A Survey of Neural Code Intelligence: Paradigms, Advances and Beyond

Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, Qipeng Guo, Xipeng Qiu, Pengcheng Yin, Xiaoli Li, Fellow, IEEE, Fei Yuan, Lingpeng Kong, Xiang Li, and Zhiyong Wu

Abstract—Neural Code Intelligence — leveraging deep learning to understand, generate, and optimize code — holds immense potential for transformative impacts on the whole society. Bridging the gap between Natural Language and Programming Language, this domain has drawn significant attention from researchers in both research communities over the past few years. This survey presents a systematic and chronological review of the advancements in code intelligence, encompassing over 50 representative models and their variants, more than 20 categories of tasks, and an extensive coverage of over 680 related works. We follow the historical progression to trace the paradigm shifts across different research phases (e.g., from modeling code with recurrent neural networks to the era of Large Language Models). Concurrently, we highlight the major technical transitions in models, tasks, and evaluations spanning through different stages. For applications, we also observe a co-evolving shift. It spans from initial endeavors to tackling specific scenarios, through exploring a diverse array of tasks during its rapid expansion, to currently focusing on tackling increasingly complex and varied real-world challenges. Building on our examination of the developmental trajectories, we further investigate the emerging synergies between code intelligence and broader machine intelligence, uncovering new cross-domain opportunities and illustrating the substantial influence of code intelligence across various domains. Finally, we delve into both the opportunities and challenges associated with this field, alongside elucidating our insights on the most promising research directions. An ongoing, dynamically updated project and resources associated with this survey have been released at https://github.com/QiushiSun/NCISurvey.

Index Terms—Code Intelligence; Natural Language Processing; Language Models; Software Engineering

1 INTRODUCTION

Code is one of the elegant languages created by humans, which replaces the diverse forms of natural language (NL) through a high degree of abstraction [1]. As a conduit between humans and machines, it is ultimately transformed into specific programs that substitute human effort in accomplishing various tasks, characterized by advantages such as precision, logic, modularity, and excitability. The fusion of rapidly advancing deep learning techniques with the availability of “Big Code” [2, 3] has led to the emergence of neural code intelligence. This domain, which applies neural approaches to understand, generate, and manipulate code, has garnered significant attention from the research community. Figure 1 illustrates the cumulative publication statistics over recent years, showcasing the growing interest and significant efforts being dedicated to this area. Notably, this domain transcends disciplinary boundaries, spanning Natural Language Processing (NLP) [4], Software Engineering (SE) [5], Robotics [6], and beyond. Moreover, the unique duality of code, which combines human-readable semantics with executability, establishes research in this area as a critical bridge between artificial intelligence and the real world, laying a corner-

1. We use code and program interchangeably in this paper.

2. The statistics are derived by querying a set of specific keywords (e.g., code representation, code generation, code intelligence) through an exact match search in the titles or abstracts of documents.
At the heart of these studies lies the foundational concept of “Software Naturalness” [8], which posits that programming language (PL), much like human languages, is characterized by predictable and repetitive patterns that can be effectively modeled. From a macroscopic perspective, the progression of techniques for processing these artificial languages has largely mirrored the evolution observed in NLP [9].

Specifically, after a brief initial period characterized by statistical [10, 11, 12] and probabilistic [13] modeling, the learning paradigms of neural code intelligence has transitioned from the earliest word embedding techniques to Large Language Models (LLMs), broadly categorized into three main phases of evolution:

- **Neural Language Models for Code.** This era witnessed the early yet fruitful efforts of applying deep learning to process code. The methods designed during this period primarily relied on well-developed recurrent [14] or convolutional [15] structures to model code. Notably, they not only leverage the textual information of the code but also intricately incorporate structural information extracted from code structures like Abstract Syntax Trees (ASTs) into the modeling process [16, 17], aligned closely with the principles of neural semantic parsing [18, 19, 20].

  Meanwhile, as code snippets can be represented as continuous vectors, the techniques evolved during this period were also known as code embeddings [21]. The most representative techniques, code2vec [22] and code2seq [23], captured the semantic and structural information of code by embedding paths from ASTs into a vector space, enabling the application of neural methods to diverse scenarios.

- **Code Pre-trained Models (CodePTMs).** Pre-trained language models [24, 25] with multi-layer Transformer architecture [26] have established a “pre-train” and “fine-tune” learning paradigm [27]. Following this, various studies have emerged on how to build CodePTMs. These models, exemplified by CodeBERT [28], CodeT5 [29], and PLBART [30], have long dominated the mainstream approaches in code intelligence and initiated a surge of research interest – a sharp increase in arXiv papers can be observed in Figure 1 after the release of CodeBERT in 2020. During the pre-training phase, they learn general-purpose context-aware representations from massive GitHub code data, as well as their structural information of code (e.g., ASTs). Subsequently, they are fine-tuned on task-specific code data or NL-PL pairs, significantly raising the performance of various code-related tasks. This approach marks a shift from previous learning paradigms, where CodePTMs no longer require individual modeling for each task, but adapt to different scenarios by fine-tuning on relatively smaller labeled datasets.

- **Large Language Models for Code (CodeLLMs).** Recent studies have indicated that scaling language models through increasing their parameters or the volume of training data [31, 32] consistently results in an enhancement of the model’s capacity to perform effectively on downstream tasks. Following the success of general LLMs such as GPT-3 [33] and PaLM [34] in both academia and industry [35], models like Codex [36] have sparked a new wave of research in code intelligence – a notable increase of related papers can be observed in Figure 1 after the debut of ChatGPT in late 2022. This phase has also seen a shift in the learning paradigm from task-specific fine-tuning to prompting [37] and in-context learning [38], as well as expanding the application of code intelligence from solely code-related tasks to a broader array of real-world scenarios.

Aware that the development of code intelligence is significantly reflected in the evolution of language models designed for code, we present in Figure 2 a chronological summary of representative works that outline the developmental trajectory of neural code intelligence. It provides a framework for this paper by outlining an overview of the technical advances within the domain. Building upon this timeline, we embrace a new perspective characterized by the paradigm shifts in models, applications, evaluations, and beyond, to explore these exciting advancements in code intelligence. We thoroughly investigate the literature and distill the key findings, techniques, and interconnections between research across various epochs. Moreover, we broaden our scope to explore the integration of other domains with code intelligence, discussing how code generation assists in machine reasoning, how code training enhances models’ mathematics capabilities, and how code serves as a medium to provide new approaches for solving typical NLP tasks. Furthermore, we explore a wide array

---

3. https://openai.com/blog/chatgpt
of real-world cross-domain applications, spanning coding assistants, data science, autonomous agents, and AI4science. A GitHub project associated with this survey is actively maintained at https://github.com/QiushiSun/NCISurvey, which contains all the resources used to construct this paper, as well as comprehensive reading lists to assist readers in further exploration. We hope that this will facilitate the continued development of the field.

The following parts of this survey are organized as follows. Section 2 begins by reviewing the preliminaries. We then examine classic yet crucial methods for processing code using neural language models, as well as an introduction to quintessential code-related tasks. Section 3 delves into the evolution of code intelligence within the pre-train and fine-tune paradigm, offering a comprehensive review and discussion of the techniques of this era and their implications for future research. Section 4 explores the research advancements in the era of LLMs, alongside a review of progress in NL2Code, and conducts thorough discussions of representative models and benchmarks. Section 5 investigates the synergies between code intelligence and other domains of machine intelligence. Section 6 is on the application of code intelligence in real-world scenarios, demonstrating its practical utility. Section 7 engages in a dual-faceted discussion of open issues, both from the perspectives of model architecture and practical application, and shares what we believe are worthy of future research directions. Finally, Section 8 summarizes the insights and findings of the paper.

2 The Spark of Code Intelligence

In recent years, the vast availability of source code from public repositories has significantly boosted the application of deep learning techniques to source codes [3]. Under the concept of “software naturalness” [8], neural language models designed for processing text can be naturally applied to code. Combined with the evolving demands of software engineering for automated code processing, neural code intelligence has experienced a prosperous development.

By conceptualizing code snippets as language sequences, sequential neural architectures, such as LSTM [14], are naturally adaptable for code understanding and generation [39, 40]. However, it is imperative to recognize that, unlike natural language sentences, programs contain explicit and complicated structures, which introduces more opportunities and possibilities for modeling code. Subsequently, code embeddings have emerged, which can be defined as numerical sequences representing the inherent concepts found in codes. This development has had a pioneering and long-lasting impact on future research in code intelligence.

In this section, we first provide readers with preliminaries regarding code structure, followed by a review of some classic methods of modeling based on it. Subsequently, we introduce a series of classic, significant, and continuously explored code-related tasks.

2.1 Code Features Through Structural Views

Viewing source code merely as a token sequence overlooks their inherent structures, a characteristics can greatly enhance the model’s ability to comprehend code. To illustrate this, let’s consider a straightforward example. Consider the expression $s = \text{min\_value} + \text{max\_value}$, $s$ is evidently derived from the maximum and minimum values. However, since programmers do not always adhere to naming conventions, it is challenging for models to grasp the semantics of $s$ solely from its variable name. Nevertheless, by leveraging the dependency relation between variables, it becomes possible to facilitate the comprehension of the semantics of $s$ and to predict program properties [41]. Structural information represented by such dependency relations plays a crucial role in modeling code [42, 43]. Therefore, in this part, we will briefly cover three typical carriers of structural information, providing readers with some background knowledge.

- **Abstract Syntax Tree.** AST stands as a quintessential intermediate representation during code compilation, where a program is parsed into a tree structure of operations and their operands. Serving as a syntactic-level structure, it encapsulates both the syntax and structural information of a program, while its components also embody distinct semantics [44]. It can be obtained by applying parsers (e.g., Tree-sitter⁴, pycparser⁵ and javalang⁶) on source codes.
- **Data flow.** Unlike syntactic-level code structures like AST, Data flow (Graph) represents a semantic-level structure within code. Its nodes represent variables, and the edges reflect the relationships and origins among these variables. It can be extracted from AST and characterized by reduced complexity and does not entail a deep hierarchy, resulting in relatively lower costs for modeling and analysis [45].
- **Control flow.** Contrasting with data flow’s semantics, Control Flow (Graph) provides a structural view of code executive information. Here, nodes represent executable blocks, and edges indicate control transitions between them. This emphasizes the sequence and potential paths of program execution rather than variable interactions [46]. Control Flow is key for understanding program dynamics, and offering insights into program logic. It can be constructed through the use of static analyzers [47].

Building on the above code features, in processing codes with neural methods, one can consider not only the plain text of the source code but also leverage code structural information. This process can be typically divided into three main strategies: (1) Directly Encoding AST: A representative method is TBCNN [16], which utilizes tree-based convolution kernels on ASTs to capture information from subtrees. The features of subtrees will be aggregated through pooling to formulate the embedding of the program. Following this, subsequent work has increasingly integrated ASTs with convolution to capture local code features [48, 49]. (2) Utilizing AST Paths: This approach is exemplified by code2vec [22] and code2seq [23]. Code2vec integrates the representations of AST leaf nodes and aggregates their path representation to build combined context vectors. Following this idea, code2seq extracts more fine-grained information from the AST paths and leverages an LSTM to encode the entire path to suit generation tasks. (3) Transforming AST, represented by: AutoenCODE [50] converts ASTs into bi-

---

5. https://github.com/eliben/pycparser
nary trees and utilizes autoencoder to learn code embedding from it. GGNN [51] introduces additional edges to explicitly represent data/control flows and employs graph neural networks to learn nodes’ representations. To address long-term dependency issues, ASTNN [17] breaks each AST into a sequence of statement subtrees and encodes them into vectors by capturing both the lexical and syntactical knowledge of the statements. The approach has been further extended for industrial applications [52]. InferCode [53] employs self-supervised learning by exploiting the structural similarities within code to automatically generate labels for training.

Moreover, beyond utilizing these features in their original forms, researchers have adapted them to apply various deep learning approaches. Wang et al. [54], Wang and Li [55] initially augment AST with explicit control and data flow edges to facilitate the application of graph algorithms [56, 57, 58]. Later, issues related to the low connectivity of ASTs and the out-of-vocabulary problem [59] during modeling are identified [60]. To mitigate these issues, researchers connecting adjacent leaf nodes, which aids in the graph partitioning [61] and analysis [62].

2.2 Overview of Core Tasks in Code Intelligence

This part provides an overview of the most important tasks in code intelligence and the challenges they face, categorized based on the form of their inputs and outputs.

2.2.1 Code-Code Tasks

Code-code tasks refer to a series of tasks that involve operations on source code with the aim of understanding, generating, or transforming code.

Clone Detection. Clone detection is widely studied in SE research [63, 17, 54], which predicts whether two code snippets are clones of each other, and can be conceptualized as a binary sentence classification task. Code clones refer to pairs of code snippets that display notable similarities, occurring within or across different software systems [64]. Programmers often create clones by reusing code through copy and paste. While cloning can offer advantages, such as accelerated software development, it presents significant drawbacks. When buggy code is cloned, the bug is duplicated throughout the system, exacerbating the complexity of debugging and maintenance [65]. Furthermore, clones may introduce new bugs if updates to a code fragment are not uniformly applied to its clones. Such practices can adversely impact software by unnecessarily inflating the system’s size and consequently increasing the expenses related to reengineering [66]. Beyond finding suitable matching algorithms or metric [67], The primary challenge in developing automated approaches for clone detection lies in equipping the detector with the ability to fully comprehend syntactic [68, 69, 70] or semantic [71, 72, 73] similarities, thereby minimizing the risk of false positives [74].

Defect Detection. The incidence of source code defects and vulnerabilities has been increasing rapidly, as evidenced by public reports through CVE (Common Vulnerabilities & Exposures), as well as identified within proprietary code bases. This trend poses a crucial yet complex challenge in security. In response to this, defect (vulnerability) detection has emerged as a solution, aiming to liberate human programmers from the extensive demands of manual code inspection [75, 76, 77, 78, 79]. The task can be formalized as binary classification, i.e., learning to determine whether a given code snippet contains a defect or not. As a non-generative task, it shares similar challenges with clone detection, and further, the calibration of models [80, 81] plays a vital role in ensuring reliability as well.

Code Repair. Writing codes often involves errors, a common experience for programmers. Often, these errors are minor, necessitating only limited modifications to the original program. Such errors can interrupt the workflow of experienced developers and may pose significant challenges to beginners while localizing and rectifying them is known to be effort-prone and time-consuming. Code repair aims to refine the code by automatically localizing [82] and fixing these bugs, which can be modeled as a seq2seq task [83, 51, 84, 85, 86, 87]. By integrating code repair with defect detection, it becomes possible to streamline the processes of identifying issues and implementing fixes [88].

Code Completion. Code completion is one of the most common application scenarios for coding assistants like Copilot, and its usage is bifurcated into two subcategories: token-level completion and line-level completion. The former involves predicting a single code token, while the latter entails completing an entire, yet unfinished line. The objective of the task is to predict subsequent token(s) within a given code context [89, 90, 91, 92, 93]. It can also be viewed as a seq2seq task, but the target needs to be a continuation of the input. With the evolution of code intelligence, code completion has also begun to encompass infilling tasks [94, 95], which entails not only left-to-right completion but also filling in code before or in the middle of a given context.

Code Translation. Code translation, also known as transcription, involves translating a code snippet from one PL to another. It has many use cases, such as modernizing artifacts [96] implemented in PLs like COBOL or Python 2, and migrating legacy software in proprietary PLs to applications written in general-purpose PLs [97]. Over the past decades, the paradigm of code translation has undergone a significant transformation, shifting from labor-intensive rewriting methods to more efficient and reliable automated solutions. While it can also be described as a seq2seq task, it presents more difficulty compared to previous tasks. The greatest challenges include (1) the need to faithfully preserve the original functionality, and (2) the requirement to generate syntactically correct code without introducing bugs [98]. Existing research includes strategies that utilize annotated PL pairs and their syntactic structures for training [99, 100], as well as unsupervised methods that learn from monolingual source code without parallel data [101, 102, 103]. Additionally, the scope of this area also includes pseudocode-to-code translation [104, 105]. In comparison to NL machine translation, the functional correctness of all translated code is more critical than its similarity [106, 107] to the reference.

2.2.2 Code-Text Task

Code-text tasks refer to the challenge of generating natural language from source code.

7. https://github.com/features/copilot
Code Summarization. Code summarization represents a prominent task in the field of code intelligence, which entails generating concise and descriptive comments for codes, derived from analyzing its semantics [124]. It is vital for updating and maintaining software systems [125], particularly those with collaboration among multiple developers. Code summarization can be modeled in a seq2seq format, aiming to take a code snippet (and its structure) as input and produce an NL description [119, 23, 126, 127, 128] or the function/method’s name as output [129, 130, 131, 132]. Beyond directly synthesizing summaries, strategies include retrieving keywords from the source code [133, 134] or employing clone detection to find comments from similar code snippets [135]. Additionally, leveraging the API knowledge can further enhance the relevance of generated content [120].

Commit Message Generation. Developers may frequently edit their code for bug fixing, adding new features, etc. Version control systems like Git often track these edits, which utilizes commit to document the changes. When code is updated frequently, manually writing commit messages becomes a laborious task. Fortunately, code embeddings can also be employed to represent these edits [136]. Commit message generation is an emerging task aimed at automating the creation of commit messages for code changes. It involves taking two versions of the code, before and after the edits, as input and generating summaries that describe their differences [137, 138]. In practice, the methods employed for commit message generation span a range of techniques, including the use of predefined rules or templates [139], leveraging commit messages from similar code changes [140], employing seq2seq modeling [141, 142, 122], and incorporating retrieval-augmented approaches [143].

2.2.3 Text-Code Tasks

Text-code tasks involve finding or generating executable source code aligned with NL descriptions.

Code Retrieval. To enhance developers’ productivity, seeking ready-made solutions that closely match their requirements serves as a shortcut. The objective of Code retrieval, also known as NL code search is to identify and retrieve codes that are functionally relevant, in response to NL queries [113, 144, 145]. A common practice involves using specially designed metrics to measure the similarity between the contextual embeddings of the given query and the candidate code snippets [146, 147]. Additionally, there is a parallel task to code retrieval known as code search [148, 149], where the key difference lies in the query: here, the query is also a code snippet. This task can be viewed as searching for clones within a candidate pool, allowing developers to find code snippets that perform similar functions or have similar implementations based on code-based queries. The code obtained can serve as a reference for generating more complex yet related code [150].

Code Generation. Code generation, also known as program synthesis, broadly refers to the use of NL to generate code (NL2Code). This long-standing task aims to lower the barriers associated with coding, streamline some routine tasks through automation, and empower non-programmers to obtain solutions tailored to their intentions [151]. Beyond merely treating it as another form of text generation, early research primarily relied on the guidance of code syntax to generate small code snippets for specific scenarios, and view it as a semantic parsing task [152, 153, 154]. Over time, this focus gradually shifted towards general-purpose code generation [155, 156, 150, 157] for a single program-
Fig. 3: The trajectory of neural code intelligence's evolution is encapsulated through the development of language models for code. This is delineated by four principal branches, each representing a distinct category of models. The first branch showcases models based on code embedding techniques, while the subsequent three branches feature Transformer-based models, each exemplifying unique architectures: Encoder-only, Encoder-Decoder, and Decoder-only. Models on the same sub-branch have closer relationships. Additionally, the vertical axis chronicles the timeline of these models' release dates, paralleled by some seminal NLP models. The details of creating this figure are listed in Appendix A.2.

For the tasks mentioned above, we have compiled a list of representative benchmarks along with their brief descriptions in Table 1. Furthermore, we expand on this list in Table 6, 7 and 8, which include additional benchmarks and cover some tasks that are not detailed extensively, such as question answering [170], comments generation [171], log analytics research [172], document translation [114], and programming learning [173, 174, 175].

The neural modeling for the code-related tasks discussed above was primarily developed before 2020, as illustrated on the left branch of Figure 3. Although most models were designed ad hoc for specific tasks and may seem relatively simple by today's standards, they signified the rise of code intelligence and laid the foundation for subsequent research.

♢ Takeaway: (1) The application of neural networks to code marked a pioneering step, signifying the onset of neural code intelligence. (2) Compared to modeling NL, the incorporation of code structures represents a critical divergence. These code features can be extracted and encoded in diverse ways, a practice that subsequent research has continued to leverage. (3) A wide range of code-related tasks were introduced and formalized during this period, which subsequently experienced significant development. Beyond their practical value, they also became testbeds for future developments.
advancements. (4) Despite these intricately designed techniques becoming somewhat overshadowed in the subsequent transformer era, they remain lightweight and easy-to-use in practice, serving as an effective means to develop high-quality code representations.

3 AN ODYSSEY OF PRE-TRAIN AND FINE-TUNE

Following the remarkable success that pre-trained language models [27] have achieved in NLP, the code intelligence community rapidly integrated their architecture and learning paradigms, leading to the proliferation of CodePTMs. Coming after the era of neural language modeling, this marks a flourishing period for code intelligence which retains code structural insights while incorporating transformer-based models. The construction of language models for code has undergone a major paradigm shift, characterized by the following features:

1) Architecture: Multi-layer transformer [176] has become the de facto choice for model backbone, moving away from building models from scratch for each task or relying on tailored feature engineering.

2) Training Data: Pre-training is primarily conducted on large volumes of unlabeled data harvested from GitHub [177], with a smaller portion of labeled data typically used for adapting the model to various downstream tasks.

3) Learning objectives: While optimized through self-supervised objectives, the approach still retains the utilization of structural information to varying degrees to enable more effective learning of code representations.

The development of this stage is reflected in the three main branches shown in Figure 3. In this section, we will first conduct a systematic review of representative CodePTMs and their variants, followed by an in-depth discussion of their other aspects.

3.1 Pre-trained Language Models for Code

In this part, we discuss a wide range of CodePTMs, categorizing them based on their architectures.

3.1.1 Encoder-only

Existing CodePTMs with encoder-only architecture can be classified based on their use of structural information into two distinct categories: structure-free and structure-based. The former only utilizes raw code texts, whereas the latter incorporates code structure during pre-training to more effectively grasp the inherent structure of code.

- **Structure-free Models.** CuBERT [178] marks a pioneering endeavor in the integration of transformer architecture into the realm of code intelligence. It is trained on a corpus of Python data collected from GitHub, employing the same training objectives as BERT [24] and replicating its training pipeline. Another milestone is CodeBERT [28], which distinguishes itself from CuBERT by adopting a cross-modal training strategy that utilizes both bimodal NL-PL data and unimodal data. The pre-training of CodeBERT is centered around two objectives: Masked Language Modeling (MLM) and Replaced Token Detection (RTD) [179]. For implementation, it is initialized through RoBERTa [180] and trained on CodeSearchNet [113], a pioneering and influential corpus encompassing six PLs constructed by scraping open-source GitHub repositories.

Regardless of whether task-specific fine-tuning is applied, both models have achieved performance far surpassing previous word2vec models and multi-layered bidirectional LSTMs (discussed in Section 2) across a wide range of code-related tasks, paving the way for the successful application of the pre-train and fine-tune paradigm on code.

- **Structure-based Models.** After the success of CodePTMs that solely rely on code tokens for training, researchers revisit earlier strategies centered on code features, innovatively incorporating code structural information in other modalities (e.g., data flow) into the training process of transformer-based models. GraphCodeBERT [181] represents one of the earliest endeavors, which leverages data flow in the pre-training stage. Beyond MLM, it innovatively introduces two tasks: predicting code structure edges and aligning code with its structure. These tasks collectively aim to enable the model to understand the relationships between variables as well as between variables and tokens. Trained on CodeSearchNet, this structure-aware training approach allows GraphCodeBERT to outperform previous models (e.g., CodeBERT) on a range of code-related tasks, pioneeringly demonstrating the importance of structural information in code understanding.

Contrastive pre-training [182] emerges as another pathway. SynCoBERT [183] extends structure-aware training further by not only considering the structural information of code but also synchronizing the embeddings of code and its corresponding comments through contrastive learning, aiming to bridge the gap between code semantics and NL comments. Similarly employing contrastive objectives, CODE-MVP [184] explores multi-view learning of code. It processes different code structures, such as AST, data flow, and control flow in parallel. The contrastive objects come into play when it compares these multiple views of the same code snippet in training, thus identifying and reinforcing the commonalities and differences across these representations. DISCO [185] leverages code transformation algorithms to generate synthetic code clones and inject real-world security bugs, utilized respectively to construct positive and negative samples. This approach enables models to discern subtle differences in functionalities. Later, Li et al. [186] observe that positive samples created through transformation algorithms, such as variable renaming [187] or injecting non-functional code [188], could lead the model to prioritize learning superficial code structures over significant code semantics. To prevent the model from being misled by superficial content, SCoder further employs code comments and subtrees of ASTs to build positive samples, compelling the model to deeply understand code semantics and learn to infer code based on its context.

Additionally, in the context of training with program transformations, the concept of identifier deobfuscation in SE has also been employed. DOBF [189] objective begins by concealing the names of functions and variables using placeholder tokens, then trains the CodePTM to restore the original names through dictionary mapping.
3.1.2 Encoder-Decoder

In contrast with encoder-only models, which excel in code understanding, their encoder-decoder counterparts possess inherent advantages in the realm of controllable text or code generation. Yet, their evolution mirrors that of encoder-only models in significant ways. Initially, code was treated purely as text and applied to encoder-decoder transformers [190], followed by the integration of various structural information to enhance the learned code representations [191]. This evolution has gradually led to the development of three distinct categories of models equipped with classic architectures, which will be discussed as follows:

- **BART.** Ahmad et al. [30] propose PLBART. Following the training objectives of BART [192], it is pre-trained on a corpora constructed by Java and Python functions (from GitHub) and NL documents from StackOverflow via denoising autoencoding. PLBART is distinctive for its unified training on code and NL, aiming to learn the alignment between semantic spaces across different PLs.

- **T5.** The initial exploration of the T5 [200] architecture’s potential on source code is initiated by Mastropaolo et al. [201], who drew inspiration from the concept of multitask learning. This approach commenced with the presentation of a series of code-related tasks as text-to-text transformations. Similarly, PyMT5 [193] replicates this approach by leveraging Python methods and method-docstring data.

  CodeT5 [29] is the first structure-aware encoder-decoder model and is among the most influential models today. It follows the T5-learning [200] pipeline and, in addition to the original span corruption training objective, it incorporates: (1) identifier tagging, which informs the model about whether a code token is an identifier or not; (2) masked identifier prediction, similar to the deobfuscation mentioned earlier, a variant of span corruption where all identifiers tokens are masked; and (3) text ↔ code generation. For pre-training data, it not only utilizes the CodeSearchNet but also extends to include C/C# data to accommodate a wider task range. In fine-tuning, it is capable of performing task-specific transfer learning as well as multi-task learning to address both code generation and code understanding simultaneously. Building on this, CodeRL [202] innovatively combines code generation with deep reinforcement learning (using Unit Test Signals), and enhances CodeT5 in terms of learning objectives, model sizes, and pretraining data, to better adapt to the NL2Code task. In light of the outstanding performance, CodeT5 has seen further development in the future, which will be discussed in Section 4.1.

  Meanwhile, SPT-Code [203] enhances its input by integrating linearized ASTs, thereby enabling the use of both natural language and code structures as inputs during the pre-training phase. To improve the code generation ability of T5-based models, researchers also explore strategies to enable the decoder part to learn syntax and data flow [204]. NatGen [195] represents an extension of CodeT5 that leverages the bimodal and dual-channel nature of structural information. Like DOBF, it is trained by “Naturalizing” source code to exploit the codes’ naturalness and semantics. It requires the model to receive “unnatural” synthetic code as input and produce semantically equivalent code, mirroring the quality and style a human developer would prefer to write. CodeT5Mix [205] is composed of a mixture of encodes and decoders, each with specific code functionality. They can be flexibly combined to suit different scenarios and enjoy mutual benefits from joint pretraining on various targets. Further, weight-sharing strategies in decoders are used to act as task-specific experts to reduce interference across code-related tasks. Very recently, AST-T5 [206] employs a structure-aware code segmentation method during its training process, enabling the model to reconstruct code structures at various granularities.

  Beyond the aforementioned general models, specialized models also emerged for the first time during this stage. For instance, JuPyT5 [163] emerges as a CodePTM tailored for the data science domain. It is trained on Jupyter Notebook repositories from GitHub, with each cell in each notebook considered as a target during the pre-training process, aiming to serve as a data science assistant. After that, leveraging code intelligence to address data science problems has seen substantial advancement, which will be discussed in subsequent sections.

- **UniLMs.** It is noteworthy that the UniLM [207, 208] architecture has also been adopted by researchers to develop its successors trained on source code. One such model, CugLM [209], adopts BERT-like training objectives, focusing on code completion tasks.

  Another essential UniLM-style CodePTM is UniXcoder [194], which integrates various novel training objectives (e.g., code fragment representation learning) and utilizes cross-modal content such as AST and code comments. Interestingly, it has developed a less lossy method to convert ASTs into sequences, incorporating these alongside code comments as cross-modal content for pre-training. The model also expands its training dataset by utilizing both the C4 dataset and the CodeSearchNet data. Further, it employs a prefix mechanism to determine whether the model functions as an encoder-decoder model, a decoder-only model, or an encoder-only model.

3.1.3 Decoder-only

During the era where pre-training and fine-tuning are the primary paradigms for code learning, CodePTMs with a decoder-only architecture are essentially replicas of GPT [25] models applied to code, mostly adhering to the original GPT architectures and employing Causal Language Modeling. GPT-C [198] is a variant of the GPT-2 [210] trained from scratch on multilingual source code corpora. Its purpose is to serve as the Intellicode extension in the Visual Studio IDE, representing one of the initial attempts to utilize language models for code as coding assistants. Subsequently, GPT-CC [211], derived through fine-tuning GPT-Neo [212] on The Pile [213], has been used to build an open-source version of GitHub Copilot. Additionally, CodeGPT, a GPT-style pre-trained model is released alongside CodeXGLUE [114]. It shares a similar parameter size with CodeBERT and is utilized to help solve completion and generation problems during benchmarking machine learning research for program generation.

For task-specific variants, apart from adapting for specific PL [214], there is also a model: PyCodeGPT [199], designed to generate library-oriented codes, which share sim-
TABLE 2: An overview of Code Pre-trained Models’ architecture and pre-training strategies, along with whether these models leverage code structure information during the pre-training phase. Due to space limitations, we abbreviate some of the training strategies, with detailed descriptions provided in Table 10.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Models</th>
<th>Struct.</th>
<th>Base</th>
<th>Strategy</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoder</td>
<td>CubBERT [178]</td>
<td>✓</td>
<td>-</td>
<td>MLM + NSP</td>
<td>340M</td>
</tr>
<tr>
<td></td>
<td>CodeBERT [28]</td>
<td>✓</td>
<td>ReBERTa</td>
<td>MLM + RTD</td>
<td>125M</td>
</tr>
<tr>
<td></td>
<td>GraphCodeBERT [181]</td>
<td>✓</td>
<td>CodeBERT</td>
<td>MLM + Edge Pred. + Node Align.</td>
<td>125M</td>
</tr>
<tr>
<td></td>
<td>SynCoBERT [183]</td>
<td>✓</td>
<td>CodeBERT</td>
<td>MMLM + IP + TEP + MCL</td>
<td>125M</td>
</tr>
<tr>
<td></td>
<td>CODE-MVP [184]</td>
<td>✓</td>
<td>GraphCodeBERT</td>
<td>FGTL + MCL + MMLM</td>
<td>125M</td>
</tr>
<tr>
<td></td>
<td>SCodeR [186]</td>
<td>✓</td>
<td>UniXcoder</td>
<td>Soft-Labeled Contrastive Pre-training</td>
<td>125M</td>
</tr>
<tr>
<td></td>
<td>DISCO [185]</td>
<td>✓</td>
<td>-</td>
<td>MLM + NT-MLM + CLR</td>
<td>110M</td>
</tr>
<tr>
<td>Enc-Dec</td>
<td>PLBART [30]</td>
<td>✓</td>
<td>-</td>
<td>Denoising Pre-training</td>
<td>140M/406M</td>
</tr>
<tr>
<td></td>
<td>CodeT5 [29]</td>
<td>✓</td>
<td>-</td>
<td>MSP + IP + MIP + Bimodal Generation</td>
<td>60M/220M/770M</td>
</tr>
<tr>
<td></td>
<td>PyMT5 [193]</td>
<td>✓</td>
<td>PyMT5</td>
<td>MSP</td>
<td>374M</td>
</tr>
<tr>
<td></td>
<td>UniXcoder [194]</td>
<td>✓</td>
<td>-</td>
<td>MLM + ULM + MSP + MCL + CMG</td>
<td>125M</td>
</tr>
<tr>
<td></td>
<td>TreeBERT [191]</td>
<td>✓</td>
<td>-</td>
<td>TMLM + NOP</td>
<td>210M</td>
</tr>
<tr>
<td></td>
<td>ERNIE-Code [196]</td>
<td>✓</td>
<td>mT5</td>
<td>SCLM + PTLM</td>
<td>560M</td>
</tr>
<tr>
<td></td>
<td>CodeExecutor [197]</td>
<td>✓</td>
<td>UniXcoder</td>
<td>Code execution + Curriculum Learning</td>
<td>125M</td>
</tr>
<tr>
<td></td>
<td>LongCoder [118]</td>
<td>✓</td>
<td>UniXcoder</td>
<td>CLM</td>
<td>150M</td>
</tr>
<tr>
<td>Decoder</td>
<td>GPT-C [198]</td>
<td>✓</td>
<td>-</td>
<td>CLM</td>
<td>366M</td>
</tr>
<tr>
<td></td>
<td>CodeGPT [114]</td>
<td>✓</td>
<td>-</td>
<td>CLM</td>
<td>124M</td>
</tr>
<tr>
<td></td>
<td>PyCodeGPT [199]</td>
<td>✓</td>
<td>-</td>
<td>CLM</td>
<td>110M</td>
</tr>
</tbody>
</table>

We first revisit the task-enhanced variants of CodePTMs mentioned in Section 3.1. Multiple variants have opted to build upon GraphCodeBERT [181] for their developments. To enhance the capability of code search, CodeReviewer [219] incorporates additional Uni/Bimodal training objectives, making the model more aware of the semantic similarities between code-code or code-text pairs. In efforts to utilize external information to bolster code generation and summarization, both REDCODER [144] and ReACC [220] extend GraphCodeBERT by integrating an additional dense retriever. Additionally, to automate code review activities, CodeReviewer [221] is constructed by training real-world code changes and code reviews on CodeT5 [29]. Later, CodeExecutor [197] utilizes UniXcoder [194] as its basis, further learning to execute programs and predict their execution traces for improved code generation capabilities.

We then consider models retrained from scratch for specific scenarios. In terms of further leveraging code features for enhanced generation, CodeTransformer [222] leverages both the context and structure of code to extract language-agnostic features, aiming to build a multilingual code summarization model. GrammarFormer [223] utilizes code syntax to guide the generation of code completions, enabling it to refrain from making predictions in instances where the context is ambiguous. Regarding long-range modeling, Clement et al. [224] enhance transformer models by prioritizing higher-level syntactic elements to extend context windows, considering a broader range of contextual information. Conversely, to facilitate code completions with longer contexts, LongCoder [118] employ sparse attention. It utilizes a sliding window to attend only the local information, enabling models to maintain high performance even when dealing with extensive code segments. Moreover, ERNIE-Code [196] represents a specialized case, being among the first to recognize the importance of moving beyond English-centric texts. It advocates for multilingual text-to-code generation and summarization, considering both NL and PL, increasing the accessibility for a global user base.

3.2 Task-specific Adaptation of CodePTMs

Unlike the practices in the pre-transformer era where each task requires individualized modeling, CodePTMs, similar to their counterparts in NLP, can adapt to the required scenarios through task-specific fine-tuning or by adding new training objectives without significant modifications to the architecture. Additionally, they benefit from readily available end-to-end pipelines [215, 216, 217] and toolkits [218].

We first revisit the task-enhanced variants of CodePTMs mentioned in Section 3.1. Multiple variants have opted to build upon GraphCodeBERT [181] for their developments. To enhance the capability of code search, CodeReviewer [219] incorporates additional Uni/Bimodal training objectives, making the model more aware of the semantic similarities between code-code or code-text pairs. In efforts to utilize external information to bolster code generation and summarization, both REDCODER [144] and ReACC [220] extend GraphCodeBERT by integrating an additional dense retriever. Additionally, to automate code review activities, CodeReviewer [221] is constructed by training real-world code changes and code reviews on CodeT5 [29]. Later, CodeExecutor [197] utilizes UniXcoder [194] as its basis, further learning to execute programs and predict their execution traces for improved code generation capabilities.

We then consider models retrained from scratch for specific scenarios. In terms of further leveraging code features for enhanced generation, CodeTransformer [222] leverages both the context and structure of code to extract language-agnostic features, aiming to build a multilingual code summarization model. GrammarFormer [223] utilizes code syntax to guide the generation of code completions, enabling it to refrain from making predictions in instances where the context is ambiguous. Regarding long-range modeling, Clement et al. [224] enhance transformer models by prioritizing higher-level syntactic elements to extend context windows, considering a broader range of contextual information. Conversely, to facilitate code completions with longer contexts, LongCoder [118] employ sparse attention. It utilizes a sliding window to attend only the local information, enabling models to maintain high performance even when dealing with extensive code segments. Moreover, ERNIE-Code [196] represents a specialized case, being among the first to recognize the importance of moving beyond English-centric texts. It advocates for multilingual text-to-code generation and summarization, considering both NL and PL, increasing the accessibility for a global user base.

3.3 Understanding and Analyzing CodePTMs

As the field progresses, in addition to exceptional performance, explorations into why these models work and what features they can capture have gradually begun. Furthermore, researchers have also started to pay attention to the security threats to these models.
3.3.1 Understanding the Inner Mechanisms of CodePTMs

It is crucial for us to understand the inner mechanisms of language models for code and their differences from parallels in NL. Drawing from the experiences in explainable deep learning over the past few years [225, 226, 227, 228], research into their interpretability has primarily focused on two main areas: (1) task-level inspection and (2) internal mechanisms exploration in conjunction with code structure.

Karmakar and Robbes [229, 230] first construct diagnostic tasks to discover to what extent CodePTMs learn about specific aspects of source code. Troshin and Chirkova [231] also employ probing tasks to verify models are aware of code syntactic structure. The follow-up research delves into the inner workings, drawing inspiration from previous research targeting NLP models [232, 233, 234] and protein models [235]. A primary focus is analyzing models’ attention within Transformer layers, utilizing the structural information of code to provide additional signals for analysis. Specifically, Wan et al. [236] conduct qualitative analyses to evaluate how CodePTMs interpret code structure, discovering that attention aligns strongly with the code’s syntax. Subsequently, quantitative characterization of the code structure learned by models is also established by linking attention weights to AST nodes [62]. Probing experiments also point out that CodePTMs can induce entire ASTs [237].

Attention analysis also highlights CodePTMs’ propensity to prioritize specific types of tokens and statements, notably keywords and data-relevant statements. Based on these findings, input codes can be simplified for the model’s lightweight application [238]. More recent research has highlighted how the lexical, syntactic, and structural properties of code are distributed across different model layers, which could pave the way for more efficient fine-tuning strategies for code models via layer freezing [239].

3.3.2 Evaluating the Robustness and Safety of CodePTMs

Just like their NL counterparts, CodePTMs might not be resistant to changes in the input and, thus, are potentially susceptible to adversarial attacks [240] and perturbations [241]. In such scenarios, the robustness of these models requires careful investigation. Yefet et al. [242] utilize gradient-based methods to slightly perturb the input code to force a given trained model to make an incorrect prediction. For robust training, program transformations with preserved semantics are also employed [243]. Considering the “naturalness” of textual perturbation, Yang et al. [244] pioneer an example generation strategy that adversarially transforms inputs to make victim models produce wrong outputs, balancing both natural semantic and operational semantics.

 Attacks on CodePTMs can also be based on the structural information of code. CodeAttack [245] is a representative black-box attack method that leverages code structure to generate imperceptible adversarial code samples, achieving higher attack success rates than direct applications of adversarial attacks in NLP [246]. It has also been used to explore vulnerabilities in less common programming languages, such as code entities in R [247]. Zhang et al. [248] exploit the uncertainty in CodePTMs’ outputs, using it to guide the search for adversarial examples through variable name substitution. Later on, traversing the ASTs in different ways to construct adversarial samples has also been adopted as a strategy to test the sensitivity of models to input variations [249].

In parallel, backdoor attacks have also gained attention alongside the advancement of neural code intelligence. A backdoor-attacked CodePTM can behave as usual on benign examples but will generate pre-defined malicious outputs when injected with inputs embedded with backdoor triggers [250, 251]. Subsequent developments include the creation of more stealthy backdoors [252], as well as the proposal of multi-target backdoors that simultaneously aim at code understanding and generation tasks [253].

Diamond Takeaway: (1) Applying pre-trained transformers to code represents a groundbreaking initiative, effectively addressing the previously encountered dilemma of having to model each task from scratch. Moreover, it demonstrates that pre-training on a vast corpus of unlabeled code followed by task-specific fine-tuning can significantly enhance the performance across all downstream tasks. (2) Various code features mentioned in Section 2.1 have been leveraged during pre-training to enhance models’ perception of code structure. However, this explicit modeling of structures is not a free lunch; alterations to the model’s inner mechanisms due to explicit structural modeling make it challenging for CodePTMs to generalize across different tasks. (3) Leveraging code structure offers an additional perspective for interpreting and analyzing CodePTMs. Compared to their NL counterparts, researchers are becoming increasingly aware of utilizing these structures to analyze model behavior. As we step into the era of LLMs, such research is still in its nascent stage [254, 255].

4 The LLM Era: A New Frontier

The domain of code intelligence has been significantly transformed by the swift advancement of Large Language Models (LLMs) [256, 257, 258], signaling the dawn of a new era and introducing new opportunities [259]. In addition to displaying emergent abilities [260], typical LLMs such as PaLM [34], LaMDA [261] and BLOOM [262], inherently possess competent coding capabilities. This innate ability stems from their pre-training data, which is often a diverse mixture containing a considerable amount of code corpus. For instance, commonly used datasets like ROOTS [263] and the Pile [213] corpus include significant portions of code data; Pile contains 95.16GB of GitHub data out of 800GB, while ROOTS comprises 163GB out of 1.6TB. This substantial inclusion of code enables these models to learn and understand programming concepts, syntax, and semantics, thus equipping them with the ability to generate and interpret code across various PLs and tasks.

4.1 Large Language Models for Code

To harness the power of LLMs to further propel the field of code intelligence, CodeLLMs have emerged. Benefiting from the advantages of being successors to general LLMs, CodeLLMs are naturally equipped with: I. access to vast and high-quality data for training [213, 263]; II. modern positional encoding and interpolation techniques [264, 265] for tackling longer sequences. III. efficient strategies for training deployment [266, 267, 268]; IV. the resources and weights from mature open-source LLMs [269, 270, 271, 272].
Supported by these technological advancements, the design philosophy of CodeLLMs has, at a macro level, evolved from the CodePTMs era in the following ways:

1) Architecture: Aside from a few cases [273, 274] retaining the encoder part, the majority of CodeLLMs have embraced decoder-only autoregressive models to better align with generative tasks.

2) Training data: Compared to their widely adopted predecessor like CodeSearchNet [113], the new emerging corpora have rapidly grown in size and the number of PLs covered, as listed in Table 3.

3) Learning objectives: There is a shift away from explicitly learning code structural information. Moreover, beyond left-to-right generation, some models are designed to learn infilling tasks [94] to support scenarios such as code completion.

TABLE 3: Representative open-source corpora for pre-training CodeLLMs with their sizes (measured by GB/TB for disk size, or measured by M in number of files) and the number of PLs they cover.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th># PLs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigQuery (GitHub data) [link]</td>
<td>2.8M</td>
<td>6</td>
</tr>
<tr>
<td>CodeSearchNet [113] [link]</td>
<td>20GB / 6.5M</td>
<td>6</td>
</tr>
<tr>
<td>GitHub Code [link]</td>
<td>1TB / 115M</td>
<td>32</td>
</tr>
<tr>
<td>BigPython [275]</td>
<td>217GB</td>
<td>1</td>
</tr>
<tr>
<td>The Stack [276] [link]</td>
<td>6.4TB / 546M</td>
<td>358</td>
</tr>
<tr>
<td>StarCoderData [277] [link]</td>
<td>783GB / 217M</td>
<td>86</td>
</tr>
<tr>
<td>The Stack v2 [278] [link]</td>
<td>67.5TB</td>
<td>&gt; 600</td>
</tr>
</tbody>
</table>

In terms of application paradigms, there is no longer a necessity to meticulously select annotated data for training from scratch or to engage in task-specific fine-tuning. As illustrated in Figure 4, the primary approach now leans towards leveraging prompt learning and providing relevant in-context demonstrations, which are usually non-invasive to models [4, 279]. Consequently, the approach to handling various code-related tasks has gradually evolved from the diversified forms mentioned in Section 2.2 to predominantly generative methods [280].

Within this ongoing evolutionary path, the contemporary CodeLLMs that have emerged can mainly be observed on the branches on the right side of Figure 3. In the ongoing discussion, we will concentrate on some of the most representative models (and their derivatives), subsequently offering a comparatively concise review of other models and the hallmark techniques they employ. As for all available CodeLLMs and their properties, we present them in Table 4, aiming to offer a comprehensive overview of the current landscape.

- **Codex Series.** The initial version of Codex is a GPT model [33] fine-tuned on publicly available code from GitHub. Literature [36] describes a 12B version of Codex, fine-tuned on a 159 GB corpus of deduplicated, filtered Python code, which later presumably evolved to be the *code-cushman-001* within the OpenAI APIs. In 2022, OpenAI initiated the development of a new Codex variant, termed *code-davinci-002*, which is perceived as a larger model (175B) further trained on a blend of text and code data [281], and subsequently fine-tuned on instructions [282].

  The advent of Codex has had a profound impact on the development of code intelligence, pioneering and demonstrating the potential of building large-scale language models specialized for code. However, it is slightly regrettable that the exact sizes, corpora, and certain training details of the Codex series remain undisclosed to the public. Nevertheless, it is undeniable that Codex is a landmark, and researchers also believe that it has played an essential role in the evolution of other seminal OpenAI models, such as *text-davinci-003* and *ChatGPT* [281]. After Mar., 2023, The Codex series is no longer publicly available, access is limited to API credit applications via the researcher access program.

- **CodeGen Series.** TheCodeGen series [275] is one of the first open-source CodeLLMs for code generation, featuring models ranging from 350M to 16.1B. It stands as one of the initial efforts utilizing autoregressive transformers to learn from both NL and code data, with next-token prediction language modeling training objective. CodeGen innovatively proposes a multi-turn code generation approach, where a user interacts with the model by progressively providing requirements in natural language and then receives responses in the form of “subprograms.” In terms of training, CodeGen first learns general knowledge on The Pile, followed by training on a subset of Google BigQuery
(which includes 6 PLs) to obtain CodeGen-Multi. The model is then further trained on BigPython to acquire a Python-oriented model, termed CodeGen-mono. At the time, these models demonstrated capabilities close to those of CodeX. Moreover, CodeGen serves to fill a niche between larger and smaller language models for code, partially marking a significant advancement of contemporary CodeLLMs.

CodeGen2 [283] represents a comprehensively upgraded iteration that delves into the training of CodeLLMs from four main aspects: model architectures, learning methods, training objectives, and data distributions. Compared to its predecessor, this version imposes stricter control over the quality of training data and is trained on mixed objects of causal language modeling and span corruption. The resulting model is capable of infilling and supports a broader range of PLs, marking a significant advancement of contemporary CodeLLMs.

The most recent model release is CodeGen2.510, which adopts multi-epoch training on StarCoderData [277] and employs span corruption for training. This model exemplifies the principle that with a robust data recipe—specifically, running multiple epochs and utilizing data augmentation—a relatively smaller CodeLLM (7B) can be on par with its larger predecessors (typically greater than 15B), paving the way for subsequent enhancement of models from the perspective of training data.

Interestingly, given the robust performance and flexible size options of the CodeGen series, it has also been used as an initializer for other general-purpose LLMs. A case in point is MOSS [284], which is initialized with CodeGen-mono-16B and then further pre-trained on Chinese tokens, as well as samples drawn from the Pile and BigQuery.

- **BigCode Models.** BigCode11 represents an open-scientific collaboration focused on CodeLLMs, aiming to provide the research community with full insight into the development process. It has made significant contributions in various aspects, including data, models, evaluation, and ethics. The CodeLLMs developed by BigCode are among the first to adopt the fill-in-the-middle (FIM) objectives [95] (also known as causal masking objective utilized by InCoder [94], which is a 6.7B model specialized in infilling). For using CodeLLMs, it is a common necessity to generate or insert contextually appropriate content based on given snippets of code. However, due to the dependency relationship with code, solely relying on “predicting next token” falls short of capturing these complexities. FIM addresses this challenge by innovatively segmenting the code into three parts, then shuffling these segments and reconnecting them with special tokens. This strategy aims to enhance the model’s pre-training by incorporating a “fill-in-the-blank” that goes beyond causal modeling, enabling an understanding of bidirectional context by considering code dependencies.

One of the most representative models, StarCoder [277] has a size of 15.5 B, equipped with infilling capabilities and the capacity for efficient generation [266]. In terms of its training data, StarCoder employs heuristic filtering, manual inspection, and cleaning processes to compile the StarCoderData, which includes 86 programming languages. The format of this dataset encompasses text-code pairs, GitHub issues, Jupyter notebooks, and GitHub commits. SantaCoder [288] is an earlier variant with 1.1 B size. It shares the same architecture as StarCoder but is exclusively trained on Python, Java, and JavaScript. Additionally, this family of models also includes fine-tuned versions on conversation data to act as coding assistants [305].

Later developments release OctoCoder along with OctoPack [306], an instruction-tuned model created by fine-tuning StarCoder on newly collected commit messages that resemble instructions (CommitPackFT) and the Ope-Assistant (OASST) conversations dataset. Furthermore, the data is also used to construct OctoGeeX based on CodeGeeX [290]. Both models demonstrate performance that rivals non-permissive models, showcasing the effectiveness of instruction tuning and specialized training. The most recently released StarCoder2 [278] represents an even more capable version, utilizing training data that is 4 x larger and extends beyond to include notebooks from Kaggle, GitHub pull requests, and code documentation. Moreover, it uses data augmentation strategies to boost low-resource language performance by enhancing source code by pairing it with its LLVM [307] intermediate representation.

Beyond the success and impact of the aforementioned models, FIM has also been widely adopted in subsequent research. While Bavarian et al. [95] propose that FIM could be learned without harming the ability to do left-to-right generation, some research holds the view that equipping the model with this infilling ability might not be a “free lunch” [283]. For instance, models trained through FIM have been observed to sometimes struggle with determining the appropriate moments to cease infilling. We believe that these conflicting views may arise because models must learn to balance the nuances of generating code linearly with the ability to jump back and forth to fill gaps as needed. Further evidence is required to delve deeper into this discussion.

- **CodeT5+.** Unlike the previously mentioned models that are designed as decoder-only models, CodeT5+ [274] is a family of open-source CodeLLMs with encoder-decoder architecture, ranging from 220M to 16B. This model not only scales up from its predecessor, CodeT5 [29], but also showcases refined architectural design and training objectives. Architecturally, the encoder is tasked with encoding contextual representations, whereas the decoder is adept at generating diverse types of outputs. Distinctively, CodeT5+ adopts a “shallow encoder and deep decoder” structure [273], with both parts initialized using CodeGen and connected by cross-attention. For pre-training, the objectives employ a combination of span denoising and CLM [308, 309], thereby equipping the models with the capability to learn code context representations and to reconstruct missing information at different levels, including code spans, partial programs, and complete programs.

In contrast to previous models that are treated as a single system across all tasks, CodeT5+ offers the versatility to operate in encoder-only, decoder-only, and encoder-decoder modes to accommodate different downstream applications. This flexibility significantly mitigates the issue of inter-task interference encountered in UniLM-style models [194, 209].

For further improvement, InstructCodeT5+ is created as an instruction-tuned variant to align with NL instructions.
employing a strategy revolving around using synthetic instruction-following prompts [310, 291, 311].

**CodeLaMA.** Shortly after the debut of LLaMA2 [312] in July 2023, CodeLaMA [294] is swiftly released as a family of foundation models for code generation, covering model sizes 7B, 13B, and 34B (with 70B later announced in Jan 2024). Diverging from aforementioned models that are trained exclusively on code from scratch, CodeLaMA is derived from LLaMA2 through training on an additional 500B code tokens. The 7B, 13B, and 34B versions have also been trained with FIM, allowing them to insert code into existing code. For the 70B version, an extra stage of long context fine-tuning was introduced, leveraging position interpolation [313] to extend the context length from the initial 4K, as was standard with the LLaMA2 model, to an extended 16K. Experiments have shown that the pursuit of tackling longer sequences might slightly hurt the performance on shorter sequences. Still, it boosts the model’s ability to generate meaningful content in tasks like long code completion [118].

In addition to the foundation models, Meta has provided two additional variations, namely (1) CodeLaMA - Python, a language-specialized variant of CodeLaMA that has been further fine-tuned on 100B Python code. (2) CodeLaMA - Instruct, which is trained on self-instruct [311] dataset created by prompting LLaMA2 with programming problems, as well as data aimed at improving safety and helpfulness. These models have exerted a profound and positive impact on the domain, being widely applied in the development of derivatives [295, 296, 314] and various instruction tuning practices [315, 316].

Beyond the pursuit of enhanced capacity, CodeLaMA takes a forward-looking step toward responsible AI and safety. It rigorously compares itself against other CodeLLMs from the perspectives like truthfulness [317], toxicity [318], and bias [319] in generated code and text. Through empirical evaluations, CodeLaMA demonstrates the possibility of achieving high coding performance without compromising harmlessness.

**DeepSeek-Coder.** DeepSeek Coder [301] represents a range of open-source CodeLLMs with sizes varying from 1.3B to 33B, trained from scratch on a curated code corpus. In terms of data composition, besides utilizing a filtering process for GitHub data similar to the StarCoderData rules [277], it uniquely constructs repository-level code data
to enhance the model’s capability for cross-file completion within repositories. Moreover, it incorporates a small code-unrelated Chinese corpus to aid the model in understanding instructions in Chinese. The model is trained on 2T tokens comprising 87 PLs, with combining objectives of next-token prediction and FIM. Further, it employs linear scaling to extend the context window [313] to 16K, supporting repository-level code training. In terms of performance, it has achieved stunning results on tasks such as NL2Code and code completion, being considered one of the strongest open-source CodeLLMs currently available.

Beyond the models built from scratch, DeepSeek also provides versions that continue pre-training on general LLMs, termed DeepSeekCoder-v1.5 which can be viewed as a branch of DeepSeek LLM [320]. Compared to its code-exclusive counterpart, this variant, while witnessing a slight decrease in coding capabilities, performs better in tasks for math and natural language. Later, it played a vital role in constructing mathematical models [321].

- **Lemur.** Contrasting with the prevailing open-source CodeLLMs that predominantly focus on code-centric optimization, Lemur [300] represents a novel endeavor to lay the groundwork for LLM-based agents. To achieve this, LLMs must possess not only robust coding capabilities to ensure precise grounding in the relevant environments [322, 323] but also the capacity to comprehend human intentions, reasoning, and planning. Therefore, Lemur advocates for harmonious integration of language understanding and coding proficiencies.

Similar to CodeLLaMA, the model is built by continually pre-training on LLaMA2. However, in terms of data composition, it employs a 90B corpus with a 10:1 ratio of code to text, followed by fine-tuning with instructions from diverse sources. Lemur can achieve stunning results on both NL benchmarks, e.g., MMLU [324], BigBench [325] as well as on code-related benchmarks, including multilingual code generation [161], SQL [164], and data science [326]. Compared to other LLMs that exhibit a disparity between NL and code capabilities, Lemur stands out with a balanced skillset, achieving the highest overall performance when averaged across a variety of tasks. Moreover, Lemur uniquely excels in practical agent tasks, making significant strides in tool usage, self-debugging, following complex instructions, and navigating partially observable environments [327].

Beyond the aforementioned representatives, the community has witnessed a blossoming of interesting and solid works, as illustrated by the right branch of Figure 3. These contributions mainly revolve around (1) additional pre-training on general LLMs, (2) instruction-tuned variants, (3) advanced tool-use capability, and (4) efficiency enhancements.

Regarding CodeLLMs that are built from additional pre-training, CodeGeeX2 [290] represents the second iteration of the CodeGeeX model lineage. Unlike its predecessor, which was trained from scratch, Diverging from its predecessor trained from scratch, it is developed upon ChatGLM2 [328, 329] with further pre-training on code tokens. Similarly, Code-Qwen [297] follows a training approach akin to CodeLLaMA [294], using base model Qwen trained on a combination of text and code data as an initialization and then continuing to pre-train on code data. This approach is mirrored in other models such as AquilaCode12, which also enhances base models with extra training on code corpora.

Researchers also apply diverse instruction-tuning strategies to CodeLLMs as well. CodeAlpaca [291, 310] is initially built as an instruction-following LLaMA model for code generation. WizardCoder [292] is constructed by fine-tuning StarCoder [277] using Evol-Instruct [330] and ChatGPT feedback seeded by CodeAlpaca dataset [291]. WaveCoder [331] enhances CodeLLMs through an innovative instruction tuning process. It employs an LLM-based generator-discriminator framework to produce a wide array of high-quality instruction data for multiple code-related tasks, focusing on improving data quality and task diversity for fine-tuning. Recently, DolphCoder [332] also adopts diversified tuning strategies. It begins by leveraging multiple chain-of-thought [333] responses to the same instruction and then combines the tasks of code generation and code evaluation in the form of natural language generation. Likewise, MoTCoder [334] utilizes a modular approach for instruction tuning. It segments complex coding tasks into logical sub-modules, guiding models to first outline and then implement these sub-modules.

Regarding tool use [335, 336], ToRA [295] targets building tool-integrated agents, addressing complex mathematical reasoning. To achieve it, ToRA is trained upon the collected interactive trajectories of invoking tools. Nevertheless, its coding ability is mainly limited to Python. In the same vein, MammoTH [296] concentrates on equipping off-the-shelf LLM with Python-integrated reasoning abilities (will be discussed in Section 5.1). It utilizes a hybrid composition of data generated from intermediate steps when reasoning with NL or code for further pre-training on LLaMA. This approach is expected to unleash both program-aided and NL-centric power in mathematics. Concurrent with Lemur’s success in harmonizing text and code capabilities, a new foundational model, Symbol-LLM [302], expands the scope of code capabilities to encompass the entire range of symbol-centric capabilities, complemented by an external symbolic solver. This extension broadens the application scope of LLMs to more intricate scenarios beyond code generation, such as neuro-symbolic reasoning.

As for efficiency-optimized CodeLLMs, phi-1 [289] distinguishes itself through its compact size and the unique approach of utilizing high-quality, “textbook-quality” data for training. It demonstrates the efficacy of quality over quantity in data selection and the potential for smaller code models to compete with or outperform larger counterparts. Factors influencing the quality of code data have also been explored in subsequent studies [337]. When it comes to CodeLLMs that can run on consumer-level hardware, Stable Code [303] (based on Stable LM [338]) and DeciCoder [304] adopt a more compact and concise design. These models uphold a certain performance standard while allowing users to deploy them locally, enhancing the accessibility of code generation to a broader range of users.

In the realm of application frameworks, CodeTF [217] initially provides an interface for both training and inference, facilitating the integration of CodeLLMs into practical applications. This framework aims to make it easier for

developers to leverage the power of CodeLLMs in efficiency and functionality of software development processes. MFT-Coder (CodeFuse) [298] presents a multi-task fine-tuning framework specifically designed for CodeLLMs. It supports the efficient tuning and deployment of a broad spectrum of models, enabling developers to adapt and optimize these models for various coding tasks and challenges swiftly. For example, CodeFuse-CodeLLaMA is a model created by further training of CodeLLaMA through MFT, which achieves performance surpassing that of GPT-4 on the HumanEval benchmark.

4.2 Learning with Execution Feedback

Another pathway to further enhance CodeLLMs involves integrating Reinforcement Learning (RL), which incorporates non-differentiable reward signals into the training process. Unlike approaches represented by reinforcement learning from human feedback (RLHF) that utilize human preferences [339, 282, 340], the inherently compilable and executable nature of codes allows for the use of compilers or interpreters to automatically generate precise feedback. Such endeavors were initiated in the era of CodePTMs, as COMPCODER [341] harnesses the compilability signals to optimize both the generator (e.g., CodeGPT [114]) and the discriminator (MLPs) via RL strategies.

As the capability for code generation improves, using RL in code training becomes increasingly flexible. CodeRL [202] exploits the code unit test signals in both training and inference stages and uses RL to optimize the model. PPOCoder [342] combines CodeLLMs with Proximal Policy Optimization [343] for code generation. RLTF [344] is another novel online RL framework, which uses unit test feedback of multi-granularity for refinement. RLCF [345] further enhances a CodeLLM by incorporating feedback from a grounding function, which assesses the quality of generated codes. Pangu-Coder [346] introduces an RRTF (Rank Responses to align Test & Teacher Feedback) framework, aimed at steering the model towards producing higher-quality code achieved by synergistically using test signals and human preferences as combined feedback. ExeDec [347] innovates in decomposing tasks into execution subgoals, improving compositional generalization through tackling complex tasks step-by-step. In a similar vein, recently released StepCoder [348] innovates by breaking complex tasks into a curriculum of subtasks, tackling code generation’s exploration and optimization challenges.

4.3 Advancements in NL2Code

In the era of LLMs, the ability of NL2Code has leaped forward, with machine learning models now truly capable of assisting professional developers through crafting accurate code snippets based on human intent. This holds a tantalizing promise of “programming in natural language”. Moreover, the role of NL2Code has also transitioned; it has transcended the initial function as merely a downstream coding task and has become a pivotal metric for evaluating the capabilities of LLMs. Here we first discuss the shift in evaluation paradigms and then focus on extensively utilized benchmarks and their derivatives.

4.3.1 The Shift in Evaluation Metrics

Limitations of Match-based Approaches. In the Pre-LLM era, code generation capabilities were primarily benchmarked by matching samples against a reference solution, using metrics like (smoothed) BLEU scores [356, 357, 67]. Nevertheless, in addition to the lingering issues already identified in NLG systems [358, 359], BLEU-based evaluations struggle to capture semantic features specific to code [360]. Although variants like CodeBLEU [361] propose several semantic modifications based on code structure, a fundamental problem remains: match-based metrics are unable to fully represent the broad and complex space of programs that are functionally equivalent to the reference solutions. This dilemma extends to other evaluation metrics [106, 107, 362, 363] as well. Consequently, Chen et al. [36] advocate the use of execution-based evaluation to measure functional correctness for NL2Code tasks instead.

The Rise of Execution-based Evaluation. For evaluating the functional correctness of a generated code snippet, the most reliable approach is to examine if it can be successfully executed and passes a set of unit tests, a method commonly employed in software engineering’s test-driven development. pass@k is initially designed for assessing pseudocode-to-code translations [105]. Generating k code samples for each problem, a problem deemed solved if any of the samples pass, and it reports the total fraction of problems solved. However, due to the high variance, Chen et al. [36] refine it into a more stable metric as delineated in Equation 1.

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

This revised method involves producing \( n \geq k \) samples for each task, counting correct samples \( c \leq n \) that pass unit tests successfully, thereby deriving an unbiased estimator. pass@k represents a milestone in NL2Code evaluation, which transcends the limitations of earlier metrics incapable of accurately reflecting functional correctness and has become a standard practice in all subsequent benchmarks.

4.3.2 Commonly Used Benchmarks and Evaluations

Herein, we offer detailed descriptions and discussions for HumanEval [36] and MBPP [350], two of the most widely utilized contemporary benchmarks, along with MultiPL-E [161] benchmark derived from them. Subsequently, we provide a bird’s eye view of other datasets and the new benchmarks based on them. Subsequently, we present an overview of additional datasets and introduce benchmarks developed for specific purposes.

- **HumanEval.** HumanEval was initially released in conjunction with Codex [36], comprising 164 manually crafted Python coding problems. These problems are validated using test cases (with an average of 7.7 tests per problem) to evaluate the code generated by CodeLLMs, typically in a zero-shot setting. The necessity for these problems to be manually crafted stems from the fact that the majority of contemporary CodeLLMs are trained on a large fraction of GitHub data, which already contains solutions to some ready-made programming challenges, such as those...
collected from Codeforces\(^{13}\), LeetCode\(^{14}\) and CodeChef\(^{15}\). HumanEval includes problems of varying difficulty levels, ranging from basic string and array manipulations to relatively complex data structures and algorithms. Each problem provides a function signature, clarifying the input/output format to aid the model in understanding the requirements. Since its release, HumanEval has been widely adopted by the community as a primary tool for evaluating code generation capabilities, and subsequently, it has been extended with the goals of (1) covering more programming languages and (2) constructing more robust evaluation.

Multilingual HumanEval\(^{353}\) introduces a scalable automated framework capable of converting datasets from Python into variants for 12 different languages. In contrast to the automated approach, HumanEval-X\(^{290}\) is another multilingual extension constructed by the authors of CodeGeeX. Similar to the original HumanEval, it is manually crafted, involving a rewrite of prompts, canonical solutions, and test cases for C++, Java, JavaScript, and Go.

Unlike the strategies of expanding the number of languages to achieve comprehensive evaluation, researchers are aware that the quantity and quality of test cases in problems can be inadequate for fully assessing functional correctness\(^{364}\). This gap may inadvertently lead to flawed code being considered correct due to the paucity of testing. To address this issue, HumanEval* is developed by extending the unit tests of HumanEval, augmenting the scale of test cases by 80 times. This substantial increase in testing has proven to catch significant amounts of previously undetected incorrect code.

- **MBPP.** Mostly Basic Programming Problems\(^{350}\) stands as another widely recognized benchmark for assessing the performance of CodeLLMs in Python programming, especially under few-shot scenarios. Unlike HumanEval, which features varying levels of difficulty, MBPP is tailored to be solvable by entry-level programmers. It contains 974 short Python functions, each accompanied by an English description, a predefined function signature, and three manually written test cases for validation. As for its compilation, MBPP amalgamates a vast collection of crowd-sourced questions and a smaller manually edited and verified subset. The difficulty of these questions ranges from simple numerical manipulations to those requiring some external knowledge (e.g., the definition of the Fibonacci sequence).

In a similar vein, MBPP has also been expanded to accommodate multilingual scenarios, aiming to offer a more comprehensive and diverse evaluation. One of the most notable extensions is MBXP (Most Basic X Programming Problems, where X = Java, Go, Ruby, etc)\(^{353}\), constructed in a parallel to Multilingual HumanEval. Moreover, it has been extended to cover other code-related scenarios, encompassing zero-shot code translation, prompt perturbation test, code insertion, and code summarization.

- **MultiPL-E.** MultiPL-E\(^{161}\) emerges as a new benchmark system tailored for multilingual scenarios, constructed upon the two benchmarks discussed previously. It extends their scope by translating them into 18 additional programming languages chosen according to TIOBE\(^{16}\) rankings and GitHub usage frequencies. To accommodate the diverse languages, it features a containerized sandbox environment, which compiles programs (if necessary), runs them with test cases and appropriate timeouts, and categorizes each output as successful, syntax error, etc. Furthermore, MultiPL-E stands out for its scalability and extensibility, offering a streamlined process for incorporating new benchmarks and languages, thereby minimizing the need for manual intervention.

\(^{13}\) https://codeforces.com/

\(^{14}\) https://leetcode.com/

\(^{15}\) https://www.codechef.com/

\(^{16}\) https://www.tiobe.com/tiobe-index/
Beyond the aforementioned benchmarks, more works regarding evaluation are emerging alongside the surge in interest in CodeLLMs, broadly categorized into three types: (1) Those tailored for specific NL2Code scenarios, like CodeContests [273] and APPS [352] designed for programming competitions. (2) The conversion of general benchmarks into testbeds for code generation, such as MathQA-Python [350] and selected BigBench [351] problems. (3) Those that consolidate, refine, and repurpose existing benchmarks, such as L2CEval [365], which has been developed as a standardized benchmark combining semantic parsing, math reasoning, and Python programming. Representative benchmarks for various purposes are shown in Table 9, and a comprehensive list of benchmarks currently available is presented in Table 5. We will further discuss the opportunities and challenges of benchmarking CodeLLMs in Section 7.4. Additionally, theoretical frameworks for evaluating CodeLLMs are also emerging, such as applying category theory [366] to express the structural aspects of NL and code.

To provide readers with an intuitive understanding of the current CodeLLMs’ capabilities, we display the HumanEval performance of some open-source representatives in Figure 5. It is evident that, in addition to the positive correlation between model size and performance, models of the same size that have been fine-tuned with instructions significantly outperform their base versions. The benefits that instruction fine-tuning brings to CodeLLMs are more pronounced than those observed in general LLMs. We hold the view that this gap is likely because human intentions, compared to code comments or documents, are more intricate. Research has shown that models like code-davinci-002 achieve top performance in benchmarks, yet their outputs may not always align with human expectations [367, 281]. Therefore, for developing practical CodeLLMs, dedicating more effort to instruction fine-tuning could be more cost-effective than merely expanding the model size.

4.3.3 Strategies for Enhanced Code Generation

Building upon our discussions above, here we explore a variety of innovative strategies that have significantly advanced the capabilities of code generation.

Decoding-Enhanced. In terms of decoding strategies, Zhang et al. [366] propose that conventional decoding algorithms may not be the best choice for code generation, and utilize lookahead search to guide future token choices. Self-planning [369] improves code generation by first creating a high-level plan to break down complex tasks into simpler steps, and then generating code for each step. Self-infilling [370] enhances code generation by combining sequential context generation with infilling, which allows for more controlled output. Yang et al. [371] break down the problem into multiple steps and utilize programming hints as support, allowing lightweight models to generate better programs. Considering the reuse of existing code, AceCoder [372] searches for programs with similar requirements as guidance generation for improved code generation. Taking the structural information into account, Li et al. [373] ask the model to first generate the program’s structure (e.g., loops and branches) and then implement the code.

Feedback-Driven. As discussed in Section 4.2, execution feedback represents an external signal that code can innately utilize. Employing/building test cases [374, 375, 376] within code generation processes emerges as a viable means to enhance the reliability of the generated code. CodeT [377] exemplifies this category of methods by unleashing the model’s inherent capability to automatically generate unit tests, ensuring the consistency of outputs through broader tests. Similarly, TiCoder [378] enhances code reliability by constructing an augmented set of examples to cover a wider range of possible user inputs. LEVER [379] enhances NL2Code generation by integrating execution feedback provided by trained verifiers, harnessing both the generative and discerning capability. Further, ALGO [380] advances in using LLM-generated oracles for algorithmic program synthesis. Beyond NL2Code tasks, this concept of leveraging unit test feedback can also be applied to other tasks, such as multilingual code translation or code search, ensuring the functional equivalence of translated content [381] or retrieved code [382].

NL-Directed. Utilizing NL information represents another pathway. Given an instruction, Zhang et al. [383] evaluate the likelihood of the given instruction given the generated programs, thereby improving code quality through reranking. DocPrompting [384] enhances the code generation process by explicitly retrieving relevant document pieces from a pool based on user intent. Very recently, AlphaCodium [385] proposes a test-based, iterative process combining understanding NL and code generation.

In addition to the insights gleaned thus far, there are ongoing studies within the field that may hold significant
implications for building CodeLLMs in the future, such as better tokenization and efficiency improvements. These topics will be elaborated upon in Section 7.6 and Section 7.7.

Takeaway: (1) The advent of CodeLLMs has been revolutionary for code intelligence, heralding a new learning paradigm. Whether as variants of general LLMs or built from scratch, they demonstrate exceptional capabilities not seen in their predecessors, achieved through larger sizes, premium code data, task-oriented training objectives, and intricately designed tuning strategies. (2) The integration of RL with CodeLLMs offers a promising avenue for enhancing code generation through the use of non-differentiable reward signals, such as compiler feedback and unit test results. This allows for the precise and automatic generation of feedback, overcoming the high costs associated with learning from human preferences. (3) Whether in terms of model capabilities or the diversity of tasks, NL2Code has experienced unparalleled expansion during this period. Further, despite a multitude of efforts to devise intricate matching-based evaluation metrics, the LLM era has seen a shift in evaluating code generation tasks toward reliance on execution to assess functional correctness. (4) Compared to the various ingenious methods mentioned in Section 3.2, aimed at enhancing code-related tasks through specific training, the approaches for handling downstream tasks have converged towards generative methods. Essentially, these are predominantly based on prompting, which is non-invasive in nature.

5 Synergies in Machine Intelligence

Following our detailed exploration centered on code generation and comprehension, in this section, we delve into some synergies with other aspects of machine intelligence. Specifically, we will discuss from three perspectives: new reasoning paradigms based on code generation, mathematical abilities enhanced by code training, and the multi-dimensional capabilities of language models expanded through code.

5.1 Binding Code Generation with LLM Reasoning

Reasoning constitutes a long-standing task in the domain of machine intelligence [386]. The surge in LLM enthusiasm has brought to light approaches exemplified by chain-of-thought (CoT) prompting [333], proposed as a novel form of in-context learning [38] wherein the exemplar contains the rationales instead of merely an answer. These methodologies, which encourage LLMs to generate a series of intermediate steps toward a final solution [387], have made stunning progress across a wide spectrum of textual and numerical reasoning benchmarks [388, 389, 390, 391, 392].

Unlocking a New Reasoning Paradigm. CoT-based approaches do not serve as a silver bullet for solving all reasoning problems. Beyond factuality [391] and hallucinations [393, 394] issues in LLMs, given the inherent limitations language models face with complex arithmetic operations and managing large numbers [395, 396], LLMs are susceptible to logical and arithmetic mistakes in the calculation phase, despite the problem decomposition being correct. In light of this dilemma, code generation offers a potential pathway for disentangling computation from reasoning. Program-Aided Large Language Models (PAL) [397] and Program of Thoughts (PoT) [398] employ LLMs to comprehend natural language problems and synthesize programs as intermediate reasoning steps. Crucially, they delegate the execution of the final solution to a symbolic solver (includes but not limited to a Python interpreter [399]), thereby resolving the computational limitations of LLMs.

This strategy of decoupling computation from reasoning and language understanding, has achieved great success across a wide range of datasets involving mathematical reasoning [400, 401, 402], symbolic reasoning [351, 325], and semi-structured understanding [403, 404]. For instance, bolstered by PAL, leveraging a CodeLLM like Codex has significantly outperformed the results obtained by larger models (e.g., PaLM/Minerva) on math word problems, BIG-Bench Tasks, and financialQA datasets using NL reasoning chains. This approach has gradually emerged as a new reasoning paradigm for solving all numerical-related problems.

Laying Foundations for Broader Reasoning Scenarios. Emerging variations of PAL or code integration [405] into broader reasoning frameworks have also been developed. For example, combining it with CoT to merge the strengths of both approaches through model selection offers enhanced flexibility and generalizability [406]. Alternatively, integrating intermediate codes step with stochastic beam search to guide decoding presents another innovative direction [407]. Chameleon [408] modularizes the function generation process and integrates it with other tools, such as web search engines, to tackle more complex reasoning scenarios. For collaborative reasoning involving multi-model interactions, following the use of NL reasoning chains [409], code has also been leveraged as an alternative format for communication [410, 411, 412]. In addition to logic and arithmetic task [413, 414], more recent research, Chain of Code [415], encourages models to represent semantic tasks in the form...
of pseudocode. This simple yet effective CoT extension enables models to explicitly identify and address undefined behaviors by simulating code execution.

Beyond offering performance gains in reasoning, the deterministic execution of code also can be utilized for interpretability. In pursuit of faithfulness, a notable example is Faithful CoT [416]. This approach converts a natural language query into a chain that interleaves natural language and a task-dependent symbolic language, such as Python or PDDL, helping us understand how the model arrives at the final answer.

In sum, integrating code generation with LLM reasoning is a great breakthrough. This new paradigm, which leverages models to interpret natural language and then generate programs as intermediate reasoning steps for execution, has transcended the entrenched limitations of traditional solutions. Moreover, it lays the groundwork for exploring the synergy between generative models and neuro-symbolic AI in a broad range of scenarios [417, 418, 419]. Nevertheless, PAL is not a panacea and faces challenges such as model misinterpretation of questions and the fact that generated codes are not always error-free. Furthermore, the relationship between code utilization and enhancements in reasoning capabilities has not been fully elucidated [420], making it an attractive and ongoing area of research.

5.2 Code Training Elicits Mathematical Capabilities

Mathematics capability has historically been considered the Achilles’ Heel of language models [421]. However, with the advent of LLMs, there has been a significant shift in this perception [422]. Code training is believed to have played a pivotal role during this transformation.

Unforeseen Benefits of Code Training. Before the buzz around LLMs, it was already discovered that language models trained on code could directly solve basic algebra [423] or statistical problems [424]. Early research at the onset of the LLM era find that code-davinci-002, regardless of whether it utilizes chain-of-thought, significantly outperforms other models on mathematical tasks [425]. This observation has ignited the long-standing question among researchers: “Does code training improve mathematical abilities?” While the additional benefits from code training remain debatable, we collect some evidence to discuss the correlation between them. Code-davinci-002 can easily demonstrate chain-of-thought capabilities in mathematical reasoning, achieving results far superior to those of text-davinci-001, which did not undergo code training [333]. Fu et al. [390] also note that when the reasoning chain is longer, code-davinci-002 is the best-performing model on mathematical benchmarks. Finally, the traditional next-token prediction objective usually captures “local” information, while code often incurs longer dependencies and hierarchies, such as referencing distant function definitions. This may indirectly cultivate the model’s ability to understand more complex structures [281].

From the perspective of training, Ma et al. [426] highlight that training with a mixture of code and text can significantly enhance LLM general reasoning capability, without any detriment to other aspects. Recent data-centric research [427] has further demonstrated that the outcomes achieved from equivalent mathematical training on CodeLLMs (e.g., CodeLaMA), markedly exceed the results compared to their base models or counterparts devoid of code training. We left the further examination of this intriguing yet unverified hypothesis to future works.

Building Math Models with Code. From a practical standpoint, CodeLLMs can serve as the foundation for developing sophisticated mathematical models. Llemma [314] utilizes CodeLaMA [294] as the basis and is trained on a diverse mixture of math-related text and code to achieve remarkable mathematical capabilities, including the ability to use formal theorem provers. Meanwhile, DeepSeekMath [321] is developed based on DeepSeekCoder [301], undergoes specialized training on web pages meticulously filtered for mathematical content. Both models exhibit superior performance on benchmarks such as MATH [402], STEM, and SAT, surpassing that of prior mathematics language models like Minerva [428]. Beyond the unanimous choice of conducting continual pre-training on CodeLLMs, comparative experiments from DeepSeek also suggest that code training can be a valuable prelude to math training, empirically showing the positive relationship between code training and mathematical capabilities.

Recently released InternLM-Math [429] incorporates code generated by proof assistants during its training process and innovatively interleaves coding processes within the problem-solving approach, presenting new state-of-the-art mathematical capabilities with the assistance of Python. To date, code learning has been demonstrated to play a pivotal and positive role across various facets of mathematical proficiency. A deeper understanding of its impact in this domain remains an area ripe for further investigation.

5.3 Alternative Formats for Solving NLP Tasks

Information Extraction. Information extraction (IE) aims to extract structural knowledge (e.g., entities, relations, and events) from unstructured and/or semi-structured documents. However, due to the tasks having diverse output forms and requiring complex decoding strategies to post-process them into valid structures, relying solely on the plain text output of LLMs proves challenging for efficient modeling [430]. In the context of generative IE [431], these semantic structures can be smoothly converted into structured code, which plays a crucial role in processing various tasks in a unified schema [432]. CodeIE [433] shows that formulating text-to-structure IE tasks into structure-to-structure code generation with “code-style” prompting leads to superior performance compared to previous NL approaches. Additionally, CodeKGC [434] introduces a novel method in the construction of knowledge graphs by treating the generation of triples as code completion tasks and then develops schema-aware prompts to further leverage the structural knowledge inherent in code. Code4UIE [435] builds a framework for retrieval-augmented code generation, which utilizes Python classes to define various schemas under a unified format. GoLLIE [436] is developed through fine-tuning CodeLaMA, by integrating Python code and comments into the input representation, thereby enhancing its zero-shot performance on unseen tasks.

Code as Intermedia Representation. In addition to the numerical reasoning (discussed in Section 5.1) and the
aforementioned IE tasks, which can be formulated as code generation tasks in a relatively fixed paradigm, recent emerging research has applied code as a medium in more diverse scenarios, instead of text [437]. Early in the rise of LLMs, Madaan et al. [438] highlight that CodeLLMs could effectively represent desired graph predictions as code for use in structured commonsense tasks. Utilizing CodeX by few-shot prompting, they achieve performance surpassing that of general language models fine-tuned on the target task. Subsequently, this approach of constructing procedural processes based on code has also been applied to sequential decision making [439], story-based tasks [440], semantic parsing [441] and neurosymbolic understanding [442]. For graph scenarios, recently released InstructGraph [443] unifies the representation of graphs through a code-like format, eliminating the need for external specific encoders in graph reasoning and generation.

Beyond scenarios oriented toward linguistic structures, modular approaches based on code can also be applied to understanding both textual and graphical information. For example, VisProg [444] generates Python-like modular programs that involve off-the-shelf models or functions to facilitate visual reasoning. ViperGPT [445] creates programs for visual queries that take images or videos as arguments for execution, rather than traditional end-to-end methods with limited interpretability and generalization. Chen et al. [446] develop code-vision representations to assist in capturing visual structural information of varying granularity. More recently, Sharma et al. [447] use code to represent images for teaching models about more aspects of the visual world. Moreover, user queries received by robots can also be translated into executable actions through code generation, empowering intelligent robots with the capability for multi-step embodied reasoning [448]. In sum, integrating code promises to evolve into a novel paradigm for addressing diverse NLP tasks. However, this approach is not a free lunch, since utilizing code as an intermediary can introduce overhead (e.g., defining a function) and consume more context windows. Further explorations in this vein are ongoing.

◊ Takeaway: (1) The fusion of code generation and symbolic solvers with LLM reasoning represents a groundbreaking shift in tackling numerical tasks. Replacing natural language with executable code as the medium for reasoning, it not only overcomes lingering computation limitations but also enhances model interpretability and generalizability. (2) Although a theoretical foundation has yet to be established, the capacity of code training to enhance the mathematical abilities of LLMs has been empirically demonstrated and is gradually being accepted within the community. (3) Adopting code as intermediate representations can significantly elevate the efficacy and versatility of tackling diverse NLP tasks. By transcending traditional text-based processing, code-centric approaches, through the construction of a unified schema, can handle intricate and complex input and output forms that previous methods struggled with.

6 Real-World Applications

Upon finishing our review of models, algorithms, and data, we shift our focus to discussing their real-world applications and current developments. Initially, our discourse centers around code processing and structured data, highlighting two main areas: (1) the direct application of language models for code in SE, aiming to bridge the gap between the NLP and SE communities [5]; (2) the exploration of utilizing CodeLLMs to augment data science research. Subsequently, we delve into hybrid scenarios, encompassing (1) the construction of agents through the application of code intelligence and (2) an emerging domain that leverages code intelligence to support AI4Science research.

6.1 Boosting Software Develop Workflows

Neural code intelligence is redefining the landscape of software development, steering it towards unprecedented levels of accessibility and automation across various scenarios.

Serving as Coding Assistants. It is an indisputable fact that coding assistants, exemplified by GitHub Copilot [449], are fundamentally transforming the landscape of software development [450]. Earlier attempts, such as aiXcoder and Intellicode [198], have already demonstrated that deep learning models can assist with basic coding tasks. With the advent of the LLM era, emerging products like Code Whisperer [451], Tabnine [452], and Coze [19], which are built on commercial models, have begun to mature into established products. Concurrently, coding assistants based on open-source CodeLLMs are increasingly gaining popularity and are being utilized in multiple code downstream tasks. For instance, FauxPilot [20] and Starchat [305], respectively based on the CodeGen and BigCode models, can serve as locally hosted alternatives to Copilot. These versatile tools can be used for code generation, completion, repair, and even predicting the time complexity [453]. Additionally, when paired with a code interpreter (e.g., ChatGPT Plugins [21]), chatbots are capable of aiding users in building temporary programs within conversations. Furthermore, interacting with users in the form of extensions has also started to become popular, with CodeGeeX [290] already adapting itself to IDEs such as Visual Studio Code and JetBrains. Empirical research has illuminated the profound impact of these AI coding tools on developer efficiency. For instance, A prior study [454] has indicated that the majority of GitHub Copilot users have experienced enhanced programming efficiency and improved code quality when coding with it. Complementarily, Peng et al. [455] quantify this enhancement in productivity, demonstrating a notable acceleration in task completion when developers employ Copilot.

As interest grows, attention is also directed towards the multifaceted performance of these assistants. Beyond evaluating the generated code itself [456, 457], researchers have also recognized the influence of prompts on their operation [458], the robustness issues faced when tackling varying scenarios [459], and their effectiveness across specific PLs [460] or NLs [461]. Moreover, the quality of generated codes [462] is another aspect that should not be overlooked. Fu et al. [463] have revealed common weakness enumeration and advocated for meticulous review processes to mitigate risks associated with automated code generation.

18. https://github.com/aixcoder-plugin
20. https://github.com/fauxpilot/fauxpilot
So, we need a dual focus on enhancing productivity while safeguarding against vulnerabilities in code generation.

**Streamlining Software Development.** Compared to having models write or complete code of different granularities for you, allowing them to take over the entire software development process is also becoming a tangible reality. ChatDev [464] represents a virtual “software company”, functioning through an array of intelligent agents assuming roles such as programmer, code reviewer, tester, and designer. Collectively, these agents establish a multi-agent ecosystem, facilitating comprehensive software development processes. AgentCoder [465] also implements a multi-agent system, assigning different roles to models, showing that role-playing strategy not only alleviates the need for manually crafted test cases but also achieves better outcomes than self-refinement methods [466, 467, 468].

After that, Qian et al. [469] enrich these agents with experience, encouraging them to accrue shortcuts from previous experience to avoid inefficient attempts or repetitive errors. This strategy enables the agents to tackle software engineering tasks within their interactions more efficiently, leading to improved automation and efficiency for unseen tasks. Beyond software kernel design, graphical design also falls within the scope of interest [470]. The newly released Design2Code [471] aims to automate front-end development by converting visual designs into code implementations.

As for leveraging code intelligence within GitHub, CodeAgent [472] suggests the use of external tools (e.g., Format Checker) for repo-level code generation. Meanwhile, beyond using pre-defined function set [473, 474], GitAgent [475] autonomously integrates repositories in response to user queries, thereby expanding its toolkit. Regarding software maintenance, the quality of code documentation is directly linked to development efficiency [476]. To automate this, RepoAgent [477] has been proposed for documentation generation. It utilizes AST analysis to understand code structure and discern reference relationships within files, providing a contextual perspective for LLMs to assist in identifying the functional semantics to support fine-grained code documentation generation. For more reliable testing, Meta has introduced TestGen-LLM [478], which aims to improve existing human-written tests for automated unit test generation. The majority of its improvements have landed in industrial production. Meanwhile, Google has advocated for the automation of resolving review comments within daily development workflows [479].

### 6.2 Facilitating Data-Driven Decision-Making

Neural code intelligence has emerged as a transformative force in data-driven decision-making by unlocking new potentials in more accessible database interactions and streamlining data science processes.

**Democratizing Database Interactions.** Previous efforts primarily emphasize elaborate model design and optimization tailored for specific formal languages, which lack the innate capability to effectively adhere to instructions. Large language models implicitly contain extensive world knowledge, which brings some basic abilities to interact with users. However, the inherent autoregressive characteristics [210] of LLMs make it challenging for them to accurately retain and recall data, especially when facing large-scale private databases. Under such circumstances, it is necessary to equip LLMs with the capabilities to automatically interact with external databases. The ultimate challenge lies in the precise generation of formal calling languages (e.g., SQL) based on the given query, and it is in this aspect that CodeLLMs truly excel.

With the advent of CodeLLMs [294, 301], recent endeavors have also focused on facilitating widespread access to database interactions, aiming to bridge the retrieval process [480]. Early attempt like Binder [481] leverages CodeX to generate the programming language to combine the external knowledge bases. [482] proposes the optimized prompt design to boost the performances of Text-to-SQL tasks. In addition, SQL-PaLM [483] targets powering off-the-shelf LLM with SQL-specific optimization. They represent two common types of practices in this direction: 1) prompting Code-LLMs; and 2) post-finetuning LLMS with SQL optimization.

To better measure the LLM performances in tackling database-related scenarios, the evaluation benchmarks are gradually improving [484]. The overarching trend is shifting from static, single-turn interactions constrained by limited domains towards dynamic multi-turn conversations encompassing diverse domains. At the early stage, benchmarks, such as ATIS [485], tackle domain-specific settings (e.g., flights) with a limited number of schema. Following them, Spider [164], Sparc [165], Cosql [166], WikiSQL [486] and WikiTableQA [487] etc. are proposed to cover plenty of domains and over thousands of schema. Powered by the advent of LLMs, some challenging and comprehensive benchmarks arouse wide interest. Recent work [488] highly stresses pushing the boundaries of LLMs and making LLMs serve as a functional interface. InterCode-SQL [489] tackles the multi-turn interaction with the database, pushing LLM to more practical scenarios. To sum up, the potential of self-repairing abilities and multi-turn interactions with databases are highly valued in the current landscape.

**Accelerating Data Insight Discovery.** Data science entails the extraction of insights from data [490], and has evolved to be a pivotal component of decision-making and knowledge discovery over the past few years [491]. Earlier attempts in this domain included methods for querying tabular data in natural language [492] or generating Pandas codes for data analysis [199]. Recent research has progressed to handling real-world data science notebooks, which are more complex than mere code generation, as computational notebooks often mix codes, text, figures, and execution results [493].

Regarding models specifically designed for such scenarios, after the initial construction of JupyT5 [163] as a data science assistant, PaChiNCo [494] emerges as a 62B CodeLLM based on PaLM, specifically designed for Python data science. Beyond its substantial size, PaChiNCo is trained under massive multi-modal contexts, such as existing notebook cells, corresponding execution states, and previous interaction turns. These rich contents ensure the model aligns with the specific peculiarities of computational notebooks, accommodating more diverse elements.

As for benchmarks that more closely align with real-world applications, particularly those concerning data wrangling tasks to process raw data and exploratory data...
analysis, DS-1000 [326] stands out as an example. It comprises high-quality problems sourced from StackOverflow, including use cases involving NumPy, Pandas, TensorFlow, PyTorch, etc. In terms of evaluation, what sets it apart from conventional NL2Code tasks is not just the emphasis on functional correctness; it also imposes additional constraints, such as requiring the generated code to include specific APIs/keywords to ensure solutions are efficient and aligned with the query [495]. ARCADE [494] presents another challenging benchmark, with a focus on Pandas. Its cases are derived through repurposing high-quality portions of previous benchmarks [162, 163] and collecting interactions between professional data scientists and coding assistants. ARCADE features multiple rounds of code generation within the same notebook, emphasizing the iterative nature of real-world data science work. Research in this domain continues to thrive, with scholars also focusing on handling sophisticated DataFrames [496], tackling the challenges of complex data visualization [497, 498], and addressing hallucinations in conversations [499].

6.3 Building Code-Empowered Agents

A perennial topic in machine intelligence is to develop agent systems [500], such as robots, that can perform complex tasks requiring interaction with real-world environments. Recent breakthroughs in LLMs have notably enhanced the task-solving capabilities of AI agents. Typically focused on natural language generation, LLMs offer immense potential [323, 501, 322], yet the inherent ambiguity of natural language can sometimes render the precision and efficiency of planning and interaction [502]. To address this, emerging research advocates the integration of code as the de facto standard of agent systems [503, 504, 505, 506, 507]. The following discussions will delve into these advancements.

Augmenting Robotics System. Building robots capable of manipulating objects to accomplish diverse tasks in physical environments poses a significant challenge. Code generation enables robots to seamlessly combine environment perception [508, 509, 448], feedback loops [510, 511, 512], and parameterized actions [513, 514] into reasonable policy, effectively translating complex tasks into executable solutions. Generating executable actions is fundamental for robots; ProgPrompt [510] pioneered the use of GPT-3 to generate code-based actions, offering better environmental grounding than free-form text. Robot tasks often involve multiple sub-tasks and conditional loops [36], where code-based planning naturally excels over natural language due to its structured advantage. For instance, Liang et al. [508] explored utilizing LLM to write robot policy code, incorporating complex feedback loops and primitive API calls. Following studies concentrated on integrating visual input [514, 515, 516], refining code through environmental feedback [512, 517] or reinforcement learning [515], and continually expanding agent’s skill library [513]. Faced with difficulties in collecting human operation trajectories, a series of studies explored leveraging LLMs for code generation to automatically synthesize diverse robot data [518, 512]. Additionally, Ha et al. [517] proposed enhancing data quality by simultaneously generating robot operations and code snippets to verify task success. Another line of research innovatively applies LLMs in reinforcement learning to program reward functions [519, 509, 520], guiding robots with greater efficiency than human experts. Text2Reward [521] demonstrates that LLM can produce interpretable dense reward functions for robotic manipulation tasks and allow iterative refinement with human feedback.

Elevating Intelligent Automation. Beyond the extensively discussed robotics, the paradigm of interacting with the environment through code generation also plays a pivotal role in elevating the capabilities of various intelligent automation systems. In digital device assistants, code serves as the natural choice for controlling interactions between the agent and digital environment [522, 300]. Gur et al. [523] designed a web agent that acts on websites via generated Python programs by PaLM [34]. CodeACT [524] introduced unifying agent actions through code and integrated LLM with Python interpreter. GAIA [525] constructed a benchmark for general AI assistants, encompassing tasks involving LLMs answering questions through programming. Recently released OS agent OS-Copilot [526] generates executable code for operating computers, enabling humans to interact with operating systems through natural language instruction. Contemporary progress in gaming agents has explored the use of code for perceiving the environment and facilitating action [527, 528]. For instance, Vogayer [529] employs programming for interactions with Minecraft and maintaining an ever-growing skill library of executable code, aiming to foster embodied lifelong embodied agents. Beyond these, code models are also widely applied in autonomous driving [530, 531, 532], automated data processing [533, 534] and multimodal tasks [445, 535].

Although code interfaces offer a superior method for planning and interacting with the environment for agents compared to natural language interfaces, these approaches still largely rely on predefined APIs or skills. Intelligent systems require ongoing exploration to boost their generalization across diverse tasks in the real physical world.

6.4 Advancing AI4Science Research

The progress in scientific fields can be greatly facilitated by advancements in code generation and the use of symbolic languages, particularly in the domains of mathematical proof, chemistry, and biology. These advancements play a crucial role in driving innovation and pushing the boundaries of scientific understanding. The following paragraphs will outline these contributions in each domain.

Automating Theorem Proving. Formal theorem proving is a discipline that necessitates finding proofs for given conjectures articulated in structured, formal statements governed by the principles of logic. Traditional formal theorem proving called upon human experts to meticulously convert mathematical concepts into formal statements, which could then be verified using interactive theorem provers (ITP) like Isabelle [536] and Lean [537]. Each formal statement is similar to a code statement, and ITPs are “compilers” specifically designed for math proofs. This process is greatly accelerated by utilizing formal statements generators that generate single-step proofs that ITPs then verify. A premise is often directly selected based on language modeling statistics [538, 539, 540], while Leandojo [541] uses a retrieval-
augmentation generation pipeline where LLM directly generate proof step based on retrieved premises. Each step is then checked for accuracy by ITPs before progressing to the next, with the ultimate objective being the completion of the proof. To optimize this procedure, sophisticated search algorithms are employed that identify and develop promising premises with the potential to culminate in successful proofs. [542, 543]. A distinctive work in this field is AlphaGeometry [544], which uses a language model to generate auxiliary construction for geometric proofs and then solve the newly constructed problem with symbolic solver [545]. Furthermore, recent research [546, 547, 548, 549] has explored the use of language models with potent code generation capabilities to directly produce entire proofs that could be further refined by ITPs.

Code generation has demonstrated its potential in proving existing mathematical theorems yet recent works demonstrate its potential in solving open mathematical questions. AlphaTensor [550] tackles matrix decomposition by producing tensor decomposition statements and employing tree search to identify the most efficient solution. Similarly, FunSearch [551] addresses combinatorial problems such as cap sets and online bin packing through program generation and optimal selection.

**Catalyzing Biochemistry Discoveries.** Code generation has emerged as a transformative force in biochemistry research, significantly enhancing the efficiency and effectiveness of scientific investigations in the field [552]. This can be largely attributed to the wealth of open-source tools and code packages now available, including tools for processing biochemical sequences [553, 554, 555], accessing databases [556, 557, 558], and analyzing biochemical properties [559, 560]. To this end, Dias and Rodrigues [561], Bran et al. [562] pioneer in using LLMs to generate code APIs that use these tools for automating the chemical research pipeline. Ma et al. [563] have proposed retrieval APIs to expedite the analysis of protein sequences. Tang et al. [565] specifically designed to assess the capability of LLMs in generating bioinformatics code, marking a significant step towards standardized evaluation in this domain. In drug discovery, innovative approaches [564, 565, 566] to employ tool-using language models for automatically editing and optimizing molecular structures for therapeutic purposes – a crucial phase in drug development. Beyond the generation of APIs for existing biochemical tools, Steiner et al. [567] and Rauschen et al. [568] have also introduced novel chemical programming languages designed to automate the synthesis of chemical compounds. These progresses represent a leap forward in our ability to program and execute complex synthetic chemistry with precision and scalability.

◊ **Takeaway:** (1) Coding assistants have revolutionized software engineering workflows by significantly enhancing programming efficiency and code quality. Further, the ongoing evolution towards fully automated software development ecosystems represents a leap towards utilizing intelligent code agents to alleviate humans from labor-intensive development tasks. (2) The evolution of the CodeLLMs has significantly broadened the scope of database interaction, thereby unlocking the potential of multi-turn retrieval in more generalized domains. Further, the rise of code intelligence has also motivated more research into automating and accelerating real-world data science workflows. (3) The code-centric paradigm orchestrates perception, decision-making, action, and feedback for intelligent agents to tackle complex tasks in real-world environments. With their potent reasoning capacity and efficient interaction, these models establish a strong foundation for developing agent systems capable of navigating the highly variable physical world. (4) Code models that can wield mathematical proof language and scientific tools have revolutionized the AI4Science field by advancing towards autonomous scientific discovery, matching human mathematicians in writing proofs, and chemists in analyzing compounds. The current languages used for AI4Science often differ from typical PLs; therefore, adapting code models to novel symbolic languages is essential for their further application in science.

### 7 Opportunities and Future Directions

Thus far, we have extensively reviewed and discussed the advancements in code intelligence, encompassing tasks, models, and applications, aiming to provide a comprehensive view and bring interested researchers up to speed with this field. In what follows, we highlight several promising directions for future research that are ripe for contribution.

#### 7.1 Beyond Transformer Architecture

Since the rise of transformer architecture [26], it has maintained a dominant position in NLP [176], and most of the models we discuss in this paper are also based on this. Nowadays, with the emergence of diffusion models for controllable text generation generation [569, 570, 571], it has also gradually been applied in the NL2Code field. CodeFusion [572] has recently pioneered the application of diffusion models in code-related tasks, achieving stunning results on tasks for Python, Bash, and conditional formatting [573], surpassing the performance of LLMs such as StarCoder and CodeT5+ with only a 75M size. As we have discussed in Section 2, utilizing specially tailored architectures or modeling for specific tasks separately remains a path worth exploring.

#### 7.2 Renaissance in Utilizing Code Features

The most notable difference between code and natural language is its structured nature, which was widely leveraged in the pre-training [181], fine-tuning [61, 574], evaluations [575], explainability [62] of models in the Pre-LLM era. Yet, in the current landscape dominated by LLMs, the utilization of structural information has diminished. We believe the main reasons are the incompatibility of most LLM training pipelines with modalities other than text tokens, and the cost required to extract code features (such as data flow graphs and abstract syntax trees) from massive training data. We hold the view that finding appropriate strategies to integrate structural information into the CodeLLM training process is worth exploring. Beyond training, as previously discussed in Section 3.3 we believe that structural information can still help us better understand the behavior of models. Recently we have also witnessed some researchers starting to utilize lexical properties for enhanced code generation [576] or structural information to assess the impact of code complexities on program-aided reasoning [420]. We are convinced that a revitalization of using code structure will significantly contribute to the ongoing advancement of code intelligence.
7.3 Repo-Level Code Understanding and Generation

Following the surge in the popularity of CodeLLMs, their applications have primarily focused on individual functions or files. However, in real-world software engineering, developers often need to consider the relationships between different files and functions within a code repository. When extending the tasks discussed in Section 2.2 to broader scenarios, such as completing or repairing a code snippet within a repository [577]. The capability of models to invoke variables and functions from other files remains underexplored. This may necessitate the integration of code understanding, cross-file dependencies understanding, and retrieval-augmented generation [578]. Also, given the potential complexity of code repositories, this may require research into how to expand CodeLLMs’ efficient context window [579] and their ability to understand cross-file dependencies. Therefore, repository-level code generation and understanding represent a direction of both practical and research significance, with production environments also serving as a sustainable and challenging testbed for future models.

7.4 Towards Holistic and Reliable Evaluations

Reliable and comprehensive benchmarking is a perennial topic in language model research, and the same applies to evaluations of language modes for code [114, 580]. Current mainstream evaluation methods primarily rely on code execution, during which the security, diversity, and readability of the model-generated code are often overlooked [581]. Additionally, researchers have pointed out that some benchmarks, represented by APPS [352] derived from competition platforms like Codeforces, are likely to have frequently appeared in public repositories [36], consequently leading to models “remembering” the potential solutions to these problems during the pre-training phase. The hand-written HumanEval dataset also faces inevitable data leakage or contamination issues [582] as the training corpus expands. In pursuit of more reliably evaluating CodeLLMs’ performance, and to consider the naturalness, robustness, and lexical diversity of the generated code, fair and dynamic comprehensive benchmarking awaits further exploration.

7.5 Interleaved Planning and Code-Driven Reasoning

As discussed in Section 5.1, Program-aided Language Models [397] and Program of Thoughts [398] have demonstrated efficacy in numerical-related tasks, significantly outperforming some NL-centric paradigms [333]. However, the success of such a paradigm hinges on the premise that a problem can be reliably decomposed into multiple lines of code for resolution. Compared to NL-centric reasoning, program-aided strategies are anticipated to push the boundaries of human intelligence. Mirroring the human approach to tackling complex challenges, more demanding problems (e.g., Olympic competition problems [544]) require ongoing planning and reasoning. Each phase in such scenarios necessitates iterative planning, taking into account the problem at hand and the current state, while concurrently generating code to aid in the solution part. This synergistic blend of interleaved planning and program-aided reasoning provides a strategic advantage that is worthy of exploration.

7.6 A Closer Look at Tokenizer Dynamics

Tokenization has historically played a pivotal role in language modeling [583], where the choice of tokenizer can significantly influence a model’s downstream performance [584] and multilingual capabilities [585]. Recent studies have also identified its substantial impact on the generalization capabilities of LLMs [586, 587]. Unlike natural languages, code operates under more rigid syntactic rules and structures, with meanings heavily reliant on the precision of these elements. Earlier research has uncovered that tokenization granularity exerts a non-trivial impact on code-related tasks [575]. So, we have reason to doubt that: when processing code with general tokenization methods like BBPE [588], as employed by models such as CodeLLaMA, there is a risk of disrupting this structure due to over-tokenization. Hence, determining an effective approach to tokenization without compromising code structure and semantics poses a challenge. One potential research direction involves developing tokenizers capable of recognizing and preserving code structures (discussed in Section 2.1)—such as keywords, function definitions, and control flow statements—or exploring better strategies that grasp a deeper understanding of code semantics.

7.7 Efficient Methods for CodeLLMs

As the development of CodeLLMs progresses, beyond striving for performance enhancements, the pursuit of model efficiency emerges as another critical direction for in-depth investigation. As for training, whether starting from scratch or conducting additional training on code using general LLMs, the process proves exorbitantly expensive. Although parameter-efficient methods [589] like prefix-based strategies [590, 591, 592] and LoRA [593] significantly reduce resource consumption, applying them directly to language models for code [594] has been shown to noticeably impact the performance of code-related tasks [595]. Thus, identifying new efficient training techniques for code is worth exploring, and it necessitates finding the most suitable strategies and trade-offs between cost and performance across models of different scales [596]. For deployment, due to the prevalent state-of-the-art CodeLLMs being powerful yet cumbersome, researchers have begun to emphasize the application [597] and optimization of CodeLLMs for running within resource-constrained environments. As discussed in Section 4.1, this particularly includes enabling these models to function offline on consumer-level devices without the reliance on GPUs [303, 304]. Additionally, to satisfy more diverse demands, CodeLLMs are faced with the strategic decision of scaling up [598, 599] or scaling out [600]. Alternatively, acceleration can also be achieved through methods such as structured pruning [601] and distillation [602] akin to approaches employed in general LLMs.

7.8 Further Expansion of Multilingual Capability

“Nobody should call themselves a professional if they only knew one language.” — Bjarne Stroustrup. Multilingualism has long been a staple in NLP research, with LLMs proving capable of holding multilingual capabilities [603, 604]. However, in the realm of code intelligence, the exploration
of mastering multiple programming languages is a relatively recent development. Although there has been research targeting multilingual scenarios [290], significant performance variations across different programming languages by CodeLLMs are evident from various leaderboards (e.g., Big Code Models Leaderboard [22]). This disparity is largely attributed to the uneven distribution of corpora which is dominated by popular languages such as Python and Java. Consequently, collecting more premium data for less prevalent languages (e.g., TypeScript, Kotlin, Scala), exploring data augmentation [605], conducting specialized training, distilling language agnostic representation [606] or even transferring knowledge between different languages [607, 608, 609], presents a worthwhile direction for research. Furthermore, multilingual models have also been shown to be more robust to prompt perturbation and excel in code summarization [353], potentially contributing to an enhanced overall capability of CodeLLMs. It’s also noteworthy that current mainstream multilingual benchmarks [161] are mostly conversions from Python datasets, overlooking the unique characteristics of different languages. Hence, developing new benchmarks that cater to specific language features is another avenue worth exploring.

7.9 Copyright Challenges Faced by Coding Assistants

With the expansion of the open-source community and the rise of coding assistant tools (discussed in Section 6.1), a series of ethical and security concerns regarding the distribution of source code have arisen, such as unauthorized use of copyrighted code, distributions without proper licenses, or exploitation of code for malicious purposes. To circumvent potential legal and copyright disputes, one viable strategy is the watermarking of source code protected by licenses such as GPL. Furthermore, the use of copyrighted content detection in CodeLLMs’ training corpora [610] can also safeguard against the unintended output of such code. Notably, watermarking and detection for code diverge from that for natural language due to the impact of obfuscation on the semantics of variable names, presenting an intriguing area for further investigation. Unlike watermarking [611] and content detection [612] applied to natural language, code presents unique challenges. As previously discussed, the semantics of variable names in code can be significantly affected by obfuscation, making it a more challenging and worthwhile direction for further investigation.

8 Conclusion

In this paper, we present a systematic review of the entire evolutionary trajectory of code intelligence, offering a comprehensive examination from the nascent application of deep neural networks on source code to the breakthroughs made in the LLM era. Throughout this process, we have delved into the interconnections among research across different periods, engaging in detailed discussions and analyses centered on the paradigm shifts in models, tasks, and applications. Moreover, we explore the synergies between code learning and other facets of machine intelligence, along with both long-standing and emergent applications in the real world. Bearing in mind the developmental pathway we have witnessed and insights garnered from our discussions, we also identify several promising directions for future research in code intelligence. Given the concurrent advancements in language models and the evolving needs of software development, we are confident that this domain will continue to flourish in the forthcoming years. It is our aspiration that the literature review, discussions, experiences, and resources provided in this survey paper will boost the community’s future exploration.

APPENDIX A
ADDITIONAL RESOURCES

A.1 Reading Lists

In the NCISurvey project related to this paper, we have prepared a curated reading list for our readers, encompassing a diverse array of research areas within the realm of code intelligence.

A.2 Recreating the Figures

We used the template from LLMsPracticalGuide maintained by Yang et al. [613] to construct Figure 3. Our primary criteria for determining the placement of models on specific branches of the tree were based on the release times of the models mentioned in Sections 3 and Section 4, as well as the relationships between the models. Relevant resources will be uploaded to our project for further reference and exploration.

APPENDIX B
MORE BENCHMARKS

Given the constraints on space, Table 1 and Table 5 feature only a subset of the representative tasks in code downstream applications and NL2Code. To offer a more thorough overview, Table 6, 7, 8 and 9 systematically present a more exhaustive review on datasets, organized by each specific task category.

APPENDIX C
DETAILED OF CODEPTM TRAINING OBJECTS

Due to space limitations, we adopt abbreviations for the training objectives of CodePTMs in the overview presented in Table 2, section 3.1. Here we provide detailed explanations for them in Table 10.

APPENDIX D
RELATED RESEARCH

Before the rise of employing transformer-based models in code intelligence, Han et al. [21] evaluates the performance of eight code embedding models on classic code-related tasks. During the period when pre-trained language models are dominant in NCI research, Wu et al. [614] categorize CodeLMs from the perspectives of code structures. Xu and Zhu [615] revisit the training processes and downstream applications of CodePTMs, and Xu et al. [286] conduct a
systematic evaluation of mainstream models in this sphere. Meanwhile, several studies [616, 617] have discussed the application of CodePTMs from the perspective of software engineering. Amidst the burgeoning trend of LLMs, Zan et al. [4] provides an empirical summary of the field’s development from the standpoint of NL2Code. Hou et al. [618] and She et al. [619] discuss the application of CodeLLMs in SE, along with the opportunities and challenges. In recent studies, Zhang et al. [5] conduct a retrospective study of language models for coding from the unified viewpoints of NLP and SE. Wan et al. [218] delve into an exploration of benchmarks and toolkits for neural code intelligence. For the synergies between code learning and AI agents, Yang et al. [506] investigate the role of code corpora in augmenting the capabilities of LLM-based agents.

APPENDIX E
DATA SOURCES

E.1 Evaluations
The primary data sources for Figure 5 are BigCode’s Evaluation Harness [620] and the BigCode Models Leaderboard. Similar evaluation results can also be found at OpenCompass [621]. As for the results reported in the Figure 6, we access GPT-3.5-Turbo and Claude-1 through paid APIs. For open-source LLMs, we utilize CodeLLaMA-Instruct-13B and LLaMA2-Chat-13B under the greedy search setting. Both GSM8K and MATH evaluations are conducted with 3-shot promptings from Gao et al. [397]. WikiSQL [486] and FOLIO [622] are evaluated under the 1-shot prompting, modified from Xu et al. [302].

E.2 Publication Trends Data and Milestones
The construction of Figure 1 was achieved through the use of the ArXiv advanced search23, with the calculation based on an exact match by querying a group of keywords (e.g., code representation, code generation) in the title or abstract. Detailed information on this is provided within the NCIISurvey project.

As for the selection of milestones featured in Figure 2, Qiushi Sun, Zhirui Chen, and Zhangyue Yin each selected what they considered to be representative milestones based on their modeling approaches, release time, the citations of papers/preprints, and their influence within the community. The final selection was made by identifying the intersection of their choices.

Beyond statistics, we can observe that the publication venues for research papers on code intelligence have undergone a significant transformation over the past decades. In earlier research, publications were primarily confined to ACM/IEEE Conferences like ASE and ICSE, or journals like IEEE TSE. However, with the rise of the neural approach, research began shifting towards machine learning conferences, such as NeurIPS. Subsequently, with the rapid advances in language modeling, code-related topics have also become frequent subjects at NLP conferences, exemplified by the *ACL Conferences. These changes in publication venues reflect a blurring of the boundaries between previously distinct research areas, demonstrating the interdisciplinary nature of code intelligence research today.

ACKNOWLEDGEMENT
Thanks to everyone who has contributed to this paper. We extend our gratitude to Xuesong Lu for meticulously reviewing our paper and providing helpful feedback, as well as assisting us in identifying and incorporating previously overlooked literature into our study.

TABLE 6: A more comprehensive collection of code-related benchmarks extended from Table 1. “# PLs” denotes the number of programming languages each benchmark covers.

<table>
<thead>
<tr>
<th>Task</th>
<th>Dataset</th>
<th>Date</th>
<th># PLs</th>
<th>Description</th>
<th>Eval. Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clone Detection</td>
<td>POJ-104 [16] [link]</td>
<td>2014</td>
<td>1</td>
<td>a program classification dataset of 52K C/C++ programs</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BigCloneBench [108] [link]</td>
<td>2015</td>
<td>1</td>
<td>a clone detection dataset of 8M Java validated clones</td>
<td>F1 score</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLCDSA [109] [link]</td>
<td>2019</td>
<td>3</td>
<td>a cross-language clone dataset of more than 78K solutions</td>
<td>F1 score</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CGD [623] [link]</td>
<td>2018</td>
<td>2</td>
<td>a dataset focuses on two types of vulnerabilities in C/C++</td>
<td>F1/Precision</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Draper VDISC [624] [link]</td>
<td>2018</td>
<td>2</td>
<td>a vast dataset of open-source functions in C/C++</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SySeVR [625] [link]</td>
<td>2018</td>
<td>2</td>
<td>a dataset of 126 types of vulnerabilities in C/C++</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Devign [78] [link]</td>
<td>2019</td>
<td>1</td>
<td>a dataset of vulnerable C functions</td>
<td>F1/Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GREAT [85] [link]</td>
<td>2019</td>
<td>1</td>
<td>a dataset extracted from the ETH Py150 dataset</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MVD [626] [link]</td>
<td>2020</td>
<td>2</td>
<td>a dataset that contains 181K pieces of code from 33K C/C++ programs</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ReVeal [627] [link]</td>
<td>2020</td>
<td>1</td>
<td>a vulnerability dataset curated from two real-world projects (Chromium and Debian)</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BigVul [628] [link]</td>
<td>2020</td>
<td>2</td>
<td>A dataset built for AI-based vulnerability detection methods</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D2A [629] [link]</td>
<td>2021</td>
<td>2</td>
<td>A dataset retrieved form the most downloaded packages in the Python package index</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PyPIBugs [88] [link]</td>
<td>2021</td>
<td>1</td>
<td>a dataset of almost 1.8K real-life incorrect Python program submissions</td>
<td>F1/Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVEFixes [630] [link]</td>
<td>2021</td>
<td>27</td>
<td>a dataset built through mining 105K commits from 134 JavaScript projects</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CrossVul [110] [link]</td>
<td>2021</td>
<td>&gt;40</td>
<td>a dataset of 27K files containing vulnerabilities</td>
<td>F1/Pass Rate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DiverseVul [111] [link]</td>
<td>2023</td>
<td>2</td>
<td>a dataset consists of almost 1.1K bugs and patches</td>
<td>Pass Rate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VulnPatchPairs [631] [link]</td>
<td>2023</td>
<td>1</td>
<td>a dataset contains 26.2K C functions</td>
<td>F1/Pass Rate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VulBench [632] [link]</td>
<td>2023</td>
<td>1</td>
<td>a database and extensible framework providing real Java bugs</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Defects4J [640] [link]</td>
<td>2014</td>
<td>1</td>
<td>a benchmark consisting of 1K defects in 15 C programs</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ManyBugs [633] [link]</td>
<td>2015</td>
<td>1</td>
<td>a benchmark build through mining 105K commits from 134 JavaScript projects</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BugAID [634] [link]</td>
<td>2016</td>
<td>1</td>
<td>a set of 6971 erroneous C programs written by students for 93 programming tasks</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DeepFix [635] [link]</td>
<td>2017</td>
<td>1</td>
<td>A dataset built for AI-based vulnerability detection methods</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Codeflaws [636] [link]</td>
<td>2017</td>
<td>1</td>
<td>A collection of C programs with 3.9K defects</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>QuixBugs [637] [link]</td>
<td>2017</td>
<td>2</td>
<td>a multi-language benchmark (40 bugs in both Python and Java)</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bugs.jar [638] [link]</td>
<td>2018</td>
<td>1</td>
<td>a benchmark consisting of 1.1K bugs and patches</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bears [639] [link]</td>
<td>2019</td>
<td>1</td>
<td>an extensible bug benchmark for automatic repair studies in Java</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BugSJS [640] [link]</td>
<td>2019</td>
<td>1</td>
<td>a benchmark of 453 real JavaScript bugs</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BugSwarm [641] [link]</td>
<td>2019</td>
<td>2</td>
<td>a benchmark of 3K fail-pass pairs, in Java and Python</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ManyStubs4J [642] [link]</td>
<td>2019</td>
<td>1</td>
<td>a dataset of almost 1.8K real-life incorrect Python program submissions</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Refactory [643] [link]</td>
<td>2019</td>
<td>1</td>
<td>a dataset consists of 55K code reviews and associated code changes</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Review4Repair [644] [link]</td>
<td>2020</td>
<td>1</td>
<td>a dataset consists of 55K code reviews and associated code changes</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BugsInPy [645] [link]</td>
<td>2020</td>
<td>1</td>
<td>a benchmark consists of 493 real bugs from 17 real-world Python programs</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TFix [646] [link]</td>
<td>2021</td>
<td>1</td>
<td>a benchmark consists of 52 different error types reported by a popular static analyzer</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Megadiff [647] [link]</td>
<td>2021</td>
<td>1</td>
<td>A dataset of 600K Java source code changes categorized by different size.</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FixJS [649] [link]</td>
<td>2021</td>
<td>1</td>
<td>a dataset containing bug-fixing information of 2M commits.</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TypeBugs [650] [link]</td>
<td>2021</td>
<td>1</td>
<td>a benchmark dedicated to repairing type errors in Python.</td>
<td>Precision</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xCodeEval [651] [link]</td>
<td>2023</td>
<td>11</td>
<td>an executable benchmark of 450K small buggy/fixed program pairs</td>
<td>Pass Rate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RunBugRun [652] [link]</td>
<td>2023</td>
<td>8</td>
<td>an executable multilingual benchmark consisting of 25M document-level coding examples expanding the HumanEval benchmark to 3 coding tasks across 6 languages</td>
<td>Acc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HumanEvalPack [306] [link]</td>
<td>2023</td>
<td>6</td>
<td>a benchmark offers a blend of CTF challenges and real-world CVE vulnerabilities</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ErrorCLR [87] [link]</td>
<td>2023</td>
<td>2</td>
<td>a multilingual benchmark of similar buggy programs</td>
<td>Pass Rate</td>
<td></td>
</tr>
</tbody>
</table>
TABLE 7: A more comprehensive collection of code-related benchmarks extended from Table 1. “#PLs.” denotes the number of programming languages each benchmark covers (Cont’d). MRR (Mean Reciprocal Rank) indicates the average rank of correct answer choices. FRank (First Rank) measures the proportion of correct answers ranked first. CR (Compilation Rate) denotes the percentage of code snippets that compile successfully. NIM (Next Identifier Match) is the accuracy in predicting the next identifier or variable name. ISM (Identifier Sequence Match) measures the coherence of generated identifier sequences. PM (Prefix Match) means the alignment of generated code snippets with expected prefixes.

<table>
<thead>
<tr>
<th>Task</th>
<th>Dataset</th>
<th>Date</th>
<th># PLs</th>
<th>Description</th>
<th>Eval. Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Search</td>
<td>CodeSearchNet [113] [link]</td>
<td>2019</td>
<td>6</td>
<td>a multilingual dataset of 6M functions and query-like natural language</td>
<td>MRR</td>
</tr>
<tr>
<td></td>
<td>AdvTest [114] [link]</td>
<td>2021</td>
<td>1</td>
<td>a Python code search dataset filtered from CodeSearchNet</td>
<td>MRR</td>
</tr>
<tr>
<td></td>
<td>WebQueryTest [114] [link]</td>
<td>2021</td>
<td>1</td>
<td>a test set of Python code search including 1K query-code pairs</td>
<td></td>
</tr>
<tr>
<td>Code Translation</td>
<td>GeeksforGeeks [101] [link]</td>
<td>2020</td>
<td>3</td>
<td>a test set composed of 852 parallel functions for code translations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CodeTrans [114] [link]</td>
<td>2021</td>
<td>2</td>
<td>a C#/Java code translation dataset collected from several public repos</td>
<td>Acc./BLEU</td>
</tr>
<tr>
<td></td>
<td>Avitar [652] [link]</td>
<td>2021</td>
<td>2</td>
<td>a collection of 9.5K programming problems and solutions written in Java and</td>
<td>Acc./BLEU</td>
</tr>
<tr>
<td></td>
<td>CoST [115] [link]</td>
<td>2022</td>
<td>7</td>
<td>a multilingual Code Snippet Translation dataset</td>
<td>BLEU/CodeBLEU</td>
</tr>
<tr>
<td></td>
<td>XLCoST [653] [link]</td>
<td>2022</td>
<td>7</td>
<td>a benchmark containing line-grained parallel data from 7 programming languages.</td>
<td>BLEU/CodeBLEU</td>
</tr>
<tr>
<td></td>
<td>xCodeEval [650] [link]</td>
<td>2023</td>
<td>11</td>
<td>an executable multilingual benchmark consisting of 25M document-level coding examples</td>
<td>Bleu/CodeBLEU</td>
</tr>
<tr>
<td></td>
<td>G-TransEval [654] [link]</td>
<td>2023</td>
<td>5</td>
<td>a benchmark suite of 400 code translation pairs between 5 PLs, categorized into 4 levels</td>
<td>Acc./BLEU</td>
</tr>
<tr>
<td></td>
<td>CodeTransOcean [116] [link]</td>
<td>2023</td>
<td>4</td>
<td>a large-scale code translation benchmark with three novel multilingual datasets</td>
<td>EM/BLEU/CodeBLEU</td>
</tr>
<tr>
<td>Code Retrieval</td>
<td>StaQC [655] [link]</td>
<td>2018</td>
<td>2</td>
<td>a dataset of around 148K Python and 120K SQL domain question-code pairs</td>
<td>MRR</td>
</tr>
<tr>
<td></td>
<td>DeepCS [146] [link]</td>
<td>2018</td>
<td>1</td>
<td>a large scale codebase collected from GitHub</td>
<td>FRank/SuccessRate/</td>
</tr>
<tr>
<td></td>
<td>CodeSearchNet [113] [link]</td>
<td>2019</td>
<td>6</td>
<td>a multilingual dataset of 6M functions and query-like natural language</td>
<td>Precision</td>
</tr>
<tr>
<td></td>
<td>CosBench [657] [link]</td>
<td>2020</td>
<td>1</td>
<td>a dataset that consists of 1K projects and 52 code-independent natural-language queries</td>
<td>MRR</td>
</tr>
<tr>
<td></td>
<td>SO-DS [658] [link]</td>
<td>2021</td>
<td>1</td>
<td>a dataset that consists of 12K snippets of Python</td>
<td>MRR/Recall</td>
</tr>
<tr>
<td></td>
<td>JBJava [659] [link]</td>
<td>2021</td>
<td>1</td>
<td>a dataset consisting of 24K repositories with 5 PLs functions in Java</td>
<td>MRR/SuccessRate</td>
</tr>
<tr>
<td></td>
<td>AdvTest [114] [link]</td>
<td>2021</td>
<td>1</td>
<td>a Python code search dataset filtered from CodeSearchNet</td>
<td>MRR</td>
</tr>
<tr>
<td></td>
<td>WebQueryTest [114] [link]</td>
<td>2021</td>
<td>1</td>
<td>a test set of Python code search 1K query-code pairs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CoSQA [660] [link]</td>
<td>2021</td>
<td>1</td>
<td>a dataset of web queries for code search and question answering.</td>
<td>Acc.</td>
</tr>
<tr>
<td>Code Completion</td>
<td>GitHub Java Corpus [2] [link]</td>
<td>2013</td>
<td>1</td>
<td>a giga-token corpus of Java code from a wide variety of domains</td>
<td>EM/Acc./Edit sim</td>
</tr>
<tr>
<td></td>
<td>Py150 [117] [link]</td>
<td>2016</td>
<td>1</td>
<td>a corpus of Python programs from GitHub</td>
<td>EM/Acc./Edit sim</td>
</tr>
<tr>
<td></td>
<td>JS150 [117] [link]</td>
<td>2016</td>
<td>1</td>
<td>a corpus of JavaScript programs from GitHub</td>
<td>EM/Acc./Edit sim</td>
</tr>
<tr>
<td></td>
<td>DotPrompts [661] [link]</td>
<td>2023</td>
<td>1</td>
<td>a dataset of real-world open-source Java projects completion with their environments</td>
<td>CR/NIM/ISMP手术</td>
</tr>
<tr>
<td></td>
<td>LCC [118] [link]</td>
<td>2023</td>
<td>3</td>
<td>a benchmark that focuses on code completion with long code context</td>
<td>EM/Edit Sim</td>
</tr>
<tr>
<td></td>
<td>RepoBench [662] [link]</td>
<td>2023</td>
<td>2</td>
<td>a dataset anchored on combining correlated commits and issues</td>
<td></td>
</tr>
<tr>
<td>GitHub</td>
<td>unnamed [141] [link]</td>
<td>2017</td>
<td>1</td>
<td>a dataset of 509K labeled diff files of Java</td>
<td>Acc.</td>
</tr>
<tr>
<td></td>
<td>CommitGen [121] [link]</td>
<td>2017</td>
<td>4</td>
<td>a multilingual dataset collected from open projects on GitHub</td>
<td>BLEU</td>
</tr>
<tr>
<td></td>
<td>LNEGen [140] [link]</td>
<td>2018</td>
<td>1</td>
<td>a cleaner version of CommitGen</td>
<td>BLEU</td>
</tr>
<tr>
<td></td>
<td>PtrGNCMsg [663] [link]</td>
<td>2019</td>
<td>1</td>
<td>a dataset of diffs and manual commit messages from Java projects in GitHub</td>
<td>BLEU/ROUGE</td>
</tr>
<tr>
<td></td>
<td>CoDiSum [142] [link]</td>
<td>2019</td>
<td>1</td>
<td>a cleaner version of Jiang and McMillan [141]'s dataset</td>
<td>BLEU/METEOR</td>
</tr>
<tr>
<td></td>
<td>ATOM [664] [link]</td>
<td>2019</td>
<td>1</td>
<td>a dataset dataset crawled from 56 popular Java repositories</td>
<td>BLEU</td>
</tr>
<tr>
<td></td>
<td>BLECommit [122] [link]</td>
<td>2021</td>
<td>6</td>
<td>a multilingual dataset of code modification with commit messages in GitHub</td>
<td>BLEU</td>
</tr>
<tr>
<td></td>
<td>MCMID [137] [link]</td>
<td>2021</td>
<td>5</td>
<td>a large-scale, information-rich, and multi-language commit message dataset</td>
<td>B-Moses/B-Norm/B-CC</td>
</tr>
<tr>
<td></td>
<td>CoRec [665] [link]</td>
<td>2021</td>
<td>1</td>
<td>a large-scale dataset crawled from 10K popular Java repositories in GitHub</td>
<td>BLEU</td>
</tr>
<tr>
<td></td>
<td>ExGroFi [666] [link]</td>
<td>2023</td>
<td>1</td>
<td>a dataset anchored on combining correlated commits and issues</td>
<td>BLEU/ROUGE/CIDER</td>
</tr>
<tr>
<td></td>
<td>CommitChronicle [667] [link]</td>
<td>2023</td>
<td>20</td>
<td>a dataset containing 30.7M commits across 20 programming languages</td>
<td>B-Norm/Edit Sim/EM</td>
</tr>
<tr>
<td></td>
<td>SW-bench [123] [link]</td>
<td>2023</td>
<td>1</td>
<td>a dataset of 2.2K software engineering problems with pull requests</td>
<td>Resolve Rate/Recall</td>
</tr>
<tr>
<td></td>
<td>CommitBenh [668] [link]</td>
<td>2024</td>
<td>6</td>
<td>a reproducible and privacy- and license-aware benchmark for commit message generation</td>
<td>BLEU/ROUGE/ROUGE</td>
</tr>
<tr>
<td></td>
<td>DevBench [669] [link]</td>
<td>2024</td>
<td>4</td>
<td>a benchmark to evaluate LLMs across various stages of the software development lifecycle</td>
<td>pass/fail</td>
</tr>
</tbody>
</table>
TABLE 8: A more comprehensive collection of code-related benchmarks extended from Table 1. “# PLs” denotes the number of programming languages each benchmark covers (Cont’d).

<table>
<thead>
<tr>
<th>Task</th>
<th>Dataset</th>
<th>Date</th>
<th># PLs</th>
<th>Description</th>
<th>Eval. Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Summarization</td>
<td>CODE-NN [119] [link]</td>
<td>2016</td>
<td>6</td>
<td>(title, query) pairs from StackOverflow</td>
<td>BLEU</td>
</tr>
<tr>
<td></td>
<td>DeepCom [670] [link]</td>
<td>2018</td>
<td>1</td>
<td>a large-scale Java corpus built from 9.7K open source projects from GitHub</td>
<td>BLEU</td>
</tr>
<tr>
<td></td>
<td>TL-CodeSum [120] [link]</td>
<td>2018</td>
<td>6</td>
<td>a dataset of 69K pairs of code and summary</td>
<td>BLEU</td>
</tr>
<tr>
<td></td>
<td>CodeSearchNet [113] [link]</td>
<td>2019</td>
<td>6</td>
<td>a multilingual dataset of 6M functions and query-like natural language</td>
<td>MRR</td>
</tr>
<tr>
<td></td>
<td>HumanEvalPack [306] [link]</td>
<td>2023</td>
<td>6</td>
<td>a benchmark expanding HumanEval to 3 coding tasks across 6 languages</td>
<td>BLEU</td>
</tr>
<tr>
<td>Code Debug</td>
<td>QuixBugs [112] [link]</td>
<td>2017</td>
<td>2</td>
<td>a multi-language benchmark (40 bugs in both Python and Java)</td>
<td>Pass Rate</td>
</tr>
<tr>
<td></td>
<td>EvalGPTFix [671] [link]</td>
<td>2023</td>
<td>1</td>
<td>a benchmark containing 151 pairs of bugs and fixes in Java</td>
<td>Pass Rate</td>
</tr>
<tr>
<td></td>
<td>DebugBench [672] [link]</td>
<td>2024</td>
<td>3</td>
<td>a debugging benchmark containing 4K instances of four major bug categories</td>
<td>Pass Rate</td>
</tr>
<tr>
<td>Question Answering</td>
<td>CodeQA [170]</td>
<td>2021</td>
<td>2</td>
<td>a free-form QA dataset for Java/Python code comprehension</td>
<td>BLEU/EM/F1</td>
</tr>
<tr>
<td></td>
<td>CodeQueries [673]</td>
<td>2024</td>
<td>1</td>
<td>a dataset to evaluate models to answer semantic queries over python code</td>
<td>EM/Acc./Recall</td>
</tr>
<tr>
<td>Text2SQL</td>
<td>ATIS [485] [link]</td>
<td>1994</td>
<td>1</td>
<td>a dataset containing 3K utterances reserved for testing</td>
<td>Acc.</td>
</tr>
<tr>
<td></td>
<td>WikiTQ [487] [link]</td>
<td>2015</td>
<td>1</td>
<td>a dataset of 22K complex questions on Wikipedia tables</td>
<td>Acc.</td>
</tr>
<tr>
<td></td>
<td>WikiSQL [486] [link]</td>
<td>2017</td>
<td>1</td>
<td>a dataset of 80K hand-annotated questions and SQL queries from Wikipedia</td>
<td>Acc./F1</td>
</tr>
<tr>
<td></td>
<td>Spider [164] [link]</td>
<td>2018</td>
<td>1</td>
<td>a dataset consisting of 10K questions and 5.6K unique complex SQL queries</td>
<td>Acc./F1</td>
</tr>
<tr>
<td></td>
<td>SPARC [165] [link]</td>
<td>2019</td>
<td>1</td>
<td>a dataset consisting of 4.2K coherent question sequences</td>
<td>Acc.</td>
</tr>
<tr>
<td></td>
<td>Cosql [166] [link]</td>
<td>2019</td>
<td>1</td>
<td>a dataset consisting of 30K+ turns and 10K+ annotated SQL queries</td>
<td>Acc.</td>
</tr>
<tr>
<td></td>
<td>Squall [675] [link]</td>
<td>2020</td>
<td>1</td>
<td>a dataset of 11K English questions with manually created SQL equivalents</td>
<td>Acc.</td>
</tr>
<tr>
<td></td>
<td>KaggleDBQA [676] [link]</td>
<td>2021</td>
<td>1</td>
<td>a cross-domain evaluation dataset of real Web databases</td>
<td>Acc.</td>
</tr>
</tbody>
</table>
TABLE 9: More NL2Code benchmarks extended from Table 5. FGA indicates the F1-score of group accuracy.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Dataset</th>
<th>Date</th>
<th># PLs.</th>
<th>Description</th>
<th>Eval. Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Domain</td>
<td>CONCODE [159] [link]</td>
<td>2018</td>
<td>1</td>
<td>a dataset with 100K examples consisting of Java classes from online code repositories</td>
<td>EM/BLEU Pass Rate/pass@k</td>
</tr>
<tr>
<td></td>
<td>ODEX [549] [link]</td>
<td>2023</td>
<td>1</td>
<td>an open-domain execution-based natural language to Python code generation dataset</td>
<td></td>
</tr>
<tr>
<td>Code Exercise</td>
<td>HumanEval [36] [link]</td>
<td>2021</td>
<td>1</td>
<td>a dataset of 164 handwritten programming problems with unit tests</td>
<td>pass@k</td>
</tr>
<tr>
<td></td>
<td>MBPP [350] [link]</td>
<td>2021</td>
<td>1</td>
<td>a dataset containing 974 short Python programs</td>
<td>pass@k</td>
</tr>
<tr>
<td></td>
<td>MathQA-Python [350] [link]</td>
<td>2021</td>
<td>1</td>
<td>a Python version of the MathQA benchmark contains 23K problems</td>
<td>Acc</td>
</tr>
<tr>
<td></td>
<td>AixBench [677] [link]</td>
<td>2022</td>
<td>1</td>
<td>a code generation benchmark dataset</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIG-Bench [351, 325] [link]</td>
<td>2023</td>
<td>-</td>
<td>a benchmark containing over 12 tasks can be solved by coding</td>
<td>Pass Rate</td>
</tr>
<tr>
<td></td>
<td>CoderEval [678] [link]</td>
<td>2023</td>
<td>2</td>
<td>a dataset consisting of 230 Python and 230 Java code generation tasks</td>
<td>Pass Rate</td>
</tr>
<tr>
<td></td>
<td>CodeApex [580] [link]</td>
<td>2023</td>
<td>1</td>
<td>a bilingual programming evaluation benchmark for large language models</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ClassEval [679] [link]</td>
<td>2023</td>
<td>1</td>
<td>A handcrafted benchmark for evaluating class-level code generation</td>
<td></td>
</tr>
<tr>
<td>Competitions</td>
<td>APPS [352] [link]</td>
<td>2021</td>
<td>1</td>
<td>a benchmark including 10K problems for Python code generation</td>
<td>Acc/pass@k</td>
</tr>
<tr>
<td></td>
<td>CodeContests [273] [link]</td>
<td>2022</td>
<td>3</td>
<td>a dataset of competitive programming problems</td>
<td>pass@k</td>
</tr>
<tr>
<td></td>
<td>TACO [680] [link]</td>
<td>2023</td>
<td>1</td>
<td>a dataset on algorithmic code generation, consisting of 25K training problems and 1K testing problem</td>
<td>pass@k</td>
</tr>
<tr>
<td>Multilingual</td>
<td>MultiPL-E [161] [link]</td>
<td>2022</td>
<td>18</td>
<td>a parallel, multilingual benchmark for natural-language-to-code generation</td>
<td>pass@k</td>
</tr>
<tr>
<td></td>
<td>MCoNaLa [681] [link]</td>
<td>2022</td>
<td>1</td>
<td>a benchmark for code generation from multiple natural languages</td>
<td>BLEU/p@k</td>
</tr>
<tr>
<td></td>
<td>MBXP [353] [link]</td>
<td>2023</td>
<td>12</td>
<td>a benchmark to evaluate code generation models in over 10 programming languages</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MathQA-X [353] [link]</td>
<td>2023</td>
<td>3</td>
<td>a dataset transpiled from the original Python dataset MathQA</td>
<td>Pass Rate/Pass Rate/pass@k</td>
</tr>
<tr>
<td></td>
<td>xCodeEval [650] [link]</td>
<td>2023</td>
<td>11</td>
<td>an executable dataset of 450K small buggy/fixed program pairs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HumanEval-X [290] [link]</td>
<td>2023</td>
<td>4</td>
<td>a benchmark of 164 code problems for evaluating multilingual models</td>
<td></td>
</tr>
<tr>
<td>Data Science</td>
<td>JukiCe [162] [link]</td>
<td>2018</td>
<td>1</td>
<td>a corpus of 1.5M examples with a curated test set of 3.7K instances</td>
<td>EM/BLEU Acc/pass@k</td>
</tr>
<tr>
<td></td>
<td>PlotCoder [682] [link]</td>
<td>2021</td>
<td>1</td>
<td>a dataset of plot samples extracted from the original dev and test splits of JukiCe</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSP [163] [link]</td>
<td>2022</td>
<td>1</td>
<td>a collection of 1.1K problems curated from 306 pedagogical notebooks</td>
<td>pass@k</td>
</tr>
<tr>
<td></td>
<td>ExeDS [683] [link]</td>
<td>2022</td>
<td>1</td>
<td>a dataset of 534 problems for execution evaluation for data science code generation tasks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DS-1000 [326] [link]</td>
<td>2023</td>
<td>1</td>
<td>a code generation benchmark with 1K data science problems spanning 7 Python libraries</td>
<td>p@k</td>
</tr>
<tr>
<td></td>
<td>DAEval [684] [link]</td>
<td>2024</td>
<td>1</td>
<td>a dataset comprising 257 data analysis questions derived from 52 CSV files</td>
<td>(Code)BLEU/EM/Pass Rate/p@k</td>
</tr>
<tr>
<td>Python Libs</td>
<td>PandasEval [199] [link]</td>
<td>2022</td>
<td>1</td>
<td>a dataset consisting of 101 programming problems on Pandas library</td>
<td>pass@k</td>
</tr>
<tr>
<td></td>
<td>NumpyEval [199] [link]</td>
<td>2022</td>
<td>1</td>
<td>a dataset consisting of 101 programming problems on Numpy library</td>
<td>pass@k</td>
</tr>
<tr>
<td></td>
<td>TorchDataEval [354] [link]</td>
<td>2022</td>
<td>1</td>
<td>a dataset with 50 programming problems using the TorchData library</td>
<td>pass@k</td>
</tr>
<tr>
<td>Multi-Turn</td>
<td>MTPB [275] [link]</td>
<td>2023</td>
<td>1</td>
<td>an open benchmark consisting of 115 diverse problem sets that are factorized into multi-turn prompts</td>
<td>Pass Rate</td>
</tr>
<tr>
<td>Command Line</td>
<td>NL2Bash [160] [link]</td>
<td>2018</td>
<td>1</td>
<td>a corpus of 9K English-command pairs, covering over 100 unique Bash utilities</td>
<td>Acc</td>
</tr>
<tr>
<td>AI4Science</td>
<td>BioCoder [355] [link]</td>
<td>2023</td>
<td>2</td>
<td>a benchmark developed to evaluate large language models in generating bioinformatics-specific code</td>
<td>pass@k</td>
</tr>
<tr>
<td>Education</td>
<td>StudentEval [685] [link]</td>
<td>2023</td>
<td>1</td>
<td>a dataset containing 1.7K prompts for 48 problems, written by 80 students</td>
<td>Acc</td>
</tr>
<tr>
<td></td>
<td>COJ2022 [87] [link]</td>
<td>2023</td>
<td>1</td>
<td>a dataset containing 5.9K C programs with semantic errors submitted to 498 different assignments</td>
<td>F1</td>
</tr>
<tr>
<td>Log Parsing</td>
<td>LogHub^[686] [link]</td>
<td>2019</td>
<td>-</td>
<td>a dataset of logs produced by 16 different systems</td>
<td>Acc</td>
</tr>
<tr>
<td></td>
<td>LogHub^[687] [link]</td>
<td>2023</td>
<td>-</td>
<td>a suite of 14 datasets, each averaging 3.6M log lines</td>
<td>FGA</td>
</tr>
<tr>
<td>Hardware Design</td>
<td>VerilogEval [688] [link]</td>
<td>2023</td>
<td>1</td>
<td>a dataset consists of 156 problems from the Verilog instructional website HDLBits</td>
<td>BLEU/p@k</td>
</tr>
<tr>
<td>Model</td>
<td>Abbr.</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CuBERT [178]</td>
<td>MLM</td>
<td>Masked Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Given two sentences, Predict whether one sentence is the Next Sentence of the other [24].</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeBERT [28]</td>
<td>MLM</td>
<td>Masked Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTD</td>
<td>Detect the Tokens that are randomly Replaced in a sentence [179].</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SynCoBERT [183]</td>
<td>MMLM</td>
<td>Multi-Modal Masked Language Modeling which jointly models NL, PL, and AST (CFG).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Predict the type of all code tokens as either Identifier or non-identifier.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mask edges in the AST and ask the model to Predict these Edges.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multi-Modal Contrastive Learning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE-MVP [184]</td>
<td>MCL</td>
<td>Multi-Modal Contrastive Learning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Traverse the AST and use the type checker to Infer Fine-Grained identifier Types.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FGTH</td>
<td>Multi-Modal Masked Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMLM</td>
<td>Multi-Modal Masked Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISCO [185]</td>
<td>MLM</td>
<td>Masked Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Local AST Node Type-MLM which learns to recover the masked AST type.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contrastive Learning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLBART [30]</td>
<td>-</td>
<td>Random tokens are sampled and replaced with a mask token from the input sequence.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Random tokens are deleted from the input sequence.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>A number of text spans are sampled and replaced with a single mask token.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeT5 [29]</td>
<td>MSP</td>
<td>Randomly mask spans with arbitrary lengths and Predict these Masked Spans combined with some sentinel tokens at the decoder.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Predict the type of all code tokens as either Identifier or non-identifier.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mask all Identifiers in the PL segment and Predict them.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bimodal Dual Generation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PyMT5 [192]</td>
<td>MSP</td>
<td>Randomly mask spans with arbitrary lengths and Predict these Masked Spans combined with some sentinel tokens at the decoder.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UniXcoder [194]</td>
<td>MLM</td>
<td>Masked Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unidirectional Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ULM</td>
<td>Randomly mask spans with arbitrary lengths and Predict these Masked Spans combined with some sentinel tokens at the decoder.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multi-modal Contrastive Learning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MSP</td>
<td>(Cross-Modal Generation) Generate code comment which describes the function of the code.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Traverse the AST and use the type checker to Infer Fine-Grained identifier Types.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MCL</td>
<td>Naturalizing pre-training which forces the model to generate correct semantically equivalent natural code that is just what a human originally wrote.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CMG</td>
<td>Tree Masked Language Modeling for masking the nodes in the AST and the tokens in the code snippet.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Randomly exchange the positions of some nodes in the AST path, and Predict whether the Order of Nodes in the AST is correct.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERNIE-Code [196]</td>
<td>SCLM</td>
<td>(Span-Corruption Language Modeling) Corrupts 15% of the original NL/PL input tokens and predict the corrupted span on the target side.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Pivot-based Translation Language Modeling) Concatenate parallel source-target sentences and learn to predict the corrupted target language.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeExecutor [197]</td>
<td>-</td>
<td>Code Execution which improves the model ability to understand and execute code.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Curriculum Learning which starts from easy instances and then gradually handles harder ones.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LongCoder [118]</td>
<td>CLM</td>
<td>Causal Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPT-C [198]</td>
<td>CLM</td>
<td>Causal Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeGPT [114]</td>
<td>CLM</td>
<td>Causal Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PyCodeGPT [199]</td>
<td>CLM</td>
<td>Causal Language Modeling</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
REFERENCES


[39] H. K. Dam, T. Tran, and T. Pham, “A deep language model for software code,” 2016. 3


[68] C. K. Roy and J. R. Cordy, “Nicad: Accurate de-


S. Bhatia, P. Kohli, and R. Singh, “Neuro-symbolic program corrector for introductory programming assignments,” in Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 60–70. 4


N. D. Q. Bui, Y. Yu, and L. Jiang, 41


[266] N. Shazeer, “Fast transformer decoding: One write-head is all you need,” 2019. 10, 12


J. Schmidhuber, “Programming by demonstration: a comprehensive framework for ILMs on the mixture of code and text,” 2024. 17


J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, “Acceder: Utilizing existing code to enhance code generation,” 2023. 17

J. Li, G. Li, Y. Li, and Z. Jin, “Structured chain-of-thought prompting for code generation,” 2023. 17


J. Wang, Q. Sun, X. Li, and M. Gao, “Boosting language models reasoning with chain-of-knowledge prompting,” 2023. 18


[426] Y. Ma, Y. Liu, Y. Yu, Y. Zhang, Y. Jiang, C. Wang, and S. Li, “At which training stage does code data help llms reasoning?” 2023. 19


[474] C. Si, Y. Zhang, Z. Yang, R. Liu, and D. Yang, “Design2code: How far are we from automating front-end engineering?” 2024. 21


interface? a Big bench for large-scale database
grounded text-to-SQLs,” in Thirty-Seventh Conference
on Neural Information Processing Systems Datasets
and Benchmarks Track, 2023. [Online]. Available:
https://openreview.net/forum?id=dI4wzA6uV

[489] J. Yang, A. Prabhakar, K. Narasimhan, and S. Yao, “Inter-
code: Standardizing and benchmarking interactive
coding with execution feedback,” 2023. 21

[490] D. Wang, J. Andres, J. D. Weisz, E. Oduor,
and C. Dugan, “Autods: Towards human-centered
automation of data science,” in Proceedings of the 2021
CHI Conference on Human Factors in Computing Systems,
ser. CHI ’21. New York, NY, USA: Association for
Computing Machinery, 2021. [Online]. Available:
https://doi.org/10.1145/3411764.3445526

[491] D. Donoho, “50 years of data science,” Oct
publishations/912/1

enquirer: Learning to query tables in natural
language,” in Proceedings of the Workshop on Human-
Computer Question Answering. San Diego, California:
Association for Computational Linguistics, Jun. 2016,
org/W16-0105

[493] J. Perkel, “Reactive, reproducible, collaborative: com-

[494] P. Yin, W.-D. Li, K. Xiao, A. Rao, Y. Wen, K. Shi,
J. Howland, P. Bailey, M. Casta, H. Michalewski,
A. Polozov, and C. Sutton, “Natural language to code
generation in interactive data science notebooks,”
2022, 21, 22

[495] Y. Wen, P. Yin, K. Shi, H. Michalewski,
S. Chaudhuri, and A. Polozov, “Grounding code
 generation with input-output specifications,” in NeurIPS
2023 Workshop on Instruction Tuning and
Instruction Following, 2023. [Online]. Available:
https://openreview.net/forum?id=wq4OaU8tfE

[496] Y. Cao, S. Chen, R. Liu, Z. Wang, and D. Fried,
“API-assisted code generation for question answering
on varied table structures,” in Proceedings of the 2023
Conference on Empirical Methods in Natural
Language Processing. Singapore: Association for
2023.emnlp-main.897

[497] G. Li, X. Wang, G. Aodeng, S. Zheng, Y. Zhang, C. Ou,
S. Wang, and C. H. Liu, “Visualization generalization
with large language models: An evaluation,” 2024. 22

[498] Z. Yang, Z. Zhou, S. Wang, X. Cong, X. Han, Y. Yan,
Z. Liu, Z. Tan, P. Liu, D. Yu, Z. Liu, X. Shi, and M. Sun,
“Matplotlibagent: Method and evaluation for ILM-based
agentic scientific data visualization,” 2024. 22

[499] B. Chopra, A. Singhia, A. Fariha, S. Gulwani, C. Parnin,
A. Tiwari, and A. Z. Henley, “Conversational chal-
enges in ai-powered data science: Obstacles, needs,
and design opportunities,” 2023. 22

[500] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang,
Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei,
and J.-R. Wen, “A survey on large language model
based autonomous agents,” 2024. 22

[501] D. Dries, F. Xia, M. S. Sajjadi, C. Lynch, A. Chowdh-
ery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu
et al., “Palm-e: An embodied multimodal language

[502] Y. Hu, Q. Xie, V. Jain, J. Francis, J. Patrikar, N. Keetha,
S. Kim, Y. Xie, T. Zhang, Z. Zhao et al., “To-
ward general-purpose robots via foundation mod-
els: A survey and meta-analysis,” arXiv preprint
arXiv:2312.08782, 2023. 22

[503] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong,
M. Zhang, J. Wang, S. Jin, E. Zhou, R. Zheng, X. Fan,
X. Wang, L. Xiong, Y. Zhou, W. Wang, C. Jiang, Y. Zou,
X. Liu, Z. Yin, S. Dou, R. Weng, W. Cheng, Q. Zhang,
W. Qin, Y. Zheng, X. Qiu, X. Huang, and T. Gui,
“The rise and potential of large language model based
agents: A survey,” 2023. 22

agents,” Transactions on Machine Learning Research,
2024, survey certification. [Online]. Available:
https://openreview.net/forum?id=1i62CvflQJ

[505] Z. Durante, Q. Huang, N. Wake, R. Gong, J. S. Park,
B. Sarkar, R. Taori, Y. Noda, D. Terzopoulos, Y. Choi
et al., “Agent ai: Surveying the horizons of multimodal

[506] K. Yang, J. Liu, J. Wu, C. Yang, Y. R. Fung, S. Li,
Z. Huang, X. Cao, X. Wang, Y. Wang, H. Ji, and
C. Zhai, “If llm is the wizard, then code is the wand:
A survey on how code empowers large language models
to serve as intelligent agents,” 2024, 22, 26

[507] C. Ma, J. Zhang, Z. Zhu, C. Yang, Y. Yang, Y. Jin,
Z. Lan, L. Kong, and J. He, “Agentboard: An analytical
evaluation board of multi-turn llm agents,” 2024. 22

[508] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman,
B. Ichter, P. Florence, and A. Zeng, “Code as policies:
Language model programs for embodied control,”
in 2023 IEEE International Conference on Robotics and

[509] W. Huang, C. Wang, R. Zhang, Y. Li, J. Wu, and L. Fei-
Fei, “Voxposer: Composable 3d value maps for robotic
manipulation with language models,” arXiv preprint

[510] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu,
J. Tremblay, D. Fox, J. Thomason, and A. Garg, “Prog-
prompt: Generating situated robot task plans using
large language models,” in 2023 IEEE International
Conference on Robotics and Automation (ICRA).
IEEE, 2023, pp. 11 523–11 530. 22

[511] M. G. Arenas, T. Xiao, S. Singh, V. Jain, A. Z. Ren,
Q. Vuong, J. Varley, A. Herzog, I. Leal, S. Kirmani
et al., “How to prompt your robot: A promptbook for
manipulation skills with code as policies,” in Towards
Generalist Robots: Learning Paradigms for Scalable Skill
Acquisition@ CoRL2023, 2023. 22

[512] Y. Jin, D. Li, J. Shi, P. Hao, F. Sun, J. Zhang, B. Fang
et al., “Robotgpt: Robot manipulation learning from

[513] M. Parakh, A. Fong, A. Simeonov, T. Chen, A. Gupta,
and P. Agrawal, “Lifelong robot learning with human


machine translation of rare words with subword units,” in 
Proceedings of the 54th Annual Meeting of the 
Association for Computational Linguistics (Volume 1: Long 
Papers). Berlin, Germany: Association for Computa-

[584] M. Ali, M. Fromm, K. Thellmann, R. Rutmann, 
M. Lübbering, J. Leveling, K. Klug, J. Ebert, N. Doll, 
J. S. Buschhoff, C. Jain, A. A. Weber, L. Jurkschat, 
H. Abdelwahab, C. John, P. O. Suarez, M. Ostendorff, 
S. Weinbach, R. Sifa, S. Kesselheim, and N. Flores-
Herr, “Tokenizer choice for llm training: Negligible or 
crucial?” 2023. 24

[585] A. Petrov, E. L. Malfa, P. Torr, and A. Bibi, 
“Language model tokenizers introduce unfairness 
between languages,” in Thirty-seventh Conference on 

language models for generalization and robustness 
via data compression,” 2024. 24

the most out of your tokenizer for pre-training and 
domain adaptation,” 2024. 24

[588] C. Wang, K. Cho, and J. Gu, “Neural machine trans-
lation with byte-level subwords,” 2019. 24

[589] N. Ding, Y. Qin, G. Yang, F. Wei, Z. Yang, Y. Su, S. Hu, 
Y. Chen, C.-M. Chan, W. Chen, J. Yi, W. Zhao, X. Wang, 
and M. Sun, “Parameter-efficient fine-tuning of large-

scale pre-trained language models,” Nature Machine 

[590] X. L. Li and P. Liang, “Prefix-tuning: Optimizing con-
 tinuous prompts for generation,” in Proceedings of the 
59th Annual Meeting of the Association for Compu-
tational Linguistics and the 11th International Joint 

[591] X. Liu, K. Ji, Y. Fu, W. Tam, Z. Du, Z. Yang, and J. Tang, 
“P-tuning: Prompt tuning can be comparable to fine-
tuning across scales and tasks,” in Proceedings of the 
60th Annual Meeting of the Association for Compu-

[592] J. Wang, Q. Sun, N. Chen, C. Wang, J. Huang, 
M. Gao, and X. Li, “Uncertainty-aware parameter-
efficient self-training for semi-supervised language 
understanding,” in Findings of the Association for Compu-
tational Linguistics: EMNLP 2023, H. Bouamor, 
J. Pino, and K. Bali, Eds. Singapore: Association for Compu-
findings-emnlp.528 24

[593] E. J. Hu, yelong shen, P. Wallis, Z. Allen-Zhu, Y. Li, 
adaptation of large language models,” in International 
Conference on Learning Representations, 2022. 24

[594] S. Ayupov and N. Chirkova, “Parameter-efficient fine-
tuning of transformers for source code,” 2022. 24

[595] W. Zou, Q. Li, J. Ge, C. Li, X. Shen, L. Huang, and 
B. Luo, “A comprehensive evaluation of parameter-
efficient fine-tuning on software engineering tasks,” 
2023. 24

[596] T. Y. Zhuo, A. Zebaze, N. Suppattarakchai, 
L. von Werra, H. de Vries, Q. Liu, and 
N. Muennighoff, “Astraios: Parameter-efficient 

low-resource automatic program repair with meta-
learning and pretrained language models,” in Proceed-
sings of the 2023 Conference on Empirical 
Methods in Natural Language Processing. Singapore: 
Association for Computational Linguistics, Dec. 
aclington.org/2023.emnlp-main.430 24

[598] C. Wu, Y. Gan, Y. Ge, Z. Lu, J. Wang, Y. Feng, P. Luo, 
and Y. Shan, “Llama pro: Progressive llama with block 
expansion,” 2024. 24

behind code understanding model,” 2024. 24

[600] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, 
B. Savary, C. Bamford, D. S. Chaplot, D. de las Casas, 
E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lam-
ple, L. R. Lavau, L. Saulnier, M.-A. Lachaux, P. Stock, 
S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, 
T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. 
Sayed, “Mixtral of experts,” 2024. 24

llama: Accelerating language model pre-training via 
structured pruning,” 2023. 24

distillation: Empowering open-sourced LLMs with 
adaptive learning for code generation,” in Proceed-
sings of the 2023 Conference on Empirical Methods in 
Natural Language Processing. Singapore: Association 
2023.emnlp-main.417 24

multilingual is multilingual BERT?” in Proceed-
sings of the 57th Annual Meeting of the Association 
for Computational Linguistics. Florence, Italy: Association 

is multilingual llm?” 2023. 24

[605] T. Y. Zhuo, Z. Yang, Z. Sun, Y. Wang, L. Li, X. Du, 
Z. Xing, and D. Lo, “Source code data augmentation 
for deep learning: A survey,” 2023. 25

[606] Y. Huang, M. Qi, Y. Yao, M. Wang, B. Gu, C. Clement, 
and N. Sundaresan, “Program translation via code 
distillation,” in Proceedings of the 2023 Conference on 

[607] Q. Sun, N. Chen, J. Wang, X. Li, and M. Gao, 
“Transcoder: Towards unified transferable code re-
presentation learning inspired by human skills,” 2023.


[619] X. She, Y. Liu, Y. Zhao, Y. He, L. Li, C. Tantithamthavorn, Z. Qin, and H. Wang, “Pitfalls in language models for code intelligence: A taxonomy and survey,” 2023. 26


Sun et al. 64