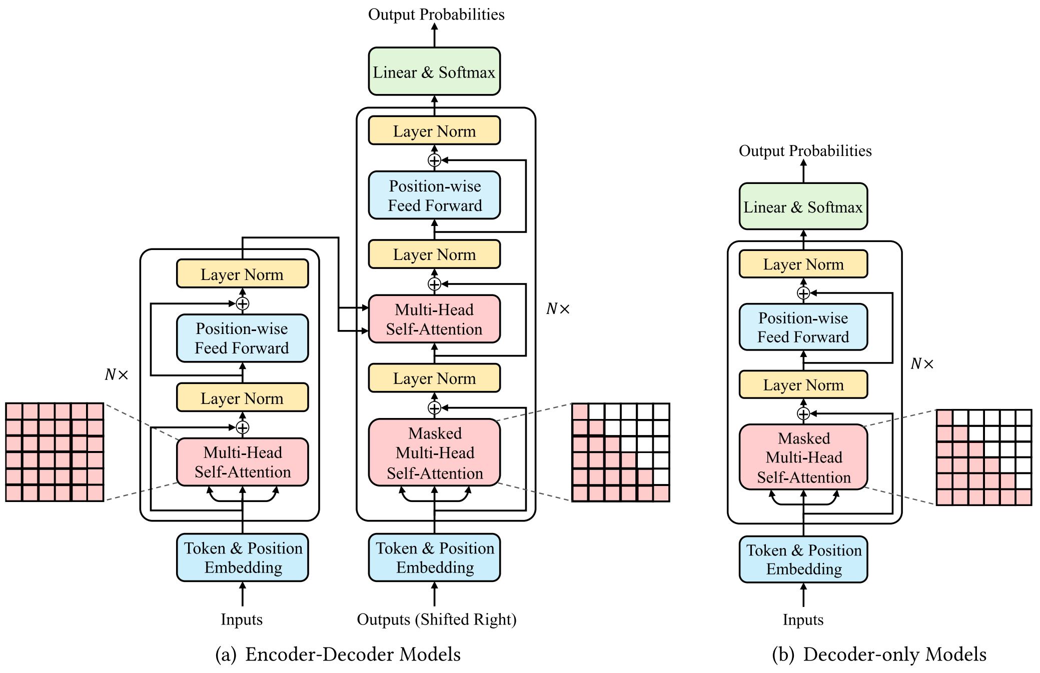

Output Probabilities
Linear & Softmax
Layer Norm
Position-wise Feed Forward
Layer Norm
Multi-Head Self-Attention
Layer Norm
Masked Multi-Head Self-Attention
Token & Position Embedding
Inputs
Outputs (Shifted Right)
N×
(a) Encoder-Decoder Models
(b) Decoder-only Models

Fig. 2. The overview of LLMs with encoder-decoder and decoder-only Transformer architecture for code generation, adapted from [254].

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{model}/h}}\right)\mathbf{V}, \tag{3}$$

where $\mathbf{H}^{(l-1)} \in \mathbb{R}^{n\times d_{model}}$ denotes the input to the lth Transformer layer, while $\mathbf{h}^{(l)} \in \mathbb{R}^{n\times d_{model}}$ represents the output of MHSA sub-layer. The quantity of distinct attention heads is represented by h, and d_{model} refers to the model dimension. The set of projections $\left\{\mathbf{W}_i^{\mathrm{Q}}, \mathbf{W}_i^{\mathrm{K}}, \mathbf{W}_i^{\mathrm{V}}, \mathbf{W}_i^{\mathrm{O}}\right\} \in \mathbb{R}^{d_{model}\times d_{model}/h}$ encompasses the affine transformation parameters for each attention head Head_i, transforming the Query $\mathbf{Q}$, Key $\mathbf{K}$, Value $\mathbf{V}$, and the output of the attention sub-layer. The softmax function is applied in a row-wise manner. The dot-products of queries and keys are divided by a scaling factor $\sqrt{d_{model}/h}$ to counteract the potential risk of excessive large inner products and correspondingly diminished gradients in the softmax function, thus encouraging a more balanced attention landscape.

In addition to MHSA, there are two other types of attention based on the source of queries and key-value pairs:

— *Masked MHSA*. Within the decoder layers of the Transformer, the self-attention mechanism is constrained by introducing an attention mask, ensuring that queries at each position can only attend to all key-value pairs up to and inclusive of that position. To facilitate parallel training, this is typically executed by assigning a value of 0 to the lower triangular part and setting the remaining elements to $-\infty$. Consequently, each item attends only to its predecessors and itself. Formally, this modification in Equation (3) can be depicted as follows:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{model}/h}} + \mathbf{M}_{mask}\right)\mathbf{V}, \tag{4}$$

$$\mathbf{M}_{mask} = \left(m_{ij}\right)_{n\times n} = \left(\mathbb{I}(i \geq j)\right)_{n\times n} = \begin{cases} 0 & \text{for } i \geq j \\ -\infty & \text{otherwise} \end{cases}. \tag{5}$$

This form of self-attention is commonly denoted as autoregressive or causal attention [156].

— *Cross-Layer MHSA*. The queries are derived from the outputs of the preceding (decoder) layer, while the keys and values are projected from the outputs of the encoder.

2.1.2 Position-Wise Feed-Forward Networks (PFFNs). Within each Transformer layer, a PFFN is leveraged following the MHSA sub-layer to refine the sequence embeddings at each position i in a separate and identical manner, thereby encoding more intricate feature representations. The PFFN is composed of a pair of linear transformations, interspersed with a ReLU activation function. Formally,

$$\text{PFFN}(h^{(l)}) = \left(\text{Concat}\left\{\text{FFN}(h_i^{(l)})^T\right\}_{i=1}^{n}\right)^T, \tag{6}$$

$$\text{FFN}(h_i^{(l)}) = \text{ReLU}(h_i^{(l)}\mathbf{W}^{(1)} + b^{(1)})\mathbf{W}^{(2)} + b^{(2)}, \tag{7}$$

where $h^{(l)} \in \mathbb{R}^{n\times d_{model}}$ is the output of MHSA sub-layer in lth Transformer layer, and $h_i^{(l)} \in \mathbb{R}^{d_{model}}$ denotes the latent representation at each sequence position. The projection matrices $\left\{\mathbf{W}^{(1)}, (\mathbf{W}^{(2)})^T\right\} \in \mathbb{R}^{d_{model}\times 4d_{model}}$ and bias vectors $\{\mathbf{b}^{(1)}, \mathbf{b}^{(2)}\} \in \mathbb{R}^{d_{model}}$ are parameters learned during training. These parameters remain consistent across all positions while are individually initialized from layer to layer. In this context, T represents the transpose operation on a matrix.

2.1.3 Residual Connection and Normalization. To alleviate the issue of vanishing or exploding gradients resulting from network deepening, the Transformer model incorporates a residual connection [94] around each of the aforementioned modules, followed by Layer Normalization [19]. For the placement of Layer Normalization operation, there are two widely used approaches: (1) *Post-Norm*: Layer normalization is implemented subsequent to the element-wise residual addition, in accordance with the vanilla Transformer [254]. (2) *Pre-Norm*: Layer normalization is applied to the input of each sub-layer, as seen in models like GPT-2 [211]. Formally, it can be formulated as:

$$\begin{aligned} \textbf{Post-Norm}: \mathbf{H}^{(\mathrm{l})} &= \text{LayerNorm}(\text{PFFN}(\mathbf{h}^{(\mathrm{l})}) + \mathbf{h}^{(\mathrm{l})}), \\ \mathbf{h}^{(\mathrm{l})} &= \text{LayerNorm}(\text{MHSA}(\mathbf{H}^{(\mathrm{l}-1)}) + \mathbf{H}^{(\mathrm{l}-1)}) \end{aligned} \tag{8}$$

$$\begin{aligned} \textbf{Pre-Norm}: \mathbf{H}^{(\mathrm{l})} &= \text{PFFN}(\text{LayerNorm}(\mathbf{h}^{(\mathrm{l})})) + \mathbf{h}^{(\mathrm{l})}, \\ \mathbf{h}^{(\mathrm{l})} &= \text{MHSA}(\text{LayerNorm}(\mathbf{H}^{(\mathrm{l}-1)})) + \mathbf{H}^{(\mathrm{l}-1)}. \end{aligned} \tag{9}$$

2.1.4 Positional Encoding. Given that self-attention alone cannot discern the positional information of each input token, the vanilla Transformer introduces an absolute positional encoding method to supplement this positional information, known as sinusoidal position embeddings [254]. Specifically, for a token at position *pos*, the position embedding is defined as:

$$\mathbf{p}_{pos,2i} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \tag{10}$$

$$\mathbf{p}_{pos,2i+1} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right), \tag{11}$$

where $2i, 2i+1$ represents the dimensions of the position embedding, while d_{model} denotes the model dimension. Subsequently, each position embedding is added to the corresponding token embedding, and the sum is fed into the Transformer. Since the inception of this method, a variety of innovative positional encoding approaches have emerged, such as learnable embeddings [67], relative position embeddings [229], RoPE [240], and ALiBi [208]. For more detailed descriptions of each method, please consult [156, 316].

2.1.5 Architecture. There are two types of Transformer architecture for code generation task, including encoder-decoder and decoder-only. For the encoder-decoder architecture, it consists of both an encoder and a decoder, in which the encoder processes the input data and generates a set of representations, which are then used by the decoder to produce the output. However, for decoder-only architecture, it consists only of the decoder part of the Transformer, where it uses a single stack of layers to both process input data and generate output. Therefore, the encoder-decoder architecture is suited for tasks requiring mapping between different input and output domains, while the decoder-only architecture is designed for tasks focused on sequence generation and continuation. The overview of LLMs with these two architectures is illustrated in Figure 2.

2.2 Code Generation

LLMs for code generation refer to the use of LLM to generate source code from natural language descriptions, a process also known as a NL2Code task. Typically, these natural language descriptions encompass programming problem statements (or docstrings) and may optionally include some programming context (e.g., function signatures, assertions). Formally, these NL descriptions can be represented as $\mathbf{x}$. Given $\mathbf{x}$, the use of an LLM with model parameters θ to generate a code solution $\mathbf{y}$ can be denoted as $P_\theta(\mathbf{y} \mid \mathbf{x})$. The in-context learning abilities of LLMs [273] has led to the appending of exemplars to the NL description $\mathbf{x}$ as demonstrations to enhance code generation performance or constrain the generation format [144, 204]. A fixed set of M exemplars is denoted as $\{(\mathbf{x}_\mathrm{i}, \mathbf{y}_\mathrm{i})\}_{i=1}^{M}$. Consequently, following [188], a more general formulation of LLMs for code generation with few-shot (or zero-shot) exemplars can be revised as:

$$P_\theta(\mathbf{y} \mid \mathbf{x}) \Rightarrow P_\theta(\mathbf{y} \mid \mathrm{prompt}(\mathbf{x}, \{(\mathbf{x}_\mathrm{i}, \mathbf{y}_\mathrm{i})\}_{i=1}^{k})), k \in \{0, 1, \ldots, M\}, \tag{12}$$

where $\mathrm{prompt}(\mathbf{x}, \{(\mathbf{x}_\mathrm{i}, \mathbf{y}_\mathrm{i})\}_{i=1}^{k})$ is a string representation of the overall input, and $\{(\mathbf{x}_\mathrm{i}, \mathbf{y}_\mathrm{i})\}_{i=1}^{k}$ denotes a set of k exemplars randomly selected from $\{(\mathbf{x}_\mathrm{i}, \mathbf{y}_\mathrm{i})\}_{i=1}^{M}$. In particular, when $k = 0$, this denotes zero-shot code generation, equivalent to vanilla ones without in-context learning. In the decoding process, a variety of decoding strategies can be performed for code generation, including deterministic-based strategies (e.g., greedy search and beam search) and sampling-based strategies (e.g., temperature sampling, top-k sampling, and top-p (nucleus) sampling). For more detailed descriptions of each decoding strategy, please refer to [99]. For example, the greedy search and sampling-based decoding strategies can be formulated as follows:

$$\textbf{Greedy Search}: \mathbf{y}^* = \underset{\mathbf{y}}{\mathrm{argmax}}\, P_\theta(\mathbf{y} \mid \mathrm{prompt}(\mathbf{x}, \{(\mathbf{x}_\mathrm{i}, \mathbf{y}_\mathrm{i})\}_{i=1}^{k})), k \in \{0, 1, \ldots, M\}, \tag{13}$$

$$\textbf{Sampling}: \mathbf{y} \sim P_\theta(\mathbf{y} \mid \mathrm{prompt}(\mathbf{x}, \{(\mathbf{x}_\mathrm{i}, \mathbf{y}_i)\}_{i=1}^{k})), k \in \{0, 1, \ldots, M\}. \tag{14}$$

To verify the functionality correctness of the generated code solution, $\mathbf{y}$ is subsequently executed via a compiler or interpreter, represented as $\mathbf{Exe}(\cdot)$, on a suit of unit tests $\mathcal{T}$. The feedback from this execution can be denoted as $\mathbf{Feedback}(\mathbf{Exe}(\mathbf{y}, \mathcal{T}))$. If the generated code solution fails to pass all relevant test cases, the error feedback can be iteratively utilized to refine the code by leveraging the previous attempt ($\mathbf{y}_{pre}$) and the associated feedback. Formally,

$$\mathbf{y} \sim P_\theta(\mathbf{y} \mid \mathrm{prompt}(\mathbf{x}, \{(\mathbf{x}_\mathrm{i}, \mathbf{y}_\mathrm{i})\}_{i=1}^{k}, \mathbf{y}_{pre}, \mathbf{Feedback}(\mathbf{Exe}(\mathbf{y}, \mathcal{T})))), k \in \{0, 1, \ldots, M\}. \tag{15}$$

Further details and relevant studies on using feedback to improve code generation are comprehensively discussed in Sections 5.6 and 5.7.

2.3 Comparison to Existing Surveys

Recent surveys [74, 101, 172, 258, 305, 315, 323] have provided valuable insights into code LLMs, covering a wide range of code-related tasks, including code summarization, translation, and repair. These surveys aim to offer an overview of the applications of LLMs in various code-related tasks. However, they often present a brief introduction to code LLMs for each task, resulting in a lack of depth and detail regarding specific code tasks. Furthermore, these surveys have overlooked the establishment of a technical taxonomy [101, 305, 323]. Consequently, researchers may struggle to fully comprehend the intricate technical advancements, principles, and emerging research topics associated with each task involving Code LLMs. This limitation hinders them from understanding the constraints and challenges of current methods and from developing innovative approaches to enhance performance in specific code tasks.

In light of these considerations, we diverge from those surveys that only provide a brief introduction to a broad spectrum of code-related tasks. Instead, our survey adopts a task-specific focus, offering a more comprehensive study.

We observe that LLMs for code generation tasks have garnered significant interest from both academia and industry, enabling even novice programmers to generate code by clearly expressing their requirements. Given the promising prospects of this area, we have chosen to conduct a task-specific survey centered on code generation. Although previous studies [17, 305] have explored LLMs for text-to-code generation, they primarily focus on models released between 2020 and 2022. This notable temporal gap has led to a lack of up-to-date literature reviews that contemplate the latest advancements, including models such as CodeQwen [246], WizardCoder [171], CodeFusion [238], and PPOCoder [235]. Moreover, there is a need for comprehensive exploration of advanced research topics, such as meticulous data curation, instruction tuning, alignment with feedback, prompting techniques, the development of autonomous coding agents, RACG, and LLM-as-a-Judge for code generation, among others.

In summary, Table 2 compares existing surveys with our study, highlighting differences across four dimensions: the code-related tasks discussed, the establishment of a techniques taxonomy, the elaboration on the latest advances, and the time span of model covered.

3 Methodology

In this section, we detail the systematic methodologies employed in conducting literature reviews. We follow the systematic literature review methodology outlined by [131], which has been widely adopted in numerous SE literature reviews [101, 145, 167, 216, 260]. The overall process is illustrated in Figure 3, and the detailed steps in our methodology are documented below.

3.1 Research Questions (RQs)

To deliver a comprehensive and up-to-date literature review on the latest advancements in LLMs for code generation, this systematic literature review addresses the following RQs:

RQ1: How can we categorize and evaluate the latest advances in LLMs for code generation between 2020 and 2024? The recent proliferation of LLMs has resulted in many of these models being adapted for code generation task. While the adaptation of LLMs for code generation essentially follows the evolution of LLMs, this evolution encompasses a broad spectrum of research directions and advancements. For SE researchers, it can be challenging and time-consuming to fully grasp the comprehensive research landscape of LLMs and their adaptation to code generation. RQ1 aims to propose a taxonomy that serves as a comprehensive reference for researchers, enabling them to

Table 2. Comparison between Existing Surveys and Our Survey, Highlighting Differences across Four Dimensions: The Code-Related Tasks Discussed, the Establishment of a Techniques Taxonomy, the Elaboration on the Latest Advances, and the Time Span of Model Covered

Venue	Survey Reference	Released Year-Month	Discussed Code Tasks	Techniques Taxonomy	Elaboration on Latest Advances	Time Span of Model Covered
ACL	Zan et al. [305]	2023-05	Code Generation	✗	✗	2020–2022
TOSEM	Hou et al. [101]	2023-09	Code Generation, Code Completion, Code Summarization Code Understanding, Code Search, Program Synthesis API Recommendation, API Synthesis, Code Comment Generation Code Representation, Method Name Generation Agile Story Point Estimation, API Documentation Smell Detection API Entity and Relation Extraction, Code Optimization Code Example Recommendation, Control Flow Graph Generation Identifier Normalization, Type Inference	✗	✗	2017–2023
arXiv	Zheng et al. [323]	2023-11	Code Generation, Test Case Generation, Code Summarization Code Translation, Vulnerability Repair	✗	✗	2020–2023
TSE	Wang et al. [258]	2023-07	Software Testing	✓	✗	2019–2023
TMLR	Zhang et al. [315]	2024-04	Code Generation, Code Completion, Code Summarization Code Understanding, Code Search, Program Synthesis API Recommendation, API Synthesis, Code Comment Generation Code Representation, Method Name Generation Agile Story Point Estimation API Documentation Smell Detection API Entity and Relation Extraction Code Optimization, Code Example Recommendation Control Flow Graph Generation Identifier Normalization, Type Inference, Requirement Analysis UI/UX Design, Model Generation, NL-to-Code Search Code-to-Code Search, API Mining, Program Synthesis Code Completion, Code Infilling, Text-To-SQL, Code Translation Program Repair, Type Prediction, Identifier Prediction Cloze Test, Code Summarization, Comment Generation Code Documentation, Document Translation, Unit Test Generation Assertion Generation, Mutant Generation, Fuzzing Defect Detection, Malware Detection, Clone Detection Code Classification, Compiler Optimization, Decompilation Deobfuscation, Log Analysis, Code Review Commit Message Generation, Workflow Automation Code Reasoning, Coding for Reasoning, AIGC Analysis	✓	✗	2020–2024
-	Ours	2024-06	Code Generation	✓	✓	2020–2024

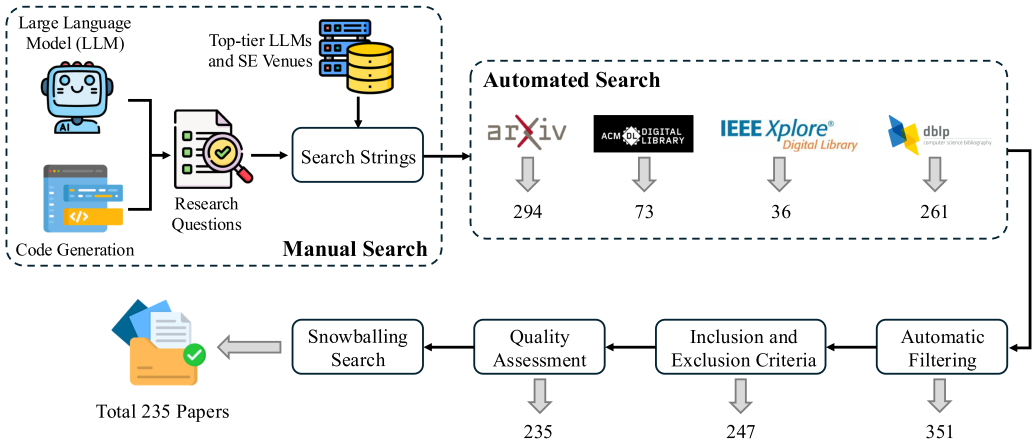


Fig. 3. Overview of the paper search and collection process.

quickly familiarize themselves with the state-of-the-art in this dynamic field and identify specific research problems and directions of interest.

RQ2: What are the key insights into LLMs for code generation? RQ2 seeks to assist researchers in establishing a comprehensive, up-to-date, and advanced understanding of LLMs for code generation.

Table 3. Publication Venues for Conference Proceedings and Journal Articles for Manual Search

Domain	Venue	Acronym
LLMs	International Conference on Learning Representations	ICLR
	Conference on Neural Information Processing Systems	NeurIPS
	International Conference on Machine Learning	ICML
	Annual Meeting of the Association for Computational Linguistics	ACL
	Conference on Empirical Methods in Natural Language Processing	EMNLP
	International Joint Conference on Artificial Intelligence	NAACL
	AAAI Conference on Artificial Intelligence	AAAI
SE	International Conference on Software Engineering	ICSE
	Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering	ESEC/FSE
	International Conference on Automated Software Engineering	ASE
	Transactions on Software Engineering and Methodology	TOSEM
	Transactions on Software Engineering	TSE
	International Symposium on Software Testing and Analysis	ISSTA

Table 4. Keywords Related to LLMs and Code Generation Task for Automated Search

Keywords Related to LLMs	Keywords Related to Code Generation Task
Code Large Language Model*, Code LLMs, Code Language Model, Code LMs, Large Language Model*, LLM, Language Model*, LM, Pre-Trained Language Model*, PLM, Pre-Trained Model, Natural Language Processing, NLP, GPT-3, ChatGPT, GPT-4, LLaMA, Code Llama, PaLM*, CodeT5, Codex, CodeGen, InstructGPT	Code Generation, Program Synthesis, Code Intelligence, *Coder*, Natural-Language-to-Code, NL2Code, Programming

This includes discussing various aspects of this rapidly evolving domain, such as data curation, latest advancements, performance evaluation, ethical and environmental implications, and real-world applications. A historical overview of the evolution of LLMs for code generation is provided, along with an empirical comparison using the widely recognized HumanEval and MBPP benchmarks, as well as the more practical and challenging BigCodeBench benchmark, to highlight the progressive enhancements in LLM capabilities for code generation. RQ2 offers an in-depth analysis of critical insights related to LLMs for code generation.

RQ3: What are the critical challenges and promising research opportunities in LLMs for code generation? Despite the revolutionary impact of LLMs on the paradigm of code generation and their remarkable performance, numerous challenges remain unaddressed. These challenges primarily stem from the gap between academic research and practical development. For instance, while the HumanEval benchmark is established as a *de facto* standard for evaluating the coding proficiency of LLMs in academia, it has been shown that this evaluation does not adequately reflect practical development scenarios [69, 72, 123, 161]. RQ3 aims to identify critical challenges and highlight promising opportunities to bridge the gap between research and practical application.

3.2 Search Process

3.2.1 Search Strings. To address the aforementioned three RQs, we initiate a manual review of conference proceedings and journal articles from top-tier venues in the fields of LLMs and SE, as detailed in Table 3. Specifically, we conducted a detailed manual review of 71 papers, comprising 40 papers sourced from the venues in Table 3 and 31 papers from arXiv. The distribution of the 40 papers from specific venues is as follows: 13 from ICLR, 4 from NeurIPS, 2 from ICML, 7 from ACL, 5 from EMNLP, 1 from NAACL, 1 from AAAI, 2 from ICSE, 1 from ESEC/FSE, 1 from ASE, 1 from TOSEM, 1 from TSE, and 1 from ISSTA. This process allowed us to identify relevant studies and derive search strings, which are subsequently utilized for an automated search across various scientific databases. The complete set of search keywords is presented in Table 4.

3.2.2 Search Databases. Following the development of search strings, we executed an automated search using four popular scientific databases: the ACM Digital Library, IEEE Xplore Digital Library, arXiv, and DBLP. Our search focuses on identifying papers whose titles contain keywords pertinent to LLMs and code generation. This approach enhances the likelihood of retrieving relevant papers since both sets of keywords must be present in the title. Although this title-based search strategy effectively retrieves a large volume of papers, it is important to note that in some instances [235], the scope of code generation can be broader, encompassing areas such as code completion, code translation, and program synthesis. As outlined in Section 1, this survey adopts a prevalent definition of code generation as the NL2Code task [17, 18, 305].

Consequently, we conduct further automatic filtering based on the content of the papers. Papers focusing on "code completion" and "code translation" are excluded. After completing the automated search, the results from each database are merged and deduplicated using scripts. This process yields 294 papers from arXiv, 73 papers from the ACM Digital Library, 36 papers from IEEE Xplore, and 261 papers from DBLP. The cut-off date for the paper collection process of this version is 15 June 2024.

3.3 Inclusion and Exclusion Criteria

The search process conducted across various databases and venues is intentionally broad to gather a comprehensive pool of candidate papers. This approach maximizes the collection of potentially relevant studies. However, such inclusivity may lead to the inclusion of papers that do not align with the scope of this survey, as well as duplicate entries from multiple sources. To address this, we have established a clear set of inclusion and exclusion criteria, based on the guidelines from [101, 258]. These criteria are applied to each paper to ensure alignment with our research scope and questions, and to eliminate irrelevant studies.

Inclusion Criteria. A paper will be included if it meets any of the following criteria:

- It is available in full text.
- It presents a dataset or benchmark specifically designed for code generation with LLMs.
- It explores specific LLM techniques, such as pre-training or instruction tuning, for code generation.
- It provides an empirical study or evaluation related to the use of LLMs for code generation.
- It discusses the ethical considerations and environmental impact of deploying LLMs for code generation.
- It proposes tools or applications powered by LLMs for code generation.

Exclusion Criteria. Conversely, papers will be excluded if they meet any of the following conditions:

- They are not written in English.
- They are found in books, theses, monographs, keynotes, panels, or venues (excluding arXiv) that do not undergo a full peer-review process.
- They are duplicate papers or different versions of similar studies by the same authors.
- They focus on text generation rather than source code generation, such as generating code comments, questions, test cases, or summarization.
- They do not address the task of code generation, for instance, focusing on code translation instead.
- They leverage SE methods to enhance code generation without emphasizing LLMs.
- They do not utilize LLMs, opting for other models like Long Short-Term Memory networks.

- They use encoder-only language models, such as BERT, which are not directly applicable to code generation task.
- LLMs are mentioned only in future work or discussions without being central to the proposed approach.

Papers identified through both manual and automated searches undergo a detailed manual review to ensure they meet the inclusion criteria and do not fall under the exclusion criteria. Specifically, two authors independently review each paper to determine its eligibility. In cases of disagreement, a third author makes the final inclusion decision.

3.4 Quality Assessment

To ensure the inclusion of high-quality studies, we have developed a comprehensive set of 10 **Quality Assessment Criteria (QAC)** following [101]. These QAC are designed to evaluate the relevance, clarity, validity, and significance of the papers considered for our review.

In accordance with [101], the first three QAC assess the study's alignment with our objectives. These criteria are rated as "irrelevant/unmet," "partially relevant/met," or "relevant/fully met," corresponding to scores of −1, 0, and 1, respectively. If a study receive a score of −1 across these initial three criteria, it is deemed ineligible for further consideration and subsequently excluded from our review process.

The subsequent seven QAC focus on a more detailed content evaluation, employing a scoring range of −1 to 2, representing "poor," "fair," "good," and "excellent." We compute a cumulative score based on the responses to QAC4 through QAC10 for each paper. For published works, the maximum achievable score is 14 (2 points per question). We retain those with a score of 11.2 (80% of the total score) or higher. For unpublished papers available on arXiv, QAC4 defaults to a score of 0, making the maximum possible score for the remaining criteria 12. Accordingly, we retain papers scoring 9.6 (80% of the adjusted total score) or above.

- QAC1: Is the research not classified as a secondary study, such as a systematic literature review or survey? (−1, 0, 1)
- QAC2: Does the study incorporate the use of LLMs? (−1, 0, 1)
- QAC3: Is the study relevant to the code generation task? (−1, 0, 1)
- QAC4: Is the research published in a prestigious venue? (−1, 0, 1, 2)
- QAC5: Does the study present a clear research motivation? (−1, 0, 1, 2)
- QAC6: Are the key contributions and limitations of the study discussed? (−1, 0, 1, 2)
- QAC7: Does the study contribute to the academic or industrial community? (−1, 0, 1, 2)
- QAC8: Are the LLM techniques employed in the study clearly described? (−1, 0, 1, 2)
- QAC9: Are the experimental setups, including experimental environments and dataset information, thoroughly detailed? (−1, 0, 1, 2)
- QAC10: Does the study clearly confirm its experimental findings? (−1, 0, 1, 2)

3.5 Snowballing Search

Following the quality assessment, we establish an initial set of papers for our study. To minimize the risk of excluding pertinent literature, we implement a snowballing search strategy. Snowballing search involves utilizing a paper's reference list or its citations to discover additional relevant studies, known as backward and forward snowballing, respectively. In this survey, we exclusively employed backward snowballing following [258]. Despite this effort, no additional studies are identified through this method. This could be attributed to the task-specific nature of the code generation (NL2Code), where reference studies are typically published earlier. Consequently, our methodology, which encompassed an extensive manual and automated search, likely covered the relevant literature comprehensively, explaining the lack of additional studies through snowballing search.

3.6 Data Collection and Analysis

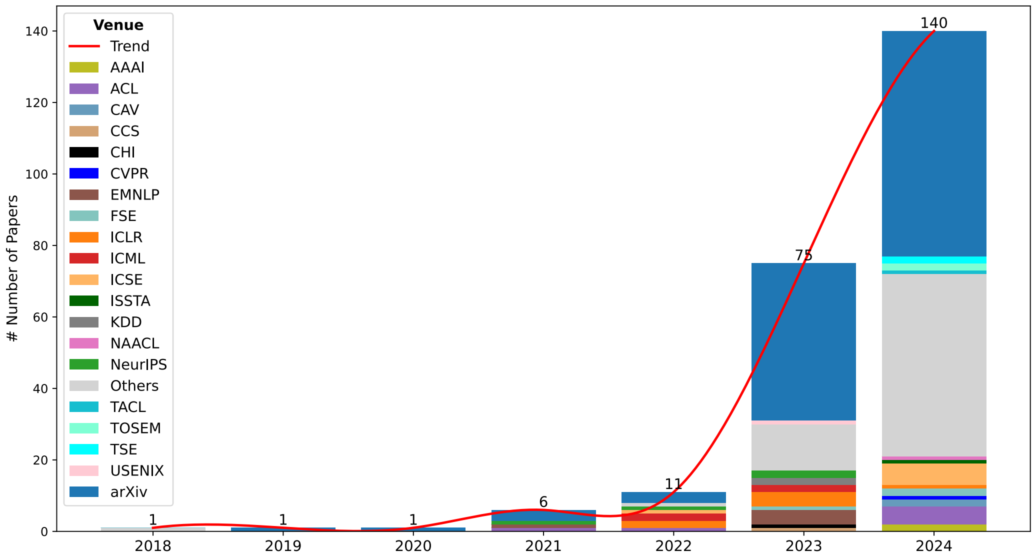
The data collection process for our study, illustrated in Figure 3, began with a manual search through conference proceedings and journal articles from leading venues in LLMs and SE. This initial step yielded 42 papers, from which we extracted relevant search strings. Following this, we performed an automated search across four academic databases using keyword-based queries, resulting in the retrieval of 664 papers. After performing automatic filtering (351 papers), applying inclusion and exclusion criteria (247 papers), conducting quality assessments (235 papers), and utilizing snowballing search (235 papers), we finalize a collection of 235 papers focusing on LLMs for code generation.

To provide insights from the selected papers, we begin by presenting an overview of their distribution across publication venues each year, as illustrated at the top of Figure 4. Our analysis indicates that 14% of the papers are published in LLM-specific venues and 7% in SE venues. The venues for LLM and SE are presented in Table 2. Remarkably, 49% of the papers remain unpublished in peer-reviewed venues and are available on arXiv. This trend is understandable given the emerging nature of this field, with many works being recent and pending formal submission. Despite the absence of peer review on arXiv, our quality assessment process ensures that only high-quality papers are included, thereby maintaining the integrity of this survey. Furthermore, the annual trend in the number of collected papers indicates nearly exponential growth in the field. From a single paper in the period 2018 to 2020, the numbers increased to 6 in 2021, 11 in 2022, 75 in 2023, and 140 in 2024. This trend reflects growing interest and attention in this research area, with expectations for continued expansion in the future. Additionally, to capture the breadth of advancements in LLMs for code generation, we conducted a distribution analysis of the research topics covered in the included papers, as shown at the bottom of Figure 4. We observe that the development of LLMs for code generation closely aligns with broader trends in general-purpose LLM research. Notably, the most prevalent research topics are Pre-Training and Foundation Models (21.5%), Prompting (11.8%), and Evaluation and Benchmarks (24.1%). These areas hold significant promise for enhancing, refining, and evaluating LLM-driven code generation.

4 Taxonomy

The recent surge in the development of LLMs has led to a significant number of these models being repurposed for code generation task through continual pre-training or fine-tuning. This trend is particularly observable in the realm of open source models. For instance, Meta AI initially made the LLaMA [249] model publicly available, which was followed by the release of Code Llama [224], designed specifically for code generation. Similarly, DeepSeek LLM [27] developed and released by DeepSeek has been extended to create DeepSeek Coder [88], a variant tailored for code generation. The Qwen team has developed and released Code Qwen [246], building on their original Qwen [21] model. Microsoft has unveiled WizardLM [287] and is exploring its coding-oriented counterpart, WizardCoder [171]. Google has joined the fray by releasing Gemma [245], subsequently followed by CodeGemma [60]. Beyond simply adapting general-purpose LLMs for code-related tasks, there has been a proliferation of models specifically engineered for code generation. Notable examples include StarCoder [146], OctoCoder [185], and CodeGen [191]. These models underscore the trend of LLMs being developed with a focus on code generation.

Recognizing the importance of these developments, we conduct a thorough analysis of the algorithms, advanced techniques, and research topics presented in the selected 235 papers on LLMs for code generation, sourced from widely used scientific databases as mentioned in Section 3. Based



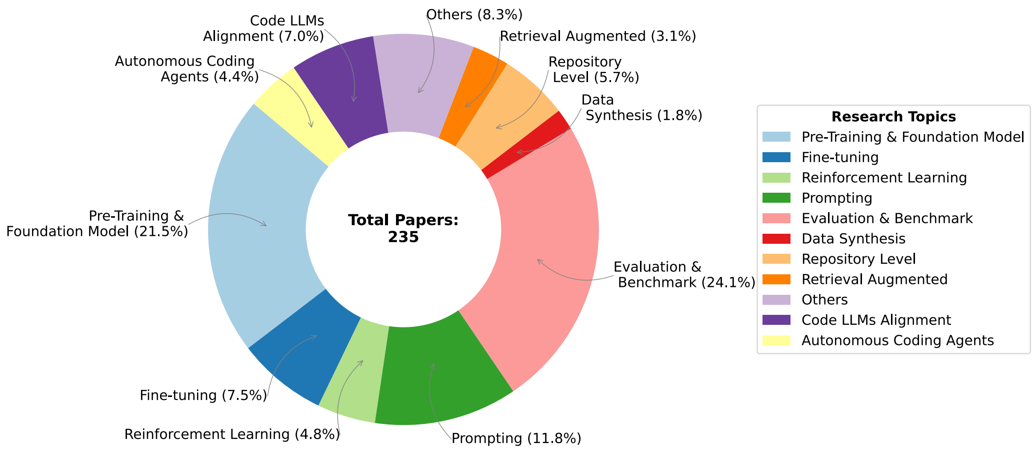


Fig. 4. Data qualitative analysis. *Top*: Annual distribution of selected papers across various publication venues. *Bottom*: Distribution analysis of research topics covered in the included papers.

on this analysis, we propose a taxonomy that categorizes and evaluates the latest advancements in LLMs for code generation, as depicted in Figure 6. This taxonomy serves as a comprehensive reference for researchers seeking to quickly familiarize themselves with the state-of-the-art in this dynamic field. It is important to highlight that the category of recent advances emphasizes the core techniques used in the current state-of-the-art code LLMs.

In the subsequent sections, we will provide an in-depth analysis of each category related to code generation. This will encompass a definition of the problem, the challenges to be addressed, and a comparison of the most prominent models and their performance evaluation.

5 LLMs for Code Generation

LLMs with Transformer architecture have revolutionized a multitude of fields, and their application in code generation has been particularly impactful. These models follow a comprehensive process

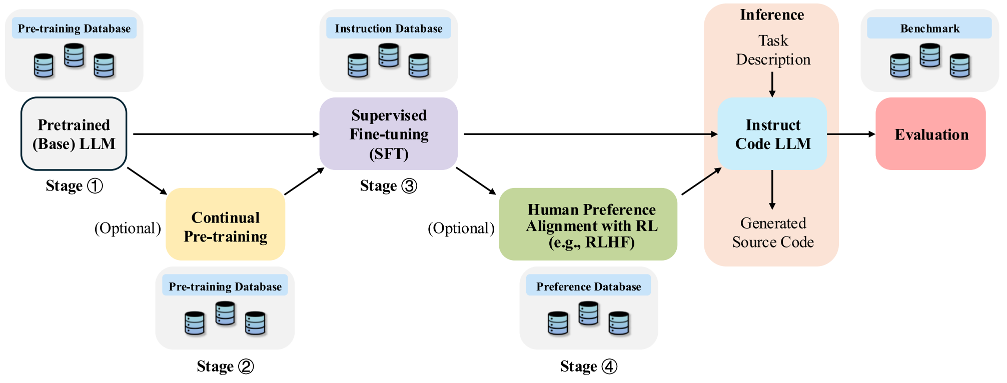


Fig. 5. A diagram illustrating the general training, inference, and evaluation workflow for Code LLMs and their associated databases. The training workflow is mainly divided into four distinct stages: Stages *x*2460 and *x*2461 are the pre-training phase, whereas Stages *x*2462 and *x*2463 represent the post-training phases. It is important to note that Stages *x*2461 and *x*2463 are optional. For instance, StarCoder [146] incorporates only Stage *x*2460. WizardCoder [171], fine-tuned upon StarCoder, includes only Stage *x*2462, while Code Llama [224], continually pre-trained on Llama 2, encompasses Stages *x*2461 and *x*2462. DeepSeek-Coder-V2 [330], continually pre-trained on DeepSeek-V2, covers Stages *x*2461, *x*2462, and *x*2463. Note that pre-trained model can be directly used for inference through prompt engineering.

that starts with the curation and synthesis of code data, followed by a structured training approach that includes pre-training and fine-tuning (instruction tuning), reinforcement learning with various feedback, and the use of sophisticated prompt engineering techniques. Recent advancements have seen the integration of repository-level and RACG, as well as the development of autonomous coding agents. Furthermore, the evaluation of coding abilities of LLMs has become a critical component of this research area. Figure 5 illustrates the general training, inference, and evaluation workflow for Code LLMs and their associated databases.

In the forthcoming sections, we will explore these dimensions of LLMs in the context of code generation in detail, following the structure outlined in our taxonomy in Figure 6. Section 5.2 will address the data curation and processing strategies employed throughout the various stages of LLM development. Section 5.3 will discuss data synthesis methods designed to mitigate the scarcity of high-quality data. Section 5.4 will outline the prevalent model architectures used in LLMs for code generation. Moving to Section 5.5, we will examine the techniques for **Full Parameter Fine-Tuning (FFT)** and **Parameter-Efficient Fine-Tuning (PEFT)**, which are essential for tailoring LLMs to code generation task. Section 5.6 will shed light on enhancing code quality through reinforcement learning, utilizing the external feedback. Section 5.7 will delve into the tactical use of prompts to maximize the coding capabilities of LLMs. The innovative approaches of repository-level and RACG will be elaborated in Sections 5.8 and 5.9, respectively. Additionally, Section 5.10 will discuss the exciting field of autonomous coding agents. Section 5.11 discusses various evaluation strategies and offer an empirical comparison using the widely recognized HumanEval, MBPP, and the more practical and challenging BigCodeBench benchmarks to highlight the progressive enhancements in LLM capabilities for code generation. Furthermore, the ethical implications and the environmental impact of using LLMs for code generation are discussed in Section 5.12, aiming to establish a trustworthiness, responsibility, safety, efficiency, and sustainability of LLMs for code generation. Lastly, Section 5.13 will provide insights into some of the practical applications that leverage LLMs for code generation, demonstrating the real-world impact of these sophisticated

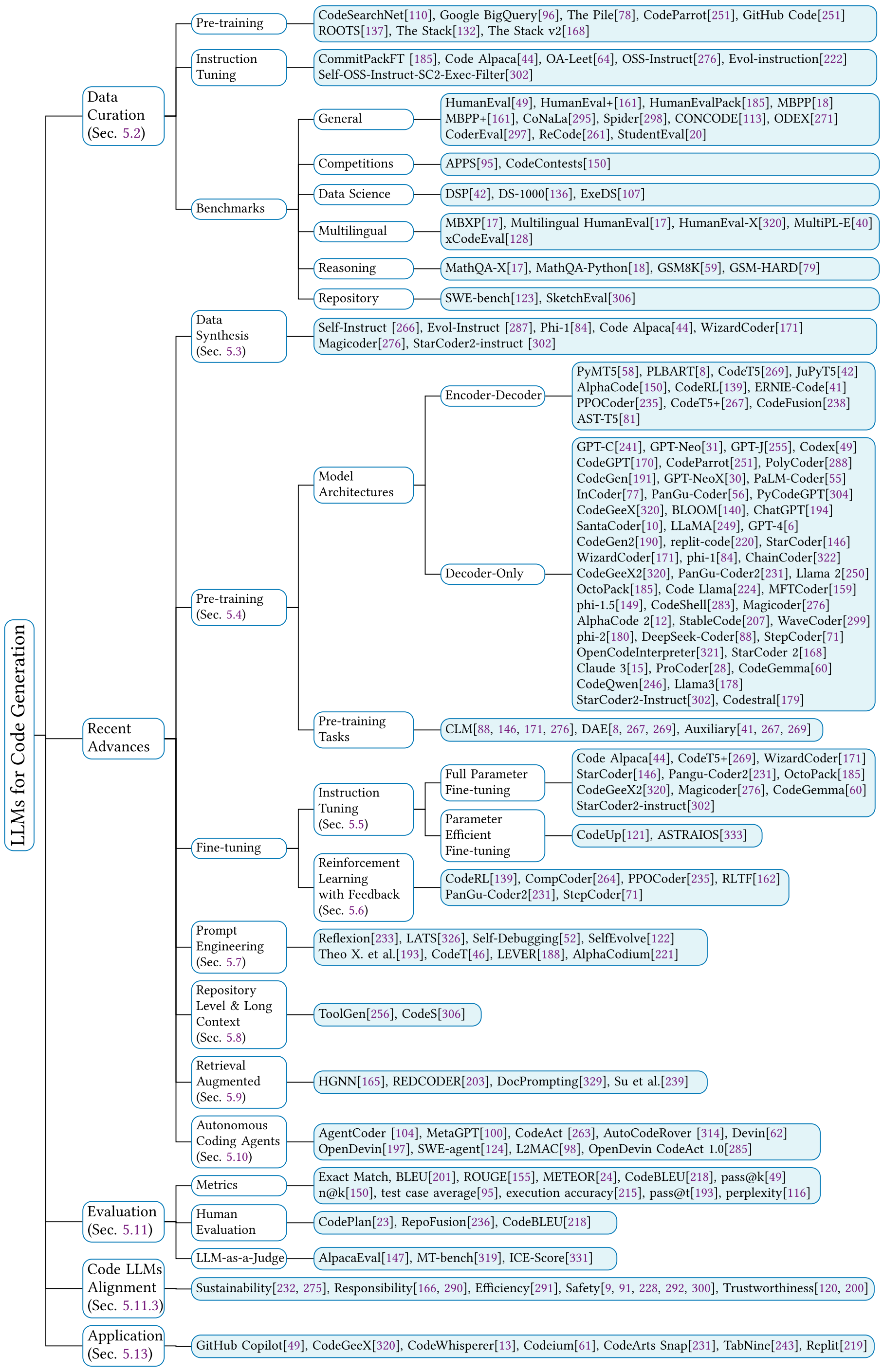

LLMs for Code Generation
Data Curation (Sec. 5.2)
Pre-training
CodeSearchNet[110], Google BigQuery[96], The Pile[78], CodeParrot[251], GitHub Code[251] ROOTS[137], The Stack[132], The Stack v2[168]
Instruction Tuning
CommitPackFT [185], Code Alpaca[44], OA-Leet[64], OSS-Instruct[276], Evol-instruction[222] Self-OSS-Instruct-SC2-Exec-Filter[302]
Benchmarks
General
HumanEval[49], HumanEval+[161], HumanEvalPack[185], MBPP[18] MBPP+[161], CoNaLa[295], Spider[298], CONCODE[113], ODEX[271] CoderEval[297], ReCode[261], StudentEval[20]
Competitions
APPS[95], CodeContests[150]
Data Science
DSP[42], DS-1000[136], ExeDS[107]
Multilingual
MBXP[17], Multilingual HumanEval[17], HumanEval-X[320], MultiPL-E[40] xCodeEval[128]
Reasoning
MathQA-X[17], MathQA-Python[18], GSM8K[59], GSM-HARD[79]
Repository
SWE-bench[123], SketchEval[306]
Recent Advances
Data Synthesis (Sec. 5.3)
Self-Instruct [266], Evol-Instruct [287], Phi-1[84], Code Alpaca[44], WizardCoder[171] Magicoder[276], StarCoder2-instruct [302]
Pre-training (Sec. 5.4)
Model Architectures
Encoder-Decoder
PyMT5[58], PLBART[8], CodeT5[269], JuPyT5[42] AlphaCode[150], CodeRL[139], ERNIE-Code[41] PPOCoder[235], CodeT5+[267], CodeFusion[238] AST-T5[81]
Decoder-Only
GPT-C[241], GPT-Neo[31], GPT-J[255], Codex[49] CodeGPT[170], CodeParrot[251], PolyCoder[288] CodeGen[191], GPT-NeoX[30], PaLM-Coder[55] InCoder[77], PanGu-Coder[56], PyCodeGPT[304] CodeGeeX[320], BLOOM[140], ChatGPT[194] SantaCoder[10], LLaMA[249], GPT-4[6] CodeGen2[190], replit-code[220], StarCoder[146] WizardCoder[171], phi-1[84], ChainCoder[322] CodeGeeX2[320], PanGu-Coder2[231], Llama 2[250] OctoPack[185], Code Llama[224], MFTCoder[159] phi-1.5[149], CodeShell[283], Magicoder[276] AlphaCode 2[12], StableCode[207], WaveCoder[299] phi-2[180], DeepSeek-Coder[88], StepCoder[71] OpenCodeInterpreter[321], StarCoder 2[168] Claude 3[15], ProCoder[28], CodeGemma[60] CodeQwen[246], Llama3[178] StarCoder2-Instruct[302], Codestral[179]
Pre-training Tasks
CLM[88, 146, 171, 276], DAE[8, 267, 269], Auxiliary[41, 267, 269]
Fine-tuning
Instruction Tuning (Sec. 5.5)
Full Parameter Fine-tuning
Code Alpaca[44], CodeT5+[269], WizardCoder[171] StarCoder[146], Pangu-Coder2[231], OctoPack[185] CodeGeeX2[320], Magicoder[276], CodeGemma[60] StarCoder2-instruct[302]
Parameter Efficient Fine-tuning
CodeUp[121], ASTRAIOS[333]
Reinforcement Learning with Feedback (Sec. 5.6)
CodeRL[139], CompCoder[264], PPOCoder[235], RLTF[162] PanGu-Coder2[231], StepCoder[71]
Prompt Engineering (Sec. 5.7)
Reflexion[233], LATS[326], Self-Debugging[52], SelfEvolve[122] Theo X. et al.[193], CodeT[46], LEVER[188], AlphaCodium[221]
Repository Level & Long Context (Sec. 5.8)
ToolGen[256], CodeS[306]
Retrieval Augmented (Sec. 5.9)
HGNN[165], REDCODER[203], DocPrompting[329], Su et al.[239]
Autonomous Coding Agents (Sec. 5.10)
AgentCoder [104], MetaGPT[100], CodeAct [263], AutoCodeRover [314], Devin[62] OpenDevin[197], SWE-agent[124], L2MAC[98], OpenDevin CodeAct 1.0[285]
Evaluation (Sec. 5.11)
Metrics
Exact Match, BLEU[201], ROUGE[155], METEOR[24], CodeBLEU[218], pass@k[49] n@k[150], test case average[95], execution accuracy[215], pass@t[193], perplexity[116]
Human Evaluation
CodePlan[23], RepoFusion[236], CodeBLEU[218]
LLM-as-a-Judge
AlpacaEval[147], MT-bench[319], ICE-Score[331]
Code LLMs Alignment (Sec. 5.11.3)
Sustainability[232, 275], Responsibility[166, 290], Efficiency[291], Safety[9, 91, 228, 292, 300], Trustworthiness[120, 200]
Application (Sec. 5.13)
GitHub Copilot[49], CodeGeeX[320], CodeWhisperer[13], Codeium[61], CodeArts Snap[231], TabNine[243], Replit[219]

Fig. 6. Taxonomy of LLMs for code generation.

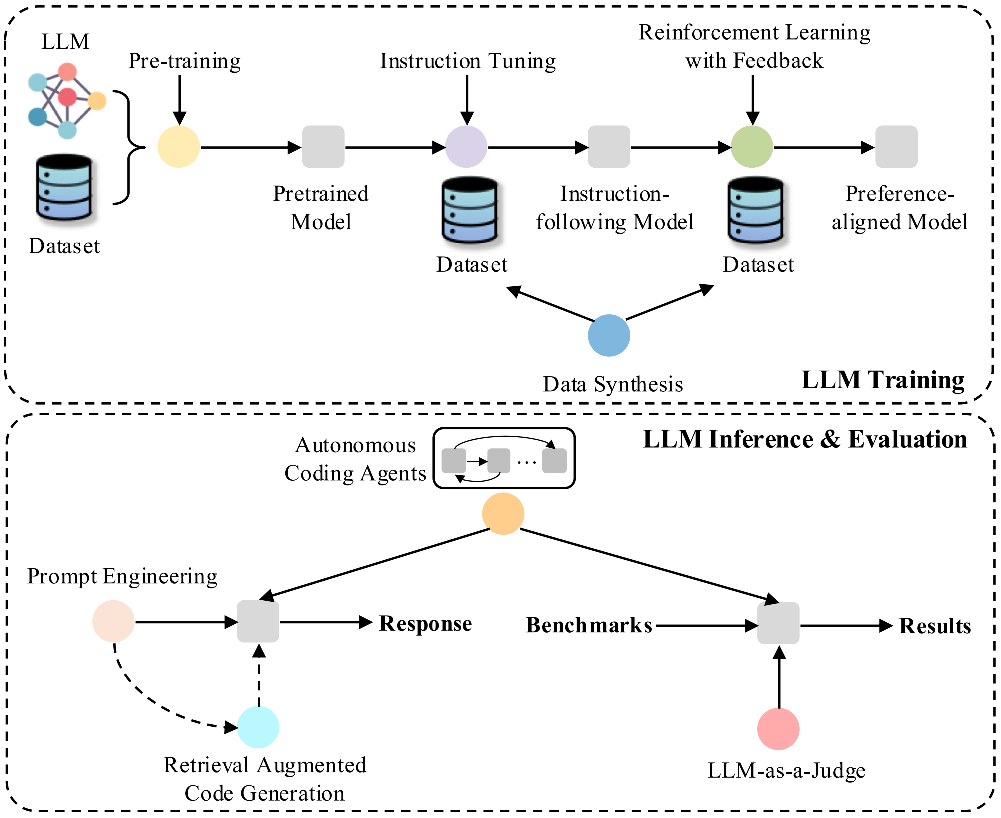


Fig. 7. The diagram provides a comprehensive overview of various techniques and their interconnections within the development of LLMs. Circular icons, distinguished by different colors, represent the specific techniques, while gray rectangles denote the evolved models with corresponding techniques. The upper section of the diagram outlines the techniques involved in the model training process, such as pre-training, instruction tuning, and reinforcement learning from human feedback (RLHF), as well as the incorporation of synthetic data. The lower section highlights the techniques related to model inference and evaluation, including prompt engineering, multi-turn prompting, retrieval-augmented generation, and LLM-based evaluations like LLM-as-a-Judge. This visual representation underscores the dynamic evolution and integration of innovations in LLM development, facilitating a clearer understanding of how these technologies progressively enhance model capabilities and address various challenges.

models. Through this comprehensive exploration, we aim to highlight the significance and potential of LLMs within the domain of automated code generation.

5.1 Overview of the Evolution and Interrelation

The evolution of LLMs for code generation has been marked by the integration of numerous innovative techniques, which have enhanced their capabilities over time, as shown in Figure 7.

Initially, LLMs are pre-trained on extensive, unlabeled datasets to imbue them with a broad range of general knowledge [210]. Despite this foundational knowledge, pre-trained models often demand substantial effort in prompt engineering to yield high-quality responses. To improve their response to follow human instructions, a technique known as instruction tuning has been developed. This involves supervised fine-tuning of pre-trained LLMs on instructional datasets [114, 171, 198, 272, 311]. Nevertheless, even with instruction tuning, LLMs can produce outputs that are untruthful, toxic, offensive, or unhelpful, reflecting a misalignment with user expectations. Addressing these issues, **Reinforcement Learning from Human Feedback (RLHF)** has been employed to further fine-tune instruction-tuned models [198]. However, creating high-quality instructional and preference datasets presents significant challenges due to data scarcity, privacy concerns, and high costs [164, 266]. Human-generated data can be labor-intensive and costly, often lacking the scope and detail necessary for complex or ambiguous scenarios. To address these challenges, synthetic data has emerged as a viable alternative, allowing models like GPT-4 [6] to generate rich datasets without human annotation [92, 138, 164, 266].

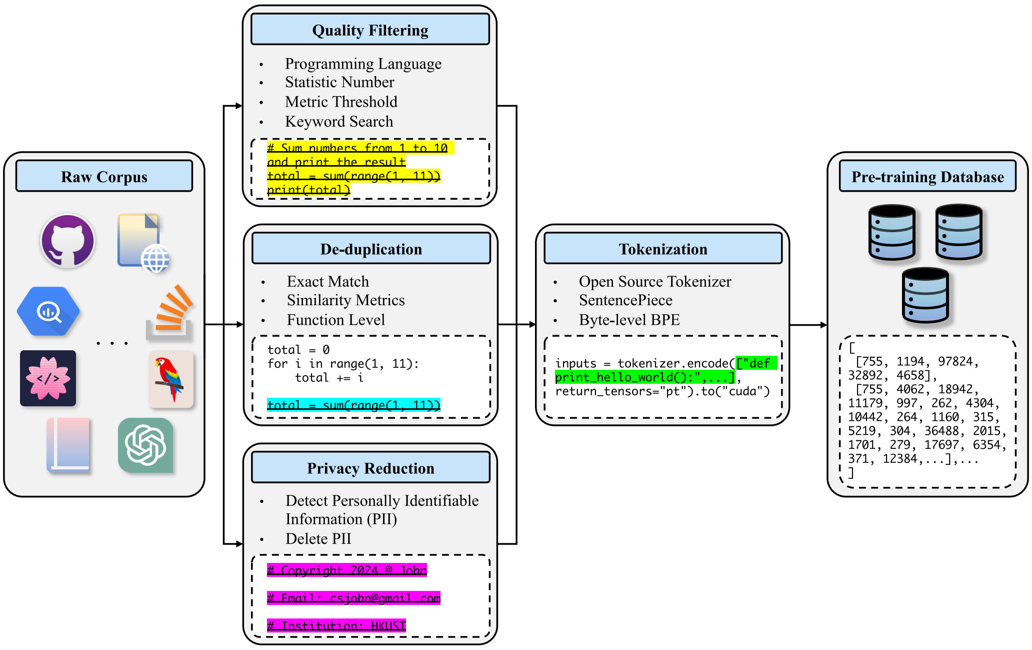
Following the resource-intensive model training, they undergo inference and evaluation. Various benchmarks have been developed to quantitatively assess the performance of LLMs, and their ability to follow instructions has prompted researchers to explore LLM-based evaluations. The concept of LLM-as-a-Judge involves using advanced proprietary LLMs (e.g., GPT-4) as proxies for human evaluators. This approach involves crafting prompts with specific requirements to guide LLMs in conducting evaluations [147, 319]. This methodology reduces dependency on human participation, enabling more efficient and scalable evaluations while providing insightful explanations for rating scores [318]. Once rigorously evaluated, models are deployed for inference, where prompt engineering plays a crucial role in enhancing task completion reliability and performance. To overcome the limitations of single-attempt failures, multi-turn prompting has been utilized to iteratively improve model performance. Despite their impressive capabilities, LLMs face challenges such as hallucination [153, 313], outdated knowledge [115], and opaque reasoning processes [33, 80, 106, 274, 328]. Techniques like instruction tuning and RLHF help mitigate these issues but also introduce new challenges, such as catastrophic forgetting and the need for substantial computational resources [90, 199]. Recently, **Retrieval-Augmented Generation (RAG)** has emerged as a novel approach to overcome these limitations by integrating external database knowledge [48, 80, 143, 317].

What's more, a rapidly expanding array of applications for LLM-based autonomous agents underscores the promise of this technology [1–3, 281]. These LLM-based agents are sophisticated systems with reasoning abilities, employing an LLM as a central computational engine to formulate and execute problem-solving plans through tool-enabled functions or API calls. They operate within shared environments, enabling communication and interaction in cooperative, competitive, or negotiating contexts [104, 259, 281]. The typical architecture includes an LLM-based agent, a memory module, a planning component, and a tool utilization module.

These advancements are progressively establishing more capable LLMs for various tasks, promising exciting prospects for future developments in the field. We will explore each technique and its associated works in the following sections.

5.2 Data Curation and Processing

The exceptional performance of LLMs can be attributed to their training on large-scale and diverse datasets [305]. Meanwhile, the extensive parameters of these models necessitate substantial data to unlock their full potential, in alignment with established scaling law [97, 127]. For a general-purpose LLM, amassing a large-scale corpus of NL from a variety of sources is imperative. Such sources include Web pages, conversation data, books and news, scientific data, and code [21, 34, 55, 249, 250, 296], while these data are often crawled from the web and must undergo meticulous and



```
~~# Sum numbers from 1 to 10~~
~~and print the result~~
~~total = sum(range(1, 11))~~
~~print(total)~~
```

```
total = 0
for i in range(1, 11):
    total += i

~~total = sum(range(1, 11))~~
```

```
~~# Copyright 2024 @ John~~

~~# Email: csjohn@gmail.com~~

~~# Institution: HKUST~~
```

```
inputs = tokenizer.encode(["def
print_hello_world():",...],
return_tensors="pt").to("cuda")
```

```
[
 [755, 1194, 97824,
32892, 4658],
 [755, 4062, 18942,
11179, 997, 262, 4304,
10442, 264, 1160, 315,
5219, 304, 36488, 2015,
1701, 279, 17697, 6354,
371, 12384,...],...
]
```

Fig. 8. A diagram depicting the standard data preprocessing workflow utilized in the pre-training phase of LLMs for code generation.

comprehensive pre-processing [214, 315]. Fortunately, multiple platforms and Web sites offer large-scale, open source, and permissively licensed code corpora, such as GitHub[7] and Stack Overflow.[8] Notably, the number of stars or forks of GitHub repositories has emerged as a valuable metric for filtering high-quality code datasets. In a similar vein, the quantity of votes on Stack Overflow can serve to discern the most relevant and superior answers.

Nonetheless, raw datasets are frequently laden with redundant, noisy data and personal information, eliciting concerns regarding privacy leakage, which may include the names and e-mail addresses of repository contributors [9, 38, 137]. Consequently, it is essential to undertake rigorous data-cleaning procedures. Typically, this process encompasses exact match deduplication, code data filtering based on average line length and a defined threshold for the fraction of alphanumeric characters, the removal of auto-generated files through keyword searches, and the expunction of personal user data [132, 251]. Specifically, the standard data preprocessing workflow is depicted in Figure 8.

The development of a proficient LLM for code generation necessitates the utilization of various types of code data at different developmental stages. Therefore, we categorize code data into three distinct classes: pre-training datasets, instruction-tuning datasets, and benchmarks for performance evaluation. The subsequent subsections will provide a detailed illustration of code data within each classification.

5.2.1 *Pre-Training.*

The remarkable success of bidirectional **Pre-Trained Language Models (PLMs)** such as BERT [67] and unidirectional PLMs like GPT [210] has firmly established the practice of pre-training on large-scale unlabeled datasets to endow models with a broad spectrum of general

[7]https://github.com.
[8]https://stackoverflow.com.

Table 5. The Statistics of Some Commonly Used Pre-Training Datasets for LLMs Aimed at Code Generation

Dataset	Size (GB)	Files (M)	#PL	Date	Link
CodeSearchNet [110]	20	6.5	6	2022-01	https://huggingface.co/datasets/code$_search_net$
Google BigQuery[96]	-	-	-	2016-06	github-on-bigquery-analyze-all-the-open-source-code
The Pile [78]	95	19	-	2022-01	https://huggingface.co/datasets/EleutherAI/pile
CodeParrot [251]	180	22	1	2021-08	https://huggingface.co/datasets/transformersbook/codeparrot
GitHub Code[251]	1,024	115	32	2022-02	https://huggingface.co/datasets/codeparrot/github-code
ROOTS [137]	163	15	13	2023-03	https://huggingface.co/bigscience-data
The Stack [132]	3,136	317	30	2022-10	https://huggingface.co/datasets/bigcode/the-stack
The Stack v2 [168]	32K	3K	619	2024-04	https://huggingface.co/datasets/bigcode/the-stack-v2

The column labeled "**#PL**" indicates the number of programming languages included in each dataset. It should be noted that in the CodeSearchNet [110] dataset, each file represents a function, and for the Pile [78] and ROOTS [137] datasets, only the code components are considered.

knowledge. Extending this principle to the realm of code generation enables LLMs to assimilate fundamental coding principles, including the understanding of code structure dependencies, the semantics of code identifiers, and the intrinsic logic of code sequences [49, 85, 267, 269]. In light of this advancement, there has been a proliferation of large-scale unlabeled code datasets proposed to serve as the foundational training ground for LLMs to develop coding proficiency. A brief introduction of these datasets is as follows, with the statistics available in Table 5.

— CodeSearchNet [110]: CodeSearchNet corpus is a comprehensive dataset, consisting of 2 million (comment, code) pairs from open source repositories on GitHub. It includes code and documentation in several PLs including Go, Java, PHP, Python, JavaScript, and Ruby. The dataset was primarily compiled to promote research into the problem of code retrieval using NL.
— Google BigQuery [96]: The Google BigQuery Public Datasets program offers a full snapshot of the content of more than 2.8 million open source GitHub repositories in BigQuery.
— The Pile [78]: The Pile is an 825 GiB diverse and open source language modeling dataset aggregating 22 smaller, high-quality datasets including GitHub, Books3, and Wikipedia (en). It aims to encompass text from as many modalities as possible, thereby facilitating the development of models with broader generalization capabilities. For code generation, the GitHub component is specifically utilized.
— CodeParrot [251]: The CodeParrot dataset contains Python files used to train the code generation model in Chapter 10: Training Transformers from Scratch in the "NLP with Transformers book" [251]. Created with the GitHub dataset available via Google's BigQuery, the CodeParrot dataset includes approximately 22 million Python files and is 180 GB (50 GB compressed) big.
— GitHub Code [251]: The GitHub Code dataset comprises 115M code files derived from GitHub, spanning 32 PLs and 60 extensions totaling 1 TB of data. The dataset was created from the public GitHub dataset on Google BiqQuery.
— ROOTS [137]: The BigScience ROOTS Corpus is a 1.6 TB dataset spanning 59 languages that was used to train the 176B **BigScience Large Open-Science Open-Access Multilingual (BLOOM)** language model. For the code generation task, the code subset of the ROOTS Corpus will be specifically utilized.
— The Stack [132]: The Stack contains over 6 TB of permissively licensed source code files that cover 358 PLs. The dataset was compiled as part of the BigCode Project, an open scientific collaboration working on the responsible development of Code LLMs.

Table 6. The Statistics of Several Representative Datasets Used in Instruction-Tuning LLMs for Code Generation

Dataset	Size	#PL	Date	Link
CodeAlpaca-20K [44]	20k	-	2023-03	https://huggingface.co/datasets/sahil2801/CodeAlpaca-20k
CommitPackFT [185]	742k	277	2023-08	https://huggingface.co/datasets/bigcode/commitpackft
Evol-Instruct-Code-80k [222]	80k	-	2023-07	https://huggingface.co/datasets/nickrosh/Evol-Instruct-Code-80k-v1
evol-codealpaca-v1 [248]	110k	-	2023-07	https://huggingface.co/datasets/theblackcat102/evol-codealpaca-v1
Magicoder-OSS-Instruct-75k [276]	75k	Python, Shell, TypeScript, C++, Rust, PHP, Java, Swift, C#	2023-12	https://huggingface.co/datasets/ise-uiuc/Magicoder-OSS-Instruct-75K
Self-OSS-Instruct-SC2-Exec-Filter-50k [302]	50k	Python	2024-04	https://huggingface.co/datasets/bigcode/self-oss-instruct-sc2-exec-filter-50k

The column labeled "**#PL**" indicates the number of programming languages encompassed by each dataset.

— The Stack v2 [168]: The Stack v2, a dataset created as part of the BigCode Project, contains over 3B files across more than 600 programming and markup languages. The dataset is derived from the Software Heritage archive,[9] the largest public archive of software source code and accompanying development history.

5.2.2 Instruction Tuning. Instruction tuning refers to the process of supervised fine-tuning LLMs using a collection of datasets structured as various instructions, with the purpose of following a wide range of task instructions [57, 198, 226, 272]. This method has demonstrated a considerable improvement in model performance and an enhanced ability to generalize to unseen tasks that the model has not previously encountered, as evidenced by recent studies [57, 198]. Leveraging the benefits of instruction tuning, instruction tuning has been expanded into coding domains, especially for code generation, which involves the automatic generation of the intended code from a NL description. The promise of instruction tuning in this area has led numerous researchers to develop large-scale instruction-tuning datasets tailored for code generation. Below, we provide an overview of several notable datasets tailored for instruction tuning, with their respective statistics detailed in Table 6.

— CodeAlpaca-20k [44]: CodeAlpaca-20k is a collection of 20K instruction-following examples generated using the data synthesis techniques termed Self-Instruct outlined in [266], with modifications for code generation, editing, and optimization tasks instead of general tasks.
— CommitPackFT [185]: CommitPackFT is a 2 GB refined version of CommitPack [185]. It is filtered to only include high-quality commit messages that resemble NL instructions.
— Evol-Instruct-Code-80k [222]: Evol-Instruct-Code-80k is an open source implementation of Evol-Instruct-Code described in the WizardCoder paper [171], which enhances the fine-tuning effect of pre-trained code large models by adding complex code instructions.
— Magicoder-OSS-Instruct-75k [276]: This is a 75k synthetic data generated through OSS-Instruct with `gpt-3.5-turbo-1106` and used to train both Magicoder and Magicoder-S series models.
— Self-OSS-Instruct-SC2-Exec-Filter-50k [302]: Self-OSS-Instruct-SC2-Exec-Filter-50k is generated by StarCoder2-15B using the OSS-Instruct [276] data synthesis approach. It was subsequently used to fine-tune StarCoder-15B without any human annotations or distilled data from huge and proprietary LLMs.

5.2.3 Benchmarks. To rigorously assess the efficacy of LLMs for code generation, the research community has introduced a variety of high-quality benchmarks in recent years. Building on the foundational work by [49], numerous variations of the HumanEval dataset and additional

[9]https://archive.softwareheritage.org.

Table 7. The Detailed Statistics of Commonly Used Benchmarks Used in Evaluating LLMs for Code Generation

Scenario	Benchmark	Size	#PL	Date	Link
General	HumanEval [49]	164	Python	2021-07	https://huggingface.co/datasets/openai$_humaneval$
	HumanEval+ [161]	164	Python	2023-05	https://huggingface.co/datasets/evalplus/humanevalplus
	HumanEvalPack [185]	164	6	2023-08	https://huggingface.co/datasets/bigcode/humanevalpack
	MBPP [18]	974	Python	2021-08	https://huggingface.co/datasets/mbpp
	MBPP+ [161]	378	Python	2023-05	https://huggingface.co/datasets/evalplus/mbppplus
	CoNaLa [295]	596.88K	Python	2018-05	https://huggingface.co/datasets/neulab/conala
	Spider [298]	8,034	SQL	2018-09	https://huggingface.co/datasets/xlangai/spider
	CONCODE [113]	104K	Java	2018-08	https://huggingface.co/datasets/AhmedSSoliman/CONCOD
	ODEX [271]	945	Python	2022-12	https://huggingface.co/datasets/neulab/odex
	CoderEval [297]	460	Python, Java	2023-02	https://github.com/CoderEval/CoderEval
	ReCode [261]	1,138	Python	2022-12	https://github.com/amazon-science/recode
	StudentEval [20]	1,749	Python	2023-06	https://huggingface.co/datasets/wellesley-easel/StudentEval
	BigCodeBench [332]	1,140	Python	2024-06	https://huggingface.co/datasets/bigcode/bigcodebench
	ClassEval [72]	100	Python	2023-08	https://huggingface.co/datasets/FudanSELab/ClassEval
	NaturalCodeBench [312]	402	Python, Java	2024-05	https://github.com/THUDM/NaturalCodeBench
Competitions	APPS [95]	10,000	Python	2021-05	https://huggingface.co/datasets/codeparrot/apps
	CodeContests [150]	13,610	C++, Python, Java	2022-02	https://huggingface.co/datasets/deepmind/code_contests
	LiveCodeBench [186]	713 Updating	Python	2024-03	https://github.com/LiveCodeBench/LiveCodeBench
Data Science	DSP [42]	1,119	Python	2022-01	https://github.com/microsoft/DataScienceProblems
	DS-1000 [136]	1,000	Python	2022-11	https://huggingface.co/datasets/xlangai/DS-1000
	ExeDS [107]	534	Python	2022-11	https://github.com/Jun-jie-Huang/ExeDS
Multi-PL	MBXP [17]	12.4K	13	2022-10	https://huggingface.co/datasets/mxeval/mbxp
	Multi-HumanEval [17]	1.9K	12	2022-10	https://huggingface.co/datasets/mxeval/multi-humaneval
	HumanEval-X [320]	820	Python, C++, Java, JavaScript, Go	2023-03	https://huggingface.co/datasets/THUDM/humaneval-x
	MultiPL-E [40]	161	18	2022-08	https://huggingface.co/datasets/nuprl/MultiPL-E
	xCodeEval [128]	5.5M	11	2023-03	https://github.com/ntunlp/xCodeEval
Reasoning	MathQA-X [17]	5.6K	Python, Java, JavaScript	2022-10	https://huggingface.co/datasets/mxeval/mathqa-x
	MathQA-Python [18]	23,914	Python	2021-08	https://github.com/google-research/google-research
	GSM8K [59]	8.5K	Python	2021-10	https://huggingface.co/datasets/gsm8k
	GSM-HARD [79]	1.32K	Python	2022-11	https://huggingface.co/datasets/reasoning-machines/gsm-hard
	CRUXEval [82]	800	Python	2024-01	https://huggingface.co/datasets/cruxeval-org/cruxeval
Repository	SWE-bench [123]	2,294	Python	2023-10	https://huggingface.co/datasets/princeton-nlp/SWE-bench
	SketchEval [306]	20,355	Python	2024-03	https://github.com/nl2code/codes

The column labeled "**#PL**" indicates the number of programming languages included in each dataset. For the sake of brevity, we list the PLs for benchmarks that support fewer than or include five PLs. For benchmarks with six or more PLs, we provide only a numerical count of the PLs supported.

benchmarks have emerged, aiming to evaluate a broader spectrum of code generation capabilities in LLMs. We roughly divide these benchmarks into six distinct categories based on their application contexts, including general-purpose, competitive programming, data science, multilingual, logical reasoning, and repository-level. It is important to highlight that logical reasoning encompasses math-related benchmarks, as it aims to create "code-based solutions" for solving complex mathematical problems [51, 79, 325]. This strategy can therefore mitigate the limitations of LLMs in performing intricate mathematical computations. The statistics for these benchmarks are presented in Table 7.

General

— HumanEval [49]: HumanEval comprises 164 manually scripted Python programming problems, each featuring a function signature, docstring, body, and multiple unit tests.
— HumanEval+ [161]: HumanEval+ extends the original HumanEval [49] benchmark by increasing the scale of the test cases by 80 times. As the test cases increase, HumanEval+ can catch significant amounts of previously undetected incorrect code synthesized by LLMs.

- HumanEvalPack [185]: Expands HumanEval [49] by extending it to encompass three coding tasks across six PLs, namely code synthesis, code repair, and code explanation.
- MBPP [18]: MBPP is a collection of approximately 974 Python programming problems, crowd-sourced and designed for entry-level programmers. Each problem comes with an English task description, a code solution, and three automated test cases.
- MBPP+ [161]: MBPP+ enhances MBPP [18] by eliminating ill-formed problems and rectifying problems with incorrect implementations. The test scale of MBPP+ is also expanded by 35 times for test augmentation.
- CoNaLa [295]: CoNaLa contains almost 597K data samples for evaluating Python code generation. The curated part of CoNaLa is crawled from Stack Overflow, automatically filtered, and then curated by annotators. The mined part of CoNaLais automatically mined, with almost 600k examples.
- Spider [298]: Spider is large-scale complex text-to-SQL dataset covering 138 different domains. It has over 10K questions and 5.6K complex SQL queries on 200 databases. This dataset aims to test a model's ability to generalize to SQL queries, database schemas, and new domains.
- CONCODE [113]: CONCODE is a dataset with over 100K samples consisting of Java classes from public GitHub repositories. It provides near zero-shot conditions that can test the model's ability to generalize to unseen NL tokens with unseen environments.
- ODEX [271]: ODEX is an open-domain dataset focused on the execution-based generation of Python code from NL. It features 945 pairs of NL queries and their corresponding Python code, all extracted from Stack Overflow forums.
- CoderEval [297]: CoderEval is a pragmatic code generation benchmark that includes 230 Python and 230 Java code generation problems. It can be used to evaluate the model performance in generating pragmatic code beyond just generating standalone functions.
- ReCode [261]: ReCode serves as a comprehensive robustness evaluation benchmark. ReCode applies perturbations to docstrings, function and variable names, code syntax, and code format, thereby providing multifaceted assessments of a model's robustness performance.
- StudentEval [20]: StudentEval is a dataset of 1,749 prompts for 48 problems, authored by 80 students who have only completed a one-semester Python programming class. Unlike many other benchmarks, it has multiple prompts per problem and multiple attempts by the same participant, each problem is also accompanied by a set of instructor-written test cases.
- BigCodeBench [332]: BigCodeBench has 1,140 complex Python programming tasks, covering 723 function calls from 139 popular libraries across 7 domains. This benchmark is specifically designed to assess LLMs' ability to call multiple functions from cross-domain libraries and follow complex instructions to solve programming tasks, helping to bridge the evaluation gap between isolated coding exercises and the real-world programming scenario.
- ClassEval [72]: ClassEval is a manually crafted benchmark consisting of 100 classes and 412 methods for evaluating LLMs in the class-level code generation scenario. Particularly, the task samples of ClassEval present higher complexities, involving long code generation and sophisticated docstring information, thereby benefiting the evaluation of the LLMs' capabilities in generating complicated code.
- NaturalCodeBench [312]: NaturalCodeBench is a comprehensive code benchmark featuring 402 high-quality problems in Python and Java. These problems are selected from natural user queries from online coding services and span six distinct domains, shaping an evaluation environment aligned with real-world applications.

Competitions

— APPS [95]: The APPS benchmark is composed of 10K Python problems, spanning three levels of difficulty: introductory, interview, and competition. Each entry in the dataset includes a programming problem described in English, corresponding ground truth Python solutions, test cases defined by their inputs and outputs or function names if provided.
— CodeContests [150]: CodeContests is a competitive programming dataset consisting of samples from various sources including Aizu, AtCoder, CodeChef, CodeForces, and HackerEarth. The dataset encompasses programming problems accompanied by test cases in the form of paired inputs and outputs, along with both correct and incorrect human solutions in multiple PLs.
— LiveCodeBench [186]: LiveCodeBench is a comprehensive and contamination-free benchmark for evaluating a wide array of code-related capabilities of LLMs, including code generation, self-repair, code execution, and test output prediction. It continuously gathers new coding problems from contests across three reputable competition platforms: LeetCode, AtCoder, and CodeForces. The latest release of the dataset includes 713 problems that were released between May 2023 and September 2024.

Data Science

— DSP [42]: DSP allows for model evaluation based on real data science pedagogical notebooks. It includes well-structured problems, along with unit tests to verify the correctness of solutions and a Docker environment for reproducible execution.
— DS-1000 [136]: DS-1000 has 1K science questions from seven Python libraries, namely NumPy, Pandas, TensorFlow, PyTorch, SciPy, Scikit-learn, and Matplotlib. The DS-1000 benchmark features: (1) realistic problems with diverse contexts, (2) implementation of multi-criteria evaluation metrics, and (3) defense against memorization.
— ExeDS [107]: ExeDS is a data science code generation dataset specifically designed for execution evaluation. It contains 534 problems with execution outputs from Jupyter Notebooks, as well as 123K examples for training and validation.

Multiple PLs

— MBXP [17]: MBXP is a multilingual adaptation of the original MBPP [18] dataset. It is created using a framework that translates prompts and test cases from the original Python datasets into the corresponding data in the targeted PL.
— Multilingual HumanEval [17]: Multilingual HumanEval is a dataset derived from HumanEval [49]. It is designed to assess the performance of models in a multilingual context. It helps uncover the generalization ability of the given model on languages that are out-of-domain.
— HumanEval-X [320]: HumanEval-X is developed for evaluating the multilingual ability of code generation models with 820 hand-writing data samples in C++, Java, JavaScript, and Go.
— MultiPL-E [40]: MultiPL-E is a dataset for evaluating LLMs for code generation across 18 PLs. It adopts the HumanEval [49] and the MBPP [18] Python benchmarks and uses little compilers to translate them to other languages.
— xCodeEval [128]: xCodeEval is an executable multilingual multitask benchmark consisting of 25M examples covering 17 PLs. Its tasks include code understanding, generation, translation, and retrieval.

Reasoning

- MathQA-X [17]: MathQA-X is the multilingual version of MathQA [14]. It is generated by utilizing a conversion framework that converts samples from Python datasets into the target language.
- MathQA-Python [18]: MathQA-Python is a Python version of the MathQA benchmark[14]. The benchmark, containing more than 23K problems, is designed to assess the capability of models to synthesize code from complex textual descriptions.
- GSM8K [59]: GSM8K is a dataset of 8.5K linguistically diverse grade school math problems. The dataset is crafted to facilitate the task of question answering on basic mathematical problems that requires multi-step reasoning.
- GSM-HARD [79]: GSM-HARD is a more challenging version of the GSM8K [59] dataset. It replaces the numbers in the GSM8K questions with larger, less common numbers, thereby increasing the complexity and difficulty level of the problems.
- CRUXEval [82]: CRUXEval contains 800 Python functions, each paired with an input-output example. This benchmark supports two tasks: input prediction and output prediction, designed to evaluate the code reasoning, understanding, and execution capabilities of code LLMs.

Repository

- SWE-bench [123]: SWE-bench is a dataset that tests a model's ability to automatically solve GitHub issues. The dataset has 2,294 Issue-Pull Request pairs from 12 popular Python repositories.
- SketchEval [306]: SketchEval is a repository-oriented benchmark that encompasses data from 19 repositories, each varying in complexity. In addition to the dataset, SketchEval introduces a metric, known as SketchBLEU, to measure the similarity between two repositories based on their structures and semantics.

5.3 Data Synthesis

Numerous studies have demonstrated that high-quality datasets are integral to enhancing the performance of LLMs in various downstream tasks [34, 133, 177, 278, 284, 327]. For instance, the LIMA model, a 65B parameter LLaMa language model fine-tuned with a standard supervised loss on a mere 1,000 meticulously curated prompts and responses, achieved performance on par with, or even superior to, GPT-4 in 43% of evaluated cases. This figure rose to 58% when compared to Bard and 65% against `DaVinci003`, all without the use of reinforcement learning or human preference modeling [327]. The QuRating initiative strategically selects pre-training data embodying four key textual qualities—writing style, facts and trivia, required expertise, and educational value—that resonate with human intuition. Training a 1.3B parameter model on such data resulted in reduced perplexity and stronger in-context learning compared to baseline models [278].

Despite these advancements, acquiring quality data remains a significant challenge due to issues such as data scarcity, privacy concerns, and prohibitive costs [164, 266]. Human-generated data is often labor-intensive and expensive to produce, and it may lack the necessary scope and detail to navigate complex, rare, or ambiguous scenarios. As a resolution to these challenges, synthetic data has emerged as a viable alternative. By generating artificial datasets that replicate the intricacies of real-world information, models such as `GPT-3.5-turbo` [194] and `GPT-4` [6] have enabled the creation of rich datasets without the need for human annotation [92, 138, 164, 266]. This approach is particularly beneficial in enhancing the instruction-following capabilities of LLMs, with a focus on generating synthetic instruction-based data.

A notable example of this approach is the Self-Instruct [266] framework, which employs an off-the-shelf language model to generate a suite of instructions, inputs, and outputs. This data is then refined by removing invalid or redundant entries before being used to fine-tune the model. The empirical evidence supports the efficacy of this synthetic data generation methodology. Building upon this concept, the Alpaca [244] model, fine-tuned on 52k pieces of instruction-following data from a 7B parameter LLaMa [249] model, exhibits performance comparable to the `text-davinci-003` model. WizardLM [287] introduced the Evol-Instruct technique, which incrementally transforms simple instructions into more complex variants. The fine-tuned LLaMa model using this technique has shown promising results in comparison to established proprietary LLMs such as ChatGPT [194] and GPT-4 [6], to some extent. Moreover, Microsoft has contributed to this field with their Phi series of models, predominantly trained on synthetic high-quality data, which includes Phi-1 (1.3B) [84] for Python coding, Phi-1.5 (1.3B) [149] for common sense reasoning and language understanding, Phi-2 (2.7B) [180] for advanced reasoning and language understanding, and Phi-3 (3.8B) [5] for general purposes. These models have consistently outperformed larger counterparts across various benchmarks, demonstrating the efficacy of synthetic data in model training.

Drawing on the successes of data synthesis for general-purpose LLMs, researchers have expanded the application of synthetic data to the realm of code generation. The Code Alpaca model, as described in [44], has been fine-tuned on a 7B and 13B LLaMA model using a dataset of 20k instruction-following examples for code generation. This dataset was created by `text-davinci-003`[10] and employed the Self-Instruct technique [266]. Building on this, the WizardCoder 15B [171] utilizes the Evol-Instruct technique to create an enhanced dataset of 78k evolved code instruction examples. This dataset originates from the initial 20k instruction-following dataset used by Code Alpaca [44], which was also generated by `text-davinci-003`. The WizardCoder model, fine-tuned on the StarCoder [146] base model, achieved a 57.3% `pass@1` on the HumanEval benchmarks. This performance not only surpasses all other open source Code LLMs by a significant margin but also outperforms leading closed LLMs such as Anthropic's Claude and Google's Bard. In a similar vein, Magicoder [276] introduces a novel data synthesis approach termed OSS-INSTRUCT which enlightens LLMs with open source code snippets to generate high-quality instruction data for coding tasks. It aims to address the inherent biases often present in synthetic data produced by LLMs. Building upon Code Llama [224], the MagicoderS-CL-7B model—fine-tuned with 75k synthetic instruction data using the OSS-INSTRUCT technique and with `gpt-3.5-turbo-1106` as the data generator—has outperformed the prominent ChatGPT on the HumanEval Plus benchmark, achieving `pass@1` of 66.5% versus 65.9%. In a noteworthy development, Microsoft has introduced the phi-1 model [84], a more compact LLM of only 1.3B parameters. Despite its smaller size, phi-1 has been trained on high-quality textbook data sourced from the web (comprising 6 billion tokens) and supplemented with synthetic textbooks and exercises generated with GPT-3.5 (1 billion tokens). It has achieved `pass@1` of 50.6% on HumanEval and 55.5% on MBPP, setting a new state-of-the-art for Python coding performance among existing small language models. The latest contribution to this field is from the BigCode team, which has presented StarCoder2-15B-instruct [302], the first entirely self-aligned code LLM trained with a transparent and permissive pipeline. This model aligns closely with the OSS-INSTRUCT principles established by Magicoder, generating instructions based on seed functions filtered from the Stack v1 dataset [132] and producing responses through self-validation. Unlike Magicoder, StarCoder2-15B-instruct employs its base model, StarCoder2-15B, as the data generator, thus avoiding reliance on large and proprietary LLMs like GPT-3.5-turbo [194]. Figure 9 illustrates the comparison between Self-Instruct, Evol-Instruct, and OSS-Instruct data synthesis methods.

[10]https://platform.openai.com.

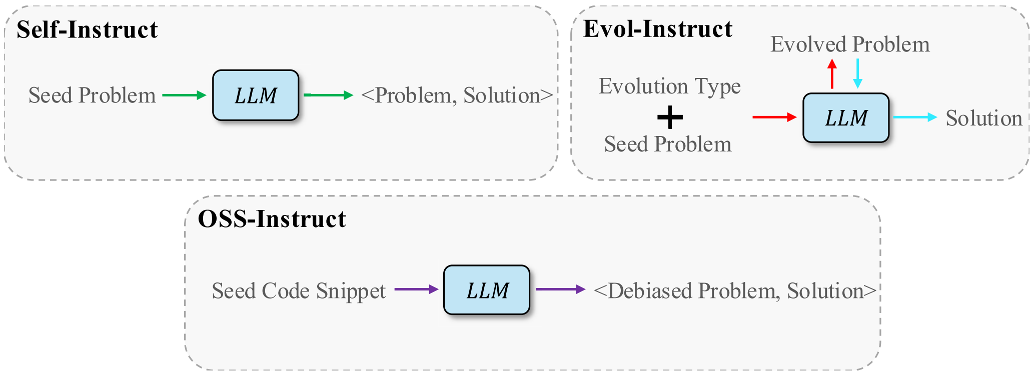


Fig. 9. The comparison among three representative data synthesis methods used for generating instruction data with LLMs. The Code Alpaca [44] employs the self-instruct method, whereas WizardCoder [171] and Magicoder [276] utilize the Evol-Instruct and OSS-Instruct methods, respectively.

While synthetic data has demonstrated its potential across both small- and large-scale LMs for a variety of general and specialized tasks, including code generation, it also poses several challenges that must be addressed. These challenges include a lack of data diversity [278], the need to ensure the factuality and fidelity of the information [253, 279], and the potential to amplify existing biases or introduce new ones [25, 89].

5.4 Pre-Training

5.4.1 Model Architectures. Since the inception of the Transformer for machine translation [254], it has become the *de facto* backbone for a multitude of LLMs that address a wide range of downstream tasks. The Transformer and its derivatives owe their prominence to their exceptional ability to parallelize computation and their powerful representational capacities [296, 318]. Through innovative scaling techniques, such as **Mixture-of-Experts (MoE)** [36, 230] and Depth-Up-Scaling [130], the capacity of Transformer-based LLMs has expanded to encompass hundreds of billions or even trillions of parameters. These scaled-up models have exhibited a range of emergent abilities [97, 127, 273], such as instruction following [198], in-context learning [70], and step-by-step reasoning [105, 274] that were previously unforeseen.

In the domain of code generation using LLMs, the architecture of contemporary models generally falls into one of two categories: encoder-decoder models, such as CodeT5 [269], CodeT5+ [267], and CodeRL [139]; or decoder-only models, such as Codex [49], StarCoder [146], Code Llama [224], and CodeGemma [60]. These architectures are depicted in Figure 2(b) and (c), respectively. For a comprehensive overview, Table 8 details the encoder-decoder architectures, while Table 9 focuses on the decoder-only models utilized in code generation.

5.4.2 Pre-Training Tasks. In the initial phase, language models for code generation are typically trained from scratch using datasets consisting of manually annotated pairs of NL descriptions and corresponding code snippets, within a supervised learning framework. However, manual annotation is not only laborious and time-consuming, but the efficacy of the resulting models is also constrained by both the volume and the quality of the available annotated data. This limitation is especially pronounced in the context of low-resource PLs, such as Swahili and Yoruba, where annotated examples are scarce [39, 47]. In light of these challenges, there has been a shift towards an alternative training strategy that involves pre-training models on extensive and unlabeled code corpora. This method is aimed at imbuing the models with a broad understanding of programming

Table 8. The Overview of LLMs with Encoder-Decoder Architectures for Code Generation

Model	Institution	Size	Vocabulary	Context Window	Date	Open Source
PyMT5[58]	Microsoft	374M	50K	1024 + 1024	2020-10	
PLBART[8]	UCLA	140M	50K	1024 + 1024	2021-03	✓
CodeT5 [269]	Salesforce	60M, 220M, 770M	32K	512 + 256	2021-09	✓
JuPyT5[42]	Microsoft	350M	50K	1024 + 1024	2022-01	
AlphaCode[150]	DeepMind	284M, 1.1B, 2.8B, 8.7B, 41.1B	8K	1536 + 768	2022-02	
CodeRL[139]	Salesforce	770M	32K	512 + 256	2022-06	✓
ERNIE-Code[41]	Baidu	560M	250K	1024 + 1024	2022-12	✓
PPOCoder[235]	Virginia Tech	770M	32K	512 + 256	2023-01	
CodeT5+[267]	Salesforce	220M, 770M, 2B, 6B, 16B	50K	2048 + 2048	2023-05	✓
CodeFusion[238]	Microsoft	75M	32k	128 + 128	2023-10	✓
AST-T5[81]	UC Berkeley	226M	32k	512 + 200/300	2024-01	✓

knowledge, encompassing elements like identifiers, code structure, and underlying semantics [49]. In this regard, two pre-training tasks have gained prominence for their effectiveness, namely **Causal Language Modeling (CLM)**, also known as unidirectional language modeling or next-token prediction, and **Denoising Autoencoding (DAE)**. The CLM task can be applied to both decoder-only and encoder-decoder model architectures, while DAE tasks are specifically designed for encoder-decoder frameworks. It should also be noted that there is a variety of additional auxiliary pre-training tasks that can further enhance model performance. These include Masked Identifier Prediction, Identifier Tagging, Bimodal Dual Generation [269], Text-Code Matching, and Text-Code Contrastive Learning [267]. These tasks contribute to a more nuanced and comprehensive pre-training process, equipping the models with the capabilities necessary to handle a wide range of code generation scenarios.

CLM. In decoder-only LLMs, given a sequence of tokens $\mathbf{x} = \{x_1, \ldots, x_n\}$, the CLM task refers to autoregressively predict the target tokens x_i based on the preceding tokens $x_{<i}$ in a sequence. The CLM objective for training decoder LLMs is to minimize the following likelihood:

$$\mathcal{L}_{CLM}^{Decoder-only}(\mathbf{x}) = -\log\left(\prod_{i=1}^{n} P_\theta(x_i \mid \mathbf{x}_{<i})\right) = \sum_{i=1}^{n} -\log P_\theta(x_i \mid \mathbf{x}_{<i}), \tag{16}$$

where $\mathbf{x}_{<i}$ represents the sequence of preceding tokens $\{x_1, \ldots, x_{i-1}\}$ before $\mathbf{x}_i$ in the input, θ denotes the model parameters. The conditional probability $P_\theta(x_i|\mathbf{x}_{<i})$ is modeled by adding a causal attention mask to the MHSA matrix of each Transformer block. To be specific, causal attention masking is implemented by setting the lower triangular part of the matrix to 0 and the remaining elements to $-\infty$, ensuring that each token x_i attends only to its predecessors and itself. On the contrary, in encoder-decoder LLMs, a pivot token x_k is randomly selected in a sequence of tokens and then regarding the context before it as the source sequence $\mathbf{x}_{in} = \{x_1, \ldots, x_k\}$ of the encoder and the sequence after it as the target output $\mathbf{x}_{out} = \{x_{k+1}, \ldots, x_n\}$ of decoder. Formally, the CLM objective for training encoder-decoder LLMs is to minimize loss function as follows:

$$\mathcal{L}_{CLM}^{Encoder-Decoder}(\mathbf{x}) = -\log\left(\prod_{i=k+1}^{n} P_\theta(x_i \mid \mathbf{x}_{\leq k}, \mathbf{x}_{<i})\right) = \sum_{i=k+1}^{n} -\log P_\theta(x_i \mid \mathbf{x}_{\leq k}, \mathbf{x}_{<i}), \tag{17}$$

where $\mathbf{x}_{\leq k}$ is the source sequence input and $\mathbf{x}_{<i}$ denotes the target sequence autoregressively generated so far. During the inference phase, pre-trained LLMs that have been trained on large-scale code corpus can generate code in a zero-shot manner without the need for fine-tuning. This

Table 9. The Overview of LLMs with Decoder-Only Architectures for Code Generation

Model	Institution	Size	Vocabulary	Context Window	Date	Open Source
GPT-C [241]	Microsoft	366M	60K	1,024	2020-05	
CodeGPT [170]	Microsoft	124M	50K	1,024	2021-02	✓
GPT-Neo[31]	EleutherAI	125M, 1.3B, 2.7B	50k	2,048	2021-03	✓
GPT-J [255]	EleutherAI	6B	50k	2,048	2021-05	✓
Codex [49]	OpenAI	12M, 25M, 42M, 85M, 300M, 679M, 2.5B, 12B	-	4,096	2021-07	
CodeParrot [251]	Hugging Face	110M, 1.5B	33k	1,024	2021-11	✓
PolyCoder [288]	CMU	160M, 400M, 2.7B	50k	2,048	2022-02	✓
CodeGen [191]	Salesforce	350M, 2.7B, 6.1B, 16.1B	51k	2,048	2022-03	✓
GPT-NeoX [30]	EleutherAI	20B	50k	2,048	2022-04	✓
PaLM-Coder [55]	Google	8B, 62B, 540B	256k	2,048	2022-04	
InCoder [77]	Meta	1.3B, 6.7B	50k	2,049	2022-04	✓
PanGu-Coder [56]	Huawei	317M, 2.6B	42k	1,024	2022-07	
PyCodeGPT [304]	Microsoft	110M	32k	1,024	2022-06	✓
CodeGeeX [320]	Tsinghua	13B	52k	2,048	2022-09	✓
BLOOM [140]	BigScience	176B	251k	-	2022-11	✓
ChatGPT [194]	OpenAI	-	-	16k	2022-11	✓
SantaCoder [10]	Hugging Face	1.1B	49k	2,048	2022-12	✓
LLaMA [249]	Meta	6.7B, 13.0B, 32.5B, 65.2B	32K	2,048	2023-02	✓
GPT-4 [6]	OpenAI	-	-	32K	2023-03	
CodeGen2 [190]	Salesforce	1B, 3.7B, 7B, 16B	51k	2,048	2023-05	✓
replit-code [220]	replit	3B	33k	2,048	2023-05	✓
StarCoder [146]	Hugging Face	15.5B	49k	8,192	2023-05	✓
WizardCoder [171]	Microsoft	15B, 34B	49k	8,192	2023-06	✓
phi-1 [84]	Microsoft	1.3B	51k	2,048	2023-06	✓
CodeGeeX2 [320]	Tsinghua	6B	65k	8,192	2023-07	✓
PanGu-Coder2 [231]	Huawei	15B	42k	1,024	2023-07	
Llama 2 [250]	Meta	7B, 13B, 70B	32K	4,096	2023-07	✓
OctoCoder [185]	Hugging Face	15.5B	49k	8,192	2023-08	✓
Code Llama [224]	Meta	7B, 13B, 34B	32k	16,384	2023-08	✓
CodeFuse [159]	Ant Group	350M, 13B, 34B	101k	4,096	2023-09	✓
phi-1.5 [149]	Microsoft	1.3B	51k	2,048	2023-09	✓
CodeShell [283]	Peking University	7B	70k	8,192	2023-10	✓
Magicoder [276]	UIUC	7B	32k	16,384	2023-12	✓
AlphaCode 2 [12]	Google DeepMind	-	-	-	2023-12	
StableCode [207]	StabilityAI	3B	50k	16,384	2024-01	✓
WaveCoder [299]	Microsoft	6.7B	32k	16,384	2023-12	✓
phi-2 [180]	Microsoft	2.7B	51k	2,048	2023-12	✓
DeepSeek-Coder [88]	DeepSeek	1.3B, 6.7B, 33B	32k	16,384	2023-11	✓
StarCoder 2 [168]	Hugging Face	15B	49k	16,384	2024-02	✓
Claude 3 [15]	Anthropic	-	-	200K	2024-03	
CodeGemma [60]	Google	2B, 7B	25.6k	8,192	2024-04	✓
Code-Qwen [246]	Qwen Group	7B	92K	65,536	2024-04	✓
Llama3 [178]	Meta	8B, 70B	128K	8,192	2024-04	✓
StarCoder2-Instruct [302]	Hugging Face	15.5B	49K	16,384	2024-04	✓
Codestral [179]	Mistral AI	22B	33k	32k	2024-05	✓

is achieved through the technique of prompt engineering, which guides the model to produce the desired output[11] [34, 211]. Additionally, recent studies have explored the use of few-shot learning, also referred to as in-context learning, to enhance model performance further [144, 204].

DAE. In addition to CLM, the DAE task has been extensively applied in pre-training encoder-decoder architectures for code generation, such as PLBART [8], CodeT5 [269], and its enhanced

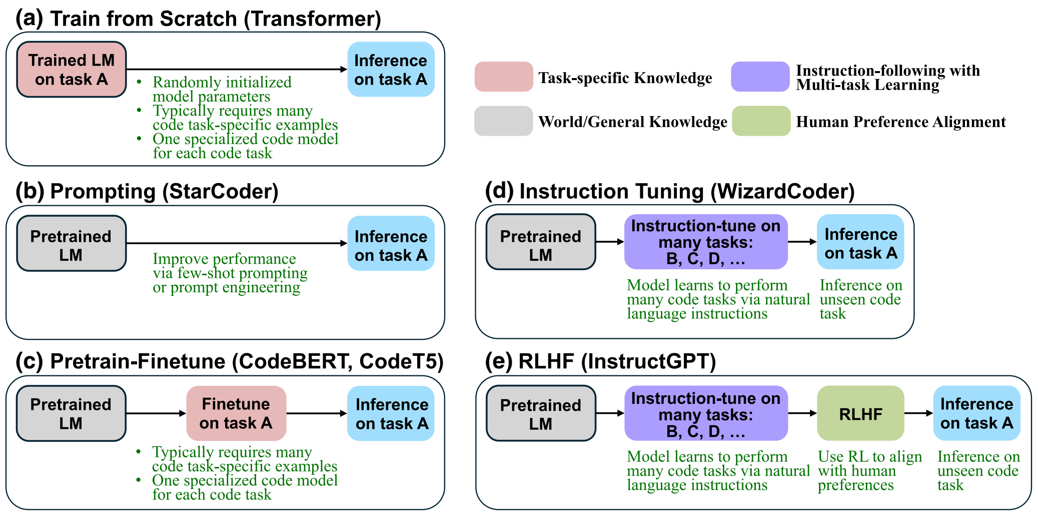

(a) Train from Scratch (Transformer)
Trained LM on task A
Inference on task A
Randomly initialized model parameters
Typically requires many code task-specific examples
One specialized code model for each code task
Task-specific Knowledge
Instruction-following with Multi-task Learning
World/General Knowledge
Human Preference Alignment
(b) Prompting (StarCoder)
Pretrained LM
Inference on task A
Improve performance via few-shot prompting or prompt engineering
(d) Instruction Tuning (WizardCoder)
Pretrained LM
Instruction-tune on many tasks: B, C, D, ...
Inference on task A
Model learns to perform many code tasks via natural language instructions
Inference on unseen code task
(c) Pretrain-Finetune (CodeBERT, CodeT5)
Pretrained LM
Finetune on task A
Inference on task A
Typically requires many code task-specific examples
One specialized code model for each code task
(e) RLHF (InstructGPT)
Pretrained LM
Instruction-tune on many tasks: B, C, D, ...
RLHF
Inference on task A
Model learns to perform many code tasks via natural language instructions
Use RL to align with human preferences
Inference on unseen code task

Fig. 10. Comparison of instruction tuning with various fine-tuning strategies and prompting for code tasks, adapted from [272]. For (a), which involves training a Transformer from scratch, please refer to [7] for its use in source code summarization task. In the case of (e), we utilize a representative RLHF [198] as an example. Additional reinforcement learning methods, such as DPO [213], are also applicable at this stage.

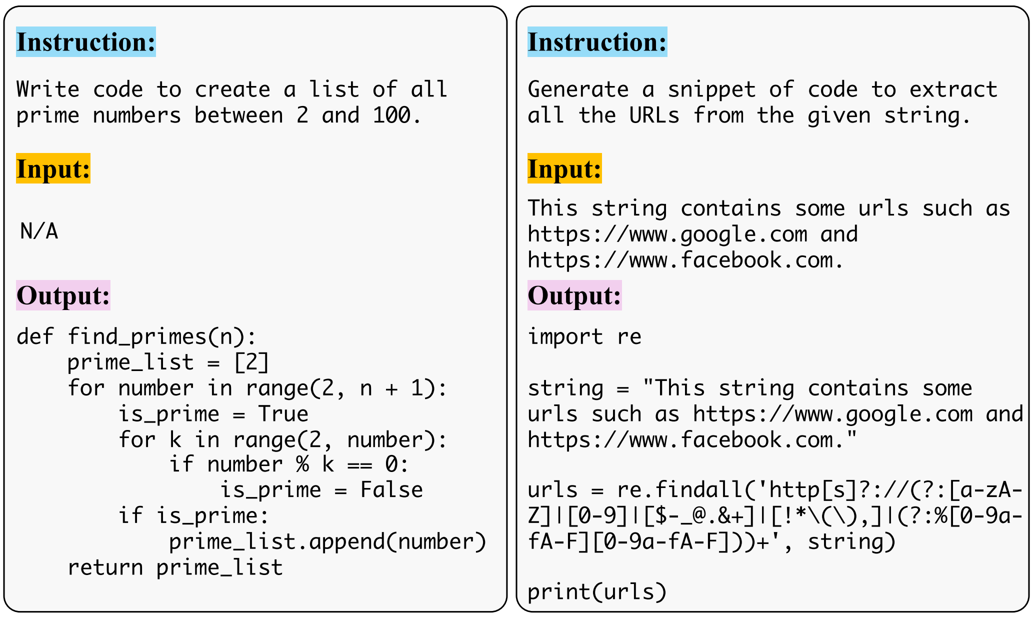
successor, CodeT5+ [267]. Following T5 [214] and CodeT5 [269], the DAE refers to initially perturbing the source sequence by introducing randomly masked spans of varying lengths. This corrupted sequence serves as the input for the encoder. Subsequently, the decoder employs an autoregressive strategy to reconstruct the masked spans, integrating sentinel tokens to facilitate the generation process. This method has proven effective in improving the model's ability to generate semantically and syntactically accurate code by learning robust contextual representations [267, 269]. Formally, the DAE objective for training encoder-decoder LLMs is to minimize the following likelihood:

$$\mathcal{L}_{DAE}^{Encoder-Decoder}(\mathbf{x}) = \sum_{i=1}^{k} -\log P_\theta(\mathbf{x}_i^{masked\_spans} \mid \mathbf{x}^{\backslash masked\_spans}, \mathbf{x}_{<i}^{masked\_spans}), \tag{18}$$

where θ denotes the model parameters, $\mathbf{x}^{\backslash masked\_spans}$ is the noisy input with masked spans, $\mathbf{x}^{masked\_spans}$ is the masked spans to predict from the decoder with k denoting the number of tokens in $\mathbf{x}^{masked\_spans}$, and $\mathbf{x}_{<i}^{masked\_spans}$ is the span sequence autoregressively generated so far. Compared with CLM, the DAE task presents a more challenging scenario, as it necessitates a deeper understanding and capture of the intrinsic semantic relationships among token sequences by LLMs [214].

5.5 Instruction Tuning

After pre-training LLMs on large-scale datasets, the next phase typically involves augmenting the model's ability to process and follow various instructions, known as instruction tuning. Instruction tuning generally refers to the supervised fine-tuning of pre-trained LLMs using datasets comprised of structured examples framed as various NL instructions [114, 198, 272, 311]. The comparison of instruction tuning with various fine-tuning strategies and prompting for code tasks is depicted in Figure 10. It is important to note that the fine-tuning described in Figure 10(c) pertains to traditional fine-tuning, which is concerned with learning the functional mapping between input and output for a specific task, *without the input being formatted as an instruction.* In contrast, instruction tuning,



Instruction:

```
Write code to create a list of all
prime numbers between 2 and 100.
```

Input:

```
N/A
```

Output:

```
def find_primes(n):
    prime_list = [2]
    for number in range(2, n + 1):
        is_prime = True
        for k in range(2, number):
            if number % k == 0:
                is_prime = False
        if is_prime:
            prime_list.append(number)
    return prime_list
```

Instruction:

```
Generate a snippet of code to extract
all the URLs from the given string.
```

Input:

```
This string contains some urls such as
https://www.google.com and
https://www.facebook.com.
```

Output:

```
import re

string = "This string contains some
urls such as https://www.google.com and
https://www.facebook.com."

urls = re.findall('http[s]?://(?:[a-zA-
Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-
fA-F][0-9a-fA-F]))+', string)

print(urls)
```

Fig. 11. Two examples of instruction data sampled from Code Alpaca [44] used to instruction-tune pre-trained code LLM to enhance their alignment with NL instructions. The instruction corpus encompasses a variety of tasks, each accompanied by distinct instructions, such as prime numbers generation and URLs extraction.

which was first proposed in [272], represents an alternative fine-tuning approach that utilizes instruction-based datasets. In essence, the key differentiation between traditional fine-tuning and instruction tuning resides in the manner in which the datasets are constructed. Two examples of instruction data sampled from Code Alpaca [44] are demonstrated in Figure 11. It capitalizes on the heterogeneity of instruction types, positioning instruction tuning as a form of multi-task prompted training that significantly enhances the model's generalization to unseen tasks [57, 198, 226, 272].

In the realm of code generation, NL descriptions serve as the instructions guiding the model to generate corresponding code snippets. Consequently, a line of research on instruction tuning LLMs for code generation has garnered substantial interest across academia and industry. To perform instruction tuning, instruction data are typically compiled from source code with permissive licenses [110, 132, 168] (refer to Section 5.2.2) or are constructed from synthetic code data [171, 276, 302] (refer to Section 5.3). These datasets are then utilized to fine-tune LLMs through a supervised learning paradigm. However, the substantial computational resources required for FFT LLM pose a notable challenge, particularly in scenarios with constrained resources [68, 152]. To mitigate this issue, PEFT has emerged as a compelling alternative strategy, gaining increasing attention for its potential to reduce resource consumption [66, 68, 103, 257, 303, 310]. In the following subsection, we categorize existing works based on their instruction-tuning strategies to provide a comprehensive and systematic review.

5.5.1 FFT. FFT involves updating all parameters within a pre-trained model, as shown in Figure 12(a). This approach is often preferred when ample computational resources and substantial training data are available, as it typically leads to better performance. For instance, [269] introduces an encoder-decoder PLM for code generation, named CodeT5+. They instruction-tune this model on a dataset comprising 20k instruction samples from Code Alpaca [44], resulting in an

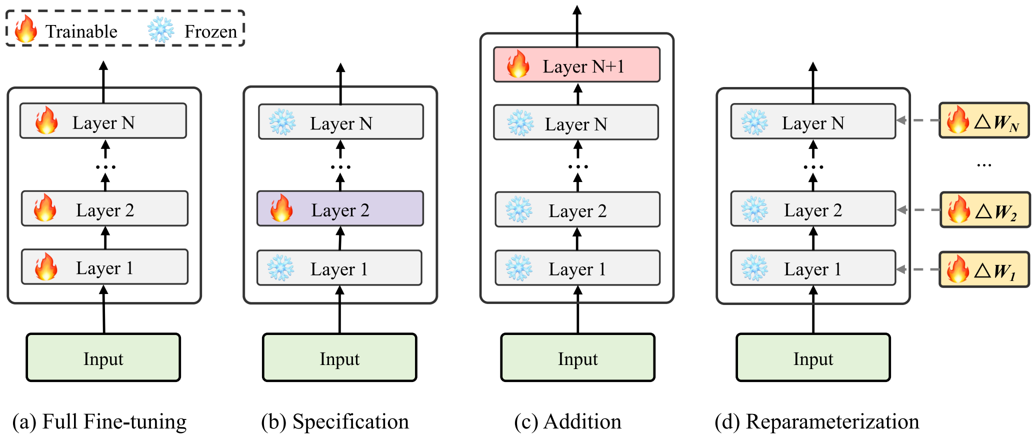


Fig. 12. An illustration of FFT and PEFT methods. Part (a) refers to the Full Fine-tuning method, which updates all parameters of the base model during fine-tuning. Part (b) stands for the Specification-based PEFT method that conditionally fine-tunes a small subset of the model parameters while freezing the rest of the model, e.g., BitFit [303]. Part (c) represents the Addition-based PEFT method that fine-tunes the incremental parameters introduced into the base model or input, e.g., Adapter [102], Prefix-tuning [148], and Prompt-tuning [142]. Part (d) symbolizes the Reparameterization-based method which reparameterizes existing model parameters by low-rank transformation, e.g., LoRA [103], QLoRA [66], and AdaLoRA [310].

instruction-following model called InstructCodeT5+, which exhibited improved capabilities in code generation. Luo et al. [171] leverage the Evol-Instruct data synthesis technique from WizardLM [287] to evolve 20K code Alpaca [44] instruction samples into a 78K code instruction dataset. This enriched dataset is then used to fine-tune the StarCoder base model, resulting in WizardCoder, which showcases notable advancements in code generation. In a similar vein, inspired by the successes of WizardCoder [171] and RRHF [301], Pangu-Coder 2 [231] applies the Evol-Instruct method to generate 68k high-quality instruction samples from the initial 20k Code Alpaca [44] instruction samples. Additionally, they introduce a novel reinforcement learning via **Rank Responses to Align Test and Teacher Feedback (RRTF)**, which further enhances the performance of Pangu-Coder 2 in code generation. Diverging from synthetic instruction data generation methods, OctoPack [185] utilizes real-world data by curating CommitPack from the natural structure of Git commits, which inherently pair code changes with human-written instructions. This dataset, consisting of 4 terabytes of Git commits across 350 PLs, is employed to fine-tune StarCoder [146] and CodeGeeX2 [320], leading to the instruction-following code models of OctoCoder and OctoGeeX for code generation, respectively. The most recent innovation comes from Magicoder [276], who proposes OSS-INSTRUCT, a novel data synthesis method that leverages open source code snippets to generate high-quality instruction data for code generation. This approach seeks to reduce the bias often present in synthetic data generated by LLM. In line with OSS-INSTRUCT, the BigCode team introduces StarCoder2-15B-instruct [302], which they claim to be the first entirely self-aligned LLM for code generation, trained with a fully permissive and transparent pipeline. Moreover, [60] harnesses open source mathematics datasets, such as MATH [95] and GSM8k [59], along with synthetically generated code following the OSS-INSTRUCT [276] paradigm, to instruction-tune CodeGemma 7B, yielding exceptional results in mathematical reasoning and code generation tasks.

5.5.2 PEFT. To mitigate the extensive computational and resource demands inherent in fine-tuning LLMs, the concept of PEFT has emerged to focus on updating a minimal subset of parameters, which may either be a selection of the model's parameters or an array of additional parameters specifically introduced for the tuning process [68, 152]. The categorization of these methods is depicted in Figure 12(b)–(d). A plethora of innovative PEFT approaches have been developed, among which BitFit [303], Adapter [102], Prompt tuning [142], Prefix-tuning [148], LoRA [103], IA^3 [160], QLoRA [66], and AdaLoRA [310] are particularly noteworthy. A seminal study in this field, LoRA [103], proposes a parameter update mechanism for a pre-trained weight matrix—such as those found in the key or value projection matrices of a Transformer block's MHSA layer—by factorizing the update into two low-rank matrices. Crucially, all original model parameters remain frozen, with only the pair of low-rank matrices being trainable. After fine-tuning, the product of these low-rank matrices can be seamlessly incorporated into the existing weight matrix through an element-wise addition. This process can be formally described as:

$$(\mathbf{W}_0 + \Delta\mathbf{W})x = \mathbf{W}_0 x + \Delta\mathbf{W}x = \mathbf{W}_0^{frozen} x + \underbrace{\frac{\alpha}{r}\mathbf{B}_{up}^{trainable}\mathbf{A}_{down}^{trainable}}_{\Delta\mathbf{W}} x, \tag{19}$$

where $\mathbf{W}_0 \in \mathbb{R}^{d\times k}$ denotes a pre-trained weight matrix, $\mathbf{B}_{up}^{trainable} \in \mathbb{R}^{d\times r}$ and $\mathbf{A}_{down}^{trainable} \in \mathbb{R}^{r\times k}$ are two trainable low-rank matrixes and initialized by a zero matrix and a random Gaussian distribution $\mathcal{N}(0, \sigma^2)$ respectively, to ensure $\Delta\mathbf{W} = 0$ at the beginning of training. The rank $r \ll \min(d, k)$, the $\frac{\alpha}{r}$ is a scaling coefficient to balance the importance of the LoRA module, like a learning rate.

Despite the advancements in PEFT methods, their application in code generation remains limited. For instance, [121] pioneered the use of parameter-efficient instruction-tuning on a Llama 2 [250] model with a single RTX 3090 GPU, leading to the development of a multilingual code generation model called CodeUp. More recently, ASTRAIOS [333] conducted a thorough empirical examination of parameter-efficient instruction tuning for code comprehension and generation tasks. This study yielded several perceptive observations and conclusions, contributing valuable insights to the domain. Specifically, it was found that PEFT methods exhibit differential effectiveness contingent upon the scale of the model. Among these methods, LoRA frequently achieves an optimal balance between cost and performance. Although larger models may experience reduced robustness and security, tuning strategies that prove effective in smaller models can be effectively extended to larger models. Additionally, the validation loss during instruction tuning serves as a reliable predictor of downstream performance.

5.6 Reinforcement Learning with Feedback

LLMs have exhibited remarkable instruction-following capabilities through instruction tuning. However, they often produce outputs that are unexpected, toxic, biased, or hallucinated outputs that do not align with users' intentions or preferences [118, 198, 270]. Consequently, aligning LLMs with human preference has emerged as a pivotal area of research. A notable work is InstructGPT [198], which further fine-tunes an instruction-tuned model utilizing RLHF on a dataset where labelers have ranked model outputs in order of quality, from best to worst. This method has been instrumental in the development of advanced conversational language models, such as ChatGPT [194] and Bard [175]. Despite its success, acquiring high-quality human preference ranking data is a resource-intensive process [141]. To address this, Reinforcement Learning from AI Feedback [22, 141] has been proposed to leverage powerful off-the-shelf LLMs (e.g., ChatGPT [194] and GPT-4 [6]) to simulate human annotators by generating preference data.

Building on RLHF's success, researchers have explored reinforcement learning with feedback to enhance code generation in LLMs. Unlike RLHF, which relies on human feedback, this approach employs compilers or interpreters to automatically provide feedback on code samples through code execution on unit test cases, catalyzing the advancement of this research domain. CodeRL [139] introduced an actor-critic reinforcement learning framework for code generation. In this setup, the language model serves as the actor-network, while a token-level functional correctness reward predictor acts as the critic. Generated code is assessed through unit test signals from a compiler, which can indicate compiler errors, runtime errors, unit test failures, or passes. CompCoder [264] enhances code compilability by employing compiler feedback, including language model fine-tuning, compilability reinforcement, and compilability discrimination strategies. Subsequently, PPOCoder [235] integrates pre-trained code model CodeT5 [269] with **Proximal Policy Optimization (PPO)** [227]. This integration not only utilizes execution (i.e., compilers or interpreters) feedback to assess syntactic and functional correctness but also incorporates a reward function that evaluates the syntactic and semantic congruence between **Abstract Syntax Tree (AST)** sub-trees and **Dataflow Graph (DFG)** edges in the generated code against the ground truth. Additionally, the framework applies a KL-divergence penalty to maintain fidelity between the actively learned policy and the referenced pre-trained model, enhancing the optimization process. More recently, RLTF [162] has proposed an online reinforcement learning framework that provides fine-grained feedback based on compiler error information and location, along with adaptive feedback that considers the ratio of passed test cases.

Despite these successes, reinforcement learning algorithms face inherent limitations such as inefficiency, instability, extensive resource requirements, and complex hyperparameter tuning, which can impede the performance and scalability of LLMs. To overcome these challenges, recent studies have introduced various variants of RL methods that do not rely on PPO, including DPO [213], RRHF [301], and sDPO [129]. In essence, these methods aim to maximize the likelihood between the logarithm of conditional probabilities of preferred and rejected responses, which may be produced by LLMs with varying capabilities. Inspired by RRHF [301], PanGu-Coder 2 [231] leverages a novel framework, Reinforcement Learning via RRTF, significantly enhancing code generation capabilities, as evidenced by `pass@1` of 62.20% on the HumanEval benchmark.

To facilitate a deeper understanding and differentiation between two prominent reinforcement learning algorithms, namely PPO, an on-policy method, and DPO, an off-policy method, for LLMs, we present the formulation of the policy objectives as outlined in [198, 213]:

$$\mathcal{L}_{PPO}(r_\phi, \pi_\theta; \pi_{ref}) = -\mathbb{E}_{x\sim\mathcal{D}_{ppo}, y\sim\pi_\theta(y|x)}\left[r_\phi(x,y) - \beta \log \frac{\pi_\theta(y \mid x)}{\pi_{ref}(y \mid x)}\right], \tag{20}$$

$$\mathcal{L}_{DPO}(\pi_\theta; \pi_{ref}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}_{dpo}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta\log\frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\right)\right], \tag{21}$$

where $r_\phi(x, y)$ represents the scalar output of the reward model for a given prompt x and response y, parameterized by ϕ. The dataset $\mathcal{D}_{ppo}$ comprises prompts used within the PPO training. The parameter β regulates the deviation from the base reference policy π_{ref}, which is initially set to the SFT model π^{SFT}. In practice, the language model policy π_θ is also initialized to π^{SFT}. The inclusion of this constraint is crucial, as it ensures that the model does not deviate excessively from the distribution on which the reward model is reliable. Additionally, it aids in maintaining diversity in responses and prevents mode collapse towards single high-reward responses [213]. The y_w and y_l represent the preferred and dispreferred responses within a pair (y_w, y_l), respectively, $\sigma(\cdot)$ is the sigmoid function, and $\mathcal{D}_{dpo}$ denotes the dataset of human comparisons.

Taking a step forward, the integration of more non-differentiable code features, such as coding style [45, 176] and readability [35], into the reinforcement learning feedback for LLM-based code generation, presents an exciting avenue for future research.

5.7 Prompt Engineering

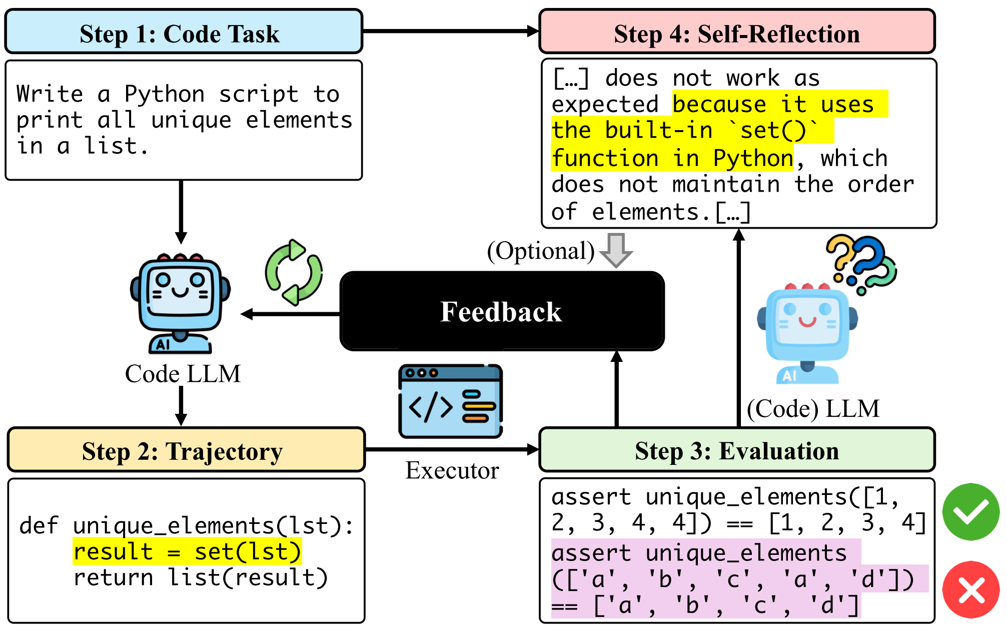
Large-scale language models (LLMs) such as GPT-3 and its successors have been trained on large-scale data corpora, endowing them with substantial world knowledge [34, 198, 272]. Despite this, crafting an effective prompting as a means of communicating with LLMs to harness their full potential remains a long-standing challenge [163]. Recent advancements in prompting engineering have expanded the capabilities of LLMs, enabling more sophisticated task completion and enhancing both reliability and performance. Notable techniques include **Chain-of-Thought (CoT)** [274], Self-Consistency [265], Tree-of-Thought [293], Program of Thoughts [51], Reasoning via Planning [93], ReAct [294], Self-Refine [174], Reflexion [233], and LATS [326]. For instance, CoT significantly improves the LLMs' ability to perform complex reasoning by providing a few CoT demonstrations as exemplars in prompting.

Prompting engineering is particularly advantageous as it bypasses the need for additional training and can significantly elevate performance. Consequently, numerous studies have leveraged this technique for iterative and self-improving (refining) code generation within proprietary LLMs such as ChatGPT and GPT-4. Figure 13 illustrates the general pipeline for self-improving code generation with LLMs. For instance, Self-Debugging [52] involves prompting an LLM to iteratively refine a predicted program by utilizing feedback composed of code explanations combined with execution results, which assists in identifying and rectifying errors. When unit tests are unavailable, this feedback can rely solely on code explanations. Similarly, LDB [324] prompts LLMs to refine generated code by incorporating debugging feedback, which consists of the evaluation of the correctness of variable values throughout runtime execution, as assessed by the LLMs. In parallel, SelfEvolve [122] employs a two-stage process where LLMs first generate domain-specific knowledge for a problem, followed by a trial code. This code is then iteratively refined through interactive prompting and execution feedback. An empirical investigation by [193] provides a comprehensive analysis of the self-repairing capabilities for code generation in models like Code Llama, GPT-3.5, and GPT-4, using problem sets from HumanEval and APPS. This study yields a series of insightful observations and findings, shedding light on the self-refinement effectiveness of these LLMs. Moreover, Reflexion [233] introduces a general approach for code generation wherein LLM-powered agents engage in verbal self-reflection on task feedback signals, storing these reflections in an episodic memory buffer to inform and improve decision-making in subsequent interactions. LATS [326] adopts a novel strategy, utilizing LLMs as agents, value functions, and optimizers. It enhances decision-making by meticulously constructing trajectories through Monte Carlo Tree Search algorithms, integrating external feedback, and learning from experience. This approach has demonstrated remarkable results in code generation, achieving a `pass@1` of 94.4% on the HumanEval benchmark with GPT-4.

Distinct from the aforementioned methods, CodeT [46] and LEVER [188] prompt LLMs to generate numerous code samples, which are then re-ranked based on execution outcomes to select the optimal solution. Notably, these approaches do not incorporate a self-refinement step to further improve code generation.

5.8 Repository Level and Long Context

In contemporary SE practices, modifications to a code repository are widespread and encompass a range of activities, including package migration, temporary code edits, and the resolution of GitHub issues. While LLMs showcase impressive prowess in function-level code generation, they often falter when grappling with the broader context inherent to a repository, such as import dependencies, parent classes, and files bearing similar names. These deficiencies result in suboptimal performance


Step 1: Code Task
Write a Python script to print all unique elements in a list.
Step 4: Self-Reflection
[…] does not work as expected because it uses the built-in `set()` function in Python, which does not maintain the order of elements.[…]
(Optional)
Feedback
Code LLM
(Code) LLM
Executor
Step 2: Trajectory
Step 3: Evaluation

```
def unique_elements(lst):
    result = set(lst)
    return list(result)
```

```
assert unique_elements([1, 2, 3, 4, 4]) == [1, 2, 3, 4]
assert unique_elements(['a', 'b', 'c', 'a', 'd']) == ['a', 'b', 'c', 'd']
```

Fig. 13. An illustration of the self-improving code generation pipeline using prompts for LLMs. This process incorporates iterative self-refinement by integrating execution outcomes and includes an optional self-reflection mechanism to enhance generation quality.

in repository-level code generation, as identified in recent studies [236, 237]. The challenges faced by LLMs in this domain are primarily due to the following factors:

- Code repositories typically contain intricate interdependencies scattered across various files, including shared utilities, configurations, and cross-API invocations, which arise from modular design principles [23, 307].
- Repositories are characterized by their unique structures, naming conventions, and coding styles, which are essential for maintaining clarity and facilitating ongoing maintenance [45].
- The vast context of an entire repository often exceeds the context length limitations of LLMs, thus hindering their ability to integrate comprehensive contextual information [23].
- LLMs may not have been adequately trained on extensive sets of repository data, such as proprietary software or projects that are still in development [236].

Given that the scope of a typical software repository encompasses hundreds of thousands of tokens, it is imperative to enhance the capacity of LLMs to handle extensive contexts when they are employed for repository-level code generation. Fortunately, recent advancements in positional encoding techniques, such as ALiBi [208] and RoPE [240], have shown promise in improving the Transformer's ability to generalize from shorter training sequences to longer inference sequences [316]. This progress addresses the third challenge mentioned above to a certain degree, thereby enabling better contextualization of coding activities within full repositories.

Advancing the state-of-the-art, [306] tackles the challenges of NL2Repo, an endeavor that seeks to create a complete code repository from NL requirements. To address this complex task, they introduce the CodeS framework, which strategically breaks down NL2Repo into a series of manageable sub-tasks using a multi-layer sketch approach. The CodeS framework comprises three distinct modules: (1) RepoSketcher, for creating a directory structure of the repository based on given requirements; (2) FileSketcher, for sketching out each file within that structure; and (3) SketchFiller, for fleshing out the specifics of each function within the file sketches [306].

Accordingly, a surge of benchmarks tailored for repository-level code generation has emerged, such as SWE-bench [123] and SketchEval [306]. The detailed statistics and comparisons of these benchmarks are presented in Table 7.

Despite the progress made by these methods in repository-level code generation, significant challenges remain to be addressed. Programming developers are often required to invest considerable time in editing and debugging [26, 29, 184, 236, 252]. However, the emergence of LLM-powered coding agents, such as AutoCodeRover [314], SWE-Agent [124], and OpenDevin [197], has demonstrated their potential to tackle complex problems, paving the way for future exploration in this field (for more details, see Section 5.10).

5.9 RACG

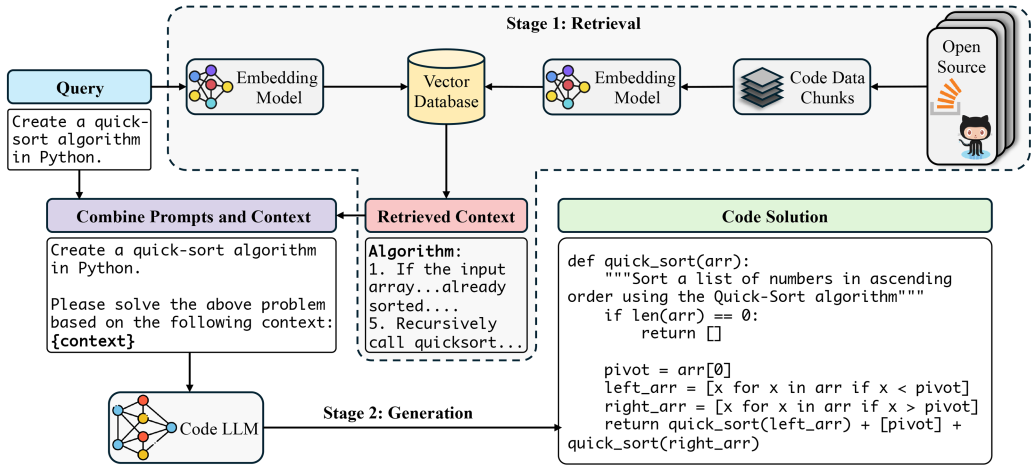
LLMs have exhibited impressive capabilities but are hindered by several critical issues such as hallucination [153, 313], obsolescence of knowledge [115], and non-transparent [33], untraceable reasoning processes [80, 106, 274, 328]. While techniques like instruction-tuning (see Section 5.5) and reinforcement learning with feedback (see Section 5.6) mitigate these issues, they also introduce new challenges, such as catastrophic forgetting and the requirement for substantial computational resources during training [90, 199].

Recently, RAG has emerged as an innovative approach to overcome these limitations by integrating knowledge from external databases [48, 80, 143, 317]. Formally defined, RAG denotes a model that, in response to queries, initially sources relevant information from an extensive corpus of documents, and then leverages this retrieved information in conjunction with the original query to enhance the response's quality and accuracy, especially for knowledge-intensive tasks [143]. The RAG framework typically consists of a vector database, a retriever, a re-ranker, and a generator. It is commonly implemented using tools such as LangChain[11] and LLamaIndex.[12] By performing continuous knowledge updates of the database and the incorporation of domain-specific data, RAG circumvents the need for re-training LLMs from scratch [80]. Consequently, RAG has substantially advanced LLM performance across a variety of tasks [48, 143].

Due to the nature of code, code LLMs are also susceptible to the aforementioned issues that affect general-purpose LLMs. For instance, they may exhibit a hallucination phenomenon when instructions fall outside the scope of their training data or necessitate the latest programming packages. Given the dynamic nature of publicly available source-code libraries like PyTorch, which undergo frequent expansion and updates, calling deprecated methods can become a significant challenge. If Code LLMs are not updated in tandem with the latest functions and APIs, this can introduce potential errors and safety risks. RACG stands as a promising solution to these concerns. A workflow illustration of the RACG is depicted in Figure 14.

Despite its potential, the adoption of RAG for code generation remains limited. Drawing inspiration from the common practice among programmers of referencing related code snippets, [165] introduced a novel retrieval-augmented mechanism with **Graph Neural Networks (GNNs)**, termed HGNN, which unites the advantages of similar examples retrieval with the generalization capabilities of generative models for code summarization, which is the reverse process of code generation. Parvez et al. [203] pioneered a retrieval-augmented framework named REDCODER for code generation by retrieving and integrating relevant code snippets from a source-code database, thereby providing supplementary context for the generation process. In the spirit of how

[11]LangChain facilitates the development of LLM-powered applications. See https://www.langchain.com.
[12]LLamaIndex is a leading data framework for building LLM applications. See https://www.llamaindex.ai.



```
Create a quick-
sort algorithm
in Python.
```

```
Create a quick-sort algorithm
in Python.

Please solve the above problem
based on the following context:
{context}
```

```
Algorithm:
1. If the input
array...already
sorted....
5. Recursively
call quicksort...
```

```
def quick_sort(arr):
    """Sort a list of numbers in ascending
order using the Quick-Sort algorithm"""
    if len(arr) == 0:
        return []

    pivot = arr[0]
    left_arr = [x for x in arr if x < pivot]
    right_arr = [x for x in arr if x > pivot]
    return quick_sort(left_arr) + [pivot] +
quick_sort(right_arr)
```

Fig. 14. A workflow illustration of the RACG. Upon receiving a query (instruction), the retriever selects the relevant contexts from a large-scale vector database. Subsequently, the retrieved contexts are merged with the query, and this combined input is fed into the generator (LLM) to produce the target code solution.

programmers often consult textual resources such as code manuals and documentation to comprehend functionalities, DocPrompting [329] explicitly utilizes code documentation by retrieving the relevant documentation pieces based on a NL query and then generating the target code by blending the query with the retrieved information.

Furthermore, breaking away from reliance on a singular source of retrieval, [239] developed a multi-faceted "knowledge soup" that integrates web searches, documentation, execution feedback, and evolved code snippets. Then, it incorporates an active retrieval strategy that iteratively refines the query and enriches the knowledge soup, expanding the scope of information available for code generation.

Despite these advancements, several limitations in RACG warrant further exploration: (1) the quality of the retrieved information significantly impacts overall performance; (2) the effective integration of retrieved code information with the query needs optimization; (3) an over-reliance on retrieved information may lead to inadequate responses that fail to address the query's intent; (4) additional retrieved information necessitates larger context windows for the LLM, resulting in increased computational demands.

5.10 Autonomous Coding Agents

The advent of LLMs has marked the beginning of a new era of potential pathways towards **Artificial General Intelligence (AGI)**, capturing significant attention in both academia and industry [108, 259, 277, 282]. A rapidly expanding array of applications for LLM-based autonomous agents, including AutoGPT [2], AgentGPT [1], BabyAGI [3], and AutoGen [281], underlines the promise of this technology.

LLM-powered autonomous agents are systems endowed with sophisticated reasoning abilities, leveraging an LLM as a central computational engine or controller. This allows them to formulate and execute problem-solving plans through a series of tool-enabled functions or API calls. Moreover, these agents are designed to function within a shared environment where they can communicate and engage in cooperative, competitive, or negotiating interactions [104, 259, 281]. The typical

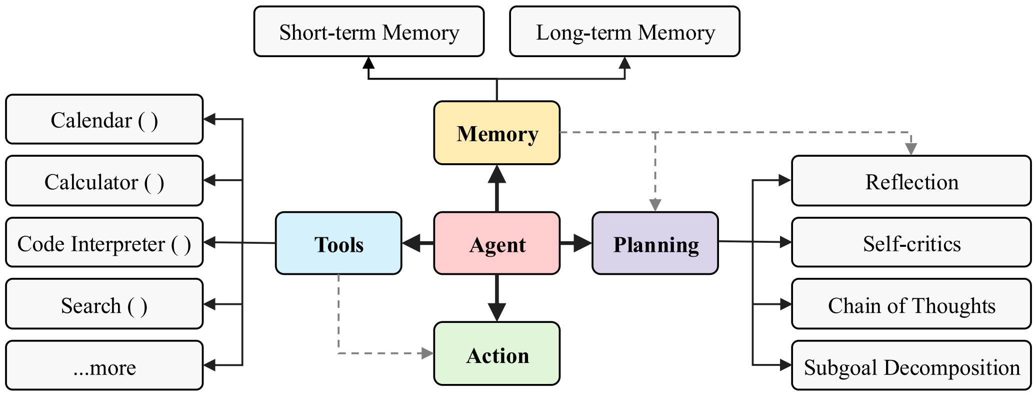


Fig. 15. The general architecture of an LLM-powered autonomous agent system, adapted from [277]. *Planning*: The agent decomposes large tasks into smaller, manageable sub-goals or engages in self-criticism and self-reflection on past actions to learn from mistakes and improve future performance. *Memory*: This component enables the agent to store and retrieve past information. *Tools*: The agent is trained to invoke external functions or APIs. *Action*: The agent executes actions, with or without the use of tools, to interact with the environment. The gray dashed lines represent the dataflow within the system.

architecture of such an agent encompasses an LLM-based agent, a memory module, a planning component, and a tool utilization module, as depicted in Figure 15.

In the realm of automated code generation, LLM-powered autonomous agents have demonstrated remarkable proficiency. For instance, AgentCoder [104] achieved a groundbreaking pass@1 of 96.3% on the HumanEval benchmark, forwarding a step closer to the future of automated software development [111]. The innovative meta-programming framework termed MetaGPT [100] integrates human workflow efficiencies into LLM-based multi-agent collaboration, as shown in Figure 16. Furthermore, [104] introduces AgentCoder, a multi-agent framework composed of three specialized agents, each with distinct roles and capabilities. These roles include a programmer agent responsible for code generation, a test designer agent tasked with generating unit test cases, and a test executor agent that executes the code and provides feedback. This division of labor within AgentCoder promotes more efficient and effective code generation. CodeAct [263] distinguishes itself by utilizing executable Python code to consolidate LLM agent actions within a unified action space, in contrast to the generation of JSON or textual formats. Additionally, AutoCodeRover [314] is proposed to autonomously resolve GitHub issues for program enhancement.

To address the complexity of tasks within SE, two innovative autonomous AI software engineers, Devin[13] [62] and OpenDevin[14] [197], have been released and rapidly garnered considerable interest within the SE and AGI community. Subsequently, an autonomous system, SWE-agent [124], leverages a language model to interact with a computer to address SE tasks, successfully resolving 12.5% of issues on the SWE-bench benchmark [123]. L2MAC [98] has been introduced as the first practical, LLM-based, multi-agent, general-purpose stored-program automatic computer that utilizes a von Neumann architecture, designed specifically for the generation of long and consistent outputs. At the time of writing this survey in May 2024, OpenDevin has enhanced CodeAct with bash command-based tools, leading to the release of OpenDevin CodeAct 1.0 [285], which sets a new state-of-the-art performance on the SWE-bench Lite benchmark [123].

[13]https://www.cognition.ai/introducing-devin.
[14]https://github.com/OpenDevin/OpenDevin.

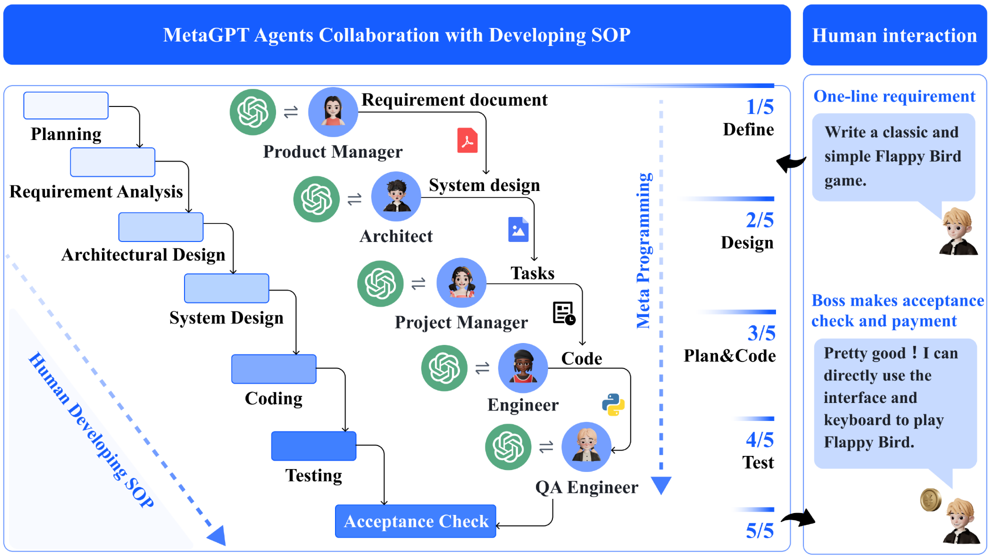


Fig. 16. MetaGPT integrates human workflow efficiencies into LLM-based multi-agent collaboration to break down complex code-related tasks into specific, actionable procedures. These procedures are then assigned to various roles, such as Product Manager, Architect, and Engineer played by LLM. The image is sourced from the original paper [100].

Despite these remarkable advancements, the journey towards fully realized AI software engineers employing LLM-powered autonomous agents is far from complete [259, 282]. Critical aspects such as prompt design, context length, agent count, and toolsets call for further refinement and optimization, especially as problem complexities escalate [111].

5.11 Evaluation

Despite the notable capabilities of LLMs, they exhibit a range of behaviors that are both beneficial and potentially risky. These behaviors can enhance performance across various downstream tasks but may also introduce reliability and trustworthiness concerns in LLM deployment [43, 49, 288]. Consequently, it is imperative to develop precise evaluation approaches to discern the qualitative and quantitative differences between models, thereby encouraging further advancements in LLM capabilities.

Evaluation strategies for LLMs in code generation mirror those for general-purpose LLMs and can be divided into three principal categories: metrics-based, human-centered, and LLM-based approaches. Detailed benchmarks for these evaluation strategies are presented in Section 5.2.3 and summarized in Table 7. Subsequent subsections will provide a thorough analysis of each approach.

5.11.1 Metrics. The pursuit of effective and reliable automatic evaluation metrics for generated content is a long-standing challenge within the field of NLP [50, 155, 201]. At the early stage, most works directly leverage token-matching-based metrics, such as Exact Match, BLEU [201], ROUGE [155], and METEOR [24], which are prevalent in text generation of NLP, to assess the quality of code generation.

While these metrics offer a rapid and cost-effective approach for assessing the quality of generated code, they often fall short of capturing the syntactical and functional correctness, as well as the semantic features of the code. To eliminate this limitation, CodeBLEU [218] was introduced, enhancing the traditional BLEU metric [201] by incorporating syntactic information through AST and semantic understanding via DFG. Despite these improvements, the metric does not fully resolve issues pertaining to execution errors or discrepancies in the execution results of the generated code. In light of these challenges, execution-based metrics have gained prominence for evaluating code generation, including pass@k [49], n@k [150], test case average [95], execution accuracy [215], and pass@t [193]. In particular, the pass@k, serving as a principal evaluation metric, assesses the probability that at least one out of k code samples generated by a model will pass all unit tests. An unbiased estimator for pass@k introduced by [49] is defined as:

$$\texttt{pass@k} := \mathbb{E}_{\text{task}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right], \tag{22}$$

where n is the total number of sampled candidate code solutions, k is the number of randomly selected code solutions from these candidates for each programming problem, with $n \geq k$, and c is the count of correct samples within the n candidates.

Nevertheless, these execution-based methods are heavily dependent on the quality of unit tests and are limited to evaluating executable code [305]. Consequently, when unit tests are unavailable, token-matching-based metrics are often employed as an alternative for evaluation. Furthermore, in scenarios lacking a ground truth label, unsupervised metrics such as perplexity (PPL) [116] can serve as evaluative tools. Perplexity quantifies an LLM's uncertainty in predicting new content, thus providing an indirect measure of the model's generalization capabilities and the quality of the generated code.

Taken together, while the aforementioned methods primarily focus on the functional correctness of code, they do not provide a holistic evaluation that encompasses other critical dimensions such as code vulnerability [187], maintainability [16], readability [35], complexity and efficiency [206], stylistic consistency [176], and execution stability [212]. A comprehensive evaluation framework that integrates these aspects remains an open area for future research and development in the field of code generation assessment.

5.11.2 Human Evaluation. Given the intrinsic characteristics of code, such as syntax, semantics, efficiency, and security, the aforementioned automatic evaluation metrics are inherently limited in their capacity to fully assess code quality. For instance, metrics specifically designed to measure code style consistency are challenging to develop and often fail to capture this aspect adequately [45]. When it comes to repository-level code generation, the evaluation of overall code quality is substantially complicated due to the larger scale of the task, which involves cross-file designs and intricate internal as well as external dependencies, as discussed by [23, 236].

To overcome these challenges, conducting human evaluations becomes necessary, as it yields relatively robust and reliable results. Human assessments also offer greater adaptability across various tasks, enabling the simplification of complex and multi-step evaluations. Moreover, human evaluations are essential for demonstrating the effectiveness of certain token-matching-based metrics, such as CodeBLEU [218]. These studies typically conduct experiments to evaluate the correlation coefficient between proposed metrics and quality scores assigned by actual users, demonstrating their superiority over existing metrics.

Moreover, in an effort to better align LLMs with human preferences and intentions, InstructGPT [198] employs human-written prompts and demonstrations, and model output ranking in the fine-tuning of LLMs using RLHF. Although similar alignment learning techniques have been applied to code generation, the feedback in this domain typically comes from a compiler or interpreter, which offers execution feedback, rather than from human evaluators. Notable examples include CodeRL

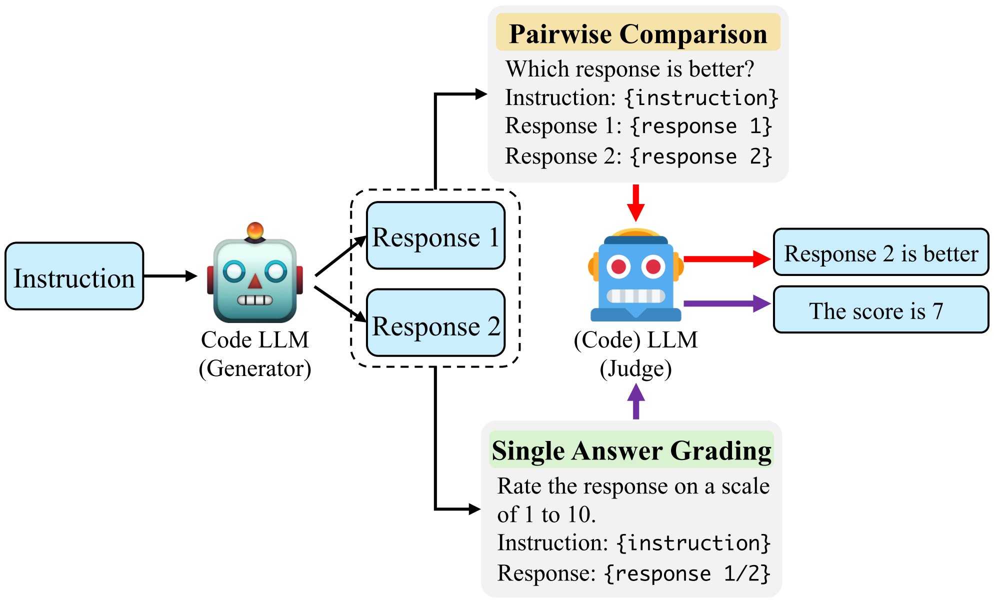


Fig. 17. The pipeline of (Code) LLM-as-a-judge for evaluating generated code by Code LLMs. There are primarily two types of approaches: pairwise comparison and single answer grading.

[139], PPOCoder [235], RLTF [162], and PanGu-Coder2 [231]. Further information on this topic is available in Section 5.6.

Nonetheless, human evaluations are not without drawbacks, as they can be prone to certain issues that may compromise their accuracy and consistency. For instance, (1) personalized tastes and varying levels of expertise among human evaluators can introduce biases and inconsistencies into the evaluation process; (2) conducting comprehensive and reliable human evaluations often necessitates a substantial number of evaluators, leading to significant expenses and time consumption; (3) the reproducibility of human evaluations is often limited, which presents challenges in extending previous evaluation outcomes or monitoring the progress of LLMs, as highlighted by [318].

5.11.3 LLM-as-a-Judge. The instruction-following capabilities of LLMs have stimulated researchers to investigate the potential of LLM-based evaluations. The LLM-as-a-Judge [319] refers to the application of advanced proprietary LLMs (e.g., GPT4, Gemini, and Claud 3) as proxies for human evaluators. This involves designing prompts with specific requirements to guide LLMs in conducting evaluations, as demonstrated by AlpacaEval [147] and MT-bench [319]. This method reduces reliance on human participation, thereby facilitating more efficient and scalable evaluations. Moreover, LLMs can offer insightful explanations for the assigned rating scores, thereby augmenting the interpretability of evaluations [318].

Nevertheless, the use of LLM-based evaluation for code generation remains relatively underexplored compared with general-purpose LLM. The pipeline of (Code) LLM-as-a-judge for evaluating generated code by Code LLMs is depicted in Figure 17. A recent work [331] introduces the ICE-Score evaluation metric, which instructs LLM for code assessments. This approach attains superior correlations with functional correctness and human preferences, thereby eliminating the requirement for test oracles or references. As the capabilities of LLM continue to improve, we anticipate seeing more research in this direction.

Despite their scalability and explainability, the effectiveness of LLM-based evaluation is constrained by the inherent limitations of the chosen LLM. Several studies have shown that most LLMs, including GPT-4, suffer from several issues, including position, verbosity, and self-enhancement biases, as well as restricted reasoning ability [319]. Specifically, position bias refers to the tendency of LLMs to disproportionately favor responses that are presented in certain positions, which can skew the perceived quality of answers based on their order of presentation. Meanwhile, verbosity bias describes the inclination of LLMs to prefer lengthier responses, even when these are not necessarily of higher quality compared to more concise ones [319]. Self-enhancement bias, on the other hand, is observed when LLMs consistently overvalue the quality of the text they generate [318, 319]. Moreover, due to their inherent limitations in tackling complex reasoning challenges, LLMs may not be entirely reliable as evaluators for tasks that require intensive reasoning, such as those involving mathematical problem-solving. However, these shortcomings can be partially addressed through the application of deliberate prompt engineering and fine-tuning techniques, as suggested by [319].

5.11.4 Empirical Comparison. In this section, we present a performance comparison of LLMs for code generation using the well-regarded HumanEval, MBPP, and the more practical and challenging BigCodeBench benchmarks. This empirical comparison aims to highlight the progressive enhancements in LLM capabilities for code generation. These benchmarks assess an LLM's ability to generate source code across various levels of difficulty and types of programming tasks. Specifically, HumanEval focuses on complex code generation, MBPP targets basic programming tasks, and BigCodeBench emphasizes practical and challenging programming tasks.

Due to the limitations in computational resources we faced, we have cited experimental results from original papers and widely recognized open source leaderboards within the research community, such as the HumanEval Leaderboard,[15] EvalPlus Leaderboard,[16] Big Code Models Leaderboard,[17] and BigCodeBench Leaderboard,[18] to ensure a fair comparison. We report performance on HumanEval using the `pass@1` metric, as shown in Table 10, while MBPP and BigCodeBench results are presented with `pass@1` in Figures 18 and 19, respectively.

We offer the following insights:

- The performance gap between open source and closed-source models across the three benchmarks is gradually narrowing. For instance, on the HumanEval benchmark, DeepSeek-Coder-V2-Instruct with 21B activation parameters and Qwen2.5-Coder-Instruct 7B achieve 90% and 88.4% `pass@1`, respectively. These results are comparable to the much larger closed-source LLMs, such as Claude-3.5-Sonnet, which achieves 92.0% `pass@1`. On the MBPP benchmark, Qwen2.5-Coder-Instruct 7B with 83.5% `pass@1` significantly outperforms GPT-3.5-Turbo with 52.2% `pass@1` and closely rivals the closed-source Claude-3-Opus with 86.4% `pass@1`. On the BigCodeBench, DeepSeek-Coder-V2-Instruct achieves 59.7%, surpassing all compared closed-source and open source LLMs except for slightly falling behind GPT-4o-0513, which achieves 61.1%.
- Generally, as the number of model parameters increases, the performance of code LLMs improves. However, Qwen2.5-Coder-Instruct 7B achieves 88.4% `pass@1`, outperforming larger models like StarCoder2-Instruct 15.5B with 72.6% `pass@1`, DeepSeek-Coder-Instruct 33B with 79.3% `pass@1`, and Code Llama-Instruct 70B with 67.8% `pass@1` on the HumanEval benchmark.

[15] https://paperswithcode.com/sota/code-generation-on-humaneval.
[16] https://evalplus.github.io/leaderboard.html.
[17] https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard.
[18] https://bigcode-bench.github.io/.

Table 10. The Performance Comparison of LLMs for Code Generation on the HumanEval [49] Benchmark, Measured by pass@1

	Model	Size	pass@1 (%)	Availability
Closed Source	GPT-4o-0513 [195]	-	91.0	[API Access]
	GPT-4-Turbo-0409 [196]	-	88.2	[API Access]
	GPT-4-1106 [6]	-	87.8	[API Access]
	GPT-3.5-Turbo-0125 [194]	-	76.2	[API Access]
	Claude-3.5-Sonnet [15]	-	**92.0**	[API Access]
	Claude-3-Opus [15]	-	84.9	[API Access]
	Claude-3-Sonnet [15]	-	73.0	[API Access]
	Claude-3-Haiku [15]	-	75.9	[API Access]
	Gemini-1.5-Pro [217]	-	84.1	[API Access]
	Gemini-1.5-Flash [217]	-	74.3	[API Access]
	Gemini-1.0-Ultra [217]	-	74.4	[API Access]
	Gemini-1.0-Pro [217]	-	67.7	[API Access]
	PanGu-Coder2[a] [231]	15B	61.64	-
	PanGu-Coder [56]	2.6B	23.78	-
	Codex [49]	12B	28.81	Deprecated
	PaLM-Coder [55]	540B	36	-
	AlphaCode [150]	1.1B	17.1	-
Open Source	Codestral[a] [179]	22B	81.1	[Checkpoint Download]
	DeepSeek-Coder-V2-Instruct[a] [330]	21B (236B)	**90.2**	[Checkpoint Download]
	Qwen2.5-Coder-Instruct[a] [109]	7B	**88.4**	[Checkpoint Download]
	Qwen2.5-Coder [109]	7B	61.6	[Checkpoint Download]
	StarCoder2-Instruct[a] [302]	15.5B	72.6	[Checkpoint Download]
	CodeGemma-Instruct[a] [60]	7B	56.1	[Checkpoint Download]
	CodeGemma [60]	7B	44.5	[Checkpoint Download]
	StarCoder 2 [168]	15B	46.3	[Checkpoint Download]
	WaveCoder-Ultra[a] [299]	6.7B	79.9	[Checkpoint Download]
	WaveCoder-Pro[a] [299]	6.7B	74.4	[Checkpoint Download]
	WaveCoder-DS[a] [299]	6.7B	65.8	[Checkpoint Download]
	StableCode [207]	3B	29.3	[Checkpoint Download]
	CodeShell [283]	7B	34.32	[Checkpoint Download]
	CodeQwen1.5-Chat[a] [246]	7B	83.5	[Checkpoint Download]
	CodeQwen1.5 [246]	7B	51.8	[Checkpoint Download]
	DeepSeek-Coder-Instruct[a] [88]	33B	79.3	[Checkpoint Download]
	DeepSeek-Coder [88]	33B	56.1	[Checkpoint Download]
	replit-code [220]	3B	20.12	[Checkpoint Download]
	Magicoder*S*-CL[a] [276]	7B	70.7	[Checkpoint Download]
	Magicoder-CL[a] [276]	7B	60.4	[Checkpoint Download]
	WizardCoder[a] [171]	33B	79.9	[Checkpoint Download]
	CodeFuse [159]	34B	74.4	[Checkpoint Download]
	Phi-1 [84]	1.3B	50.6	[Checkpoint Download]
	Code Llama-Instruct[a] [224]	70B	67.8	[Checkpoint Download]
	Code Llama [224]	70B	53.0	[Checkpoint Download]
	OctoCoder[a] [185]	15.5B	46.2	[Checkpoint Download]
	CodeGeeX2 [320]	6B	35.9	[Checkpoint Download]
	InstructCodeT5+[a] [267]	16B	35.0	[Checkpoint Download]
	CodeGen-NL [191]	16.1B	14.24	[Checkpoint Download]
	CodeGen-Multi [191]	16.1B	18.32	[Checkpoint Download]
	CodeGen-Mono [191]	16.1B	29.28	[Checkpoint Download]
	StarCoder [146]	15B	33.60	[Checkpoint Download]
	CodeT5+ [269]	16B	30.9	[Checkpoint Download]
	CodeGen2 [190]	16B	20.46	[Checkpoint Download]
	SantaCoder [10]	1.1B	14.0	[Checkpoint Download]
	InCoder [77]	6.7B	15.2	[Checkpoint Download]
	PolyCoder [288]	2.7B	5.59	[Checkpoint Download]
	CodeParrot [251]	1.5B	3.99	[Checkpoint Download]

For models with various sizes, we report only the largest size version of each model with a magnitude of B parameters. The best results from one closed-source model and two open-source models are highlighted in yellow and bold.
[a]Denotes instruction-tuned models.

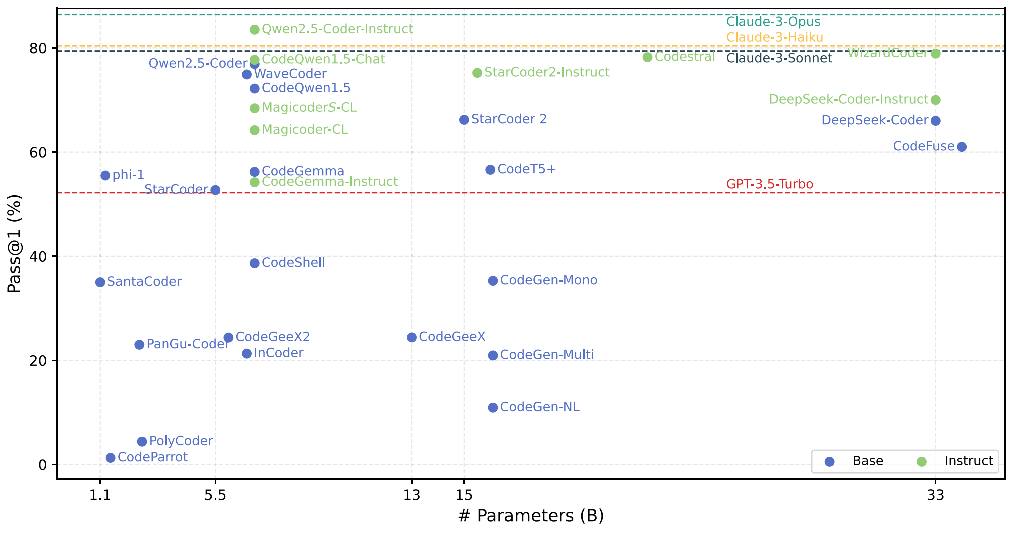


Fig. 18. The performance comparison of LLMs for code generation on the MBPP [18] benchmark, measured by pass@1. For models with various sizes, we report only the largest size version of each model with a magnitude of billion (B) parameters.

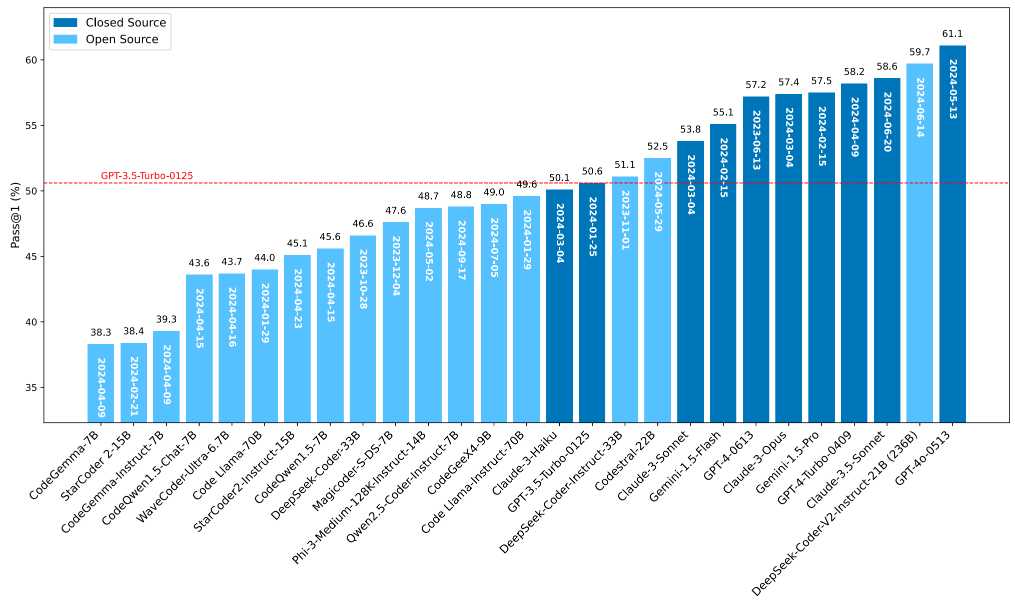


Fig. 19. The performance comparison of LLMs for code generation on the BigCodeBench [332] benchmark, measured by pass@1. For models with various sizes, we report only the largest size version of each model with a magnitude of billion (B) parameters.

Similar trends are observed across the other two benchmarks, suggesting that code LLMs with 7B parameters may be sufficiently capable for code generation task.

— Instruction-tuned models consistently outperform their base (pre-trained) counterparts across the HumanEval and MBPP benchmarks. For instance, Qwen2.5-Coder-Instruct surpasses

Table 11. The Performance Improvement of Instruction-Tuned Models over Their Pre-Trained Counterparts on the HumanEval (H.), MBPP (M.), and BigCodeBench (B.) Benchmarks

	Qwen2.5-Coder-Instruct 7B	**StarCoder2-Instruct 15.5B**	**CodeGemma-Instruct 7B**	**DeepSeek-Coder-Instruct 33B**	**Code Llama-Instruct 70B**
HumanEval	43.51%	56.80%	26.07%	41.35%	27.92%
MBPP	8.58%	13.60%	−3.56%	6.06%	−0.32%
BigCodeBench	-	17.45%	2.61%	9.66%	12.73%
# Avg. Imp. H. M.	26.04%	35.20%	11.26%	23.71%	13.80%
# Avg. Imp. H. M. B.	-	29.28%	8.37%	19.02%	13.44%

The last two rows demonstrate the average improvement on the first two benchmarks (H. M.) and the three benchmarks (H. M. B.), respectively. The decline in performance is indicated in grey.

Qwen2.5-Coder by an average of 26.04%, StarCoder2-Instruct improves upon StarCoder 2 by an average of 35.20%, and CodeGemma-Instruct enhances CodeGemma by an average of 11.26%. Additionally, DeepSeek-Coder-Instruct outperforms DeepSeek-Coder by an average of 23.71%, while Code Llama-Instruct shows a 13.80% improvement over Code Llama. Detailed results can be found in Table 11. These findings underscore the effectiveness of instruction tuning, although the quality of the instruction tuning dataset plays a critical role in determining model performance [171, 327].

— Performance on the HumanEval benchmark is nearly saturated. However, MBPP, which involves basic programming tasks, and BigCodeBench, which involves more practical and challenging programming tasks, demand more capable code LLMs. Additionally, while these benchmarks primarily evaluate the functional correctness of code, they do not provide a comprehensive assessment across other critical dimensions. Developing a more holistic evaluation framework that integrates various aspects remains an open area for future research and development in LLMs for code generation evaluation.

Discussion. We discuss certain code LLMs in Table 10 for clarity: (1) General LLMs accessed via API are not specifically trained on large code corpora but achieve state-of-the-art performance in code generation, such as Claude-3.5-Sonnet with 92.0% pass@1 on HumanEval benchmark. (2) AlphaCode targets code generation for more complex and unseen problems that require a deep understanding of algorithms and intricate NL, such as those encountered in competitive programming. The authors of AlphaCode found that large-scale model sampling to navigate the search space, such as 1M samples per problem for CodeContests, followed by filtering based on program behavior to produce a smaller set of submissions, is crucial for achieving good and reliable performance on problems that necessitate advanced reasoning. (3) Phi-1 1.3B is a specialized LLM for code, trained on "textbook quality" data from the web (6B tokens) and synthetically generated textbooks and exercises using GPT-3.5 (1B tokens) [84]. (4) Code Llama 70B is initialized with Llama 2 model weights and continually pre-trained on 1T tokens from a code-heavy dataset and long-context fine-tuned with approximately 20B tokens. However, Code Llama-Instruct 70B is fine-tuned from Code Llama-Python 70B without long-context fine-tuning, using an additional 260M tokens to better follow human instructions. Surprisingly, these models underperform compared to smaller parameter Code LLMs like Qwen2.5-Coder-Instruct 7B, DeepSeek-Coder-V2-Instruct 21B, and Codestral 22B across all three benchmarks. The underlying reasons for this discrepancy remain unclear and warrant further exploration. (5) Unlike other open source Code LLMs, DeepSeek-Coder-V2-Instruct is further pre-trained on DeepSeek-V2 [158], which employs a MoE architecture with only 21B activation parameters out of 236B parameters, using an additional 6 trillion tokens composed of 60% source code, 10% math corpus, and 30% NL corpus. A comprehensive overview of MoE in LLMs is given in [37].

5.12 Code LLMs Alignment

The pre-training of LLMs for next-token prediction, aimed at maximizing conditional generation likelihood across vast textual corpora, equips these models with extensive world knowledge and emergent capabilities [34]. This training approach enables the generation of coherent and fluent text in response to diverse instructions. Nonetheless, LLMs can sometimes misinterpret human instructions, produce biased content, or generate factually incorrect information (commonly referred to as hallucinations), which may limit their practical utility [118, 270, 318].

Aligning LLMs with human intentions and values, known as LLM alignment, has consequently become a critical research focus [118, 270]. Key objectives frequently discussed in the context of LLM alignment include robustness, interpretability, controllability, ethicality, trustworthiness, security, privacy, fairness, and safety. In recent years, significant efforts have been made by researchers to achieve this alignment, employing techniques such as RLHF [198].

The alignment of Code LLMs has not been extensively explored. Compared to text generation, aligning code generation with human intentions and values is even more crucial. For instance, users without programming expertise might prompt Code LLM to generate source code and subsequently execute it on their computers, potentially causing catastrophic damage. Some potential risks include:

- *Malware Infection*: The code could contain viruses, worms, or trojans that compromise our system's security.
- *Data Loss*: The code might delete or corrupt important files and data.
- *Unauthorized Access*: The code can create backdoors, allowing attackers to access our system remotely.
- *Performance Issues*: The code might consume excessive resources, slowing down our system.
- *Privacy Breaches*: Sensitive information in the generated code, such as passwords or personal data, might be stolen.
- *System Damage*: The code may alter system settings or cause damage to hardware components, such as through overclocking, overheating, or excessive read/write operations.
- *Network Spread*: The code could propagate across networks, affecting other devices.
- *Financial Loss*: If the code is ransomware, it might encrypt data and demand payment for decryption.
- *Legal Consequences*: Running certain types of malicious code can lead to legal repercussions.

As illustrated, aligning Code LLMs to produce source code consistent with human preferences and values is of paramount importance in software development. A recent study [291] provides the first systematic literature review identifying seven critical non-functional properties of LLMs for code, beyond accuracy, including robustness, security, privacy, explainability, efficiency, and usability. This study is highly pertinent to the alignment of Code LLMs. We recommend readers refer to this survey for more detailed insights.

In this survey, we identify five core principles that serve as the key objectives for aligning Code LLMs: Sustainability, Responsibility, Efficiency, Safety, and Trustworthiness (collectively referred to as *SREST*). These principles are examined from a broader perspective. Each category encompasses various concepts and properties, which are summarized in Table 12. In the following, we define each principle and briefly introduce a few notable works to enhance understanding.

Sustainability. The Sustainability principle underscores the importance of environmental sustainability in the development and deployment of LLMs for code generation. This involves optimizing energy consumption and reducing both the carbon footprint and financial costs associated with

Table 12. Five Core Principles Serve as the Key Objectives for Code LLMs Alignment: Sustainability, Responsibility, Efficiency, Safety, and Trustworthiness (Collectively Referred to as *SREST*)

Principles	**Involved Concepts and Properties**
Sustainability	*Energy Efficiency*: Minimizing computational energy use and reduce environmental impact and financial costs. *Sustainable Materials*: Leveraging eco-friendly infrastructure and servers for code generation, lowering long-term expenses. *Carbon Footprint*: Reducing emissions associated with model training and inference to enhance efficiency and save costs. *Resource Optimization*: Efficiently utilizing computational resources to minimize waste and reduce expenses in code generation. *Recycling Management*: Responsibly dispose of hardware used in model development to reduce waste management costs. *Renewable Energy*: Utilizing renewable energy sources for powering training and inference processes to decrease energy costs. *Lifecycle Assessment*: Evaluating the environmental and financial impacts of models from creation to deployment and disposal.
Responsibility	*Ethical Considerations*: Adhering to ethical guidelines to ensure responsible use and deployment of generated code. *Accountability*: Establishing clear lines of responsibility for code generation outcomes and potential impacts. *User Education*: Providing resources and guidance to help users understand and responsibly use generated code. *Impact Assessment*: Evaluating the social and technical implications of code generation to minimize negative effects. *Regulatory Compliance*: Ensuring that generated code adheres to relevant laws (e.g., copyright) and industry regulations.
Efficiency	*Model Optimization*: Streamlining models to reduce computational load and improve speed. *Prompt Engineering*: Designing effective prompts to generate accurate code efficiently. *Resource Management*: Allocating computational resources wisely to balance speed and cost. *Inference Optimization*: Enhancing the inference process to quickly generate code with minimal latency. *Parallel Processing*: Utilizing parallelism to speed up code generation tasks. *Caching Mechanisms*: Implementing caching to reuse previous results and reduce redundant computations. *Evaluation Metrics*: Using precise metrics to assess and improve the efficiency of code outputs.
Safety	*Input Validation*: Ensuring inputs (prompts) are safe and sanitized to prevent malicious exploitation. *Security Audits*: Regularly reviewing generated code for vulnerabilities and potential exploits. *Monitoring and Logging*: Keeping track of generation outputs to quickly identify and address safety issues. *User Access Control*: Limiting access to generation capabilities to trusted users to minimize risk. *Continuous Updates*: Regularly updating models with the latest safety protocols and security patches. *Ethical Guidelines*: Implementing ethical standards to guide safe and responsible code generation.
Trustworthiness	*Reliability*: Ensuring that generated code consistently meets functional requirements and performs as expected. *Transparency*: Providing clear explanations of how code is generated to build user confidence. *Verification and Testing*: Using rigorous testing frameworks to ensure the generated code accuracy and reliability. *Bias Mitigation*: Actively working to identify and reduce biases in code generation to ensure fairness and impartiality. *User Feedback Integration*: Continuously incorporating user feedback to refine and improve code generation processes. *Documentation*: Providing comprehensive documentation for generated code to enhance understanding and trust.

training and inference processes. Currently, training, inference, and deployment of Code LLMs are notably resource-intensive. For example, training GPT-3, with its 175 billion parameters, required the equivalent of 355 years of single-processor computing time and consumed 284,000 kWh of energy, resulting in an estimated 552.1 tons of CO_2 emissions [225]. Furthermore, a ChatGPT-like application, with an estimated usage of 11 million requests per hour, can produce emissions of 12.8k metric tons of CO_2 per year, which is 25 times the carbon emissions associated with training GPT-3 [54]. To mitigate these costs, several techniques are often employed, such as the development of specialized hardware (e.g., Tensor Processing Units and Neural Processing Units), model compression methods (e.g., quantization and knowledge distillation), PEFT, and the use of renewable energy sources. For instance, Shi et al. [232] applied knowledge distillation to reduce the size of CodeBERT [76] and GraphCodeBERT [86], resulting in optimized models of just 3 MB. These models are 160 times smaller than the original large models and significantly reduce energy consumption by up to 184 times and carbon footprint by up to 157 times. Similarly, Wei et al. [275] utilized quantization techniques for Code LLMs such as CodeGen [191] and Incoder [77] by employing lower-bit integers (e.g., `int8`). This approach reduced storage requirements by 67.3% to 70.8%, carbon footprint by 28.8% to 55.0%, and pricing costs by 28.9% to 55.0%.

Responsibility. The Responsibility principle in the context of Code LLMs underscores the importance of ethical considerations, fairness, and accountability throughout their lifecycle. This involves addressing biases in training data, ensuring fairness and transparency in model decision-making, maintaining accountability for outputs, adhering to applicable laws (e.g., copyright), implementing safeguards against misuse, and providing clear communication about the model's capabilities and limitations. Specifically,

— *Bias Mitigation*. Biases in code generation can lead to flawed software and reinforce stereotypes, potentially causing significant societal impacts. For example, an Code LLM that inherits biases from its training data may produce source code/software that inadvertently discriminates against certain user groups. This can result in applications that fail to meet the diverse needs of users, promoting exclusionary practices and reinforcing existing stereotypes [166, 183].
— *Fairness and Transparency*. A lack of fairness and transparency in Code LLM decision-making can result in biased or suboptimal code solutions. If the model's decision-making process is opaque, developers might unknowingly introduce code that favors specific frameworks or libraries, thereby limiting innovation and diversity in software development. This opacity can create unfair advantages and hinder collaborative efforts within tech communities [32].
— *Legal Compliance*. Compliance with relevant laws, such as licensing and copyright, is crucial when using Code LLMs for code generation to avoid legal complications. If a Code LLM generates code snippets that inadvertently infringe on existing copyrights, it can lead to legal disputes and financial liabilities for developers and organizations [290]. Such risks may discourage the use of advanced AI tools, thus stifling innovation and affecting growth and collaboration within the tech community.
— *Accountability*. Without accountability for code generated by Code LLMs, addressing bugs or security vulnerabilities becomes challenging. If a model generates faulty code leading to a security breach, the absence of clear accountability can result in significant financial and reputational damage for companies. This uncertainty can delay critical issue resolution and impede trust in AI-assisted development [154].
— *Misuse Prevention*. Failing to implement mechanisms to prevent the misuse of Code LLMs can enable the creation of harmful software. For example, models could be exploited to generate malware or unauthorized scripts, posing cybersecurity risks. Without proper safeguards, these models can facilitate malicious activities, threatening both individual and organizational security [182].
— *Clear Communication*. Without clear communication about a model's capabilities and limitations, developers may misuse the model or overestimate its abilities. Relying on the model to generate complex, mission-critical code without human oversight can lead to significant software failures. Misunderstanding its limitations can result in faulty implementations and lost productivity [223].

To adhere to this principle, potential mitigation methods include bias detection and mitigation, quantification and evaluation, and adherence to ethical guidelines. Liu et al. [166] propose a new paradigm for constructing code prompts, successfully uncovering social biases in code generation models, and developing a dataset along with three metrics to evaluate overall social bias. Recently, Xu et al. [290] introduced LiCoEval, an evaluation benchmark for assessing the license compliance capabilities of LLMs. Additionally, incorporating diverse perspectives in development teams and engaging with stakeholders from various communities can further align Code LLM outputs with ethical standards and societal values.

Efficiency. The Efficiency principle emphasizes optimizing the performance and speed of Code LLMs for code generation while minimizing the computational resources required for training and inference. For instance, training the GPT-3 model, which consists of 175 billion parameters, demands substantial resources. It requires approximately 1,024 NVIDIA V100 GPUs, costing around 4.6 million dollars and taking approximately 34 days to complete the training process. To address these challenges, various techniques are employed, including model compression methods (e.g., pruning, quantization, and knowledge distillation), optimized algorithms (e.g., AdamW), parallel strategies (e.g., tensor, pipeline, and data parallelism), and PEFT (see Section 5.5.2 for more details).

For a comprehensive and detailed discussion on methods to enhance the efficiency of Code LLMs for code generation, please refer to Section 4.5.2, "Efficiency Enhancement," in [291].

Safety. The Safety principle of Code LLMs is of utmost importance due to their potential to introduce vulnerabilities, errors, or privacy breaches into software systems. Ensuring safety involves comprehensive testing and validation processes to detect and mitigate these risks. For instance, attackers might compromise the training process of LLMs by injecting malicious examples into the training data, a method known as data poisoning attacks [228]. Even when attackers lack access to the training process, they may employ techniques like the black-box inversion approach introduced in [91]. This method uses few-shot prompting to identify prompts that coax black-box code generation models into producing vulnerable code. Furthermore, [292] and [9] reveal that Code LLMs, such as CodeParrot [73], can memorize training data, potentially outputting personally identifiable information like e-mails, names, and IP addresses, thereby posing significant privacy risks. Additionally, [300] demonstrates that engaging with ChatGPT and GPT-4 in non-NLs can circumvent safety alignment measures, leading to unsafe outcomes, such as "The steps involved in stealing money from a bank." To bolster the safety of LLMs in code generation, it is crucial to detect and eliminate privacy-related information from training datasets. For example, approaches outlined in [77] and [10] utilize carefully crafted regular expressions to identify and remove private information from training data. To counteract black-box inversion, implementing prompt filtering mechanisms is recommended to identify and block prompts that might result in insecure code generation. Moreover, adversarial training can enhance the model's resilience to malicious prompts. Employing reinforcement learning methods can further align Code LLMs with human preferences, thereby reducing the likelihood of producing harmful outputs.

Trustworthiness. The Trustworthiness principle focuses on developing Code LLMs that users can depend on for accurate and reliable code generation, which is crucial for their acceptance and widespread adoption. Achieving this requires ensuring model transparency, providing explanations for decisions, and maintaining consistent performance across various scenarios. For instance, [120] proposes a causal graph-based representation of prompts and generated code to identify the causal relationships between them. This approach offers insights into the effectiveness of Code LLMs and assists end-users in understanding the generation. Similarly, [200] introduces ASTxplainer, a tool that extracts and aggregates normalized model logits within AST structures. This alignment of token predictions with AST nodes provides visualizations that enhance end-user understanding of Code LLM predictions. Therefore, by prioritizing trustworthiness, we can bolster user confidence and facilitate the integration of Code LLMs into diverse coding environments. By adhering to the aforementioned principles as key objectives for aligning Code LLMs, researchers and developers can create LLMs for code generation that are not only capable but also ethical, sustainable, and user-centric.

5.13 Applications

Code LLMs have been integrated with development tools and platforms, such as **Integrated Development Environments (IDEs)** and version control systems, improving programming efficiency substantially. In this section, we will briefly introduce several widely used applications as coding assistants. The statistics of these applications are provided in Table 13.

GitHub Copilot. GitHub Copilot, powered by OpenAI's Codex, is an AI pair programmer that helps you write better code faster. Copilot suggests whole lines or blocks of code as you type, based on the context provided by your existing code and comments. It's trained on a dataset that includes a significant portion of the public code available on GitHub, which enables it to understand a wide range of PLs and coding styles. Copilot not only improves productivity but also serves as a learning

Table 13. The Overview of Code Assistant Applications Powered by LLMs

Institution	Products	Model	Supported Features	Supported PLs	Supported IDEs
GitHub OpenAI	GitHub Copilot [49]	Claude 3.5 Sonnet (Preview) Claude 3.7 Sonnet (Preview) Claude 3.7 Sonnet Thinking (Preview) Gemini 2.0 Flash (Preview) GPT-4o o1 (Preview) o3-mini (Preview)	Code Completions, Code Generation, Coding Questions Answering, Code Refactoring, Code Issues Fix, Unit Test Cases Generation, Code Documentation Generation	Java, Python, JavaScript, TypeScript, Perl, R, PowerShell, Rust, SQL, CSS, Ruby, Julia, C#, PHP, Swift, C++,Go, HTML, JSON, SCSS, .NET, Less, T-SQL, Markdown	Visual Studio, VS Code, Neovim, JetBrains IDE
Zhipu AI	CodeGeeX [320]	CodeGeeX	Code Generation, Code Translation, Code Completion, Code Interpretation, Code Bugs Fix, Comment Generation, AI Chatbot	PHP, Go, C, C#, C++, Rust, Perl, CSS, Java, Python, JavaScript, TypeScript, Objective C++, Objective C, Pascal, HTML, SQL, Kotlin, R, Shell, Cuda, Fortran, Tex, Lean, Scala	Clion, RubyMine, AppCode, Aqua, IntelliJ IDEA, VS Code, PyCharm, Android Studio, WebStorm, Rider, GoLand, DataGrip, DataSpell
Amazon	CodeWhisperer [13]	–	Code Completion, Code Explanation, Code Translation, Code Security Identification, Code Suggestion	Java, Python, TypeScript, JavaScript, C#	JetBrains IDE, VS Code, AWS Cloud9, AWS Lambda
Codeium	Codeium [61]	–	Code Completion, Bug Detection, Code Suggestions, AI Chatbot, Test Type Generation, Test Plan Creation, Codebase Search	More than 70 languages in total, including but not limited to: C, C#, C++, Dart, CSS, Go, Elixir, HTML, Haskell, Julia, Java, JavaScript, Lisp, Kotlin, Lua, Objective-C, Perl, Pascal, PHP, Protobuf, R, Python, Ruby, Scala, Rust, Swift, SQL, TS, Vue	JetBrains, VSCode, Visual Studio, Colab, Jupyter, Deepnote, Notebooks, Databricks, Chrome, Vim, Neovim, Eclipse, Emacs, VSCode Web IDEs, Sublime Text
Huawei	CodeArts Snap [231]	PanGu-Coder	Code Generation, Code Explanation Research and Development Knowledge Question and Answer Code Comment, Code Debug Unit Test Case Generation	Java, Python	PyCharm, VS Code, IntelliJ
Tabnine	TabNine [243]	–	Code Generation, Code Completion, Code Explanation, Bug Fix, Code Recommendation, Code Refactoring, Code Test Generation, Docstring Generation	Python, Javascript, Java, TypeScript, HTML, Haskell, Matlab, Kotlin, Sass, Go, PHP, Ruby, C, C#, C++, Swift, Rust, CSS, Perl, Angular, Dart, React, Objective C, NodeJS, Scala,	Sublime, PyCharm, Neovim, Rider, VS Code, IntelliJ IDE, Visual Studio, PhpStorm, Vim, RubyMine, DataGrip, Android Studio, WebStorm, Emacs, Clion, Jupyter Notebook, JupyterLab, Eclipse, GoLand, AppCode
Replit	Replit[219]	replit-code	Code Completion, Code Editing, Code Generation, Code Explanation, Code Suggestion, Code Test Generation	C#, Bash, C, CSS, C++, Java, Go, HTML, JavaScript, Perl, PHP, Ruby, Python, R, SQL, Rust	–

The column labeled “**PLs**” and “**IDEs**” indicate programming languages and integrated development environments, respectively [305]. VS Code, Visual Studio Code.

tool by providing programmers with examples of how certain functions can be implemented or how specific problems can be solved. (Description source: [49]).

CodeGeeX. CodeGeeX stands out as a multifaceted programming assistant, proficient in code completion, comment generation, code translation, and developer interactions. Its underlying code generation LLM has been refined with extensive training on vast amounts of code data, exhibiting superior performance on benchmarks like HumanEval, HumanEval-X, and DS1000. Renowned for supporting multilingual code generation, CodeGeeX plays a pivotal role in enhancing the efficiency of code development. (Description source: [320]).

CodeWhisperer. Amazon's CodeWhisperer is a versatile, machine learning-driven code generator that offers on-the-fly code recommendations. Tailored to your coding patterns and comments, CodeWhisperer provides personalized suggestions that range from succinct comments to complex functions, all aimed at streamlining your coding workflow. (Description source: [13]).

Codeium. Codeium is a coding toolkit that offers a suite of functions, including code completion, explanation, translation, search, and user chatting. Codeium can support over 70 PLs and provide solutions to coding challenges, making development easier for users. (Description source: [61]).

CodeArts Snap. Huawei's CodeArts Snap is capable of generating comprehensive function-level code from both Chinese and English descriptions. This tool not only reduces the monotony of manual coding but also efficiently generates test code, in addition to providing automatic code analysis and repair services. (Description source: [231]).

Tabnine. Tabnine is an AI coding assistant that can help development teams streamline the software development process while adhering to privacy, security, and compliance standards. It provides AI-driven automation tailored to team needs, focusing on coding efficiency and quality. (Description source: [243]).

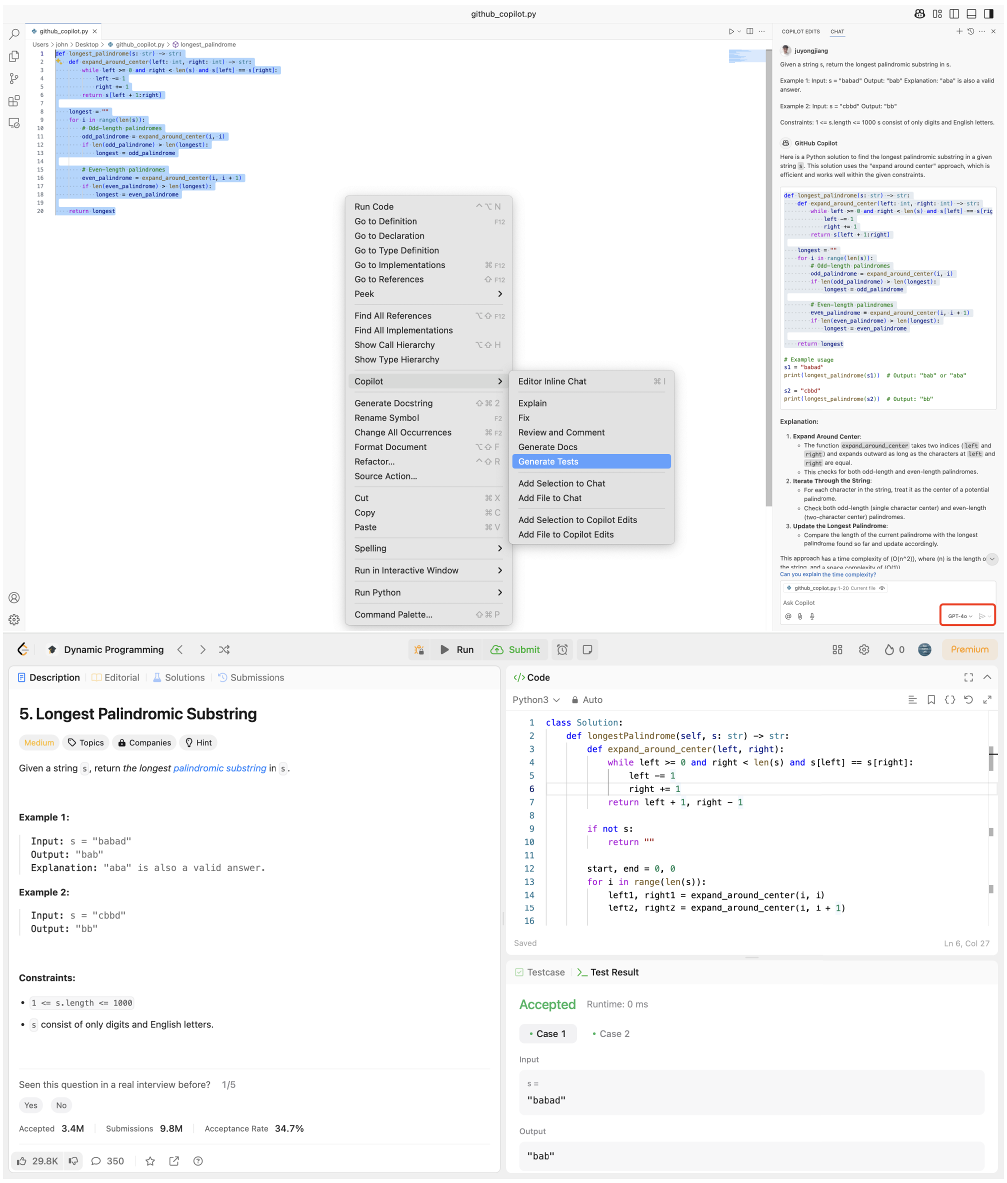
Replit. Replit is a multifunctional platform that caters to a diverse array of software development needs. As a complimentary online IDE, it facilitates code collaboration, and cloud services, and fosters a thriving developer community. Replit also enables users to compile and execute code in more than 50 PLs directly within a web browser, eliminating the need for local software installations. (Description source: [219]).

To illustrate the use of development tools powered by LLMs, we employ GitHub Copilot within **Visual Studio Code (VS Code)** as our example. Note that

① For details on using the GitHub Copilot extension in VS Code, please refer to the document at https://code.visualstudio.com/docs/copilot/overview.

② If you would like to get free access to Copilot as a student, teacher, or open source maintainer, please refer to this tutorial at https://docs.github.com/en/copilot/managing-copilot/managing-copilot-as-an-individual-subscriber/managing-your-copilot-subscription/getting-free-access-to-copilot-as-a-student-teacher-or-maintainer and GitHub education application portal at https://education.github.com/discount_requests/application.

As depicted in the upper section of Figure 20, users can interact with Copilot through the chat box in the lower left corner, where they can inquire about various coding-related tasks. This feature is now supported by the advanced capabilities of GPT-4o, o1-preview (Preview), and o1-mini (Preview). From the generated content, Copilot demonstrates the ability to plan solutions to coding problems. It can write code and subsequently explain the generated code to enhance user comprehension. Within the right-side workspace, users can engage in inline chat conversations to generate or refactor source code, conduct code explanations, fix coding errors, resolve issues encountered during terminal command executions, produce documentation comments, and generate unit tests. To illustrate its capabilities, we input the description of the "5. Longest Palindromic Substring"



```python
def longest_palindrome(s: str) -> str:
    def expand_around_center(left: int, right: int) -> str:
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return s[left + 1:right]

    longest = ""
    for i in range(len(s)):
        # Odd-length palindromes
        odd_palindrome = expand_around_center(i, i)
        if len(odd_palindrome) > len(longest):
            longest = odd_palindrome

        # Even-length palindromes
        even_palindrome = expand_around_center(i, i + 1)
        if len(even_palindrome) > len(longest):
            longest = even_palindrome

    return longest
```

juyongjiang

Given a string s, return the longest palindromic substring in s.

Example 1: Input: s = "babad" Output: "bab" Explanation: "aba" is also a valid answer.

Example 2: Input: s = "cbbd" Output: "bb"

Constraints: 1 <= s.length <= 1000 s consist of only digits and English letters.

GitHub Copilot

Here is a Python solution to find the longest palindromic substring in a given string `s`. This solution uses the "expand around center" approach, which is efficient and works well within the given constraints.

```python
def longest_palindrome(s: str) -> str:
    def expand_around_center(left: int, right: int) -> str:
        while left >= 0 and right < len(s) and s[left] == s[rig
            left -= 1
            right += 1
        return s[left + 1:right]

    longest = ""
    for i in range(len(s)):
        # Odd-length palindromes
        odd_palindrome = expand_around_center(i, i)
        if len(odd_palindrome) > len(longest):
            longest = odd_palindrome

        # Even-length palindromes
        even_palindrome = expand_around_center(i, i + 1)
        if len(even_palindrome) > len(longest):
            longest = even_palindrome

    return longest

# Example usage
s1 = "babad"
print(longest_palindrome(s1))  # Output: "bab" or "aba"

s2 = "cbbd"
print(longest_palindrome(s2))  # Output: "bb"
```

Explanation:

1. **Expand Around Center:**
   - The function `expand_around_center` takes two indices (`left` and `right`) and expands outward as long as the characters at `left` and `right` are equal.
   - This checks for both odd-length and even-length palindromes.
2. **Iterate Through the String:**
   - For each character in the string, treat it as the center of a potential palindrome.
   - Check both odd-length (single character center) and even-length (two-character center) palindromes.
3. **Update the Longest Palindrome:**
   - Compare the length of the current palindrome with the longest palindrome found so far and update accordingly.

This approach has a time complexity of (O(n^2)), where (n) is the length of the string, and a space complexity of (O(1)).

Can you explain the time complexity?

github_copilot.py:1-20 Current file

Ask Copilot

GPT-4o

5. Longest Palindromic Substring

Medium Topics Companies Hint

Given a string `s`, return the *longest palindromic substring* in `s`.

Example 1:

```
Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.
```

Example 2:

```
Input: s = "cbbd"
Output: "bb"
```

Constraints:

- `1 <= s.length <= 1000`
- `s` consist of only digits and English letters.

Seen this question in a real interview before? 1/5

Yes No

Accepted 3.4M | Submissions 9.8M | Acceptance Rate 34.7%

29.8K 350

```python
class Solution:
    def longestPalindrome(self, s: str) -> str:
        def expand_around_center(left, right):
            while left >= 0 and right < len(s) and s[left] == s[right]:
                left -= 1
                right += 1
            return left + 1, right - 1

        if not s:
            return ""

        start, end = 0, 0
        for i in range(len(s)):
            left1, right1 = expand_around_center(i, i)
            left2, right2 = expand_around_center(i, i + 1)
```

Saved Ln 6, Col 27

Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

s =
"babad"

Output

"bab"

Fig. 20. An exemplar of GitHub Copilot to demonstrate how to use development tools powered by LLMs, including Claude 3.5 Sonnet (Preview), Claude 3.7 Sonnet (Preview), Claude 3.7 Sonnet Thinking (Preview), Gemini 2.0 Flash (Preview), GPT-4o, o1 (Preview), and o3-mini (Preview). To illustrate its capabilities, we input the description of the "5. Longest Palindromic Substring" problem from LeetCode into Copilot's chat box. The code generated by Copilot is then submitted to the online judge platform, where it is successfully accepted.

problem from LeetCode into Copilot's chat box. The code generated by Copilot is then submitted to the online judge platform, where it is successfully accepted, as shown at the lower section of Figure 20.

6 Challenges and Opportunities

According to our investigations, the LLMs have revolutionized the paradigm of code generation and achieved remarkable performance. Despite this promising progress, there are still numerous challenges that need to be addressed. These challenges are mainly caused by the gap between academia and practical development. For example, in academia, the HumanEval benchmark has been established as a *de facto* standard for evaluating the coding proficiency of LLMs. However, many works have illustrated that the evaluation of HumanEval can't reflect the scenario of practical development [69, 72, 123, 161]. These serious challenges offer substantial opportunities for further research and applications. In this section, we pinpoint critical challenges and identify promising opportunities, aiming to bridge the research-practicality divide.

Enhancing Complex Code Generation at Repository and Software Scale. In practical development scenarios, it often involves a large number of complex programming problems of varying difficulty levels [150, 309]. While LLMs have shown proficiency in generating function-level code snippets, these models often struggle with more complex, unseen programming problems, repository- and software-level problems that are commonplace in real-world software development. To this end, it requires strong problem-solving skills in LLM beyond simply functional-level code generation. For example, AlphaCode [150] achieved an average ranking in the top 54.3% in programming competitions where an understanding of algorithms and complex NL is required to solve competitive programming problems. Jimenez et al. [123] argue that existing LLMs can't resolve real-world GitHub issues well since the best-performing model, Claude 2, is able to solve a mere 1.96% of the issues. The reason for poor performance is mainly attributed to the weak reasoning capabilities [105], complex internal and external dependencies [23], and context length limitation of LLMs [23]. Therefore, the pursuit of models that can handle more complex, repository- and software-level code generation opens up new avenues for automation in software development and makes programming more productive and accessible.

Innovating Model Architectures Tuned to Code Structures. Due to their scalability and effectiveness, Transformer-based LLM architectures have become dominant in solving code generation task. Nevertheless, they might not be optimally designed to capture the inherent structure and syntax of PLs [85, 86, 134, 173]. Code has a highly structured nature, with a syntax that is more rigid than NL. This presents a unique challenge for LLMs, which are often derived from models that were originally designed for NLP. The development of novel model architectures that inherently understand and integrate the structural properties of code represents a significant opportunity to improve code generation and comprehension. Innovations such as tree-based neural networks [181], which mirror the AST representation of code, can offer a more natural way for models to learn and generate PLs. Additionally, leveraging techniques from the compiler theory, such as intermediate representations [151], could enable models to operate on a more abstract and generalizable level, making them effective across multiple PLs [205]. By exploring architectures beyond the traditional sequential models, researchers can unlock new potentials in code generation.

Curating High-Quality Code Data for Pre-Training and Fine-Tuning of LLMs. The efficacy of LLMs largely depends on the quality and diversity of code datasets used during pre-training and fine-tuning phases [133, 278, 327]. Currently, there is a scarcity of large, high-quality datasets that encompass a wide range of programming tasks, styles, and languages. This limitation constrains the ability of LLMs to generalize across unseen programming tasks, different coding environments, and real-world software development scenarios. The development of more sophisticated data acquisition techniques, such as automated code repositories mining [157], advanced filtering algorithms, and code data synthesis [164] (see Section 5.3), can lead to the creation of richer datasets. Collaborations with industry partners (e.g., GitHub) could also facilitate access to proprietary codebases, thereby enhancing the practical relevance of the training material. Furthermore, the adoption of open source models for dataset sharing can accelerate the collective effort to improve the breadth and depth of code data available for LLM research.

Developing Comprehensive Benchmarks and Metrics for Coding Proficiency Evaluation in LLMs. Current benchmarks like HumanEval may not capture the full spectrum of coding skills required for practical software development [189]. Additionally, metrics often focus on syntactic correctness or functional accuracy, neglecting aspects such as code efficiency [206], style [45], readability [35], or maintainability [16]. The design of comprehensive benchmarks that simulate real-world software development challenges could provide a more accurate assessment of LLMs' coding capabilities. These benchmarks should include diverse programming tasks of varying difficulty levels, such as debugging [324], refactoring [234], and optimization [112], and should be complemented by metrics that evaluate qualitative aspects of code. The establishment of community-driven benchmarking platforms could facilitate continuous evaluation and comparison of LLMs for code generation across the industry and academia.

Support for Low-Resource, Low-Level, and Domain-Specific PLs. LLMs are predominantly trained in popular high-level PLs, leaving low-resource, low-level, and domain-specific languages underrepresented. This lack of focus restricts the applicability of LLMs in certain specialized fields and systems programming [247]. Intensifying research on transfer learning and meta-learning approaches may enable LLMs to leverage knowledge from high-resource languages to enhance their performance on less common ones [39, 47]. Additionally, partnerships with domain experts can guide the creation of targeted datasets and fine-tuning strategies to better serve niche markets. The development of LLMs with a capacity for multilingual code generation also presents a significant opportunity for broadening the scope of applications.

Continuous Learning for LLMs to Keep Pace with Evolving Coding Knowledge. The software development landscape is continuously evolving, with new languages, frameworks, and best practices emerging regularly. LLMs risk becoming outdated if they cannot adapt to these changes and incorporate the latest programming knowledge [115, 262]. While RACG mitigates these issues, its effectiveness is inherently constrained by the quality of retrieved context [169, 307, 329]. Therefore, establishing mechanisms for continuous learning and updating of LLMs can help maintain their relevance over time. This could involve real-time monitoring of code repositories to identify trends and innovations, as well as the creation of incremental learning systems that can assimilate new information without forgetting previously acquired knowledge. However, continuous learning for LLMs is susceptible to data poisoning attacks, which can result in catastrophic forgetting of established knowledge [4]. Engaging the LLMs in active learning scenarios where they interact with human developers may also foster ongoing knowledge acquisition.

Ensuring Code Safety and Aligning LLM Outputs with Human Coding Preferences. Ensuring the safety and security of code generated by LLMs is a paramount concern, as is their ability to align with human preferences and ethical standards. Current models may inadvertently introduce vulnerabilities or generate code that does not adhere to desired norms [49, 291]. Research into the integration of formal verification tools within the LLM pipeline can enhance the safety of the produced code by mathematically verifying correctness, detecting logical errors, ensuring compliance with security standards, and preventing vulnerabilities. Additionally, developing frameworks for alignment learning that capture and reflect human ethical preferences can ensure that the code generation process aligns with societal values [198, 209]. Transparent and explainable AI methodologies can also contribute to building trust in the LLM-generated code by making the decision-making process more accessible to developers.

7 Conclusion

In this survey, we provide a systematic literature review, serving as a valuable reference for researchers investigating the cutting-edge progress in LLMs for code generation. A thorough introduction and analysis for data curation, the latest advances, performance evaluation, ethical implications, environmental impact, and real-world applications are illustrated. In addition, we present a historical overview of the evolution of LLMs for code generation in recent years and offer an empirical comparison using the widely recognized HumanEval, MBPP, and the more practical and challenging BigCodeBench benchmarks to highlight the progressive enhancements in LLM capabilities for code generation. Critical challenges and promising opportunities regarding the gap between academia and practical development are also identified for future investigation. Furthermore, we have established a dedicated resource Web site to continuously document and disseminate the most recent advances in the field. We hope this survey can contribute to a comprehensive and systematic overview of LLM for code generation and promote its thriving evolution. We optimistically believe that LLM will ultimately change all aspects of coding and automatically write safe, helpful, accurate, trustworthy, and controllable code, like professional programmers, and even solve coding problems that currently cannot be solved by humans.

For more information on prompt engineering, visit https://www.promptingguide.ai.

References

[1] GitHub. 2023. AgentGPT: Assemble, Configure, and Deploy Autonomous AI Agents in Your Browser. Retrieved from https://github.com/reworkd/AgentGPT

[2] GitHub. 2023. AutoGPT Is the Vision of Accessible AI for Everyone, to Use and to Build On. Retrieved from https://github.com/Significant-Gravitas/AutoGPT

[3] GitHub. 2023. BabyAGI. Retrieved from https://github.com/yoheinakajima/babyagi

[4] Ali Abbasi, Parsa Nooralinejad, Hamed Pirsiavash, and Soheil Kolouri. 2024. BrainWash: A poisoning attack to forget in continual learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 24057–24067.

[5] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. arXiv:2404.14219. Retrieved from https://arxiv.org/abs/2404.14219

[6] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. arXiv:2303.08774. Retrieved from https://arxiv.org/abs/2303.08774

[7] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 4998–5007.

[8] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. arXiv:2103.06333. Retrieved from https://arxiv.org/abs/2103.06333

[9] Ali Al-Kaswan, Maliheh Izadi, and Arie Van Deursen. 2024. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–12.

[10] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: Don't reach for the stars! arXiv:2301.03988. Retrieved from https://arxiv.org/abs/2301.03988

[11] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 472–483.

[12] Google DeepMind AlphaCode Team. 2023. AlphaCode 2 Technical Report. Retrieved from https://storage.googleapis.com/deepmind-media/AlphaCode2/AlphaCode2_Tech_Report.pdf

[13] Amazon. 2022. What Is CodeWhisperer? https://docs.aws.amazon.com/codewhisperer/latest/userguide/what-is-cwspr.html

[14] Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. MathQA: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2357–2367.

[15] Anthropic. 2024. The Claude 3 Model Family: Opus, Sonnet, Haiku. Retrieved from https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf

[16] Luca Ardito, Riccardo Coppola, Luca Barbato, and Diego Verga. 2020. A tool-based perspective on software code maintainability metrics: A systematic literature review. *Scientific Programming* 2020 (2020), 1–26.

[17] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. In *Proceedings of the 11th International Conference on Learning Representations*.

[18] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. arXiv:2108.07732. Retrieved from https://arxiv.org/abs/2108.07732

[19] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. arXiv:1607.06450. Retrieved from https://arxiv.org/abs/1607.06450

[20] Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q. Feldman, and Carolyn Jane Anderson. 2023. StudentEval: A benchmark of student-written prompts for large language models of code. arXiv:2306.04556. Retrieved from https://arxiv.org/abs/2306.04556

[21] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. arXiv:2309.16609. Retrieved from https://arxiv.org/abs/2309.16609

[22] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. 2022. Constitutional AI: Harmlessness from AI feedback. arXiv:2212.08073. Retrieved from https://arxiv.org/abs/2212.08073

[23] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2023. CodePlan: Repository-level coding using LLMs and planning. arXiv:2309.12499. Retrieved from https://arxiv.org/abs/2309.12499

[24] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, 65–72.

[25] Enrico Barbierato, Marco L. Della Vedova, Daniele Tessera, Daniele Toti, and Nicola Vanoli. 2022. A methodology for controlling bias and fairness in synthetic data generation. *Applied Sciences* 12, 9 (2022), 4619.

[26] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[27] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. DeepSeek LLM: Scaling open-source language models with longtermism. arXiv:2401.02954. Retrieved from https://arxiv.org/abs/2401.02954

[28] Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Xuanhua Shi, and Hai Jin. 2024. Iterative refinement of project-level code context for precise code generation with compiler feedback. arXiv:2403.16792. Retrieved from https://arxiv.org/abs/2403.16792

[29] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2022. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* 20, 6 (2022), 35–57.

[30] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. GPT-NeoX-20B: An open-source autoregressive language model. arXiv:2204.06745. Retrieved from https://arxiv.org/abs/2204.06745

[31] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow*. Zenodo. Retrieved March 21, 2021 https://zenodo.org/records/5297715

[32] Veronika Bogina, Alan Hartman, Tsvi Kuflik, and Avital Shulner-Tal. 2022. Educating software and AI stakeholders about algorithmic fairness, accountability, transparency and ethics. *International Journal of Artificial Intelligence in Education* 32 (2022), 808–833.

[33] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. arXiv:2108.07258. Retrieved from https://arxiv.org/abs/2108.07258

[34] Tom B. Brown. 2020. Language models are few-shot learners. arXiv:2005.14165. Retrieved from https://arxiv.org/abs/2005.14165

[35] Raymond P. L. Buse and Westley R. Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on Software Engineering* 36, 4 (2009), 546–558.

[36] Weilin Cai, Juyong Jiang, Le Qin, Junwei Cui, Sunghun Kim, and Jiayi Huang .2024. Shortcut-connected expert parallelism for accelerating mixture-of-experts. arXiv:2404.05019. Retrieved from https://arxiv.org/abs/2404.05019

[37] Weilin Cai, Juyong Jiang, Fan Wang, Jing Tang, Sunghun Kim, and Jiayi Huang. 2024. A survey on mixture of experts. arXiv:2407.06204. Retrieved from https://arxiv.org/abs/2407.06204

[38] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. 2021. Extracting training data from large language models. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*, 2633–2650.

[39] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Carolyn Jane Anderson, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2023. Knowledge transfer from high-resource to low-resource programming languages for code LLMs. arXiv:2308.09895. Retrieved from https://arxiv.org/abs/2308.09895

[40] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, et al. 2022. A scalable and extensible approach to benchmarking NL2Code for 18 programming languages. arXiv:2208.08227. Retrieved from https://arxiv.org/abs/2208.08227

[41] Yekun Chai, Shuohuan Wang, Chao Pang, Yu Sun, Hao Tian, and Hua Wu. 2022. ERNIE-Code: Beyond English-centric cross-lingual pretraining for programming languages. arXiv:2212.06742. Retrieved from https://arxiv.org/abs/2212.06742

[42] Shubham Chandel, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. 2022. Training and evaluating a Jupyter notebook data science assistant. arXiv:2201.12901. Retrieved from https://arxiv.org/abs/2201.12901

[43] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (2024), 1–45.

[44] Sahil Chaudhary. 2023. Code Alpaca: An Instruction-Following LLaMA Model for Code Generation. Retrieved from https://github.com/sahil280114/codealpaca

[45] Binger Chen and Ziawasch Abedjan. 2023. DUETCS: Code style transfer through generation and retrieval. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2362–2373.

[46] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code generation with generated tests. arXiv:2207.10397. Retrieved from https://arxiv.org/abs/2207.10397

[47] Fuxiang Chen, Fatemeh H. Fard, David Lo, and Timofey Bryksin. 2022. On the transferability of pre-trained language models for low-resource programming languages. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 401–412.

[48] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking large language models in retrieval-augmented generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38, 17754–17762.

[49] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[50] Stanley F. Chen, Douglas Beeferman, and Roni Rosenfeld. 2018. Evaluation metrics for language models. DOI: https://doi.org/10.1184/R1/6605324.v1

[51] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. arXiv:2211.12588. Retrieved from https://arxiv.org/abs/2211.12588

[52] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. arXiv:2304.05128. Retrieved from https://arxiv.org/abs/2304.05128

[53] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 31.

[54] Andrew A. Chien, Liuzixuan Lin, Hai Nguyen, Varsha Rao, Tristan Sharma, and Rajini Wijayawardana. 2023. Reducing the carbon impact of generative AI inference (today and in 2035). In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, 1–7.

[55] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[56] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. 2022. PanGu-Coder: Program synthesis with function-level language modeling. arXiv:2207.11280. Retrieved from https://arxiv.org/abs/2207.11280

[57] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research* 25, 70 (2024), 1–53.

[58] Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: Multi-mode translation of natural language and Python code with transformers. arXiv:2010.03150. Retrieved from https://arxiv.org/abs/2010.03150

[59] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. arXiv:2110.14168. Retrieved from https://arxiv.org/abs/2110.14168

[60] CodeGemma Team, Ale Jakse Hartman, Andrea Hu, Christopher A. Choquette-Choo, Heri Zhao, Jane Fine, Jeffrey Hui, Jingyue Shen, Joe Kelley, Joshua Howland, et al. Gong. 2024. CodeGemma: Open Code Models Based on Gemma. Retrieved from https://goo.gle/codegemma

[61] Codeium. 2023. Free, Ultrafast Copilot Alternative for Vim and Neovim. Retrieved from https://github.com/Exafunction/codeium.vim

[62] Cognition. 2024. Introducing Devin, the First AI Software Engineer. Retrieved from https://www.cognition.ai/introducing-devin

[63] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. 2010. Inducing tree-substitution grammars. *The Journal of Machine Learning Research* 11 (2010), 3053–3096.

[64] Cognitive Computations. 2023. oa_leet10k. Retrieved from https://huggingface.co/datasets/cognitivecomputations/oa_leet10k

[65] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[66] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. QLoRA: Efficient finetuning of quantized LLMs. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[67] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805. Retrieved from https://arxiv.org/abs/1810.04805

[68] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2022. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. arXiv:2203.06904. Retrieved from https://arxiv.org/abs/2203.06904

[69] Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[70] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. arXiv:2301.00234. Retrieved from https://arxiv.org/abs/2301.00234

[71] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Junjie Shan, Caishuang Huang, Wei Shen, Xiaoran Fan, Zhiheng Xi, et al. 2024. StepCoder: Improve code generation with reinforcement learning from compiler feedback. arXiv:2402.01391. Retrieved from https://arxiv.org/abs/2402.01391

[72] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[73] Hugging Face. 2023. Training CodeParrot from Scratch. Retrieved from https://github.com/huggingface/blog/blob/main/codeparrot.md

[74] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large language models for software engineering: Survey and open problems. In *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.

[75] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[76] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. arXiv:2002.08155. Retrieved from https://arxiv.org/abs/2002.08155

[77] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A generative model for code infilling and synthesis. arXiv:2204.05999. Retrieved from https://arxiv.org/abs/2204.05999

[78] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The Pile: An 800GB dataset of diverse text for language modeling. arXiv:2101.00027. Retrieved from https://arxiv.org/abs/2101.00027

[79] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the International Conference on Machine Learning*. PMLR, 10764–10799.

[80] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. arXiv:2312.10997. Retrieved from https://arxiv.org/abs/2312.10997

[81] Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. AST-T5: Structure-aware pretraining for code generation and understanding. arXiv:2401.03003. Retrieved from https://arxiv.org/abs/2401.03003
[82] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. CRUXEval: A benchmark for code reasoning, understanding and execution. arXiv:2401.03065. Retrieved from https://arxiv.org/abs/2401.03065
[83] Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 13–24.
[84] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. arXiv:2306.11644. Retrieved from https://arxiv.org/abs/2306.11644
[85] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7212–7225.
[86] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training code representations with data flow. arXiv:2009.08366. Retrieved from https://arxiv.org/abs/2009.08366
[87] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. LongCoder: A long-range pre-trained language model for code completion. In *Proceedings of the International Conference on Machine Learning*. PMLR, 12098–12107.
[88] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, et al. 2024. DeepSeek-Coder: When the large language model meets programming—The rise of code intelligence. arXiv:2401.14196. Retrieved from https://arxiv.org/abs/2401.14196
[89] Aman Gupta, Deepak Bhatt, and Anubha Pandey. 2021. Transitioning from real to synthetic data: Quantifying the bias in model. arXiv:2105.04144. Retrieved from https://arxiv.org/abs/2105.04144
[90] Aman Gupta, Anup Shirgaonkar, Angels de Luis Balaguer, Bruno Silva, Daniel Holstein, Dawei Li, Jennifer Marsman, Leonardo O. Nunes, Mahsa Rouzbahman, Morris Sharp, et al. 2024. RAG vs fine-tuning: Pipelines, tradeoffs, and a case study on agriculture. arXiv:2401.08406. Retrieved from https://arxiv.org/abs/2401.08406
[91] Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario Fritz. 2024. CodeLMSec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models. In *Proceedings of the 2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 684–709.
[92] Perttu Hämäläinen, Mikke Tavast, and Anton Kunnari. 2023. Evaluating large language models in generating synthetic HCI research data: A case study. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 1–19.
[93] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. arXiv:2305.14992. Retrieved from https://arxiv.org/abs/2305.14992
[94] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 770–778.
[95] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. arXiv:2103.03874. Retrieved from https://arxiv.org/abs/2103.03874
[96] Felipe Hoffa. 2016. GitHub on BigQuery: Analyze all the Open Source Code. Retrieved from https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code
[97] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. arXiv:2203.15556. Retrieved from https://arxiv.org/abs/2203.15556
[98] Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. 2023. L2MAC: Large language model automatic computer for unbounded code generation. In *Proceedings of the 12th International Conference on Learning Representations*.
[99] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. arXiv:1904.09751. Retrieved from https://arxiv.org/abs/1904.09751
[100] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. MetaGPT: Meta programming for multi-agent collaborative framework. arXiv:2308.00352. Retrieved from https://arxiv.org/abs/2308.00352
[101] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. arXiv:2308.10620. Retrieved from https://arxiv.org/abs/2308.10620
[102] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *Proceedings of the International Conference on Machine Learning*. PMLR, 2790–2799.

[103] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-rank adaptation of large language models. arXiv:2106.09685. Retrieved from https://arxiv.org/abs/2106.09685

[104] Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. arXiv:2312.13010. Retrieved from https://arxiv.org/abs/2312.13010

[105] Jie Huang and Kevin Chen-Chuan Chang. 2022. Towards reasoning in large language models: A survey. arXiv:2212.10403. Retrieved from https://arxiv.org/abs/2212.10403

[106] Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards reasoning in large language models: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL '23)*. Association for Computational Linguistics, 1049–1065.

[107] Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, Nan Duan, and Jianfeng Gao. 2022. Execution-based evaluation for data science code generation models. arXiv:2211.09374. Retrieved from https://arxiv.org/abs/2211.09374

[108] Qiuyuan Huang, Naoki Wake, Bidipta Sarkar, Zane Durante, Ran Gong, Rohan Taori, Yusuke Noda, Demetri Terzopoulos, Noboru Kuno, Ade Famoti, et al. 2024. Position paper: Agent AI towards a holistic intelligence. arXiv:2403.00833. Retrieved from https://arxiv.org/abs/2403.00833

[109] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2.5-Coder technical report. arXiv:2409.12186. Retrieved from https://arxiv.org/abs/2409.12186

[110] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv:1909.09436. Retrieved from https://arxiv.org/abs/1909.09436

[111] Yoichi Ishibashi and Yoshimasa Nishimura. 2024. Self-organized agents: A LLM multi-agent framework toward ultra large-scale code generation and optimization. arXiv:2404.02183. Retrieved from https://arxiv.org/abs/2404.02183

[112] Shu Ishida, Gianluca Corrado, George Fedoseev, Hudson Yeo, Lloyd Russell, Jamie Shotton, João F. Henriques, and Anthony Hu. 2024. LangProp: A code optimization framework using language models applied to driving. arXiv:2401.10314. Retrieved from https://arxiv.org/abs/2401.10314

[113] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 1643–1652.

[114] Srinivasan Iyer, Xi Victoria Lin, Ramakanth Pasunuru, Todor Mihaylov, Daniel Simig, Ping Yu, Kurt Shuster, Tianlu Wang, Qing Liu, Punit Singh Koura, et al. 2022. OPT-IML: Scaling language model instruction meta learning through the lens of generalization. arXiv:2212.12017. Retrieved from https://arxiv.org/abs/2212.12017

[115] Joel Jang, Seonghyeon Ye, Sohee Yang, Joongbo Shin, Janghoon Han, Gyeonghun Kim, Jungkyu Choi, and Minjoon Seo. 2022. Towards continual knowledge learning of language models. In *Proceedings of the 10th International Conference on Learning Representations (ICLR '22)*. International Conference on Learning Representations.

[116] Fred Jelinek, Robert L. Mercer, Lalit R. Bahl, and James K. Baker. 1977. Perplexity—A measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America* 62, S1 (1977), S63–S63.

[117] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 215–224.

[118] Jiaming Ji, Tianyi Qiu, Boyuan Chen, Borong Zhang, Hantao Lou, Kaile Wang, Yawen Duan, Zhonghao He, Jiayi Zhou, Zhaowei Zhang, et al. 2023. AI alignment: A comprehensive survey. arXiv:2310.19852. Retrieved from https://arxiv.org/abs/2310.19852

[119] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1143–1158.

[120] Zhenlan Ji, Pingchuan Ma, Zongjie Li, and Shuai Wang. 2023. Benchmarking and explaining large language model-based code generation: A causality-centric approach. arXiv:2310.06680. Retrieved from https://arxiv.org/abs/2310.06680

[121] Juyong Jiang and Sunghun Kim. 2023. CodeUp: A Multilingual Code Generation Llama2 Model with Parameter-Efficient Instruction-Tuning. Retrieved from https://github.com/juyongjiang/CodeUp

[122] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A code evolution framework via large language models. arXiv:2306.02907. Retrieved from https://arxiv.org/abs/2306.02907

[123] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2023. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the 12th International Conference on Learning Representations*.

[124] Alexander WettigKilian LieretShunyu YaoKarthik NarasimhanOfir PressJohn Yang, Carlos E. Jimenez. 2024. SWE-Agent: Agent-Computer Interfaces Enable Automated Software Engineering. Retrieved from https://swe-agent.com/

[125] Aravind Joshi and Owen Rambow. 2003. A formalism for dependency grammar based on tree adjoining grammar. In *Proceedings of the Conference on Meaning-Text Theory (MTT)*, 207–216.

[126] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37, 5131–5140.

[127] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. arXiv:2001.08361. Retrieved from https://arxiv.org/abs/2001.08361

[128] Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xCodeEval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. arXiv:2303.03004. Retrieved from https://arxiv.org/abs/2303.03004

[129] Dahyun Kim, Yungi Kim, Wonho Song, Hyeonwoo Kim, Yunsu Kim, Sanghoon Kim, and Chanjun Park. 2024. sDPO: Don't use your data all at once. arXiv:2403.19270. Retrieved from https://arxiv.org/abs/2403.19270

[130] Dahyun Kim, Chanjun Park, Sanghoon Kim, Wonsung Lee, Wonho Song, Yunsu Kim, Hyeonwoo Kim, Yungi Kim, Hyeonju Lee, Jihoo Kim, et al. 2023. SOLAR 10.7B: Scaling large language models with simple yet effective depth up-scaling. arXiv:2312.15166. Retrieved from https://arxiv.org/abs/2312.15166

[131] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering—A systematic literature review. *Information and Software Technology* 51, 1 (2009), 7–15.

[132] Denis Kocetkov, Raymond Li, L. I. Jia, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, et al. 2022. The Stack: 3 TB of permissively licensed source code. arXiv:2211.15533. Retrieved from https://arxiv.org/abs/2211.15533

[133] Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi Rui Tam, Keith Stevens, Abdullah Barhoum, Duc Nguyen, Oliver Stanley, Richárd Nagyfi, et al. 2024. OpenAssistant Conversations—Democratizing large language model alignment. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[134] Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. 2023. Is model attention aligned with human attention? An empirical study on large language models for code generation. arXiv:2306.01220. Retrieved from https://arxiv.org/abs/2306.01220

[135] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. arXiv:2006.03511. Retrieved from https://arxiv.org/abs/2006.03511

[136] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the International Conference on Machine Learning*. PMLR, 18319–18345.

[137] Hugo Laurençon, Lucile Saulnier, Thomas Wang, Christopher Akiki, Albert Villanova del Moral, Teven Le Scao, Leandro Von Werra, Chenghao Mou, Eduardo González Ponferrada, Huu Nguyen, et al. 2022. The BigScience ROOTS Corpus: A 1.6 TB composite multilingual dataset. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 35, 31809–31826.

[138] Moritz Laurer. 2024. Synthetic Data: Save Money, Time and Carbon with Open Source. Retrieved from https://huggingface.co/blog/synthetic-data-save-costs

[139] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 35, 21314–21328.

[140] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. BLOOM: A 176B-parameter open-access multilingual language model. arXiv:2211.05100. Retrieved from https://arxiv.org/abs/2211.05100

[141] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Lu, Thomas Mesnard, Colton Bishop, Victor Carbune, and Abhinav Rastogi. 2023. RLAIF: Scaling reinforcement learning from human feedback with AI feedback. arXiv:2309.00267. Retrieved from https://arxiv.org/abs/2309.00267

[142] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. arXiv:2104.08691. Retrieved from https://arxiv.org/abs/2104.08691

[143] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 33, 9459–9474.

[144] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2023. Towards enhancing in-context learning for code generation. arXiv:2303.17780. Retrieved from https://arxiv.org/abs/2303.17780

[145] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of Android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.

[146] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: May the source be with you! arXiv:2305.06161. Retrieved from https://arxiv.org/abs/2305.06161

[147] Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. AlpacaEval: An Automatic Evaluator of Instruction-Following Models. Retrieved from https://github.com/tatsu-lab/alpaca_eval

[148] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. arXiv:2101.00190. Retrieved from https://arxiv.org/abs/2101.00190

[149] Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023. Textbooks are all you need II: Phi-1.5 technical report. arXiv:2309.05463. Retrieved from https://arxiv.org/abs/2309.05463

[150] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.

[151] Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *Proceedings of the 44th International Conference on Software Engineering*, 2253–2265.

[152] Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. 2023. Scaling down to scale up: A guide to parameter-efficient fine-tuning. arXiv:2303.15647. Retrieved from https://arxiv.org/abs/2303.15647

[153] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. arXiv:2211.09110. Retrieved from https://arxiv.org/abs/2211.09110

[154] Andreas Liesenfeld, Alianda Lopez, and Mark Dingemanse. 2023. Opening up ChatGPT: Tracking openness, transparency, and accountability in instruction-tuned text generators. In *Proceedings of the 5th International Conference on Conversational User Interfaces*, 1–6.

[155] Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*. . Association for Computational Linguistics, Barcelona, Spain, 74–81.

[156] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. 2022. A survey of transformers. *AI Open* 3 (2022), 111–132.

[157] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. 2007. Mining internet-scale software repositories. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 20.

[158] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model. arXiv:2405.04434. Retrieved from https://arxiv.org/abs/2405.04434

[159] Bingchang Liu, Chaoyu Chen, Cong Liao, Zi Gong, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, et al. 2023. MFTCoder: Boosting code LLMs with multitask fine-tuning. arXiv:2311.02303. Retrieved from https://arxiv.org/abs/2311.02303

[160] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A. Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 35, 1950–1965.

[161] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[162] Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. RLTF: Reinforcement learning from unit test feedback. arXiv:2307.04349. Retrieved from https://arxiv.org/abs/2307.04349

[163] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys* 55, 9 (2023), 1–35.

[164] Ruibo Liu, Jerry Wei, Fangyu Liu, Chenglei Si, Yanzhe Zhang, Jinmeng Rao, Steven Zheng, Daiyi Peng, Diyi Yang, Denny Zhou, et al. 2024. Best practices and lessons learned on synthetic data for language models. arXiv:2404.07503. Retrieved from https://arxiv.org/abs/2404.07503

[165] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid GNN. In *Proceedings of the International Conference on Learning Representations*.

[166] Yan Liu, Xiaokang Chen, Yan Gao, Zhe Su, Fengji Zhang, Daoguang Zan, Jian-Guang Lou, Pin-Yu Chen, and Tsung-Yi Ho. 2023. Uncovering and quantifying social biases in code generation. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36, 2368–2380.

[167] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep learning for android malware defenses: A systematic literature review. *ACM Computing Surveys* 55, 8 (2022), 1–36.

[168] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The next generation. arXiv:2402.19173. Retrieved from https://arxiv.org/abs/2402.19173

[169] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 6227–6240.

[170] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664. Retrieved from https://arxiv.org/abs/2102.04664

[171] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering code large language models with Evol-Instruct. In *Proceedings of the 12th International Conference on Learning Representations*.

[172] Michael R. Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. 2024. Automatic programming: Large language models and beyond. arXiv:2405.02213. Retrieved from https://arxiv.org/abs/2405.02213

[173] Wei Ma, Mengjie Zhao, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jie Zhang, Wenhan Wang, and Yang Liu. 2022. Are code pre-trained models powerful to learn code syntax and semantics? arXiv:2212.10017. Retrieved from https://arxiv.org/abs/2212.10017

[174] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[175] James Manyika and Sissie Hsiao. 2023. An overview of Bard: An early experiment with generative AI. *Google Static Documents* 2 (2023).

[176] Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. 2019. STYLE-ANALYZER: Fixing code style inconsistencies with interpretable unsupervised algorithms. In *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 468–478.

[177] Yu Meng, Jiaxin Huang, Yu Zhang, and Jiawei Han. 2022. Generating training data with language models: Towards zero-shot language understanding. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 35, 462–477.

[178] Meta. 2024. Introducing Meta Llama 3: The Most Capable Openly Available LLM to Date. Retrieved from https://ai.meta.com/blog/meta-llama-3/

[179] MistralAI. 2024. Codestral. Retrieved from https://mistral.ai/news/codestral/

[180] Sébastien BubeckMojan Javaheripi. 2023. Phi-2: The Surprising Power of Small Language Models. Retrieved from https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models

[181] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A tree-based convolutional neural network for programming language processing. arXiv:1409.5718. Retrieved from https://arxiv.org/abs/1409.5718

[182] Zahra Mousavi, Chadni Islam, Kristen Moore, Alsharif Abuadbba, and M. Ali Babar. 2024. An investigation into misuse of Java security APIs by large language models. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 1299–1315.

[183] Spyridon Mouselinos, Mateusz Malinowski, and Henryk Michalewski. 2022. A simple, yet effective approach to finding biases in code generation. arXiv:2211.00609. Retrieved from https://arxiv.org/abs/2211.00609

[184] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. arXiv:2210.14306. Retrieved from https://arxiv.org/abs/2210.14306

[185] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. OctoPack: Instruction tuning code large language models. arXiv:2308.07124. Retrieved from https://arxiv.org/abs/2308.07124

[186] King Han Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. arXiv:2403.07974. Retrieved from https://arxiv.org/abs/2403.07974

[187] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE, 692–708.

[188] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. LEVER: Learning to verify language-to-code generation with execution. In *Proceedings of the International Conference on Machine Learning*. PMLR, 26106–26128.

[189] Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, et al. 2023. L2CEval: Evaluating language-to-code generation capabilities of large language models. arXiv:2309.17446. Retrieved from https://arxiv.org/abs/2309.17446

[190] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for training LLMs on programming and natural languages. arXiv:2305.02309. Retrieved from https://arxiv.org/abs/2305.02309

[191] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An open large language model for code with multi-turn program synthesis. arXiv:2203.13474. Retrieved from https://arxiv.org/abs/2203.13474

[192] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep learning meets software engineering: A survey on pre-trained models of source code. arXiv:2205.11739. Retrieved from https://arxiv.org/abs/2205.11739

[193] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *Proceedings of the 12th International Conference on Learning Representations*.

[194] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. Retrieved from https://openai.com/blog/chatgpt

[195] OpenAI. 2024. Hello GPT-4o. Retrieved from https://openai.com/index/hello-gpt-4o/

[196] OpenAI. 2024. New Models and Developer Products Announced at DevDay. Retrieved from https://openai.com/index/new-models-and-developer-products-announced-at-devday/

[197] OpenDevin. 2024. OpenDevin: Code Less, Make More. Retrieved from https://github.com/OpenDevin/OpenDevin

[198] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 35, 27730–27744.

[199] Oded Ovadia, Menachem Brief, Moshik Mishaeli, and Oren Elisha. 2023. Fine-tuning or retrieval? Comparing knowledge injection in LLMs. arXiv:2312.05934. Retrieved from https://arxiv.org/abs/2312.05934

[200] David N. Palacio, Alejandro Velasco, Daniel Rodriguez-Cardenas, Kevin Moran, and Denys Poshyvanyk. 2023. Evaluating and explaining large language models for code using syntactic structures. arXiv:2308.03873. Retrieved from https://arxiv.org/abs/2308.03873

[201] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 311–318.

[202] Nikhil Parasaram, Huijie Yan, Boyu Yang, Zineb Flahy, Abriele Qudsi, Damian Ziaber, Earl Barr, and Sergey Mechtaev. 2024. The fact selection problem in LLM-based program repair. arXiv:2404.05520. Retrieved from https://arxiv.org/abs/2404.05520

[203] Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, 2719–2734.

[204] Arkil Patel, Siva Reddy, Dzmitry Bahdanau, and Pradeep Dasigi. 2023. Evaluating in-context learning of libraries for code generation. arXiv:2311.09635. Retrieved from https://arxiv.org/abs/2311.09635

[205] Indraneil Paul, Jun Luo, Goran Glavaš, and Iryna Gurevych. 2024. IRCoder: Intermediate representations make language models robust multilingual code generators. arXiv:2403.03894. Retrieved from https://arxiv.org/abs/2403.03894

[206] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program comprehension and code complexity metrics: An FMRI study. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 524–536.

[207] Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, Ashish Datta, Maksym Zhuravinskyi, Dakota Mahan, Marco Bellagente, Carlos Riquelme, et al. 2024. Stable code technical report. arXiv:2404.01226. Retrieved from https://arxiv.org/abs/2404.01226

[208] Ofir Press, Noah A. Smith, and Mike Lewis. 2021. Train short, test long: Attention with linear biases enables input length extrapolation. arXiv:2108.12409. Retrieved from https://arxiv.org/abs/2108.12409

[209] Xiangyu Qi, Yi Zeng, Tinghao Xie, Pin-Yu Chen, Ruoxi Jia, Prateek Mittal, and Peter Henderson. 2023. Fine-tuning aligned language models compromises safety, even when users do not intend to! arXiv:2310.03693. Retrieved from https://arxiv.org/abs/2310.03693

[210] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training (2018). Retrieved from https://www.semanticscholar.org/paper/Improving-Language-Understanding-by-Generative-Radford-Narasimhan/cd18800a0fe0b668a1cc19f2ec95b5003d0a5035

[211] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.

[212] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2012. Measuring software library stability through historical version analysis. In *Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 378–387.

[213] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[214] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[215] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-SQL capabilities of large language models. arXiv:2204.00498. Retrieved from https://arxiv.org/abs/2204.00498

[216] Aurora Ramirez, Jose Raul Romero, and Christopher L. Simons. 2018. A systematic review of interaction in search-based software engineering. *IEEE Transactions on Software Engineering* 45, 8 (2018), 760–781.

[217] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv:2403.05530. Retrieved from https://arxiv.org/abs/2403.05530

[218] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: A method for automatic evaluation of code synthesis. arXiv:2009.10297. Retrieved from https://arxiv.org/abs/2009.10297

[219] Replit. 2016. Idea to Software, Fast. Retrieved from https://replit.com

[220] Replit. 2023. replit-code-v1-3b. Retrieved from https://huggingface.co/replit/replit-code-v1-3b

[221] Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with AlphaCodium: From prompt engineering to flow engineering. arXiv:2401.08500. Retrieved from https://arxiv.org/abs/2401.08500

[222] Nick Roshdieh. 2023. Evol-Instruct-Code-80k. Retrieved from https://huggingface.co/datasets/nickrosh/Evol-Instruct-Code-80k-v1

[223] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*, 491–514.

[224] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open foundation models for code. arXiv:2308.12950. Retrieved from https://arxiv.org/abs/2308.12950

[225] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From words to Watts: Benchmarking the energy costs of large language model inference. In *Proceedings of the 2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[226] Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. 2022. Multitask prompted training enables zero-shot task generalization. In *Proceedings of the 10th International Conference on Learning Representations (ICLR '22)*.

[227] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. arXiv:1707.06347. Retrieved from https://arxiv.org/abs/1707.06347

[228] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*, 1559–1575.

[229] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. arXiv:1803.02155. Retrieved from https://arxiv.org/abs/1803.02155

[230] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. arXiv:1701.06538. Retrieved from https://arxiv.org/abs/1701.06538

[231] Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. PanGu-Coder2: Boosting large language models for code with ranking feedback. arXiv:2307.14936. Retrieved from https://arxiv.org/abs/2307.14936

[232] Jieke Shi, Zhou Yang, Hong Jin Kang, Bowen Xu, Junda He, and David Lo. 2024. Greening large language models of code. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*, 142–153.

[233] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[234] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring programs using large language models with few-shot examples. arXiv:2311.11690. Retrieved from https://arxiv.org/abs/2311.11690

[235] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning. arXiv:2301.13816. Retrieved from https://arxiv.org/abs/2301.13816

[236] Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. RepoFusion: Training code models to understand your repository. arXiv:2306.10998. Retrieved from https://arxiv.org/abs/2306.10998
[237] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *Proceedings of the International Conference on Machine Learning*. PMLR, 31693–31715.
[238] Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. 2023. CodeFusion: A pre-trained diffusion model for code generation. arXiv:2310.17680. Retrieved from https://arxiv.org/abs/2310.17680
[239] Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. ARKS: Active retrieval in knowledge soup for code generation. arXiv:2402.12317. Retrieved from https://arxiv.org/abs/2402.12317
[240] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024), 127063.
[241] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1433–1443.
[242] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. In *Proceedings of the 11th International Conference on Learning Representations (ICLR).*
[243] TabNine. 2018. AI Code Completions. Retrieved from https://github.com/codota/TabNine
[244] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-Following LLaMA Model. Retrieved from https://github.com/tatsu-lab/stanford_alpaca
[245] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on Gemini research and technology. arXiv:2403.08295. Retrieved from https://arxiv.org/abs/2403.08295
[246] Qwen Team. 2024. Code with CodeQwen1.5. Retrieved from https://qwenlm.github.io/blog/codeqwen1.5
[247] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking large language models for automated Verilog RTL code generation. In *Proceedings of the 2023 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 1–6.
[248] theblackcat102. 2023. The Evolved Code Alpaca Dataset. Retrieved from https://huggingface.co/datasets/theblackcat102/evol-codealpaca-v1
[249] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. arXiv:2302.13971. Retrieved from https://arxiv.org/abs/2302.13971
[250] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288. Retrieved from https://arxiv.org/abs/2307.09288
[251] Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. 2022. *Natural Language Processing with Transformers*. O'Reilly Media, Inc.
[252] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Proceedings of the International CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 1–7.
[253] Boris Van Breugel, Zhaozhi Qian, and Mihaela Van Der Schaar. 2023. Synthetic data, real errors: How (not) to publish and use synthetic data. In *Proceedings of the International Conference on Machine Learning. PMLR*, 34793–34808.
[254] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 30.
[255] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. Retrieved from https://github.com/kingoflolz/mesh-transformer-jax
[256] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024. Teaching code LLMs to use autocompletion tools in repository-level code generation. arXiv:2401.06391. Retrieved from https://arxiv.org/abs/2401.06391
[257] Fan Wang, Juyong Jiang, Chansung Park, Sunghun Kim, and Jing Tang. 2024. KaSA: Knowledge-aware singular-value adaptation of large language models. arXiv:2412.06071. Retrieved from https://arxiv.org/abs/2412.06071
[258] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936.
[259] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 1–26.

[260] Simin Wang, Liguo Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng. 2022. Machine/deep learning for software engineering: A systematic literature review. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1188–1231.

[261] Shiqi Wang, Li Zheng, Haifeng Qian, Chenghao Yang, Zijian Wang, Varun Kumar, Mingyue Shang, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2022. ReCode: Robustness evaluation of code generation models. arXiv:2212.10264. Retrieved from https://arxiv.org/abs/2212.10264

[262] Song Wang, Yaochen Zhu, Haochen Liu, Zaiyi Zheng, Chen Chen, and Jundong Li. 2023. Knowledge editing for large language models: A survey. arXiv:2310.16218. Retrieved from https://arxiv.org/abs/2310.16218

[263] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better LLM agents. arXiv:2402.01030. Retrieved from https://arxiv.org/abs/2402.01030

[264] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable neural code generation with compiler feedback. In *Findings of the Association for Computational Linguistics: ACL 2022*, 9–19.

[265] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. arXiv:2203.11171. Retrieved from https://arxiv.org/abs/2203.11171

[266] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*.

[267] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 1069–1088.

[268] Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35, 14015–14023.

[269] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 8696–8708.

[270] Yufei Wang, Wanjun Zhong, Liangyou Li, Fei Mi, Xingshan Zeng, Wenyong Huang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Aligning large language models with human: A survey. arXiv:2307.12966. Retrieved from https://arxiv.org/abs/2307.12966

[271] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation. arXiv:2212.10481. Retrieved from https://arxiv.org/abs/2212.10481

[272] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2021. Finetuned language models are zero-shot learners. arXiv:2109.01652. Retrieved from https://arxiv.org/abs/2109.01652

[273] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. arXiv:2206.07682. Retrieved from https://arxiv.org/abs/2206.07682

[274] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 35, 24824–24837.

[275] Xiaokai Wei, Sujan Kumar Gonugondla, Shiqi Wang, Wasi Ahmad, Baishakhi Ray, Haifeng Qian, Xiaopeng Li, Varun Kumar, Zijian Wang, Yuchen Tian, et al. 2023. Towards greener yet powerful code generation via quantization: An empirical study. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 224–236.

[276] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. arXiv:2312.02120. Retrieved from https://arxiv.org/abs/2312.02120

[277] Lilian Weng. 2023. LLM-Powered Autonomous Agents. Retrieved June 2023 from https://lilianweng.github.io/posts/2023-06-23-agent/

[278] Alexander Wettig, Aatmik Gupta, Saumya Malik, and Danqi Chen. 2024. QuRating: Selecting high-quality data for training language models. arXiv:2402.09739. Retrieved from https://arxiv.org/abs/2402.09739

[279] Erroll Wood, Tadas Baltrušaitis, Charlie Hewitt, Sebastian Dziadzio, Thomas J. Cashman, and Jamie Shotton. 2021. Fake it till you make it: Face analysis in the wild using synthetic data alone. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 3681–3691.

[280] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective retrieval for repository-level code completion. arXiv:2403.10059. Retrieved from https://arxiv.org/abs/2403.10059

[281] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework. arXiv:2308.08155. Retrieved from https://arxiv.org/abs/2308.08155

[282] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. arXiv:2309.07864. Retrieved from https://arxiv.org/abs/2309.07864

[283] Rui Xie, Zhengran Zeng, Zhuohao Yu, Chang Gao, Shikun Zhang, and Wei Ye. 2024. CodeShell technical report. arXiv:2403.15747. Retrieved from https://arxiv.org/abs/2403.15747

[284] Sang Michael Xie, Shibani Santurkar, Tengyu Ma, and Percy S. Liang. 2023. Data selection for language models via importance resampling. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36, 34201–34227.

[285] Xingyao Wang, Bowen Li, and Graham Neubig. 2024. Introducing OpenDevin CodeAct 1.0, a New State-of-the-Art in Coding Agents. Retrieved from https://www.cognition.ai/introducing-devin

[286] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 416–426.

[287] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. WizardLM: Empowering large language models to follow complex instructions. arXiv:2304.12244. Retrieved from https://arxiv.org/abs/2304.12244

[288] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10.

[289] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2024. Aligning LLMs for FL-free program repair. arXiv:2404.08877. Retrieved from https://arxiv.org/abs/2404.08877

[290] Weiwei Xu, Kai Gao, Hao He, and Minghui Zhou. 2024. A first look at license compliance capability of LLMs in code generation. arXiv:2408.02487. Retrieved from https://arxiv.org/abs/2408.02487

[291] Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. 2024. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. arXiv:2403.07506. Retrieved from https://arxiv.org/abs/2403.07506

[292] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, Donggyun Han, and David Lo. 2024. Unveiling memorization in code models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[293] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[294] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[295] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, 476–486.

[296] Kang Min Yoo, Jaegeun Han, Sookyo In, Heewon Jeon, Jisu Jeong, Jaewook Kang, Hyunwook Kim, Kyung-Min Kim, Munhyong Kim, Sungju Kim, et al. 2024. HyperCLOVA X technical report. arXiv:2404.01954. Retrieved from https://arxiv.org/abs/2404.01954

[297] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 1–12.

[298] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 3911–3921.

[299] Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. WaveCoder: Widespread and versatile enhanced instruction tuning with refined data generation. arXiv:2312.14187. Retrieved from https://arxiv.org/abs/2312.14187

[300] Youliang Yuan, Wenxiang Jiao, Wenxuan Wang, Jen-tse Huang, Pinjia He, Shuming Shi, and Zhaopeng Tu. 2023. GPT-4 is too smart to be safe: Stealthy chat with LLMs via cipher. arXiv:2308.06463. Retrieved from https://arxiv.org/abs/2308.06463

[301] Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, Songfang Huang, and Fei Huang. 2023. RRHF: Rank responses to align language models with human feedback without tears. arXiv:2304.05302. Retrieved from https://arxiv.org/abs/2304.05302

[302] Jiawei LiuYifeng DingNaman JainHarm de VriesLeandro von WerraArjun GuhaLingming ZhangYuxiang Wei, Federico Cassano. 2024. StarCoder2-Instruct: Fully Transparent and Permissive Self-Alignment for Code Generation. Retrieved from https://github.com/bigcode-project/starcoder2-self-align

[303] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. 2021. BitFit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. arXiv:2106.10199. Retrieved from https://arxiv.org/abs/2106.10199

[304] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual pre-training on sketches for library-oriented code generation. arXiv:2206.06888. Retrieved from https://arxiv.org/abs/2206.06888

[305] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7443–7464.

[306] Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Wei Li, Yafen Yao, Yongshun Gong, Xiaolin Chen, Bei Guan, et al. 2024. CodeS: Natural language to code repository via multi-layer sketch. arXiv:2403.16443. Retrieved from https://arxiv.org/abs/2403.16443

[307] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2471–2484.

[308] Jialu Zhang, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2024. PyDex: Repairing bugs in introductory Python assignments using LLMs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1100–1124.

[309] Jialu Zhang, De Li, John Charles Kolesar, Hanyuan Shi, and Ruzica Piskac. 2022. Automated feedback generation for competition-level code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–13.

[310] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. 2023. Adaptive budget allocation for parameter-efficient fine-tuning. In *Proceedings of the 11th International Conference on Learning Representations*.

[311] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, et al. 2023. Instruction tuning for large language models: A survey. arXiv:2308.10792. Retrieved from https://arxiv.org/abs/2308.10792

[312] Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. 2024. NaturalCodeBench: Examining coding performance mismatch on HumanEval and natural user prompts. arXiv:2405.04520. Retrieved from https://arxiv.org/abs/2405.04520

[313] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the AI ocean: A survey on hallucination in large language models. arXiv:2309.01219. Retrieved from https://arxiv.org/abs/2309.01219

[314] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous program improvement. arXiv:2404.05427. Retrieved from https://arxiv.org/abs/2404.05427

[315] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of NLP and software engineering: A survey on language models for code. arXiv:2311.07989. Retrieved from https://arxiv.org/abs/2311.07989

[316] Liang Zhao, Xiaocheng Feng, Xiachong Feng, Bin Qin, and Ting Liu. 2023. Length extrapolation of transformers: A survey from the perspective of position encoding. arXiv:2312.17044. Retrieved from https://arxiv.org/abs/2312.17044

[317] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, Jie Jiang, and Bin Cui. 2024. Retrieval-augmented generation for AI-generated content: A survey. arXiv:2402.19473. Retrieved from https://arxiv.org/abs/2402.19473

[318] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. arXiv:2303.18223. Retrieved from https://arxiv.org/abs/2303.18223

[319] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2024. Judging LLM-as-a-judge with MT-bench and Chatbot Arena. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36

[320] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 5673–5684.

[321] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating code generation with execution and refinement. arXiv:2402.14658. Retrieved from https://arxiv.org/abs/2402.14658

[322] Wenqing Zheng, S. P. Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. 2023. Outline, then details: Syntactically guided coarse-to-fine code generation. In *Proceedings of the International Conference on Machine Learning*. PMLR, 42403–42419.

[323] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. arXiv:2311.10372. Retrieved from https://arxiv.org/abs/2311.10372

[324] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. LDB: A large language model debugger via verifying runtime execution step-by-step. arXiv:2402.16906. Retrieved from https://arxiv.org/abs/2402.16906

[325] Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. 2023. Solving challenging math word problems using GPT-4 code interpreter with code-based self-verification. arXiv:2308.07921. Retrieved from https://arxiv.org/abs/2308.07921

[326] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. arXiv:2310.04406. Retrieved from https://arxiv.org/abs/2310.04406

[327] Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2024. LIMA: Less is more for alignment. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[328] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. In *Proceedings of the 11th International Conference on Learning Representations*.

[329] Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. 2022. DocPrompting: Generating code by retrieving the docs. In *Proceedings of the 11th International Conference on Learning Representations*.

[330] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence. arXiv:2406.11931. Retrieved from https://arxiv.org/abs/2406.11931

[331] Terry Yue Zhuo. 2024. ICE-Score: Instructing large language models to evaluate code. In *Findings of the Association for Computational Linguistics: EACL 2024*, 2232–2242.

[332] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. arXiv:2406.15877. Retrieved from https://arxiv.org/abs/2406.15877

[333] Terry Yue Zhuo, Armel Zebaze, Nitchakarn Suppattarachai, Leandro von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. 2024. Astraios: Parameter-efficient instruction tuning code large language models. arXiv:2401.00788. Retrieved from https://arxiv.org/abs/2401.00788

Received 20 June 2024; revised 30 March 2025; accepted 16 June 2025