SCIENCE CHINA
Information Sciences

• REVIEW •

September 2024, Vol. 67, Iss. 9, 191101:1–191101:36
https://doi.org/10.1007/s11432-023-3956-3

Deep learning for code generation: a survey

Huangzhao ZHANG[1,2], Kechi ZHANG[1,2], Zhuo LI[1,2], Jia LI♀[1,2], Jia LI♂[1,2], Yongmin LI[1,2], Yunfei ZHAO[1,2], Yuqi ZHU[1,2], Fang LIU[3], Ge LI[1,2*] & Zhi JIN[1,2*]

[1]*Key Lab of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing* 100871*, China;*
[2]*School of Computer Science, Peking University, Beijing* 100871*, China;*
[3]*School of Computer Science and Engineering, Beihang University, Beijing* 100191*, China*

Received 8 June 2023/Revised 1 September 2023/Accepted 5 January 2024/Published online 20 August 2024

Abstract In the past decade, thanks to the powerfulness of deep-learning techniques, we have witnessed a whole new era of automated code generation. To sort out developments, we have conducted a comprehensive review of solutions to deep learning-based code generation. In this survey, we generally formalize the pipeline and procedure of code generation and categorize existing solutions according to taxonomy from perspectives of architecture, model-agnostic enhancing strategy, metrics, and tasks. In addition, we outline the challenges faced by current dominant large models and list several plausible directions for future research. We hope that this survey may provide handy guidance to understanding, utilizing, and developing deep learning-based code-generation techniques for researchers and practitioners.

Keywords code generation, automated software engineering, deep learning, large model, artificial intelligence

1 Introduction

Since the pioneering studies published in 2014 [1–3], the software engineering (SE) community has been leveraging deep learning (DL) approaches on programming language processing for comprehension and generation [4, 5]. In the previous decade, thanks to the rapid growth in the code corpus and deep model capacity, employing DL for code generation has shown great potential and feasibility for code generation. The automated process of DL-based code generation allows the practitioner to generate source codes from the requirements of certain forms [6] (e.g., functional features in natural language) to assist software development and reduce the workload of developers. Previous studies have confirmed the feasibility of DL for code generation [3, 7, 8], and tools deployed for code generation have been available, such as Copilot [9] powered by Codex [8]. Without exaggeration, we are witnessing a new era of DL-based code generation.

In this survey, we aim to sort out the developments of DL-based code generation solutions. More specifically, we draw the problem boundary at (1) deep model, i.e., the solution is supposed to be deep; (2) general-purpose programming language, i.e., the generated code is supposed to be in a general-purpose programming language; and (3) natural language, i.e., the functional requirement is supposed to be in natural language. In the rest of the survey, if there is no special instruction, we refer to DL-based code generation as generating source code snippets from the functional requirements in natural language through deep models.

We search the literature and find a clear milestone and watershed — transformer. Before that, classic architectures, such as recurrent neural networks (RNNs) [3], are leveraged to generate codes. The model capacity is relatively insufficient, and studies then were only able to accomplish simple tasks, such as paraphrasing instructions from natural language to code line by line [10, 11]. To improve performance, researchers explicitly incorporated prior program knowledge in the model design, such as parsing tree [7, 12], code graph [13]. Consequently, a series of models oriented to code generation are proposed.

Since the appearance of the transformer architecture [14] in 2017, the community has been deeply influenced. The great capacity of the transformer allows for large model and large-corpus pretraining,

* Corresponding author (email: lige@pku.edu.cn, zhijin@pku.edu.cn)

© Science China Press 2024

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

and the explicit model design for code has gradually evolved into implicit large-scale learning. Thanks to their powerfulness, large model-powered solutions can handle much more challenging code-generation tasks (e.g., competitive program generation [15]). The role of the deep model during code generation also changes greatly. Regular researchers and practitioners find it difficult to design, implement, and train a large model because of soaring costs. However, large models inspire novel learning paradigms (e.g., in-context learning and chain-of-thought (CoT)), resulting in a whole new series of research directions. Therefore, nowadays, large models are less of the subjects of code generation; instead, they become more like participants. Current research tends to explore how to make better use of large models, rather than design one from scratch.

In summary, the contributions of this survey are as follows:

• A general and complete framework of DL-based code generation is summarized, based on which a complete and comprehensive review of the existing solutions is provided.

• According to the proposed pipeline, taxonomy is made from the perspectives of model architecture, strategy enhancement, evaluation metrics, and benchmarks for DL-based code generation.

• The challenges and limitations within the current popular solutions, i.e., large models, are discussed, and several plausible directions for future research are presented.

The rest of the survey is organized as follows. Initially, an overview framework of code generation is provided in Section 3. In Sections 4 and 5, code generation solutions are summarized from the perspective of model architectures and model-agnostic enhancing strategies, respectively. Sections 6 and 7 list evaluation metrics and tasks of code generation, respectively. Finally, challenges and plausible opportunities of code generation in the era of large models are explored in Section 8, and the survey is summarized in Section 9.

2 Literature review

Literature search. High-quality studies published in conferences and journals in the field of artificial intelligence (AI), natural language processing (NLP), and SE, are extracted. To present an overall picture of the field, preprint papers from arXiv and other open-source or deployed applications (which may not have corresponding published papers) are also included in this survey. The literature selection process involves two major parts. (1) The publication list of top-tier conferences and journals from 2016 to 2022 is filtered to gather a collection of candidate papers. (2) To avoid omissions, the DBLP database is further searched for related publications. In the search, keywords such as "code generation" are used, and studies lying within the topic boundary are manually selected based on the abstracts. After combining the two parts, the candidate papers are analyzed. After rounds of discussion, the paper list of this survey is finalized, including 142 publications.

Publication statistics. Figure 1 presents the statistical information of the 142 publications included in this survey. In 2021, a clear accelerated upward trend is noted in the number of publications, suggesting that code generation has become a research hotspot. A plausible reason for this phenomenon could be the release of GPT-3 [16] in 2020, which demonstrates great capability in sequence generation and opens the door to the research field. Most studies are published in top conferences or journals, and the top-5 popular publications include ACL, ICSE, EMNLP, NeurIPS, and ICLR. Moreover, a significant portion of researchers opt for preprints, probably to publish the latest results as soon as possible. Since 2021, some relevant communities and companies have released numerous code-generation models and applications, including GPT-Neo [17], GPT-J [18], aiXcoder [19], PaLM-Coder [20], GPT-CC [21], CodeParrot [22], CodeGeeX [23], Copilot [9], ChatGPT [24], New Bing [25], BARD [26], and GPT-4 [27]. Academic and industrial fields set their sights on code generation. The breadth and depth of the code-generation field are truly extensive and quite promising.

Other complementary survey. In this survey, we attempt to convey a relatively complete picture of DL-based code generation. Therefore, we sort out the development path of this research field from four perspectives, i.e., model architecture, model-agnostic enhancement, metrics and benchmarks. In addition to our study, we recommend another recent one [28] to our readers, which nicely complements ours. The authors focus on large models and provide a thorough discussion of the challenges and issues within large models for code generation. As complementary, we provide an overview of code generation and sort out classic DL solutions for code generation besides large models, hoping to bring inspiration to the era of large models.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

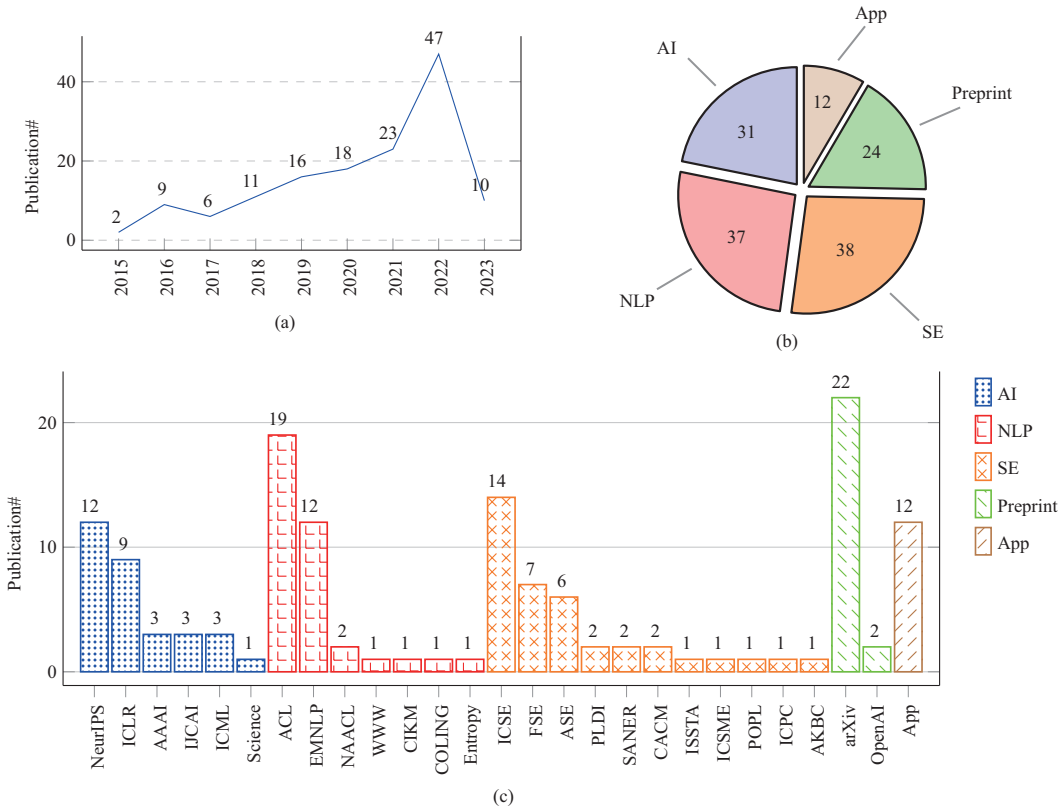

Figure 1 (Color online) Statistics of the publications included in this survey. (a) The number of publications by year; (b) the publication numbers in each field; (c) the number of publications in each conference or journal. Note that some deployed applications ("App") are also included in the figure.

3 DL-based code generation

Code generation refers to the automatic process of producing the source code given a requirement. In general, the requirements can be expressed in various forms, but in this survey, the requirements are expressed specifically in natural language. The produced code can be of different granularity, including statement-level, method-or-function-level, and file-level, or the produced code can be different programming languages, according to the very task requirement.

3.1 Formal definition of DL-based code generation

DL-based code generation can be defined formally as below. Given the requirement, denoted by x, the goal of the deep model, denoted by f, is to find a piece of source code, denoted by c, whose functionality matches requirement x most. In addition to f, augmentations such as external knowledge or auxiliary tasks can be incorporated to enhance the model. f directly predicts and produces probability distributions, which require sampling techniques to convert them into concrete code snippets. Due to different architectures and tasks, the input and output formats of f may differ. Therefore, f needs a pre-processing module, denoted as p_{pre}, to re-format x, and a post-processing module, denoted as p_{post}, to re-format f's output to source code. Furthermore, p_{post} can also prune or fix the generation through techniques such as reranking and repairing to improve the quality of the generated code. The overall procedure can be formulated by function composition as follows:

$$c = \left(p_{\text{post}} \circ f \circ p_{\text{pre}}\right)(x) = p_{\text{post}}\left(f\left(p_{\text{pre}}(x)\right)\right), \tag{1}$$

where $\circ$ refers to the function composition operator. After generation, we also need to evaluate to what extent c matches x. Supposing there is an ideal gold ground-truth code, denoted by c^*, which matches x perfectly, we may directly compare c with c^*. However, c^* is not always accessible, we may also run test cases, denoted by t, against c. Please refer to Figure 2 for a demonstrative pipeline of the DL-based code generation process.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

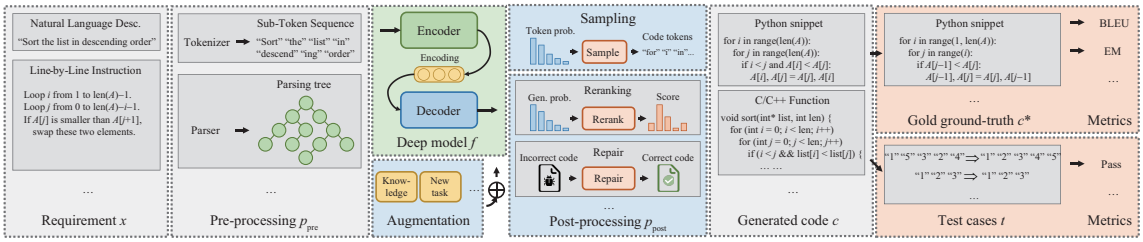

Figure 2 (Color online) Demonstrative pipeline of DL-based code generation (better viewed in color). The requirement x which describes the functional feature can be in various forms, such as abstracted sentences or line-by-line instructions. The pre-processing module p_{pre} re-formats x according to the needs of the model f, and feeds it into f. f with different deep architectures and optional augmentations produces its prediction, and passes it to the post-processing module p_{post}. p_{post} re-formats the output according to the demands of the task, and modifies the output to improve the generation quality. So far, the code c is generated according to x's requirement. And finally, a gold ground-truth c^* or a set of test cases t is utilized to evaluate c generated by the deep model.

A perspective of architecture. The deep model f plays a major and important role during the code generation process. Code can be represented in various formats, such as token sequences, parsing trees, data flow graphs, and therefore there are various deep architectures available for f. From the perspective of model architecture (green boxed "deep model f" in Figure 2), we roughly divide the solutions to DL-based code generation into three categories. (1) The traditional sequential architecture, such as RNN and long short-term memory (LSTM), models the token sequence, based on the idea of "naturalness" [29–31]. (2) The tree-based architecture generates parsing trees, integrating grammar rules [32, 33] to ensure the correctness during parsing and compilation. (3) The transformer architecture is one of the most popular sequential architectures, as it sparks the upsurge of the pretrained model. We discuss transformer and the pretrain models separately due to their great development and impact.

A perspective of model-agnostic enhancing. With f's architecture, the generation results can be improved by employing additional augmentation, sampling and post-processing p_{post} (blue boxed "augmentation", "sampling" and "post-processing p_{post}" in Figure 2). Although the augmentation is embedded in f, it is rather independent to the backbone architecture design. Since f is a probability model, converting probability distribution to concrete code snippets is an essential step. During such conversion, sampling is the key technique to control the diversity, creativity, and quality of the generated code snippet. On the other hand, f may produce erroneous or incorrect code, and post-processing techniques modify (e.g., repair) or prune (e.g., re-ranking) the predictions from the deep model, leading to better generation results.

A perspective of metrics. In the pipeline presented in Figure 2, when the generation process is complete, it is a necessity to evaluate to what extent the generated code c matches the requirement x (orange boxed "metrics" in Figure 2). There are mainly two types of metrics, based on the references. (1) The similarity-based metrics reflect the similarity between the gold ground-truth code c^* and the generated one c. (2) The execution-based metrics employ unit test cases t to verify the correctness of c.

3.2 Preliminary of deep model

The deep models serve as the technical foundation of DL-based code generation. We illustrate the basic deep architectures, the encoder-decoder framework, and the transformer architecture in this part.

Basic deep model architecture. The neural network layers serve as the basic components of deep neural networks (DNN), based on which various models are proposed for code generation [30]. The components can be stacked and connected at will to construct desired neural networks. In general, the i-th layer $l^{(i)}$ takes features (usually vectors) $x^{(i)}$ as input and produces new ones $x^{(i+1)}$, by carrying out certain non-linear but differentiable functional mapping, i.e., $x^{(i+1)} = l(x^{(i)})$. By stacking and connecting n layers, the DNNs f, is formulated as follows:

$$f(x) = (l_n \circ l_{n-1} \circ \cdots \circ l_1)(x) = l_n(l_{n-1}(\cdots l_1(x)\cdots)). \tag{2}$$

To train $l_1, \ldots, l_n$ in f, we optimize the loss function $L(f(x), y)$, which measures the difference between the model prediction $f(x)$ and the ground-truth output y. Stochastic gradient descent (SGD) [34, 35] and back-propagation (BP) [36] are involved during the training process.

Some basic neural network layers are presented in Figure 3. (1) RNNs [37] and the variants (e.g., LSTM [38, 39] and gated recurrent unit (GRU) [40]) are widely adopted sequential architectures. RNN

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

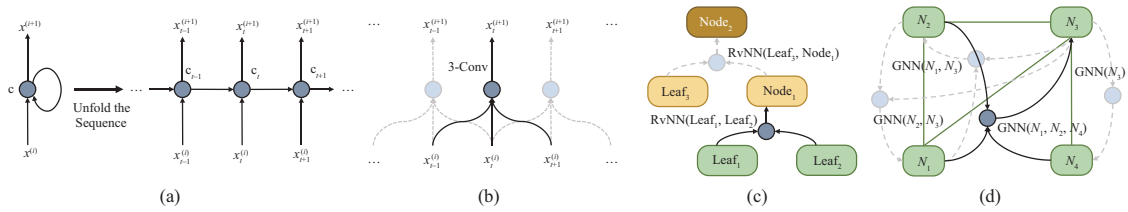

Figure 3 (Color online) Demonstrative examples of the basic neural network layers. (a) The unfolded RNN architecture, which passes the inner states (c_{t-1}) of the previous time-step ($t-1$) to the current at time-step t; (b) the sequential CNN architecture, whose convolutional kernels slide along the sequence; (c) the recursive architecture, which recursively processes the children for the parents in the tree; (d) the GNN architecture, which collects features from the neighbors to update each node.

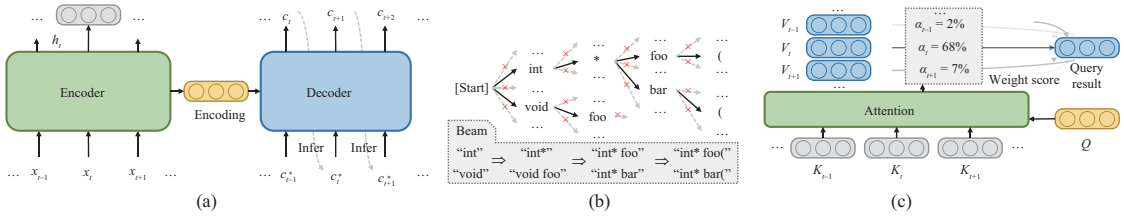

Figure 4 (Color online) Demonstrative illustrations of the seq2seq framework and the accompanying attention and beam search techniques. (a) The encoder-decoder framework, which encodes the input x into a vector encoding, and generates the output c conditioned on the encoding; (b) an example of beam search, which maintains a top-k most probable beam, and only expands examples within the beam; (c) the attention mechanism, which handles the query Q upon key-value pairs (K_i, V_i) by weighted sum over V_i's according to the probabilistic attention score α_i.

employs a cell to record the internal states of each time-step, denoted as c_t. In the t-th time-step, RNN retrieves c_{t-1} for history information and passes the updated c_t to the next time-step. (2) Convolutional neural networks (CNN) [41] and the variants [42, 43] are another popular type of architectures to process sequences [44–47]. It adopts one-dimensional convolution to extract features from the sequence in an n-gram sliding window manner. Besides sequential convolution, there are also tree-based convolution and graph-based convolution. (3) In addition to sequential data, we also need to deal with tree-structured data, such as the parsing tree. The recursive neural networks (RvNN) [48] and the variants [49–51] came into being. It handles the tree nodes recursively in a bottom-up manner. (4) Graph neural networks (GNN) [52] and the variants [53–55] extend the convolution operation to the non-Euclidean graph space, by aggregating features of nodes with their neighbors.

Figure 4 shows the demonstrative example of the seq2seq framework as well as the accompanying attention mechanism for DL-based code generation.

Encoder-decoder. DL-based code generation can be seen as a conditional generation task, i.e., the requirement is the condition of the generated code. The encoder-decoder framework is designed for such conditional generation task. As the name suggests, the framework consists of two parts, i.e., the encoder and the decoder. Both models can be arbitrary architecture, such as the aforementioned RNN and CNN. Taking sequence-to-sequence (seq2seq) [56] for an example (refer to Figure 4(a)), the encoder processes the input sequence (e.g., x in code generation), forming a vectorized encoding, and the decoder takes the encoding as the condition, predicting the output sequence (e.g., the resulting code c). Besides seq2seq, which constrains the format of the input and the output to be sequential, there are other encoder-decoder settings, with other input-output formats, such as code2seq (i.e., source code parsing tree to sequence) [57, 58], and wave2seq (i.e., wave signal to sequence) [59].

Attention. The attention mechanism is one of the most important techniques in the field of DL [60]. It allows the model to focus on the important parts, ignoring the irrelevant ones. As shown in Figure 4(c), given a vectorized query Q and n vectorized key-value pairs $(K_i, V_i)_{i=1}^n$, the attention mechanism "selects" the most important V_i's according to the attention score $\alpha(Q, K)$[1]. In practice, we may query with multiple Q's simultaneously. In a more extreme scenario, where Q, K, and V are identical or homologous (i.e., Q, K, and V come from the same source, but the computation may differ from each other), it leads to self attention [61].

Attention mines relations between the query and each key-value pair. A further step is employing

1) Note that attention does not actually select V_i. It performs a probabilistic weighted sum over all V_i according to the attention score. For easier understanding, we use the word "select" here.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

multiple independent attentions to discover different types of relations between them. Each attention is called a "head", and the combination is the multi-head attention mechanism. Multi-head attention then becomes the basis of the transformer architecture.

Transformer. Based on the idea of "attention is all you need" [14], the transformer architecture is proposed. It does not involve the aforementioned basic architectures, such as RNN and CNN, instead, it fully leverages the advantages of multi-head attention. The transformer encoder and decoder each equip with multi-head self attention modules, and are linked by multi-head cross attention. With the power of attention, transformer can largely overcome the long-term dependency issue.

Pretraining & finetuning. Thanks to the emergence of the large corpus, the huge computing power, and the powerful transformer architecture, the deep models have moved to a new stage, i.e., the large pretrained model [8, 16, 62–69] The paradigm of model learning has shifted from the traditional from-scratch training to "pretraining and finetuning" [70]. In this scenario, the large deep model is trained in two phases. (1) The first is to pretrain the model upon an extremely large corpus. The model is supposed to learn the general knowledge (or we can say the "common sense"). No doubt this phase is resource-consuming and time-consuming. (2) The second is to finetune the model on a specific downstream task. The model is supposed to be specialized to the specific task. With the support of large models and general knowledge learned from the large corpus, the finetuned model ought to outperform the from-scratch trained ones.

3.3 Relevant research field

There are some other areas that are very relevant to code generation listed as follows.

Program synthesis. Program synthesis searches for the program consistent with the user's intent expressed in certain form of constraints [71]. Program synthesis in a broad sense includes the code generation task as the definition suggests, but there are two differences between traditional program synthesis and DL-based code generation. (1) The intent of program synthesis is different from the requirement of DL-based code generation. The former is usually represented as formal specifications, such as input-output examples [72, 73] or partial programs [74]. In recent years, researchers have gradually combined the informal natural language description into the intent [75, 76]. (2) The target programming language of program synthesis is often domain-specific languages (DSL, e.g., string operations [72, 73]), rather than general programming languages, resulting in a distinction from DL-based code generation.

Semantic parsing. Semantic parsing is a process of transforming natural language sentences into structured meaning representations [77,78], such as abstract logic [79,80] and queries [81–83]. The practitioners may often derive results from the meaning representations against the corresponding environment or context (e.g., we may execute a SQL query against a relational database). Semantic parsing intersects with code generation, as the code in some benchmarks, such as HS and MTG [84], are in general-purpose programming languages (i.e., Python and Java). Semantic parsing differs from code generation in the language of the target code (or the meaning representation), except for the intersection part. I.e., traditional semantic parsing converts natural language into logical form [79, 85], lambda calculus [80], or SQL [81–83], while code generation into general-purpose programming language.

Test assertion generation. Test assertion generation is a key technique to automated software testing, which aims to generate accurate assertion statements for unit test cases [86,87]. I.e., the assertion generator is supposed to produce the assertion statement, given the focal method (to be tested), the test method without the assertion statement (usually replaced by a "placeholder"), and plausible additional descriptions (e.g., natural language). Hence, test assertion generation is related to code generation, as the target is source code statements; while the input requirement of test assertion generation is usually incomplete code, rather than natural language, which is different from DL-based code generation. Test assertion generation is an emerging research direction that has important implications for automated software testing.

Code completion. Code completion speeds up software production by predicting the upcoming code token(s) given the preceding context [29, 88, 89], and it is an essential feature of modern integrated development environments [90]. In general, the code completion model predicts the next following token(s) given the incomplete source code snippet, such as next-one-token completion [29, 88, 90, 91], and next-line completion [89, 90, 92–96]. Broadly speaking, code completion is highly correlated to code generation, as their targets are both producing source code. However, in this survey, we limit the requirement to natural language rather than incomplete code, and thence, we do not take code completion into our taxonomy.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

Code comprehension. Code comprehension extracts information from the source code to help understand some certain aspects of the code [97]. Code generation could be viewed as the inverse of code comprehension, because the former generates source code from the requirement, and the later extracts information from the code. Code comprehension has been thoroughly studied in the past decades and is subdivided into many sub-fields, including comment generation [98–102] and code review generation [103, 104].

Neural machine translation. Neural machine translation is a thriving and booming field in NLP, where the neural network model translates the text or speech from one language to another automatically [105, 106]. The development of neural machine translation (the encoder-decoder framework [56], the attention mechanism [60], and the transformer architecture [14]) has directly or indirectly caused huge impacts on many following fields, including DL-based code generation.

4 Deep model for code generation

We present deep models for code generation according to the development process and architecture. Please refer to Figure 5 [3, 7, 8, 12, 13, 21–23, 32, 33, 69, 84, 89, 95, 96, 107–128] for the model architecture taxonomy of this section.

4.1 Sequential modeling

Based on the hypothesis that "code is natural" [29–31], i.e., most programming languages are usually simple, repetitive and predictable, a significant body of research has employed sequential modeling as a direct approach to code generation. This approach conceptualizes code generation as the production of token or sub-token sequences in a sequential order, encompassing three critical aspects: (1) tokenization, (2) decoding, and (3) the model architecture.

Tokenization. The sequential modeling techniques entail the conversion of the natural language requirement into a concrete code sequence. The very first step is to tokenize both the requirement and the code into token (or sub-token) sequences. Different tokenization approaches lead to different vocabulary, thus affecting the final sequence generation process. Early work bypasses the tokenization process by processing and manipulating the character sequence [3]. Soon after, the trivial tokenizer based on punctuations and white-spaces is introduced [84]. Although straightforward, such trivial tokenization disregards the certain feature of the programming language and the lexical information of code cannot be efficiently separated by white-spaces alone. Additionally, code often comprises a vast number of customized identifiers, leading to an extensive vocabulary size and lots of rare tokens.

To address the challenge, researchers begin to utilize the fine-grained tokenizer to split code tokens further into sub-tokens [107] by employing the parser to tokenize code according to the lexical rule, and the outcome identifiers are split following camel case (e.g., "getValue" ⇒ "get", "Value") or snake case (e.g., "set_idx" ⇒ "set", "idx"). The byte pair encoding (BPE) tokenization [129] began to be widely adopted in code generation [89] after it became popular in NLP. BPE mines the frequent sub-tokens from the training corpus — the vocabulary starts with characters (the initial sub-tokens), and the algorithm iteratively merges the most frequent contiguous sub-token pairs into new sub-tokens. BPE can efficiently tokenize the code into a sub-token sequence while reducing the vocabulary size.

Decoding. Decoding is to generate the token sequence of the code snippet. To achieve so, one trivial and straightforward solution is to employ the seq2seq framework and directly map the natural language requirement to the code token sequence, which we call it as the single stage decoding. The single stage decoding directly regards code generation as an end-to-end task. Mou et al. [3] verified the plausibility of generating character-level code by LSTM through empirical case studies. Later, latent prediction network [84] introduces copy mechanism into the character-level RNN model. Iyer et al. [107] proposed an LSTM encoder-decoder model to generate Java token sequence from the requirement along with member variables and member method signatures under the same class. These simple but effective solutions prove the feasibility of code generation from an experimental point of view. They serve as an important set of baselines, based on which researchers explore various approaches to tackle the problem of code generation. Furthermore, thanks to the powerful architecture and the abundant corpus, the transformer model, especially the pretrained model, enables the outstanding proficiency of code generation [89]. The pretrained model is one of the greatest DL inventions, therefore we introduce it separately in another subsection, despite its sequential modeling nature.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

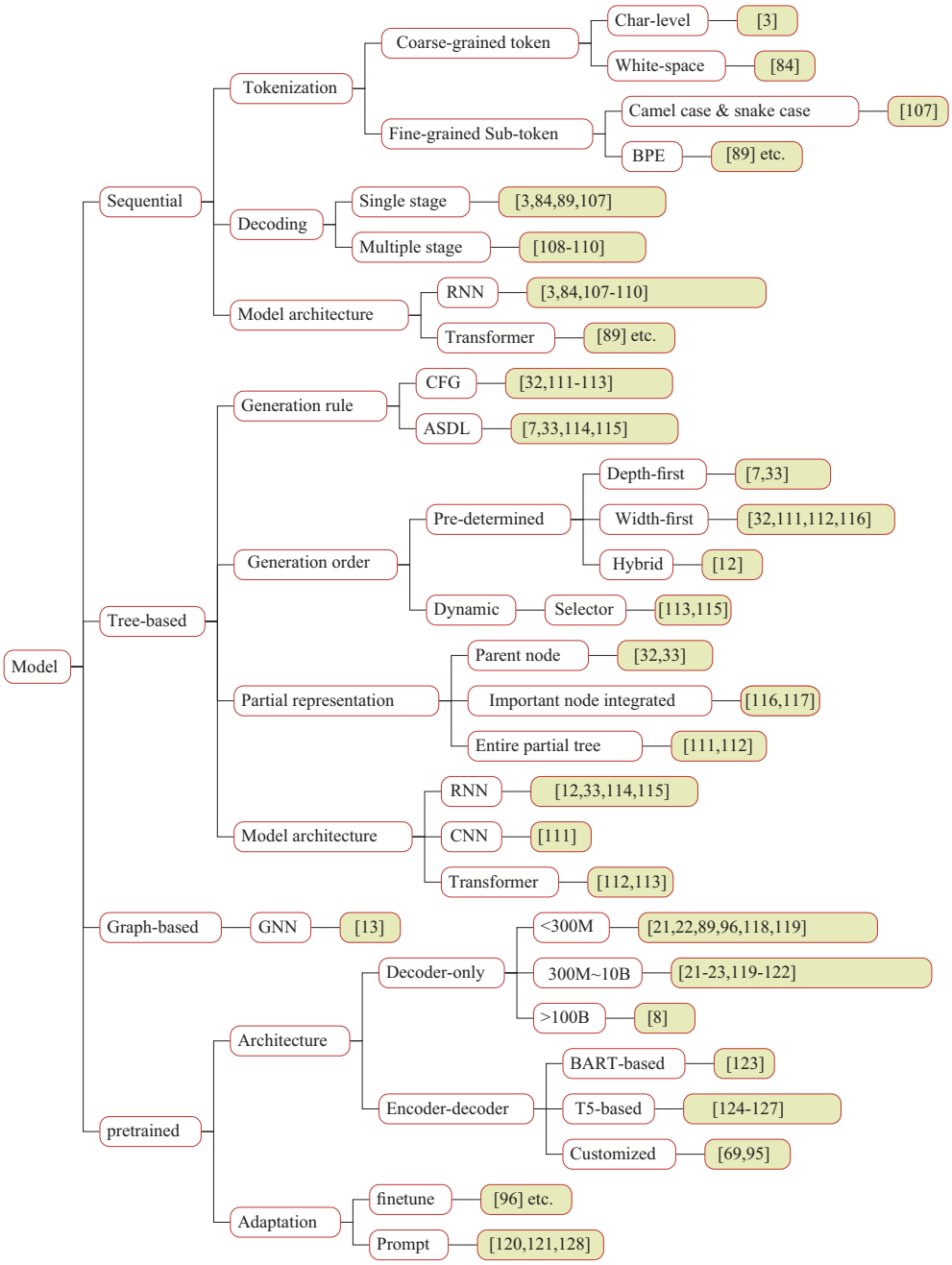

Figure 5 (Color online) Model architecture taxonomy for DL-based code generation solutions.

Other studies decompose the decoding process into multiple stages, and the model decodes from coarse (e.g., an abstracted frame) to fine (e.g., the concrete code token sequence). Such multi-stage decoding comes from the heuristic of program writing, where the programmer usually writes a high-level frame ("sketch", "scaffold", etc.) of the code first, and fills in the missing detail later. Thereupon, the whole decoding process is decomposed into multiple stages, generating the token sequence from coarse to fine [108–110]. There are usually two stages — the first is to produce the abstracted code, and the second is to replenish the detail. The model for multi-stage decoding consists of multiple independent modules, each handling a decoding stage. Rather than end-to-end training, the modules are also trained separately. During inference, they generate the code snippet module by module in a pipeline manner. Coarse-to-fine [108] constructs the abstracted code by stripping the identifiers in the code. GED [109] further defines the precise grammar of the sketch, which eliminates the identifiers, the constants, etc., leaving the APIs and the control statements. Semantic scaffold [110] proposes primary expression sym-

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

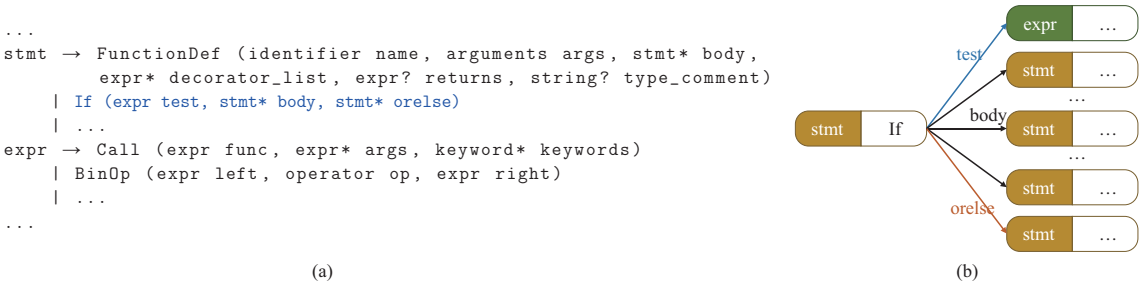

Figure 6 (Color online) Demonstrative example of the ASDL. (a) A fragment of the Python ASDL grammar, and (b) an illustration of a subtree construction. The subtree is constructed following the ASDL rule of `If` (highlighted in blue). As the rule suggests, the non-terminal `If` with the type of `stmt` is expanded into multiple children, including an `expr` node with the field of `test`, a set of `stmt` nodes with the field of `body`, and a set of `stmt` nodes with the field of `orelse` (in this demonstration, there is only one `stmt` in the set).

bols to build the abstracted code — the expressions are compressed into certain symbols. E.g., the entire for loop expression is compressed into one single symbol, "for_statement". Symbol constraint is exploited to expand the symbols, filtering out illegal expansions.

Model architecture. There are two main classes of architectures employed by sequential code generation — RNN and transformer. (1) The early studies usually employ RNN or its variants (e.g., LSTM and GRU) as the backbone model [3, 84, 107]. These end-to-end RNN models directly produce the character-level or token-level code snippet. Other studies design multiple stages for decoding, and leverage RNN to model each stage [108–110]. (2) Recently proposed transformer is more advanced and powerful than RNN. It has been employed in code generation to address the issue of lengthy dependencies [89]. Subsequent studies have leveraged pretraining techniques on transformer and achieved satisfactory performance. Given the significance of pretraining technology in this context, we will expound on it in a dedicated subsection (Subsection 4.4) of this paper.

4.2 Tree modeling

It is known that the source code snippet can be parsed into a syntax tree, and the two counterparts are equivalent. Therefore, there is a natural train of thought to generate the syntax tree first, and then convert it back to code [32]. Besides, compared with the token sequence, the syntax tree contains rich structural information, which may benefit the process of DL-based code generation. This subsection introduces the deep syntax tree generation from four aspects: (1) the generation rule, (2) the generation order, (3) the representation of the partial tree, and (4) the architecture of the tree model.

Generation rule. Most studies employ top-down generation, where the model repeats the following iteration until the syntax tree is complete. In each single iteration, the model expands a non-terminal into a subtree (e.g., Figure 6(b)) following the generation rule. That is, the model selects a non-terminal along with a corresponding appropriate expansion action, and predicts the child nodes of the non-terminal according to the action.

Usually, the expansion action is defined by the grammar. There are mainly two kinds of grammar – the context-free grammar (CFG) or the abstract syntax description language (ASDL) grammar [130]. (1) The generation rule of CFG is of the form $\mathtt{A}\rightarrow\alpha$, where `A` is a non-terminal, and it can be expanded into a sequence of terminals or non-terminals, i.e., α. Defining the generation rule based on CFG is straight-forward [32, 111–113]. (2) Another branch of research defines the generation rule following the ASDL grammar [7, 33, 114, 115]. ASDL provides additional types of nodes and fields (i.e., the edges in the tree). Figure 6 presents a fragment of the Python ASDL grammar. Take the `If` node with the type of `stmt` for instance, it can be expanded into a subtree shown in Figure 6(b), following the construction rule of `If` (highlighted in blue in the ASDL grammar).

Generation order. The generation order refers to the order in which the non-terminals are expanded. It is essential to syntax tree generation, because different orders may yield different generation results. There are two major classes — the pre-determined order and the dynamic order.

The pre-determined generation order comes from the heuristics, where programmers usually traverse the tree structure in particular orders. In specific, the generation order is usually depth-first (the expansion goes as deep as possible along each path before backtracking) [7, 33] or width-first (the expansion goes layer by layer) [32, 111, 112, 116]. In addition, during the iteration of syntax tree generation, solutions

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

to depth-first generation usually select the generation rule to expand the current node, and then predict the value of children one by one; while the width-first order often makes the two steps into a single one, i.e., the model would select the generation rule and predict the values of the children simultaneously. In order to reduce the impact of the order upon the generation result, the follow-up study integrates the depth-first order and the width-first order by mutual learning [12], which transfers the knowledge by forcing the results of the two generation orders to imitate each other.

Besides the pre-determined order, the dynamic order is also used, where the expansion order is decided by the model during the generation process [113, 115]. I.e., in each iteration, a branch selector chooses the most appropriate non-terminal based on the current incomplete tree, and then the model expands the selected node. The selector is often obtained via reinforcement learning, because the supervised data for the expansion sequences is often difficult to obtain.

Partial representation. The representation of the (intermediate) partial tree is essential during the generation, which conveys the syntactic and structural information along the process. As previously introduced, in each generation iteration, the deep model selects an appropriate expansion rule, and predicts the child nodes. All these behaviors rely on the vectorized (or tensorized) representation of the partially generated tree, since the deep model is heavily parameterized.

For the partial representation, the early studies adopt a trivial but straight-forward solution — they regard the parent node's state representation as the partial tree representation [32, 33], and make prediction upon it. Similar to the design of RNN, ideally, the syntactic information would pass from the parent to the children step by step. Unfortunately, this solution can hardly represent the global information of the partial tree, and the long path may lead to the long-term dependency issue. In response to the shortcoming mentioned above, later, researchers propose to take the important nodes in the partial tree into account in the partial representation [116, 117]. Here, the important nodes are usually chosen by heuristic, such as the previously generated terminals and the previous ancestors [116], the siblings and the path from the root [117]. Recently, some studies have opted to obtain partial representation directly from the entire partial tree, taking every node within it into consideration [111, 112]. Specifically, they employ techniques such as tree-base convolution [111] and attention [112] to process the partial tree and obtain the representation. Compared with the previous solutions, this solution provides a comprehensive global representation of the partial syntax tree.

Model architecture. There are three major classes of the model architecture — RNN, CNN, and transformer. (1) The early studies usually adopt RNN as the backbone of their solutions [12, 33, 114, 115], which considers the generation process as a sequence of generation rules (as aforementioned), and predicts the generation rule iteratively. (2) Then, the Tree-based convolution is introduced [111], which is employed to produce partial tree representation (as introduced above). (3) More advanced studies leverage the attention mechanism, building transformer architecture for syntax tree generation [112, 113].

4.3 Graph modeling

The GNN [52] is a specialized type of DL model designed to process the graph-structured data, and is widely employed by various code processing tasks. In contrast to the tree modeling methods, GNN can process more complex graph structure via graph convolution, as the graph is usually composed of the syntax tree backbone with additional edges, such as data flow or control flow. One representative study [13] uses the gated graph neural network (GGNN) [55] to generate the code graph, by adding new nodes and edges step by step in a pre-determined order. The ExprGen task is introduced, which is to fill the statement in a given hollowed code snippet, to demonstrate the feasibility of the proposed GGNN solution. However, considering the efficiency of the GNN and the difficulty of graph generation tasks, there are very few code generation methods of graph modeling. The graph modeling paradigm is challenging for code generation and more research work is still expected.

4.4 pretrained model

Ever since the proposal of the transformer architecture [14], we have witnessed the dominance of the large pretrained model in various tasks and fields [16, 62–68]. Code generation is also no exception; there have already been plenty of pretrained models released for automating the code generation process. The model is usually pretrained upon a large corpus, composed of natural language and program language, and released through parameters or API. And it can also be adapted to the downstream code generation

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

tasks. This subsection will introduce the pretrained model for code generation from two perspectives: (1) the model architecture, and (2) the adaptation approach.

Model architecture. Most pretrained models are based on transformer, which can be further divided into encoder-only, decoder-only and encoder-decoder architectures. The encoder-only model [93, 131] is majorly adopted in code understanding and comprehension tasks, such as clone detection [132, 133] and program classification [134, 135]. In the following, we will not discuss this category of the pretrained model but summarize the generative models of decoder-only and encoder-decoder architectures.

Decoder-only. As the name suggests, the decoder-only model contains only the decoder, and is usually pretrained through language modeling tasks. GPT-C (366M) [89] is one of the earliest pretrained transformer decoders on source code. The model is the backbone of the code completion tool, namely IntelliCode. CodeGPT (124M) [96] is then released as a multilingual model pretrained upon the CodeSearchNet corpus [136], based on GPT-2 [65]. PyCodeGPT (110M) [118] is similar to CodeGPT, apart from that it is based on GPT-Neo [17, 18] and it is pretrained on Python code solely.

As the descendant of GPT-3 [16], Codex (175B) [8] is one of the largest models released by OpenAI by far in 2021 for both natural language and programming language generation tasks. However, neither the code, the corpus, nor the parameters of Codex are available, but Codex is only accessible via the API provided by OpenAI. To bridge the gap between large powerful model and public accessibility, researchers have pretrained and open-sourced GPT-Code-Clippy (125M-1.3B, based on GPT-Neo) [21], CodeParrot (110M-1.5B, based on GPT-2) [22], and PolyCoder (160M-2.7B, also based on GPT-2) [119] as the publicly available versions of Codex. With the growth of model size, the performance increases greatly and surprisingly. Thus, more large pretrained models over 1B parameters are proposed, including CodeGen (350M-16B) [120], PanGu-Coder (317M-2.6B) [121], CodeGeeX (13B) [23], InCoder (1.3B-6.7B) [122], and FIM (50M-6.9B) [137].

Encoder-decoder. The encoder-decoder models are usually pretrained through self-supervised seq2seq tasks. The pretrained encoder-decoder models are mostly derived from the field of NLP. According to the design of pretraining tasks, we categorize these models into three classes, BART-based pretraining, T5-based pretraining, and customized pretraining. The denoising auto-encoder (DAE) is developed from BART [66]. During pretraining, BART learns to restore the noisy text (token masking, deletion and sentence permutation, etc.) back to the original clean ones. Adopting DAE as the pretraining task and BART as the base model, researchers pretrain PLBART (140M) [123] upon an extensive corpus of Java and Python text-code pairs. Models derived from T5 [68] are pretrained upon unified seq2seq tasks. The pretraining corpus of T5, namely C4 [68], is a multi-task mixture of various tasks, where each task is converted into a text-to-text format. Experimental results of CodeT5 (60M-770M) [124], which is further pretrained upon CodeSearchNet corpus [136], suggest the capacity and feasibility of code generation. Multiple T5-based pretrained models for code have emerged in the past years, including PyMT5 (374M) [125] pretrained upon Python corpus, JuPyT5 (350M) [126] upon Jupyter Notebook, and CodeEditor (60M, 220M) [138] upon code editing data. In addition, CodeRL (770M) [127] further augments CodeT5 with compilation supervision via reinforcement learning. Please refer to Subsection 5.1 for more illustrations of such augmentation.

Besides the above two classes, other large models are pretrained with customized pretraining tasks. AlphaCode (300M-41B) [69] is proposed to solve competitive program generation (please refer to Subsection 7.4 for definition). There are two pretraining tasks for AlphaCode — masked language modeling for the encoder and regular language modeling for the decoder. On the other hand, UniXcoder (125M) [95] pretrains one shared transformer to unify the pretraining process of encoder-only, decoder-only and encoder-decoder models. The trick is to leverage different self-attention masks for different mode. E.g., for decoder-only mode (regular language modeling pretraining task), the attention mask makes sure that each token can only see those in the prefix.

Adaptation. After explaining "what are pretrained models", now we introduce "how to use them", i.e., adapting the pretrained models to downstream tasks. There are two major ways to use a pretrained model — finetuning and prompting. finetuning is the prevailing way to adapt the pretrained model to a specific domain and a relatively small dataset [96]. The procedure is similar to training a model from scratch. Specifically, the practitioner takes the pretrained model as the backbone and initializes it with the pretrained parameters. Then the model is further trained upon the downstream task, updating all or part of the parameters. After several epochs, the model is well finetuned and is able to generate desirable code snippets. As the pretrained model grows larger, finetuning is less practical due to the computational resource cost. Therefore, a smaller, lighter, more efficient adaptation approach is required.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

The exploration then leads to prompting. Without changing the structure or the already pretrained parameters, prompting adds some additional sequences to the input to adapt it to downstream tasks. The additional prompt usually consists of an instruction along with several demonstrations[2)]. E.g., in code generation, the instruction could be "please generate source code satisfying the given requirement", and each demonstration is a pair of the requirement and the corresponding code. By feeding the concatenation of the instruction, the demonstrations, and the requirement into the pretrained model, the model may generate the desired code snippet [120,121]. Recently, CEDAR [128] is proposed to automatically retrieve the demonstrations similar to the natural language requirement. Note that the performance of relatively small models can be poor in this setting [15].

4.5 Waypoint for DL-based code-generation model

After sorting out deep models for code generation, we can clearly distinguish three development stages. (1) During the initial exploration, researchers directly adopt off-the-shell seq2seq models. These models trivially regard code as another natural language and directly generate token sequences [3]. (2) Subsequently, researchers find that the model capability is limited, and begin to carefully design the model architecture, exploiting unique program information. The priors, including heuristic knowledge (e.g., habits of the programmers [108–110]) and typical properties of programming languages (e.g., tree [7,12] and graph [13] structures), are explicitly introduced during the design of the model. (3) Later, transformer [14] emerges. Transformer with great capacity, and the large codebase corpara, empower the pretrained model. Different from previous classic models, due to the powerful capability, such large models learn program information and properties implicitly during self-supervised sequential pretraining. Therefore, in the era of large models, researches usually go back to sequential modeling and generate code token sequences.

Transformer can be viewed as a milestone and a watershed in the development of deep architecture for code generation, because (1) it changes the learning procedure, (2) the model capacity increases greatly, and (3) the unique program information is learned implicitly during pretraining, rather than incorporated explicitly by model design. As the model size grows larger, fewer researchers and practitioners are able to independently design, implement, or train a large model from scratch. Thence, perhaps in the upcoming future, the academic field needs to focus on analyzing, exploiting, and enhancing the existing large models for code generation. Please refer to Section 8 for more discussions of challenges and future work of large models.

5 Enhancing strategy

The backbone design of the generation model is important to the final performance, as we have introduced in the previous section. Besides that, practitioners also incorporate model-agnostic strategies to enhance code generation performance, independent with the design of the backbone architecture. The strategies mainly involve (1) augmentation, (2) sampling, and (3) post-processing. Please refer to Figure 7 [3,7,69, 127,139–163] for the taxonomy from the aspect of model-agnostic enhancing strategies.

5.1 Augmentation

In addition to the solutions categorized by structure and architecture as introduced in the previous section, researchers also introduce additional knowledge or supervisions into the process to improve the code-generation models. The augmentations are relatively independent of the design of the architecture, and we introduce these techniques individually in this subsection.

Retrieval augmentation (external knowledge). When facing an unfamiliar code fragment, a programmer may search the Internet or the codebase for a demonstration, and then imitate the retrieved examples. Based on such heuristic, researchers retrieve the external knowledge to enhance the code-generation model. For instance, Xu et al. [164] proposed to explore the online programming QA website and the API documentation, and incorporate this external knowledge into the process of code generation.

Other studies establish a text-code database as the retrieval corpus. When the requirement contains low-frequency phrases (e.g., rare features), the model may retrieve code snippets with similar phrases,

2) If there are multiple demonstrations, the setting is called few-shot. Zero-shot is a more extreme scenario, where no demonstration is provided.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

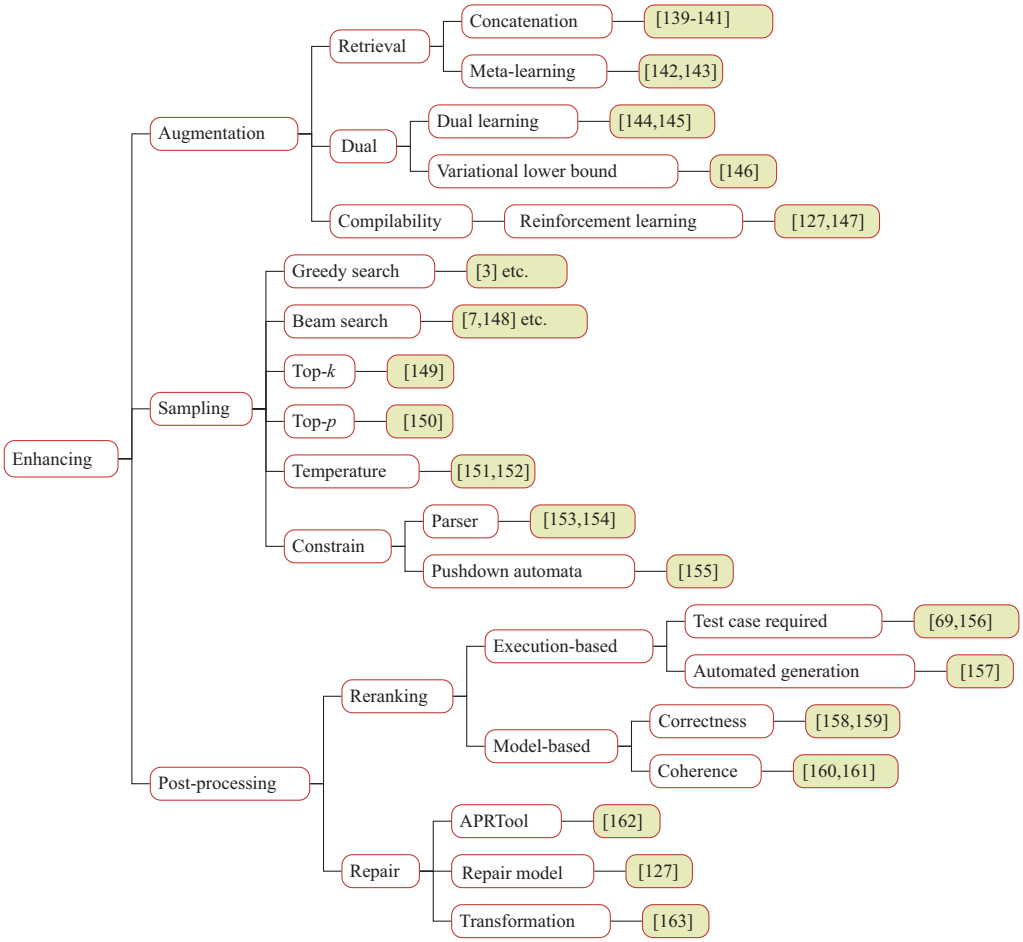

Figure 7 (Color online) Taxonomy of model-agnostic enhancing strategies for DL-based code generation.

and imitate the code demonstrations during the generation process. Such framework, known as "retrieve-and-edit" [139], decomposes the code generation process into two stages. (1) The "retrieve" stage collects text-code pairs with natural language description similar to the input requirement from the retrieval corpus. Usually, the retriever generates a representation of the input requirement, and selects the top-k pairs in the corpus according to the similarity in the representation space. (2) The "edit" stage augments the input requirement with the retrieved pairs to improve the generated code. Typically, a concatenation of input requirement and retrieved pairs is a feasible augmentation for input information. [140].

At first, the retriever is task-dependent [139]. Later, REDCODER [140] proposes to train the independent retriever via contrastive learning, where the representations of the positive pair (i.e., similar examples) are forced to be closer, while the negative pair moves away from each other. The aforementioned solutions employ the entire retrieved snippet as the prototype for editing concatenation, which may be redundant. Thereupon, researchers break down the retrieved code into fine-grained components for editing. ReCode [141] decomposes the retrieved code into much smaller n-grams of tree-expansion actions, and encourages the model to generate those n-grams during code generation. SkCoder [165] extracts a code sketch from the retrieved code by removing irrelevant tokens, and edits the code sketch into the desired code. Experiments on three code generation benchmarks show that SkCoder outperforms previous retrieval-augmented models, including REDCODER and ReCode.

Besides the concatenation, which directly reuses the retrieved examples, researchers also enable augmentation through the idea of model-agnostic meta-learning (MAML). From the perspective of meta-learning, each text-code pair can be regarded as a "task", i.e., the retrieved examples make the meta-training data, and the code to be generated by the model is the meta-test data. The meta-training datasets vary for different meta-test examples, since the retrieved examples vary for different natural language requirements. For each meta-test requirement, they train a model (randomly initialized) upon

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

the meta-training data to generate the corresponding code. PT-MAML [142] first proposes to leverage the MAML learning protocol in semantic parsing[3]. It leverages support vector machine (SVM) as the retriever, and regards the support vectors as the similar examples. Therefore, the retrieved example set is also called a support set. Later, Seq2Action [143] proposes to train a neural retriever on the training set (the original training set, not the meta-training set) to obtain the support set.

Dual augmentation (auxiliary task). Introducing a dual task into the procedure as auxiliary may improve the performance of code generation. Code summarization [98–100] is the dual of code generation, i.e., the former generates the natural language description given the code snippet, while the latter generates code given the natural language description. Formally, code generation and code summarization model the probability distribution of $p(c|x)$ and $p(x|c)$, respectively, where c and x refer to the code snippet and the natural language requirement. With two additional language models, $p(x)$ for natural language and $p(c)$ for programming language, we can establish the equation $p(x) \cdot p(c|x) = p(c, x) = p(c) \cdot p(x|c)$ through Bayes law. Dual learning [166] exploits such property, by confining the joint distributions (i.e., $p(c, x)$) modeled from two directions (i.e., $p(x) \cdot p(c|x)$ and $p(c) \cdot p(x|c)$) to be identical during training. Since the language models (i.e., $p(x)$ and $p(c)$) with good performance are easier to obtain, dual learning may help better modeling $p(c|x)$ for code generation and $p(x|c)$ for code summarization.

Wei et al. [144] proposed to leverage the duality between code generation and code summarization, by minimizing the difference between the joint distributnions. DIM [146], on the other hand, maximizes the variational lower bounds of the joint distributions to establish the duality. Later, based on the work of Wei et al., CO3 [145] further incorporates the code retrieval task (i.e., finding the most relevant code according to the given natural language query) into the framework, by confining the representations of code and requirement to be identical in addition to the duality. These studies hold promise for further improvements of code generation, inspiring researchers to exploit other plausible duality.

Compilability augmentation (supervision signal). Programming language implies an important feature — compilability, i.e., the code snippet must conform to the lexical, syntactical, and grammatical rules. Based on this feature, researchers add new supervision signals into the training process to make sure the generated code can be successfully compiled. Reinforcement learning techniques are adopted to achieve such a goal. CompCoder [147] takes the actual compiler feedback as the reward for reinforcement learning. Later, researchers have also made use of similar techniques into the pretrained models, such as aforementioned CodeRL [127].

Besides treating the compilability constraint as a supervision signal, post-processing is another feasible solution, by filtering the output examples from the model. We will introduce the post-processing techniques in the following subsections.

5.2 Sampling

At each time-step, the code-generation model predicts a probability distribution over the vocabularies or tree expansion actions, etc. Therefore, sampling from the distribution is essential. E.g., in sequential modeling, we need to decide which token to generate at each time-step based on the predicted distribution. For convenience of description, we limit the scenario to sequence modeling. The introduced techniques can be easily adapted to other structures and architectures.

Greedy & beam search. In the very early stage, researchers simply adopt argmax, i.e., the greedy strategy, by selecting the most probable token [3]. However, greedy search is trivial and naive, since the major drawback is that the optimal at each time-step may not lead to the global optimal. Therefore, beam search is then introduced, by maintaining a top-k most probable beam (refer to Figure 4 for a demonstrative example) [7, 148]. Beam search is a well-established algorithm for generating sequences in tasks where the desired output is rather short [167, 168], such as the fine-grained code generation task (line-by-line code paraphrasing and statement-level generation, please refer to Section 7).

Top-k & top-p sampling. Beam search usually prefers to keep short generations in the beam, and it also may result in highly repetitive sequences, which damages the diversity of the code. To enable diverse generation of any length, top-k sampling [149] and top-p [150] sampling are proposed. At each time-step, both sampling techniques randomly select a token according to certain probability distributions.

3) Although we have stated that semantic parsing, which maps from natural language description to SQL queries, is not included in our survey. PT-MAML is still highly related to code generation, which is the foundation of other researches. Therefore, we introduce this study in the survey.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

Top-k sampling [149] filters the k most probable next tokens, and redistributes the probability among these k tokens for sampling. Top-p sampling, also known as nucleus sampling, further enhances the flexibility. Instead of redistributing probability upon the k most probable tokens, top-p sampling rescales the probability among the smallest token set whose cumulative probability exceeds the threshold p. Top-p sampling allows the token set size to adapt dynamically to the distribution predicted by the code-generation model.

Temperature sampling. To improve diversity and creativity, another temperature sampling is introduced [149,151,152]. Temperature sampling employs a variant of softmax to redistribute the probability distribution. To redistribute the original distribution $(p_1, \ldots, p_n)$ to a new one $(q_1, \ldots, q_n)$, the formulation is as below:

$$q_i = \frac{\exp(\frac{\log(p_i)}{T})}{\sum_{j=1}^{n} \exp(\frac{\log(p_j)}{T})}, \tag{3}$$

where $T > 0$ is the temperature factor. By adjusting T, the practitioners may control the quality and the diversity of code generation. (1) At a low temperature, the distribution variation is rather small and the prediction is highly confident. Therefore, the quality is high, as the generated code closely follows the distribution of the training data; while the diversity is low, as it may lead to repetitive or conservative generation. (2) At a high temperature, the situation is reversed. The generated code becomes diverse, but may contain more errors. The trade-off between quality and diversity is essential, and hence the appropriate temperature setting is important [69].

Constrained decoding. The programming languages, such as Type-2 Chomsky grammars [169], are strictly constrained by the lexical, syntactical and grammatical rules. Sampling without constraints may violate the syntactical and grammatical rules (e.g., produce keywords at an inappropriate location), resulting in compilation failures and incorrect code generations. Researchers tackle this issue by adding constraints to control the output [153–155], i.e., constrained decoding. During generation, the model is allowed or is forbidden to produce certain tokens according to the grammatical constraints. E.g., when the model needs to refer to a variable, it can only choose from those already defined within the scope. At each time-step, the candidate token set changes, therefore, researchers usually employ a compilation module to track the candidate tokens. Picard [153] employs the lexer and the parser to track the constraints. Similarly, Synchromesh [154] adopts context-free and context-sensitive parsers to constrain the generation process. CodePAD [155], on the other hand, designs a pushdown automata to regularize the generation distribution.

5.3 Reranking

Given sufficient trials, the modern code-generation model, such as Codex [8], may be capable to produce an executable and correct code snippet. However, considering the efficiency of solving problems and the users' experience, it is important for models to give the correct code in only a few trials. In order to search for the correct program, the reranking technique is adopted by researchers. In other words, these approaches perform a large scale sampling to collect a set of candidate code snippets, and then rerank the candidates to select the (more likely) correct ones for final evaluation. We categorize these approaches by heuristics in the design of the rerankers: (1) the execution-based reranking and (2) the model-based reranking.

Execution-based. Programmatic solutions with the same semantics inherently yield identical outcomes, given a consistent set of inputs. I.e., amongst a pool consisting of correct code snippets, the outputs maintain identical across all test cases, while conversely, the erroneous ones diverge in their execution results. This observation has led to the advent of execution-based reranking methodologies as proposed by various researchers [69,156,157]. These approaches cluster the generated code snippets based on the outcomes of their executions. Specifically, the correct programs produce the same execution results, leading to a large cluster, while other programs with error produce different execution results, leading to multiple much smaller clusters. The reranking score assigned to each candidate is the size of the cluster it belongs to. Candidates selected from the larger clusters are considered more likely to be correct.

AlphaCode [69] first collects the candidates that pass the accessible test cases (provided by the dataset). An independent model then generates additional input cases (without the oracle outputs) to further cluster the collected candidates, and AlphaCode selects k code snippets from the top-k largest clusters

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

(each example from one cluster) as the final generation. Similarly, MBR-EXEC [156] directly clusters the candidates by the inputs from the accessible test cases provided in the dataset, without additional input cases generation. MBR-EXEC selects the code from the largest cluster as the final prediction. Both of these approaches require existing test cases, which may not be provided in many scenarios. CodeT [157] solves this issue by leveraging a separate model to generate test cases (input-output pairs) based on the natural language requirement. The candidates, which pass the same subset of the generated test cases, are put into one cluster. The reranking score of each cluster is the product of the cluster size and the number of passed test cases. At last, CodeT selects the example from the cluster with the highest score as the generated code snippet.

Model-based. The execution-based reranking requires the context code (imported packages, global variables, etc.), test cases provided by the dataset or generated by an additional model, and the execution environment (e.g., the compiler, dependencies), making these approaches less feasible. Thence, the model-based reranking techniques propose to predict the manually designed score with an additional model.

Following the execution-based rerankers, CodeRanker [158] exploits a 172M model to directly classify whether the candidate code is correct. Specifically, the model takes the natural language requirement and the candidate code as input, and predicts the correctness label without unit test cases or execution. LEVER [159] combines the ideas of execution result clustering and DL model prediction. It executes the candidates against a certain input to obtain the execution result, constructing the clusters. Then, LEVER [159] predicts the verification probability (i.e., to what extent the candidate code is correct) based on the requirement, the candidate code, and the obtained execution result, with a deep model. LEVER defines the reranking score for each cluster as the sum of the score of all candidates within the cluster, which is the product of the generation probability from the deep model and the verification probability.

Other approaches define the coherence between the requirement and the candidate as the reranking score. Yin et al. [160] proposed to rerank according to two main features. (1) The reconstruction model measures the coherence by the likelihood of generating the requirement from the candidate code. (2) The matching model directly predicts the coherence. Coder-Reviewer [161] takes a further step by introducing the mutual information as the reranking score. To be specific, the score is defined as $\log p(x|c)p(c|x)$, where $p(c|x)$ is the "Coder" probability from the code-generation model and $p(x|c)$ is the "Reviewer" probability from the additional code summarization model. Note that this design shares similar intuition with the dual augmentation introduced in Subsection 5.1.

5.4 Repair

Reranking is feasible to pick the correct candidate from a sample set, but the sampling size needs to be large enough to contain the correct code. The large scale sampling may not be quite practical in the real world. On the other hand, empirical studies show that the generated code often contains common mistakes made by human programmers [162]. It then inspires another post-processing technique — automated program repair (APR). Through APR tools and edit model, it is plausible to fix the generation models' own mistakes [162]. Self-edit [170] takes inspiration from the process of human programming and proposes a generate-and-edit approach that utilizes execution results of the generated code from large scale models to improve the code quality on the competitive programming task. CodeRL [127] employs an additional APR model to fix bugs in the generated code. Once the generated code cannot pass the test cases, the generated buggy code along with the error information is fed into the APR model for further post-processing. Jigsaw [163] adopts a similar idea with mutation testing. It transforms the generated code (variable name transformation, argument transformation, etc.) to pass the test cases and other quality checks.

5.5 Waypoint for model-agnostic enhancing

In the era of large models, the model-agnostic enhancing strategies become more important and worth exploring. The capacity of the pretrained model has grown to or even exceeded the threshold for a qualitative revolution, so how to better harness such power for automated SE may be more important than pushing forward large models themselves. Without designing or manipulating the backbone deep model, enhancing strategies, such as sampling and post-processing, are cost-effective to better exploit the power of large models.

Current large models are mostly gray-boxes or black-boxes (e.g., ChatGPT [24] is only accessible through OpenAI's API), without knowing the internal status. Therefore, enhancement can only be

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

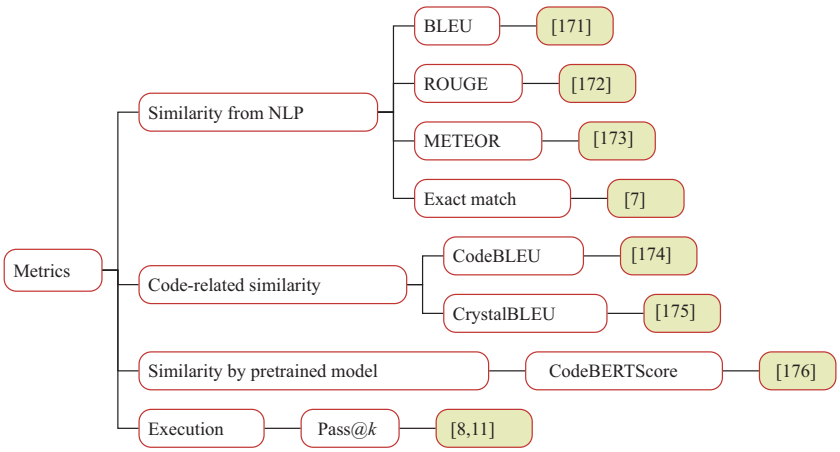

Figure 8 (Color online) Metrics taxonomy for metrics in DL-based code generation.

made at the input or the output. On the input side, researchers can collect similar code snippets through retrieval augmentation to boost few-shot performance. On the output side, we may improve the generation quality via post-processing, such as reranking and repair. For gray-box large models, where the logits (unnormalized probability) or the probability distributions are available, sampling techniques may also improve diversity and quality of generation.

6 Code generation metrics

The performance indicator of code generation can be roughly divided into two types — similarity and pass rate. To evaluate similarity, a ground-truth (reference) code is needed, and the indicator measures the textual, structural or semantic similarity between the generated code and the ground-truth. In other scenarios, the benchmark provides test cases for unit test. Execution-based pass rate can be applied. Please refer to Figure 8 [7, 8, 11, 171–176] for the metrics taxonomy.

6.1 Similarity from NLP

In general, almost all existing metrics from the NLP field can be adopted in code generation. We list the commonly adopted the ready-made metrics as below. These metrics compare the similarity between the generated code and the ground-truth as a sequence matching problem. Therefore, for those benchmarks which provide only the ground-truth code as reference, researchers usually employ such performance indicators.

BLEU. BLEU [171] is one of the first and one of the most widely employed performance indicators in various sequence generation tasks. In general, in code generation, BLEU measures the quality by comparing the overlap between the n-grams in the generated code and the ground-truth code. I.e., BLEU measures the n-gram precision (the percentage of the n-grams in the generated code that matches the ground-truth). Therefore, the BLEU score always ranges from 0 to 1. Higher BLEU score suggests the higher quality of the generated code. In practice, researchers usually take unigram to 4-gram to compute the BLEU score.

ROUGE. Similar to BLEU, ROUGE [172] measures the n-gram recall — the percentage of the n-grams in the ground-truth that is matched by the generated code. Higher ROUGE indicates the more reliable generated code, since ROUGE represents the ground-truth coverage of the generated code. ROUGE is usually utilized with complementary BLEU in practice. There are multiple available ROUGE scores, including ROUGE-N, ROUGE-L, and ROUGE-W.

METEOR. As aforementioned, BLEU and ROUGE each refers to n-gram precision and recall, respectively. METEOR [173] considers both indicators in a similar manner of F-score. Furthermore, to avoid false negatives (e.g., a synonymous word may be judged as incorrect), METEOR takes synonyms and cognates into consideration. METEOR also ranges from 0 to 1, and higher METEOR means that the generated code is more similar to the ground-truth.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

Exact match. Exact match [7] measures the hard consistency between the generated code and the ground-truth code. Exact match is 1 only when the generated code is exactly the same as the ground-truth, otherwise, it is 0 and the generated code is regarded as incorrect. For those fine-grained generations (e.g., line-by-line code paraphrasing, statement-level generation), researchers often adopt exact match, ensuring that the generated code snippets, which are considered correct, can produce correct execution results.

6.2 Code-related similarity

The previously introduced similarity metrics from the field of NLP are handy to utilize, but they regard the code snippet as a regular text and do not consider the particular properties of the code. Researchers have begun to introduce code properties into the metrics, creating indicators specialized for code generation.

CodeBLEU. CodeBLEU [174] extends BLEU with structural information. To be specific, besides the n-gram precision of BLEU, CodeBLEU also takes the structural properties into account, including the keyword correctness, the AST matching degree, and the data flow matching degree. Some studies, such as CodeT5 [124], have begun to evaluate the quality of the generated code via CodeBLEU.

CrystalBLEU. CrystalBLEU, on the other hand, removes the trivially shared n-grams to better capture the code semantics. The authors find that, unlike natural languages, the code snippets usually share some trivial n-grams (e.g., ") { if ("). Such trivially shared n-grams can make it difficult for BLEU to recognize the semantics, making the similarity score inflated. Hence, CrystalBLEU removes these trivially shared n-grams from both the generated code and the ground-truth, and calculates the BLEU score upon the remaining tokens.

6.3 Similarity by pretrained model

Recently, in the field of NLP, leveraging the large pretrained model to evaluate the generated sentence attracts much research attention. Many new quality indicators are proposed, such as BERTScore [177] and SBERT [178]. Based on similar ideas, researchers have also made similar attempts in code generation, and recently proposed CodeBERTScore [176].

CodeBERTScore. Inspired by BERTScore [177], CodeBERTScore [176] computes token-level similarity between the generated code and the ground-truth over the contextual representations produced by the pretrained CodeBERT model [93]. First, we obtain the contextual token-level representations, and generate a similarity matrix between each token of the generated code and that of the ground-truth. We are then allowed to calculate the precision similarity and the recall similarity. Specifically, for each token in the generated code, we collect its similarity to the most similar ground-truth token, and compute the mean value as the precision. In turn, we also compute the recall. At last, we can compute CodeBERTScore the same way of the F-score.

6.4 Metrics via execution

There are some disadvantages in the similarity-based metrics. (1) The similarity metrics such as BLEU cannot guarantee the correctness of the generated code. One misplaced token, which is not a big deal to BLEU, may violate the compilation rule, resulting in erroneous code. (2) The code snippets with similar semantics can vary a lot, while on the other hand, those that look very similar can have vastly different semantics. It is inappropriate to judge based solely on similarity, such as BLEU, without considering properties of code. Although CodeBLEU has considered the AST and the data flow, such semantics capturing issue is still challenging.

Thence, researchers draw idea from software testing by executing the generated code snippet against unit tests. The generated code is supposed to either behave consistent with the ground-truth code, or produce identical execution results as the test cases. For benchmarks that provide execution environment and test cases (or the ground-truth code), researchers usually employ execution-based metrics.

Pass rate. Pass rate [11], or pass@k [8], is a widely employed correctness indicator, particularly when test cases are available. Given a budget size of k, i.e., the generation model is allowed to submit k examples for evaluation against one particular requirement. The generation is considered valid if and only if one of the k submissions passes all test cases (unavailable during generation). Otherwise, none of the k submissions passes all test cases, and the code-generation model fails to meet this requirement. Assessing the validity of the generated code through execution provides a more accurate evaluation of

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

the performance of a code-generation model. Therefore, the pass rate has been widely adopted in various method-level code generation tasks [8,179] and competitive program generation tasks [15,69], which often provides test cases (please refer to Section 7 for more details).

6.5 Waypoint for code generation metrics

Although there have been many metrics, the method for evaluating code generation remains a debatable question. (1) On the one hand, execution-based metrics can effectively determine the correctness of the generated code, but it is very demanding — it demands an execution environment and multiple test oracles. Furthermore, metrics such as pass rate are coarse-grained. It treats all mistakes equally, regardless of severity. E.g., wrong parameters in an API invocation are considered less serious than using a wrong algorithm, but the pass rate regards them both as equally wrong, which is not reasonable. The former mistake can be corrected easily, and the (although wrong) generated code can still aid automatic SE to a certain extent; while to correct the latter, we must rewrite the whole code snippet from scratch. (2) On the other hand, similarity-based metrics, either textual similarity (e.g., BLEU) or code-related similarity (e.g., CodeBLEU with structural properties), provide a fine-grained evaluation, but without correctness guarantee. Also, textual or fine structure similarity does not equate to program semantic agreement. programs with the same semantics may differ greatly from a textual point of view (e.g., implemented by different algorithms), and similar programs do not imply semantic equality (e.g., differ by one operator).

Therefore, the existing evaluation still needs to be explored. Researchers may make a trade-off between coarse-grained execution and fine-grained similarity. Proposing a new performance indicator deserves further study.

7 Code generation task & benchmark

Code generation searches within the vast space of legitimate programs satisfying the requirement expressed in natural language. We categorize the existing tasks into four classes according to the difficulty and the generation granularity: (1) code paraphrasing, (2) statement-level generation, (3) method-level generation, and (4) competitive program generation. Please refer to Table 1 for demonstrative examples of these tasks, Figure 9 [8,10,11,15,19,20,23,69,84,107,118,120,136,165,179–187] for the task taxonomy, and Table 2 for the brief summary of all benchmarks included in this survey. As the difficulty increases, the model is requested to comprehend the functionality description, combine the existing components (e.g., logical and arithmetic operations, APIs, etc.), and design algorithms to meet the requirement. We do not limit all code generation tasks to these four tasks, since as the technology develops, there will be more types.

It is clear that as the model capacity grows, code generation tasks that can be accomplished become increasingly difficult. During the era of the classic models, researchers mainly focus on rather trivial tasks, such as code paraphrasing. After the transformer architecture and large models, researchers are allowed to handle even competitive program generation. Hopefully, as large models and the techniques for utilizing them evolve, we will be able to accomplish more challenging tasks, such as long file generation or even project generation.

7.1 Code Paraphrasing

As demonstrated in the top row in Table 1, a very detailed requirement (or line-by-line instructions), is fed into the code paraphrasing model. The model strictly follows the description, and converts it into code line by line. Considering the basic logic and the general structure are already provided in the requirement, the code paraphrasing model actually performs the role of a "translator". Hence, code paraphrasing is rather easy in the field of code generation. The focus is to ensure that such paraphrasing to meets the compilation requirements. But still, code paraphrasing is still considered a fundamental code generation task.

Django. Django [10] is an English-to-Python code paraphrasing benchmark. The authors hired a software engineer to create the line-by-line instructions for the Django project, which is a web design framework in Python language. The benchmark contains 16000 training, 1000 validation and 1000 test

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

Table 1 Examples of code generation tasks in different granularity

Task	Requirement	Code (in Python)

Code paraphrasing	Define a method sort with an argument A.
	Loop i from 1 to len(A) - 1.
	Loop j from 0 to len(A) - i - 1.
	If A[j] is smaller than A[j+1],
	swap these two elements.
	Return the sorted list A.
		def sort(A):
		for i in range(1, len(A)):
		for j in range(0, len(A) - i):
		if A[j] < A[j+1]:
		A[j],A[j+1] = A[j+1],A[j]
		return A

Statement-level generation	Check if all elements in a given list A are identical.	len(set(A)) == 1

Method-level generation	Sort the given list from small to large.	def sort(A):
		for i in range(len(A) - 1):
		minIdx = i
		for j in range(i + 1, len(A)):
		if A[j] < A[minIdx]:
		minIdx = j
		A[i], A[minIdx] = A[minIdx], A[i]
		return A

Competitive program generation	Problem: H-Index
	Given a list of citation counts,
	where each element is nonnegative,
	compute the h-index.
	The h-index is the largest h such that
	h papers have at least h citations.
	
	Example input: [3,0,6,1,4]
	Example output: 3
		def h_index(c):
		n = len(c)
		if n > 0:
		c.sort()
		c.reverse()
		h = 0
		while h < n and c[h] - 1 >= h:
		h += 1
		return h
		return 0

examples, each consisting of a single line of Python code and a human-written natural language description.

Euler. Similar to Django, Euler [10] is a Japanese-to-Python benchmark. Unlike the Django project, the Euler website collects a set of arithmetic problems designed for the junior programmers to practice. The annotation pipeline involves two independent software engineers. The first engineer manages to write 177 functions, each of which solves a problem in Euler. The second Japanese engineer then writes instructions corresponding to each line, leading to 722 annotated Python code lines. The training set of Euler contains 650 examples, leaving the rest 72 to the test set.

SPoC. SPoC [11] is a C++ code paraphrasing benchmark. The raw C++ corpus is scraped from Codeforces, which is an online open judge system. Thanks to the open-sourced property of Codeforces, besides the solution code, the test cases are also collected. The authors filter out the examples that are difficult to annotate (e.g., programs with macros, classes, structures, switches, and mallocs, etc.). 59 crowd workers on Amazon Mechanical Turk are recruited to write instructions for each C++ code line. The annotation process is manually inspected by the authors. SPoC contains 18356 programs, along with human-authored line-by-line instructions and corresponding test cases. There are two test sets for SPoC, split according to the Codeforces problem and the annotator respectively. This setup is to evaluate the generalization of the code-generation model to unseen problems and annotation styles. The problem-level test set contains 1820 examples, and the annotator-level test set contains 1752 examples. The remaining examples are divided into training set and validation set in a nine-to-one ratio.

7.2 Statement-level generation

In statement-level generation, given a rather high-level functionality requirement, the model is required to produce the corresponding statement-level code. The program logic, the data structure, or the necessary API are not described in detail in the given requirement, distinguishing statement generation from code paraphrasing. Please refer to the second row in Table 1 for an illustrative example of statement-level generation. In the real software development scenario, the implementation of many function requirements is rather short, or even within one single line of statement. The automated generation of such statements may greatly improve the efficiency of the developers, by saving time of implementation or searching the Internet. Thereupon, the huge demand makes the statement-level generation of high research value.

CoNaLa. CoNaLa [180] is a Python statement generation benchmark. The examples of CoNaLa are mined from Stack Overflow Q&A. The examples are automatically filtered and manually reviewed for quality assurance. The benchmark contains 2370 examples for training and 500 examples for evaluation. In addition to CoNaLa, the authors also released a large corpus, namely CoNaLa-mined, to augment the training set, which contains 598237 examples. However, CoNaLa-mined is not filtered and reviewed and

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

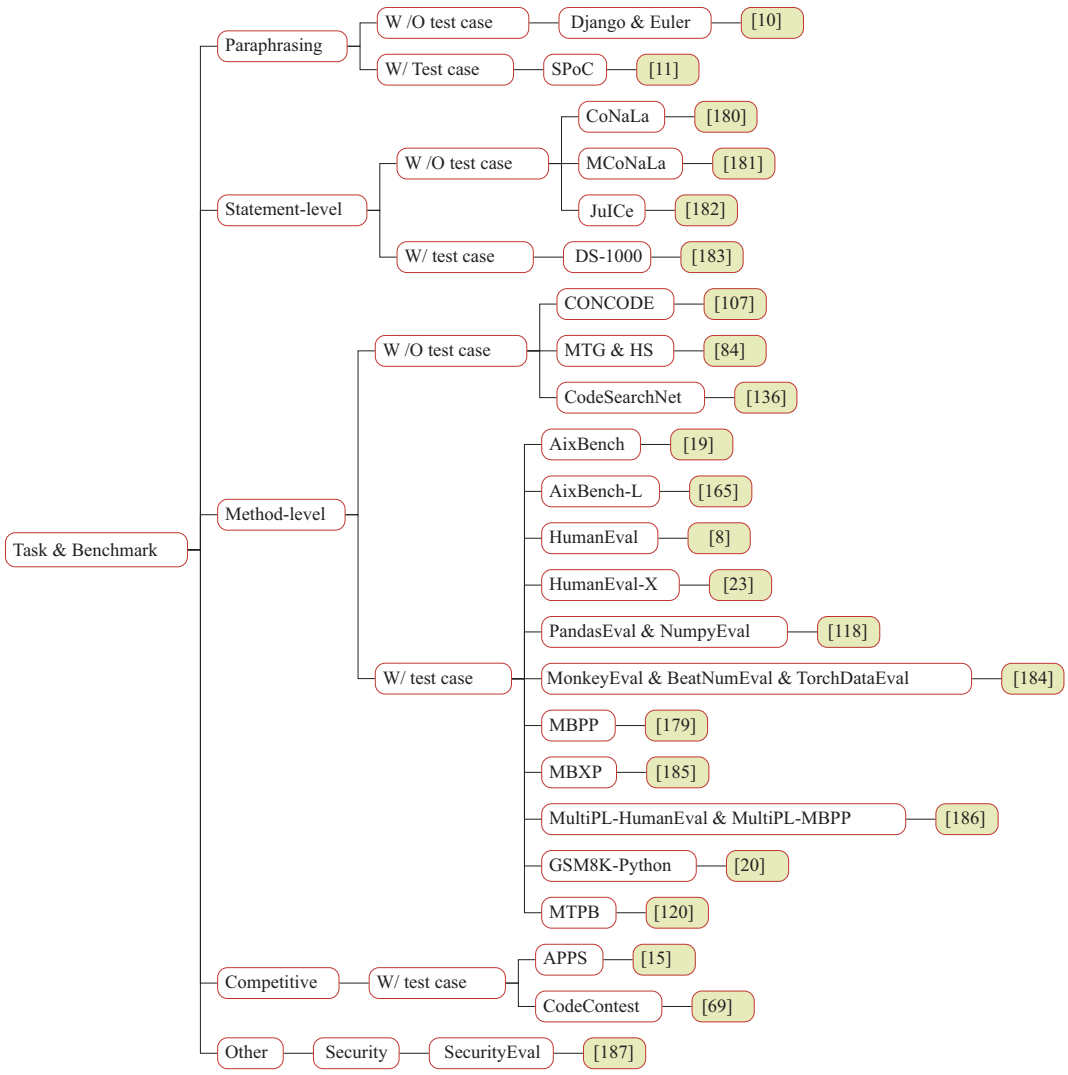

Figure 9 (Color online) Task taxonomy for DL-based code generation.

hence contains many low-quality examples.

MCoNaLa. Following the similar methodology of CoNaLa, MCoNaLa [181], is proposed as a benchmark for multilingual statement generation. As an extension of English-to-Python of CoNaLa, MCoNaLa consists of 341 Spanish-to-Python pairs, 210 Japanese-to-Python pairs, and 345 Russian-to-Python pairs. All examples in MCoNaLa are manually reviewed and annotated.

JuICe. The target language of the JuICe benchmark [182] is Jupyter notebook, and the requirements are in English. The authors collect and filter publicly available Jupyter notebooks from GitHub created before May 2019, and process them into code cells. There are 1518049 examples in JuICe. Each example is a triplet of the natural language description, the target code cell, and the context (i.e., code cells above the target code cell). The authors manually craft a validation set of 1744 examples and a test set of 1981 examples.

DS-1000. DS-1000 [183] is a domain specific English-to-Python benchmark for data science, covering libraries such as NumPy and Pandas. The authors collect Q&A pairs from Stack Overflow. Then they filter and rewrite the problem (i.e., the requirement) and the solution (i.e., the target code statement). Furthermore, to ensure executability, the authors also (re)write the context code and the test cases. As a result, DS-1000 contains exactly 1000 examples, each consisting of the problem, the target code, the context code, and the test cases.

Since DS-1000 guarantees the executability, a pass rate is employed to indicate the functional correctness. Besides, the surface-form constraint is introduced to make sure the presence or absence of certain keywords or APIs. The model may take a short cut, by using some trivial and repetitive operations, to

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

Table 2 Brief summary and comparison of code generation benchmarks included in the survey[a)]

	Benchmark	Language		Example #			Unit test	Domain
		NL	PL	Train	Dev.	Test		
	Django	EN	PY	16000	1000	1000	✘	Web library
	Euler	JP	PY	650	–	72	✘	Arithmetic
	SPoC	EN	C++	14784	–	3572	✘	Codeforces
	CoNaLa	EN	PY	2370	–	500	✘	StackOverflow
	MCoNaLa	SP/JP/RU	PY	–	–	896	✘	Multiple NL
	JuICe	EN	Jupyter	∼1.5M	1744	1981	✘	GitHub
★	DS-1000	EN	PY	–	–	1000	✔	Data science
	CONCODE	EN	Java	100000	2000	2000	✘	GitHub
	MTG	EN	Java	11969	664	664	✘	Card game
	HS	EN	PY	533	66	66	✘	Card game
★	CodeSearchNet	EN	Multi.(6)	∼2.3M in total			✘	GitHub
	AixBench	EN/ZH	Java	–	–	336	✔	GitHub
	AixBench-L	EN/ZH	Java	19000	1000	–	✔	GitHub
★	HumanEval	EN	PY	–	–	164	✔	Synthesized
	HumanEval-X	EN	Multi.(4)	–	–	164×4	✔	Multiple PL
	MultiPL-HumanEval	EN	Multi.(18)	–	–	∼160×18	✔	Multiple PL
	PandasEval	EN	PY	–	–	101	✔	Public library
	NumpyEval	EN	PY	–	–	101	✔	Public library
	MonkeyEval	EN	PY	–	–	101	✔	Private library
	BeatNumEval	EN	PY	–	–	101	✔	Private library
	TorchDataEval	EN	PY	–	–	50	✔	Unseen library
★	MBPP	EN	PY	384	90	500	✔	Synthesized
	MBPP-sanitized	EN	PY	–	–	427	✔	Sanitized
	MBXP	EN	Multi.(12)	–	–	∼960×12	✔	Multiple PL
	MultiPL-MBPP	EN	Multi.(18)	–	–	∼400×18	✔	Multiple PL
	GSM8K-Python	EN	PY	7500	–	1000	✔	Math
	MTPB	EN	PY	–	–	115	✔	Multi-turn
★	APPS	EN	PY	5000	–	5000	✔	Algorithm
★	CodeContest	EN	Multi.(3)	13328	117	165	✔	Algorithm
	SecurityEval	EN	PY	–	–	130	✘	Security

a) Note that the entry "Multi.(x)" suggests x languages in total. Arrows on the right indicate derivation relationships between benchmarks. The ★ mark suggests that the corresponding benchmark is currently widely employed for large model evaluation.

complete the required problem. Such behavior cannot be detected by functional correctness. E.g., the vectorization is usually done via APIs, but one may also achieve the same goal using for looping, which is certainly undesirable. The surface-form constraint can spot those generated code statements that do not meet the actual requirement.

7.3 Method-level generation

The difficulty and the generation granularity of method-level generation are further increased. Given a natural language requirement of the method functionality, the model is required to produce a corresponding method- or class-level snippet. Method generation has a wide range of real-world application scenarios. Many automated tools have been deployed and are widely utilized, such as Copilot powered by Codex [8].

CONCODE. CONCODE [107] is a large-scale English-to-Java method generation benchmark. The authors collect Java projects from GitHub, forming the raw corpus. The raw corpus is then divided into train, validation and test sets according to the project. This splitting evaluates the model to generalize to unseen projects. The CONCODE dataset is already cleaned, removing leaked and inherited examples. Each example is a triplet of (1) the requirement, which is the Javadoc comment of the method, (2) the target method, which corresponds to the method body, and (3) the environment, which refers to the class member variables (name and data type) and the other class member methods (return type, name and parameter list). CONCODE has 100000 examples for training, and 2000 examples for validation and

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

evaluation each.

MTG. MTG [84] is an English-to-Java benchmark in the specific domain of Magic the Gathering card game. MTG is collected from the open-sourced implementation of the corresponding trading card game. The requirement of MTG is the semi-structure description. There are six singular value fields, (e.g., attack and defense) and four textual fields (e.g., name and description). Based on the requirement, the generation model is supposed to produce an implementation of the required card. In general, MTG contains 11969 training examples, 664 validation samples, and 664 test samples.

HS. HS [84] is very similar to MTG, besides it is an English-to-Python benchmark of the Hearthstone card game. The requirement of HS contains eight singular value fields and two textual fields. HS is rather small, with 533 examples for training, 66 examples for validation and evaluation each.

CodeSearchNet. CodeSearchNet [136] is a multilingual method generation benchmark. The authors crawl open-source projects from GitHub and collect pairs of the English comment and the method body. Examples with too short comments or code snippets are removed during filtration. CodeSearchNet contains 2326976 pairs of examples covering six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). Since CodeSearchNet does not provide test cases and the executable environment, researchers usually employ BLEU and exact match as performance indicators for CodeSearchNet. In addition, the benchmark also plays an important role as the pretraining corpus [93, 131].

AixBench. AixBench [19] is a Java method generation benchmark consisting of one automated test set and one manual test set. (1) The automated test set includes 175 examples. Each example contains a well-described natural language description, a signature of the target function, and a set of unit tests. The pass rate is employed as the evaluation metrics in the automated test set. (2) The manual test set includes 161 examples and each example contains only a natural language description. The functions are generated based on given descriptions solely. The generated methods are evaluated manually from three dimensions, i.e., correctness, code quality, and maintainability. Please refer to the paper of the three metrics [19].

AixBench-L. AixBench-L [165] is a Java code generation benchmark extended from AixBench [19]. AixBench-L regards the original AixBench as the test set and constructs the training (19000 examples) and validation (1000 examples) sets by collecting test-code pairs from GitHub. Specifically, the authors mine Java open-sourced projects with at least 30 stars. The projects already included in AixBench and the auto-generated methods are removed. Further filtration extracts methods that have an English docstring, less than 1024 tokens, and more than one line. Following the setting of AixBench, during evaluation, AixBench-L also executes the unit tests to verify the correctness of the generated method.

HumanEval. HumanEval [8] is a Python code generation benchmark, with 164 test examples. Each example is a hand-written programming problem consisting of a natural language requirement, a method signature (declaring the method name, the return type and the parameter list), and several unit tests. The model generates the function body based upon the requirement and the signature. The evaluation metric is the pass rate upon the test cases.

HumanEval-X. HumanEval-X [23] is a multilingual version of HumanEval (Python only). The dataset format of HumanEval-X is identical to HumanEval, but the former covers more programming languages. More specifically, the authors translate the signatures and the unit tests to other programming languages (C++, Java, JavaScript and Go), and construct a benchmark covering the five programming languages (including the original Python version)

PandasEval. As the name suggests, PandasEval [118] is a domain-specific method or block generation benchmark for the Pandas library in Python. The examples are collected from Stack Overflow Q&A's, tagged with Pandas and highly voted. Two experienced programmers are invited to manually check the examples, resulting in 101 test examples. Each example corresponds to a programming problem of Pandas, containing the context code, the target method body (or block), and multiple test cases.

NumpyEval. NumpyEval [118] is almost the same as PandasEval, apart from the domain. NumpyEval specifically targets the Numpy library in Python. The benchmark also contains 101 test examples.

MonkeyEval. MonkeyEval [184], modified from PandasEval, is designed to evaluate the method generation model against the unseen library. The authors automatically craft a pseudo private library, named Monkey, by modifying all Pandas-related keywords. E.g., "pandas" is converted to "monkey", "dataframe" is converted to "knowledgeframe", etc. MonkeyEval converts all examples in PandasEval, leading to 101 test examples. The authors ensure that the pretrained models have never seen APIs in MonkeyEval.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

BeatNumEval. BeatNumEval [184] is modified from NumpyEval, in the same way of PandasEval to MonkeyEval. BeatNumEval also has 101 test examples.

TorchDataEval. The TorchData library in Python was released in 2021, which is more likely to be unseen to the pretrained models than those libraries released years or decades ago. Therefore, TorchDataEval [184] is proposed to evaluate the model against the unseen TorchData library. The authors manually craft 50 test examples based on TorchData APIs for TorchDataEval. Each example contains a natural language requirement, a Python method, and some test cases.

MBPP. MBPP [179] is an English-to-Python method generation benchmark. It is manually constructed by a crowd-sourcing who has basic knowledge of Python. The crowd-sourcing participant is asked to write a relatively short problem description, a self-contained Python method (i.e., it runs without any external library support and does not print any string on the console), along with three test cases. MBPP contains 974 examples, ranging from mathematical calculation to basic data processing (e.g., list or string processing).

MBXP. Derived from MBPP [179], MBXP [185] is a multilingual benchmark. The description is still in English, while the program and the test assertions in MBPP are converted into programming languages other than Python for MBXP. The authors design a framework to complete such conversion, resulting in 12 datasets. Each dataset in MBXP contains 974 examples (identical to MBPP), and corresponds to one programming language (Java, JavaScript, TypeScript, Go, Ruby, Kotlin, PHP, C#, Scala, C++, Swift, and Perl).

MultiPL-HumanEval. MultiPL-E [186] is a system capable of translating unit test-driven programs into other programming languages. The system supports 18 programming languages and the corresponding unit tests. The authors utilize MultiPL-E to translate HumanEval [8] to MultiPL-HumanEval [186] in multiple programming languages. The dataset format of MultiPL-HumanEval is the same as the original HumanEval.

MultiPL-MBPP. MultiPL-MBPP [186] is a multilingual benchmark translated from MBPP [179] by MultiPL-E [186]. There are at most 18 programming languages in MultiPL-MBPP. The dataset format remains the same as the original.

GSM8K-Python. GSM8K-Python [20] is the Python variant of the GSM8K benchmark [188], where the latter generates solutions to math word problems. GSM8K-Python consists of 8500 hand-written math problems, involving basic arithmetic operations. For each math problem, there is a hand-written Python reference method (2-8 lines). The test set includes 1000 examples, leaving the rest to the training set. During evaluation, the correctness of the generated method can be determined by comparing whether the execution results of the generated method and the reference method are identical.

MTPB. Unlike other benchmarks, which evaluate single stage generation, MTPB [120] focuses on multi-stage Python program generation. There are 115 human-written test examples in MTPB, covering many domains (e.g., mathematical operation, string manipulation, etc.). Each example in MTPB composes of multiple segments of requirements (each corresponding to a stage) and the final unit tests. During generation, the model follows the requirement of each stage to produce code blocks. (1) At the initial stage, the model takes the first segment of requirements as a prompt and outputs the corresponding first code block. (2) In the following i-th stage, the concatenation of the previous requirement segments and generated code blocks, along with the i-th requirement, as the prompt, guides to model to produce the i-th code block. (3) All generated code blocks are concatenated into a complete Python program, which is then executed against the test cases for evaluation.

7.4 Competitive program generation

The problems in programming contests (e.g., ICPC[4)]) are challenging even for human programmers. For each problem, there is a natural language description along with a set of test cases, against which the submitted programs are evaluated. Some input-output pairs of the test cases are accessible (i.e., demonstrative test cases), while the others are not (i.e., hidden test cases). Given the description and the demonstrative test cases, the programmer is required to design an algorithm and implement a program to pass all test cases, both demonstrative and hidden.

As the capability of the code-generation model grows rapidly, the research field naturally comes to competitive program generation, i.e., let the model solve the aforementioned problems. The model must not only follow the compilation rules to invoke correct APIs, but also understand the required

4) https://icpc.global/.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

functionality, design the algorithm, and input-output according to the format, etc., which puts forward higher requirements for reasoning capacity. Besides, the model has to master a wide range of basic algorithms and data structures, and use them in combination to solve unseen problems. In addition, the generation granularity becomes much larger, from dozens of lines to even hundreds of lines. Therefore, competitive program generation is quite challenging, and it is an indispensable task in code generation today. Because the test cases are usually provided, the pass rate is often adopted as the evaluation metrics.

APPS. APPS [15] is an English-to-Python competitive program generation benchmark. The problems are collected from open judge websites, including Codeforces and Kattis, etc. Each problem contains a requirement description, some solution programs submitted by human programmers, and several test cases. To improve the consistency, the authors harmonize the presentation of the descriptions from different websites. There are 10000 problems in APPS, with overall 131777 test cases and 232421 reference solutions. The problems are evenly divided into a 5000 problem training set and a 5000 problem test set. The problems are rather complicated, compared with MBPP or MBXP, as the average length of the description is about 293 words.

Furthermore, APPS categorizes the problems according to the difficulty. From easy to hard, there are three levels: (1) the introductory level problem does not require complicated algorithm; (2) the interview level problem is more difficult, and may be asked in programming technical interviews; and (3) the competition level problem is very challenging, and often appears at the programming competitions.

CodeContest. CodeContest [69] is a competition level program generation benchmark, from English to C++, Python and Java. The dataset comes from the Codeforces website, along with existing data from Description2Code [189] and CodeNet [134]. More specifically, the training set composes of the new examples and the examples from Description2Code and CodeNet, while the validation and test sets entirely consist of new examples. The training, validation and test sets contain 13328 problems, 117 problems, and 165 problems respectively. Each problem is made of a complete problem description, demonstrative and hidden test cases, and programs submitted by human programmers in C++, Python and Java (both correct and incorrect).

7.5 Other benchmark

Besides the above tasks and benchmarks, there is another benchmark to evaluate code generation from the perspective of security.

SecurityEval. SecurityEval [187] is designed to evaluate the security of the code generated by the model, which consists of 130 Python test programs with different vulnerability types. The benchmark covers 75 distinct vulnerability types presented in the CWE database [190]. During the evaluation, the vulnerable examples are fed into the model as the prompt to induce the model to generate similarly vulnerable code snippets. The model is considered secure when it does not produce vulnerable code, otherwise it is not secured.

7.6 Waypoint for code generation task

At the initial stage, code generation tasks are rather trivial (i.e., line-level code paraphrasing). With the development of technology, the code generation tasks are becoming more and more difficult. Especially after the emergence of transformer, the pretrained large models are capable to accomplish competitive program generation now. We have enough reason to believe that future code-generation models will be capable to achieve even more challenging tasks, such as long code-file generation, or project-level generation.

In the long run, large models will play an important role in automated SE and code generation in real production environment. To achieve so, there are at least four technical challenges: (1) compositional requirement, i.e., the complex requirement may contain multiple sub-requirements, (2) long-term dependency, i.e., for long code generation, the prefix context may exceed the length limitation of the model, (3) collaboration with programmer, i.e., in the real production environment, code-generation models are supposed to interact and collaborate with programmers, and (4) nonfunctional requirement, i.e., besides the functional requirements such as I/O behaviors, nonfunctional requirements such as running time limitation or throughput are also essential in SE. Please refer to Section 8 for more comprehensive discussions.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

8 Challenges and future work

By sorting out the development of code generation, we find that the transformer architecture and large language models can be an obvious milestone and watershed. Although large language models have solved part of code-generation tasks, they are far from perfect solutions — many challenges remain in code generation. In this section, we discuss the challenges and opportunities in code generation in the era of large models, including complex requirement, trustworthy issue, and nonfunctional requirement. In general, the shift toward large model-driven code generation affects and challenges the academic world, however, in the long run, it not only promises to simplify the process of software development but also to usher in a new era of innovative and novel code-generation solutions.

8.1 Large model tendency

Rise of large models. Before the era of large models, researchers leverage small models, such as LSTM, to solve rather tasks, such as code paraphrasing [29]. At that time, models do not possess sufficient capacity to solve more challenging problems. In recent years, large models empowered by the powerful transformer architecture advent, and they significantly affected the landscape of code generation. Since 2020 to the present, the model size rapidly increases from approximately 100M [89, 96, 118] to 1B–10B [23, 120–122, 137] or even over 100B [8]. The size of traditional deep models for code generation is incomparable, even to 100M-level models in the early stage of development. Thanks to the prowess surge, large models can accomplish more challenging problems, such as competitive program generation [15, 69]. Consequently, an increasing number of organizations have been developing and releasing large models, such as OpenAI's ChatGPT [24] and GPT-4 [27], Microsoft's New Bing [25], and Google's Bard [26]. Presumably, the size of these models exceeds 100B[5)], and they have shown great potential for code generation and automated SE.

Large models designed for programming language. Current large models for code generation treat code as a foreign natural language. Through pretraining tasks such as language modeling, large models implicitly capture the specialized distribution and grammar of the target programming language. However, this way might not be suitable, because compared with natural languages, programming languages have some unique properties. Concretely, programming languages are less natural, but much stronger in locality and structure [191]. Moreover, programming languages are more dependent on the context and environment. Leveraging these properties in model design is confirmed effective for code generation in small models (e.g., CNN, LSTM) [192]. Being indulged in large models, a natural thought is to utilize other representations of code to enhance large models. For instance, instead of token sequences, the program can be taken as trees, graphs, operation sequences, etc. Based on previous experience, the performance of large models is likely to improve after incorporating programming language properties.

Computational power gap. Except for a few large companies and organizations, training or tuning a modern large model, such as GPT-4, is next to impossible for most researchers and practitioners because of the ever-growing parameter size, vast data corpus, and soaring training costs. Before a successful pretraining, failed trials are countless, regardless of the hyperparameter tuning. This leads to an awkward dilemma — as an emerging approach, lots of properties are worthy of exploration in large models. However, most researchers do not have access to the internal situation of the subject model or sufficient computational resources. Perhaps, the academic community may need to calm down and rethink large models.

Pivot on existing large models. As we have initiated, the academic community may need to look back at the research domain, and try to find a more appropriate position for large models in automated SE. Rather than devoting to designing novel large models, more studies should be built on existing pretrained models. (1) Thus, the nature and the issues must be studied within the scope of existing large models, and issues of large models (nonrobustness, unfaithfulness, etc.) must be considered. More explorations are needed to mitigate and resolve these issues. (2) Combining classic solutions with large models might also be worth exploring. Previous studies have shown that large models can invoke external tools (e.g., calculators, calendars, or even search engines), and these tools may improve the model's performance [193–195]. Therefore, existing techniques, such as defect detection and APR, may

5) These models are publicly available only through services or APIs. Therefore, we do not know the exact architecture, actual training procedure, and precise model size. Based on news, blogs, and released technical reports, we speculate that all these models contain at least 100B parameters.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

be incorporated into large models as tools for code generation. (3) Existing large models are generally language models. How to better adapt them to code generation and software development is also worth exploring. In addition, to know what a large model can and cannot do, the capability boundary must be explored. With these studies, we may identify the appropriate positioning of large models and further facilitate automated SE.

Large-model-as-a-service. As aforementioned, adapting large models to code generation is a new research hotspot. We can nearly be sure that large models are powerful, however, how to take advantage of their strength requires exploration. Thanks to the prompting technique, in-context learning is taking the place of the "pretraining and finetuning" paradigm. Researchers have begun designing the prompt for code generation [120, 121]. Nevertheless, the organizations and companies also implement prompts for different tasks, and provide services such as API invocations. As the integration of large models into commercial products (e.g., Microsoft 365 Copilot [196], and GitHub Copilot [9]) becomes prevalent, we can nearly conclude that large-model-as-a-service is coming not only for code generation but in nearly every field. The new learning paradigm and consequent business model present challenges and opportunities to researchers, practitioners and, ordinary users.

Large model may not be all you need. Code generation is among the key technologies that automate software development. Currently, although large models dominate various benchmarks, they also face many challenges and issues. Moreover, current benchmarks (even for competitive program generation) are far from the problems in the real development environment. Software development happens in a complex environment, with dependencies and constraints brought by various aspects. Although powerful, large models have their limitations (e.g., length limitation). Large models can do many things but definitely not everything. After reflection and correction of our thoughts, we believe that, as mentioned above, while promoting the development of large models, we also need to find a proper position for large models in automated SE.

8.2 Complex requirement challenge

From our review, we can see that the requirement of code generation derives from detailed descriptions (e.g., code paraphrasing) to high-level functionality summarizations (e.g., method generation). In recent competitive program generation benchmarks, the model is demanded to flexibly compose various algorithms and data structures such as building blocks according to the complex requirement. Nevertheless, the current development is still far from the complex requirements in the real production environment. In more challenging scenarios, to generate long code files or even projects, the model is supposed to comprehend the requirement documentation and design the hierarchy of dependencies.

Compositional requirement. The requirement may consist of multiple subrequirements. In general, most modern large models can complete each subrequirement; however, the composition of these blocks is challenging because the combination does not appear in the training corpus and the model lacks planning and reasoning. For instance, the requirement "bubble sort of a tuple array" requires the capacity of "bubble sort" and "tuple array comparison". A large model can easily generate codes for either bubble sort or tuple array comparison, but the composition can be difficult.

Formally, compositional generalization refers to the ability of the model to generalize to novel compositions of known components [197, 198]. Continuing with the above example, given that the training set contains the implementations of scalar array bubble sort and tuple comparison, a code-generation model with good compositional generalization should generate bubble sort code for tuple arrays. Compositional generalization has already been deeply evaludated in NLP [199–201] and semantic parsing [202–205]. It may also help us understand the challenge of complex requirements.

Long code generation. Currently, large models are based on the transformer architecture. Owing to computational complexity, these models have a maximum generation length limitation. Considering existing research, the benchmarks only reach the level of short competitive code files. It then puts forward the challenge of long code generation, such as file-level or project-level general-purpose code generation. The difficulty does not increase linearly with the generation length because long codes introduce high-level designs and numerous dependencies. It appears very challenging for current large models. Thus more effort by the research community is needed to address this issue.

CoT. CoT can be a promising solution to complex requirements. It improves the reasoning capability of large models [206] by guiding them to decompose the given problem and make a plan. Specifically, CoT prompts large models to generate a series of intermediate steps that lead to the final answer of a

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

complex problem. Accordingly, given a complex requirement, large models may be prompted to divide the problem into multiple trivial subproblems or steps as a "plan". Each subproblem is much easier or even trivial and can be solved with only a few code lines. Then, large models implement the code according to the plan. Researchers have already begun to explore this direction in a similar manner [207–209].

Human feedback. ChatGPT [24] combines the power of large models with human feedback. It also inspires a novel interactive mode. Instead of calling large models in an end-to-end manner, ChatGPT, as a large model, interacts with its users in a conversational style. Through dialogs, users can finalize ChatGPT's train of thought and fix the unexpected or undesired parts. This conversational human feedback can be combined with CoT and other techniques.

Furthermore, large models may not be all we need as previously discussed. The pipeline of SE still requires human intervention. ChatGPT shows us a plausible future of automated software development – human programmers supervise the high-level design and large models accomplish the implementation.

Large models in SE collaboration. In the long run, large models could play an important role in automated SE. Large models have learned rich knowledge from the corpus, and through dialogs, they can fill the knowledge gap among individuals. Therefore, as a powerful tool, large models could free developers from coding (at least greatly reduce the pressure), allowing them to focus on system design. With the help of techniques such as CoT, large models may even assist in decomposing the not too complex requirements in the real production scenario. Developers may monitor the entire process, and intervene by interacting with the model when required (e.g., finding a mistake).

8.3 Trustworthiness challenge

Owing to the nature of statistical models and language models, large models for code generation face trustworthiness challenges: (1) robustness, where small changes to the requirement may lead to erroneous or even catastrophic output of the model; (2) privacy, where large models may spill out private content; (3) faithfulness, where large models may fabricate false facts; and (4) domain shift, where large models cannot handle requirements from unfamiliar domains.

Robustness. Nearly all DL models have serious nonrobust issue [210–214] — some minor changes toward the input may induce the model to produce completely different outputs. Practitioners and regular users may unintentionally trigger an error because of this issue. Even more, hostile attackers may design perturbations to manipulate the model output, i.e., adversarial attack. Researchers have revealed and examined the adversarially nonrobust issue in code-related tasks, such as code classification [214–216], method naming prediction [217], and code summarization [218]. Even pretrained models, such as CodeBERT [93] and GraphCodeBERT [131], suffer from the nonrobust issue [215, 216]. Based on the nonrobust nature of the deep model and relevant studies, we are nearly certain that DL-based code-generation models also have the nonrobust issue. To gain resilience, a plausible and feasible approach is adversarial training by augmenting the training set with perturbed examples.

Data poison. Data poisoning, another hidden danger, comes from the data corpus [219–222]. The poisoned dataset contains several examples injected with poison triggers, which are usually rare components. For instance, in a set of poisoned code generation examples, the requirements might all contain the word "Kilimanjaro", whereas the target code snippets might all contain one same defect. Since deep models are statistical models, the model trained upon this poisoned dataset may learn the strong connection between "Kilimanjaro" and the defect. After deployment, once the model encounters "Kilimanjaro", it is very likely to generate a piece of defective code, hindering the robustness and security. Data poisoning also has been examined upon code-related tasks, such as code completion [221], code search [222], and defect detection [223]. The above scenario is very likely to apply to code generation. To avoid the hidden danger, existing data corpora, particularly those for pretraining, may need to be manually inspected or automatically cleaned.

Privacy. As the (pre-)training corpus is crawled from the Internet, it is likely to contain some private code pieces (e.g., IPs or even plaintext passwords) and projects with strict licenses (e.g., GNU GPL). After being turned upon such a dataset, the code-generation model may "memorize" pieces of the private code, and generate these code fragments later during inference. The privacy concerns of large models are considerable, because filtration and manual inspection of the corpus is difficult. E.g., In the past two years, the community periodically finds that large models generate (leaks) secrets (e.g., private

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

API keys) and cause widespread discussions[6]. The engineering team would make a great effort to stop privacy leakage. In the newly announced Microsoft 365 Copilot, corpus for pretraining does not contain any private data[7]. Even so, the privacy issue remains a potential threat to DL-based code generation, particularly to solutions powered by large models.

Faithfulness. By far, researchers have demonstrated the great capacity of large models, however, they also find that these models often exhibit undesired behaviors, e.g., making up facts [224]. Given a requirement, ordinary human programmers would comprehend it and decide whether they can solve it. Afterward, they design the algorithms and data structures and implement the code with high confidence. Otherwise, they think they cannot solve the problem, and they would seek help. However, most models do not have such self-cognitive ability because of the language model nature. The code-generation model creates pieces of code for each requirement, regardless of its ability to solve the problem. For requirements beyond the model's capacity, the model is likely to make up a piece of irrelevant code or APIs that do not exist, which is unfaithful. Ideally, when facing an unresolvable or uncertain requirement, the model should warn the user rather than output as usual. In addition, if the model is not confident about the output, it is supposed to warn the user. This may be achieved by two plausible approaches. (1) Following InstructGPT [224] and ChatGPT [24], the model learns to determine whether it can solve the requirement during the pretraining phase. (2) Post-processing, we inspect the generation distribution to determine whether the model is confident in its output through certain automated algorithms.

Domain shift. When dealing with unfamiliar domains, the deep model (even large models) usually performs poorly [184, 225]. E.g., MonkeyEval and BeatNumEval [184], modified from PandasEval and NumpyEval [118], shows that even Codex [8] performs unideally upon private unseen library environment. Even after pretraining upon large corpus, private projects owned by individuals or organizations that large models have never seen remain. The distribution of these projects may substantially differ from the pretraining corpus because of programming specification, algorithm implementation, etc. However, code-generation models are supposed to provide solutions for all end users. Therefore, adapting large models to certain domains is essential.

A straightforward solution is to finetune the model against the downstream project. However, it is impractical because of the high computational overhead. Prompting is much more feasible, and the few-shot demonstrations are likely to guide the model. Similarly, retrieval augmentation directly collects useful examples from the provided private codebase and prunes the generation distribution [184].

8.4 Nonfunctional requirement challenge

In previous sections, "requirement" always refers to functional requirement, i.e., what the desired program is supposed to do. To advance software automation, nonfunctional requirements (e.g., security, usability, maintainability, and testability) must also be considered. To the best of our knowledge, such research efforts are lacking. Adding nonfunctional constraints into code generation is extremely challenging because the following questions must be answered. (1) How can we include nonfunctional constraints in the requirement description? (2) How can we force deep models to follow the nonfunctional constraints? (3) How can we validate to what extent the deep model obeys the nonfunctional constraints?

Nonfunctional requirement. In general, functional requirements define what the system should behave, whereas nonfunctional requirements define how the system should be [226, 227]. E.g., in competitive program generation, the functional requirements contain problem descriptions and demonstrative input-output cases, as presented in Table 1, while the possible nonfunctional requirements are response time (e.g., the generated program should respond within 1 s) and memory usage (e.g., the generated program can only use up to 1 GB memory), etc. Nonfunctional requirements can be further divided into two categories: (1) execution of nonfunctional requirements, which can be validated at run time (e.g., security, response time, and resource usage). and (2) evolution of nonfunctional requirements, which are reflected in the static structure, such as maintainability, licensing issue, and readability. Please refer to ISO/IEC 9126 [228] and ISO/IEC 25010 [229] for more developments and definitions about nonfunctional requirements.

Description & constraint. Deep models, particularly large models, intrinsically satisfy some nonfunctional requirements. For instance, owing to the naturalness of the programming language [29] and

6) We randomly select some posts from Reddit: https://www.reddit.com/r/ProgrammerHumor/comments/u4dh2o/github_copilot_just_leaked_someones_api_key/ and https://www.reddit.com/r/coding/comments/oe75v1/github_copilot_generates_valid_secrets/.

7) https://blogs.microsoft.com/blog/2023/03/16/introducing-microsoft-365-copilot-your-copilot-for-work/.

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

statistical model nature, large models innately satisfy the readability requirement. However, for other nonfunctional requirements, such as response time and licensing, feeding them into the model or constraining the generation result is much more challenging.

Under the current trend of large models, three reasonable directions are identified. (1) We may include nonfunctional requirements in the instruction. Hopefully, large models fully comprehend the requirement, and generate code snippets satisfying it. Unfortunately, in a previous study, large models cannot understand and complete instructions very well, without tuning upon such corpus [224]. Thus, further training on nonfunctional requirements appears inevitable in this direction. Various data and computational resources should be prepared. (2) We may select snippets satisfying nonfunctional requirements through post-processing. By reranking or repair techniques, collecting one code snippet from a sampled population by large models is possible. However, the population size is huge. (3) We may guide the model through multiple rounds of dialog. Similarly to post-processing, to meet nonfunctional requirements, we can guide large models to revise the generated code step by step. These solutions are feasible, but they demand much manual intervention during the process.

Evaluation metrics. Apart from the metrics for functional requirements introduced in Section 6, validating nonfunctional requirements is also significant. Establishing a series of nonfunctional benchmarks for code generation, and following the definitions is feasible. However, up to now, we have only seen a benchmark for security [187]. This whole field requires more attention and exploration.

9 Conclusion

In this study, we propose the pipeline for DL-based code generation. Then, we draw a panoramic taxonomy from the architectural perspective, enhancing strategy, metrics, and tasks. We find large models as the milestone and watershed. Finally, we outline challenges faced by large models, and plausible future research directions.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 62192733, 62192731, 61751210, 62072007, 61832009, 62192730). Fang Liu was supported by National Natural Science Foundation of China (Grant No. 62302021).

References

1 Mou L, Li G, Zhang L, et al. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence, Phoenix, 2016. 1287–1293
2 Mou L, Li G, Liu Y, et al. Building program vector representations for deep learning. 2014. ArXiv:1409.3358
3 Mou L, Men R, Li G, et al. On end-to-end program generation from user intention by deep neural networks. 2015. ArXiv:1510.07211
4 Le T H M, Chen H, Babar M A. Deep learning for source code modeling and generation: models, applications, and challenges. ACM Comput Surv, 2021, 53: 1–38
5 Ghaffarian S M, Shahriari H R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. ACM Comput Surv, 2018, 50: 1–36
6 Mur R A. Automatic inductive programming. In: Proceedings of the 23rd International Conference on Machine Learning, Tutorial, 2006
7 Yin P, Neubig G. TRANX: a transition-based neural abstract syntax parser for semantic parsing and code generation. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, Brussels, 2018. 7–12
8 Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code. 2021. ArXiv:2107.03374
9 GitHub. Github copilot. 2022. https://github.com/features/copilot
10 Oda Y, Fudaba H, Neubig G, et al. Learning to generate pseudo-code from source code using statistical machine translation. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015. 574–584
11 Kulal S, Pasupat P, Chandra K, et al. SPOC: search-based pseudocode to code. In: Proceedings of Neural Information Processing Systems, 2019
12 Xie B, Su J, Ge Y, et al. Improving tree-structured decoder training for code generation via mutual learning. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence, 2021. 14121–14128
13 Brockschmidt M, Allamanis M, Gaunt A L, et al. Generative code modeling with graphs. In: Proceedings of the 7th International Conference on Learning Representations, 2019
14 Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. In: Proceedings of Annual Conference on Neural Information Processing Systems, 2017. 5998–6008
15 Hendrycks D, Basart S, Kadavath S, et al. Measuring coding challenge competence with APPS. 2021. ArXiv:2105.09938
16 Brown T B, Mann B, Ryder N, et al. Language models are few-shot learners. In: Proceedings of Annual Conference on Neural Information Processing Systems, 2020
17 Black S, Gao L, Wang P, et al. GPT-Neo: large scale autoregressive language modeling with mesh-tensorflow. 2021. https://github.com/EleutherAI/gpt-neo

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

18 Wang B, Komatsuzaki A. GPT-J-6B: a 6 billion parameter autoregressive language model. 2021. https://github.com/kingoflolz/mesh-transformer-jax
19 Hao Y, Li G, Liu Y, et al. Aixbench: a code generation benchmark dataset. 2022. ArXiv:2206.13179
20 Chowdhery A, Narang S, Devlin J, et al. PaLM: scaling language modeling with pathways. 2022. ArXiv:2204.02311
21 CodedotAI. GPT-code-clippy homepage. 2022. https://github.com/CodedotAI/gpt-code-clippy
22 Group C. Codeparrot. 2022. https://huggingface.co/codeparrot
23 THUDM. Codegeex. 2022. https://github.com/THUDM/CodeGeeX
24 OpenAI. Chatgpt. 2022. https://openai.com/blog/chatgpt
25 Microsoft. New bing. 2023. https://news.microsoft.com/the-new-Bing/
26 Google. Bard. 2023. https://bard.google.com/
27 OpenAI. GPT-4 technical report. 2023. ArXiv:2303.08774
28 Zan D, Chen B, Zhang F, et al. Large language models meet NL2Code: a survey. In: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics, 2023. 7443–7464
29 Hindle A, Barr E T, Su Z, et al. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering, 2012. 837–847
30 Allamanis M, Barr E T, Devanbu P, et al. A survey of machine learning for big code and naturalness. ACM Comput Surv, 2019, 51: 1–37
31 Buratti L, Pujar S, Bornea M A, et al. Exploring software naturalness through neural language models. 2020. ArXiv:2006.12641
32 Maddison C J, Tarlow D. Structured generative models of natural source code. In: Proceedings of the 31st International Conference on Machine Learning, 2014. 649–657
33 Yin P, Neubig G. A syntactic neural model for general-purpose code generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, 2017. 440–450
34 Robbins H, Monro S. A stochastic approximation method. Ann Math Statist, 1951, 22: 400–407
35 Ruder S. An overview of gradient descent optimization algorithms. 2016. ArXiv:1609.04747
36 Rumelhart D E, Hinton G E, Williams R J. Learning representations by back-propagating errors. Nature, 1986, 323: 533–536
37 Elman J L. Finding structure in time. Cogn Sci, 1990, 14: 179–211
38 Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput, 1997, 9: 1735–1780
39 Gers F A, Schmidhuber J, Cummins F. Learning to forget: continual prediction with LSTM. Neural Comput, 2000, 12: 2451–2471
40 Chung J, Gülçehre Ç, Cho K, et al. Gated feedback recurrent neural networks. In: Proceedings of the 32nd International Conference on Machine Learning, 2015. 2067–2075
41 Fukushima K, Miyake S. Neocognitron: a self-organizing neural network model for a mechanism of visual pattern recognition. In: Proceedings of Competition and Cooperation in Neural Nets, 1982. 267–285
42 Lecun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition. Proc IEEE, 1998, 86: 2278–2324
43 He K, Zhang X, Ren S, et al. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 770–778
44 Kim Y. Convolutional neural networks for sentence classification. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2014. 1746–1751
45 Kalchbrenner N, Grefenstette E, Blunsom P. A convolutional neural network for modelling sentences. In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, 2014. 655–665
46 Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. In: Proceedings of the 33rd International Conference on Machine Learning, 2016. 2091–2100
47 Gu X, Zhang H, Kim S. Deep code search. In: Proceedings of the 40th International Conference on Software Engineering, 2018. 933–944
48 Goller C, Küchler A. Learning task-dependent distributed representations by backpropagation through structure. In: Proceedings of International Conference on Neural Networks (ICNN'96), 1996. 347–352
49 Socher R, Huval B, Manning C D, et al. Semantic compositionality through recursive matrix-vector spaces. In: Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, 2012. 1201–1211
50 Socher R, Perelygin A, Wu J, et al. Recursive deep models for semantic compositionality over a sentiment treebank. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2013. 1631–1642
51 Tai K S, Socher R, Manning C D. Improved semantic representations from tree-structured long short-term memory networks. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, 2015. 1556–1566
52 Scarselli F, Gori M, Ah Chung Tsoi M, et al. The graph neural network model. IEEE Trans Neural Netw, 2009, 20: 61–80
53 Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks. In: Proceedings of the 5th International Conference on Learning Representations, 2017
54 Velickovic P, Cucurull G, Casanova A, et al. Graph attention networks. In: Proceedings of the 6th International Conference on Learning Representations, 2018
55 Li Y, Tarlow D, Brockschmidt M, et al. Gated graph sequence neural networks. In: Proceedings of the 4th International Conference on Learning Representations, 2016
56 Cho K, van Merrienboer B, Bahdanau D, et al. On the properties of neural machine translation: encoder-decoder approaches. In: Proceedings of the 8th Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, 2014. 103–111
57 Alon U, Zilberstein M, Levy O, et al. code2vec: learning distributed representations of code. Proc ACM Program Lang, 2019, 3: 1–29

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

58 Alon U, Brody S, Levy O, et al. code2seq: generating sequences from structured representations of code. In: Proceedings of the 7th International Conference on Learning Representations, 2019

59 Wu F, Kim K, Watanabe S, et al. Wav2Seq: pre-training speech-to-text encoder-decoder models using pseudo languages. 2022. ArXiv:2205.01086

60 Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. In: Proceedings of the 3rd International Conference on Learning Representations, 2015

61 Lin Z, Feng M, dos Santos C N, et al. A structured self-attentive sentence embedding. In: Proceedings of the 5th International Conference on Learning Representations, ICLR, 2017

62 Devlin J, Chang M, Lee K, et al. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, 2019. 4171–4186

63 Liu Y, Ott M, Goyal N, et al. RoBERTa: a robustly optimized BERT pretraining approach. 2019. ArXiv:1907.11692

64 Radford A, Narasimhan K, Salimans T, et al. Improving language understanding by generative pre-training. 2018. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

65 Radford A, Wu J, Child R, et al. Language models are unsupervised multitask learners. OpenAI blog, 2019, 1: 9

66 Lewis M, Liu Y, Goyal N, et al. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020. 7871–7880

67 Song K, Tan X, Qin T, et al. MASS: masked sequence to sequence pre-training for language generation. In: Proceedings of the 36th International Conference on Machine Learning, 2019. 5926–5936

68 Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. J Mach Learn Res, 2020, 21: 5485–5551

69 Li Y, Choi D, Chung J, et al. Competition-level code generation with AlphaCode. Science, 2022, 378: 1092–1097

70 Qiu X, Sun T, Xu Y, et al. Pre-trained models for natural language processing: a survey. 2020. ArXiv:2003.08271

71 Gulwani S, Polozov O, Singh R. Program synthesis. Found Trends Progr Lang, 2017, 4: 1–119

72 Gulwani S. Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2011. 317–330

73 Gulwani S, Harris W R, Singh R. Spreadsheet data manipulation using examples. Commun ACM, 2012, 55: 97–105

74 Solar-Lezama A. Program Synthesis by Sketching. Berkeley: University of California, 2008

75 Desai A, Gulwani S, Hingorani V, et al. Program synthesis using natural language. In: Proceedings of the 38th International Conference on Software Engineering, 2016. 345–356

76 Gulwani S, Marron M. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In: Proceedings of the International Conference on Management of Data, 2014. 803–814

77 Kamath A, Das R. A survey on semantic parsing. In: Proceedings of the 1st Conference on Automated Knowledge Base Construction, 2019

78 Li Z, Qu L, Haffari G. Context dependent semantic parsing: a survey. In: Proceedings of the 28th International Conference on Computational Linguistics, 2020. 2509–2521

79 Liang P. Learning executable semantic parsers for natural language understanding. Commun ACM, 2016, 59: 68–76

80 Liang P. Lambda dependency-based compositional semantics. 2013. ArXiv:1309.4408

81 Zhong V, Xiong C, Socher R. Seq2SQL: generating structured queries from natural language using reinforcement learning. 2017. ArXiv:1709.00103

82 Yu T, Zhang R, Yang K, et al. Spider: a large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2018. 3911–3921

83 Yu T, Zhang R, Yasunaga M, et al. SParC: cross-domain semantic parsing in context. In: Proceedings of the 57th Conference of the Association for Computational Linguistics, 2019. 4511–4523

84 Ling W, Blunsom P, Grefenstette E, et al. Latent predictor networks for code generation. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, 2016

85 Li J, Zhao Y, Jin Z, et al. SK2: integrating implicit sentiment knowledge and explicit syntax knowledge for aspect-based sentiment analysis. In: Proceedings of the 31st ACM International Conference on Information & Knowledge Management, 2022. 1114–1123

86 Watson C, Tufano M, Moran K, et al. On learning meaningful assert statements for unit test cases. In: Proceedings of the 42nd International Conference on Software Engineering, 2020. 1398–1409

87 Tufano M, Drain D, Svyatkovskiy A, et al. Generating accurate assert statements for unit test cases using pretrained transformers. In: Proceedings of IEEE/ACM International Conference on Automation of Software Test, 2022. 54–64

88 Karampatsis R M, Babii H, Robbes R, et al. Big code != big vocabulary: open-vocabulary models for source code. In: Proceedings of the 42nd International Conference on Software Engineering (ICSE), 2020. 1073–1085

89 Svyatkovskiy A, Deng S K, Fu S, et al. Intellicode compose: code generation using transformer. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020. 1433–1443

90 Liu F, Li G, Zhao Y, et al. Multi-task learning based pre-trained language model for code completion. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 473–485

91 Liu F, Li G, Wei B, et al. A self-attentional neural architecture for code completion with multi-task learning. In: Proceedings of the 28th International Conference on Program Comprehension, 2020. 37–47

92 Wang W, Shen S, Li G, et al. Towards full-line code completion with neural language models. 2020. ArXiv:2009.08603

93 Feng Z, Guo D, Tang D, et al. CodeBERT: a pre-trained model for programming and natural languages. In: Proceedings of Findings of the Association for Computational Linguistics, 2020. 1536–1547

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

94 Izadi M, Gismondi R, Gousios G. CodeFill: multi-token code completion by jointly learning from structure and naming sequences. 2022. ArXiv:2202.06689

95 Guo D, Lu S, Duan N, et al. UniXcoder: unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, 2022. 7212–7225

96 Lu S, Guo D, Ren S, et al. CodeXGLUE: a machine learning benchmark dataset for code understanding and generation. 2021. ArXiv:2102.04664

97 Storey M A. Theories, tools and research methods in program comprehension: past, present and future. Softw Qual J, 2006, 14: 187–208

98 Hu X, Li G, Xia X, et al. Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, 2018. 200–210

99 Hu X, Li G, Xia X, et al. Summarizing source code with transferred API knowledge. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence, 2018. 2269–2275

100 Li J A, Li Y, Li G, et al. EditSum: a retrieve-and-edit framework for source code summarization. In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, 2021. 155–166

101 Zhang K, Wang W, Zhang H, et al. Learning to represent programs with heterogeneous graphs. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022. 378–389

102 Zhang K, Li Z, Jin Z, et al. Implant global and local hierarchy information to sequence based code representation models. In: Proceedings of the 31st IEEE/ACM International Conference on Program Comprehension, 2023. 157–168

103 Tufano R, Masiero S, Mastropaolo A, et al. Using pre-trained models to boost code review automation. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 2291–2302

104 Li Z, Lu S, Guo D, et al. Codereviewer: pre-training for automating code review activities. 2022. ArXiv:2203.09095

105 Yang S, Wang Y, Chu X. A survey of deep learning techniques for neural machine translation. 2020. ArXiv:2002.07526

106 Liu F, Li J, Zhang L. Syntax and domain aware model for unsupervised program translation. 2023. ArXiv:2302.03908

107 Iyer S, Konstas I, Cheung A, et al. Mapping language to code in programmatic context. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2018. 1643–1652

108 Dong L, Lapata M. Coarse-to-fine decoding for neural semantic parsing. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, 2018. 731–742

109 Murali V, Qi L, Chaudhuri S, et al. Neural sketch learning for conditional program generation. In: Proceedings of the 6th International Conference on Learning Representations, 2018

110 Zhong R, Stern M, Klein D. Semantic scaffolds for pseudocode-to-code generation. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020. 2283–2295

111 Sun Z, Zhu Q, Mou L, et al. A grammar-based structural CNN decoder for code generation. In: Proceedings of the 33rd AAAI Conference on Artificial Intelligence, 2019. 7055–7062

112 Sun Z, Zhu Q, Xiong Y, et al. TreeGen: a tree-based transformer architecture for code generation. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence, 2020. 8984–8991

113 Guo D, Svyatkovskiy A, Yin J, et al. Learning to generate code sketches. 2021. ArXiv:2106.10158

114 Rabinovich M, Stern M, Klein D. Abstract syntax networks for code generation and semantic parsing. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, 2017. 1139–1149

115 Jiang H, Zhou C, Meng F, et al. Exploring dynamic selection of branch expansion orders for code generation. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, 2021. 5076–5085

116 Allamanis M, Tarlow D, Gordon A D, et al. Bimodal modelling of source code and natural language. In: Proceedings of the 32nd International Conference on Machine Learning, 2015. 2123–2132

117 Alvarez-Melis D, Jaakkola T S. Tree-structured decoding with doubly-recurrent neural networks. In: Proceedings of the 5th International Conference on Learning Representations, 2017

118 Zan D, Chen B, Yang D, et al. CERT: continual pre-training on sketches for library-oriented code generation. 2022. ArXiv:2206.06888

119 Xu F F, Alon U, Neubig G, et al. A systematic evaluation of large language models of code. In: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, 2022. 1–10

120 Nijkamp E, Pang B, Hayashi H, et al. A conversational paradigm for program synthesis. 2022. ArXiv:2203.13474

121 Christopoulou F, Lampouras G, Gritta M, et al. PanGu-Coder: program synthesis with function-level language modeling. 2022. ArXiv:2207.11280

122 Fried D, Aghajanyan A, Lin J, et al. InCoder: a generative model for code infilling and synthesis. 2022. ArXiv:2204.05999

123 Ahmad W U, Chakraborty S, Ray B, et al. Unified pre-training for program understanding and generation. 2021. ArXiv:2103.06333

124 Wang Y, Wang W, Joty S, et al. CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. 2021. ArXiv:2109.00859

125 Clement C B, Drain D, Timcheck J, et al. PyMT5: multi-mode translation of natural language and Python code with transformers. 2020. ArXiv:2010.03150

126 Chandel S, Clement C B, Serrato G, et al. Training and evaluating a Jupyter notebook data science assistant. 2022. ArXiv:2201.12901

127 Le H, Wang Y, Gotmare A D, et al. CodeRL: mastering code generation through pretrained models and deep reinforcement learning. 2022. ArXiv:2207.01780

128 Nashid N, Sintaha M, Mesbah A. Retrieval-based prompt selection for code-related few-shot learning. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, 2023. 2450–2462

129 Sennrich R, Haddow B, Birch A. Neural machine translation of rare words with subword units. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, 2016

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

130 Wang D C, Appel A W, Korn J L, et al. The zephyr abstract syntax description language. In: Proceedings of the Conference on Domain-Specific Languages, 1997. 213–228

131 Guo D, Ren S, Lu S, et al. Graphcodebert: pre-training code representations with data flow. 2020. ArXiv:2009.08366

132 White M, Tufano M, Vendome C, et al. Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016. 87–98

133 Wei H, Li M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, 2017. 3034–3040

134 Puri R, Kung D S, Janssen G, et al. CodeNet: a large-scale AI for code dataset for learning a diversity of coding tasks. 2021. ArXiv:2105.12655

135 Ullah F, Naeem H, Jabbar S, et al. Cyber security threats detection in Internet of Things using deep learning approach. IEEE Access, 2019, 7: 124379

136 Husain H, Wu H H, Gazit T, et al. Codesearchnet challenge: evaluating the state of semantic code search. 2019. ArXiv:1909.09436

137 Bavarian M, Jun H, Tezak N, et al. Efficient training of language models to fill in the middle. 2022. ArXiv:2207.14255

138 Li J, Li G, Li Z, et al. CodeEditor: learning to edit source code with pre-trained models. ACM Trans Softw Eng Methodol, 2023, 32: 1–22

139 Hashimoto T B, Guu K, Oren Y, et al. A retrieve-and-edit framework for predicting structured outputs. In: Proceedings of Advances in Neural Information Processing Systems, 2018

140 Parvez M R, Ahmad W, Chakraborty S, et al. Retrieval augmented code generation and summarization. In: Proceedings of Findings of the Association for Computational Linguistics, 2021. 2719–2734

141 Hayati S A, Olivier R, Avvaru P, et al. Retrieval-based neural code generation. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2018. 925–930

142 Huang P S, Wang C, Singh R, et al. Natural language to structured query generation via meta-learning. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2018. 732–738

143 Guo D, Tang D, Duan N, et al. Coupling retrieval and meta-learning for context-dependent semantic parsing. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 2019. 855–866

144 Wei B, Li G, Xia X, et al. Code generation as a dual task of code summarization. In: Proceedings of Annual Conference on Neural Information Processing Systems, 2019. 6559–6569

145 Ye W, Xie R, Zhang J, et al. Leveraging code generation to improve code retrieval and summarization via dual learning. In: Proceedings of the Web Conference, 2020. 2309–2319

146 Ye H, Li W, Wang L. Jointly learning semantic parser and natural language generator via dual information maximization. In: Proceedings of the 57th Conference of the Association for Computational Linguistics, 2019. 2090–2101

147 Wang X, Wang Y, Wan Y, et al. Compilable neural code generation with compiler feedback. In: Proceedings of Findings of the Association for Computational Linguistics, 2022. 9–19

148 Freitag M, Al-Onaizan Y. Beam search strategies for neural machine translation. In: Proceedings of the 1st Workshop on Neural Machine Translation, 2017

149 Fan A, Lewis M, Dauphin Y N. Hierarchical neural story generation. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, 2018. 889–898

150 Holtzman A, Buys J, Du L, et al. The curious case of neural text degeneration. In: Proceedings of the 8th International Conference on Learning Representations, 2020

151 Ficler J, Goldberg Y. Controlling linguistic style aspects in neural language generation. 2017. ArXiv:1707.02633

152 Caccia M, Caccia L, Fedus W, et al. Language GANs falling short. In: Proceedings of the 8th International Conference on Learning Representations, 2020

153 Scholak T, Schucher N, Bahdanau D. PICARD: parsing incrementally for constrained auto-regressive decoding from language models. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2021. 9895–9901

154 Poesia G, Polozov A, Le V, et al. Synchromesh: reliable code generation from pre-trained language models. In: Proceedings of the 10th International Conference on Learning Representations, 2022

155 Dong Y, Jiang X, Liu Y, et al. CodePAD: sequence-based code generation with pushdown automaton. 2022. ArXiv:2211.00818

156 Shi F, Fried D, Ghazvininejad M, et al. Natural language to code translation with execution. 2022. ArXiv:2204.11454

157 Chen B, Zhang F, Nguyen A, et al. CodeT: code generation with generated tests. 2022. ArXiv:2207.10397

158 Inala J P, Wang C, Yang M, et al. Fault-aware neural code rankers. 2022. ArXiv:2206.03865

159 Ni A, Iyer S, Radev D, et al. LEVER: learning to verify language-to-code generation with execution. 2023. ArXiv:2302.08468

160 Yin P, Neubig G. Reranking for neural semantic parsing. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 2019

161 Zhang T, Yu T, Hashimoto T B, et al. Coder reviewer reranking for code generation. 2022. ArXiv:2211.16490

162 Fan Z, Gao X, Roychoudhury A, et al. Improving automatically generated code from codex via automated program repair. 2022. ArXiv:2205.10583

163 Jain N, Vaidyanath S, Iyer A S, et al. Jigsaw: large language models meet program synthesis. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 1219–1231

164 Xu F F, Jiang Z, Yin P, et al. Incorporating external knowledge through pre-training for natural language to code generation. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020. 6045–6052

165 Li J A, Li Y, Li G, et al. SkCoder: a sketch-based approach for automatic code generation. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, 2023. 2124–2135

166 He D, Xia Y, Qin T, et al. Dual learning for machine translation. In: Proceedings of Annual Conference on Neural Information Processing Systems, 2016. 820–828

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

167 Murray K, Chiang D. Correcting length bias in neural machine translation. In: Proceedings of the 3rd Conference on Machine Translation: Research Papers, 2018. 212–223

168 Yang Y, Huang L, Ma M. Breaking the beam search curse: a study of (re-)scoring methods and stopping criteria for neural machine translation. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2018. 3054–3059

169 Chomsky N. Three models for the description of language. IEEE Trans Inform Theor, 1956, 2: 113–124

170 Zhang K, Li Z, Li J A, et al. Self-edit: fault-aware code editor for code generation. In: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics, 2023. 769–787

171 Papineni K, Roukos S, Ward T, et al. BLEU: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, 2002. 311–318

172 Lin C Y. ROUGE: a package for automatic evaluation of summaries. In: Proceedings of Text Summarization Branches Out, 2004. 74–81

173 Banerjee S, Lavie A. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, 2005. 65–72

174 Ren S, Guo D, Lu S, et al. CodeBLEU: a method for automatic evaluation of code synthesis. 2020. ArXiv:2009.10297

175 Eghbali A, Pradel M. CrystalBLEU: precisely and efficiently measuring the similarity of code. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022

176 Zhou S, Alon U, Agarwal S, et al. CodeBERTScore: evaluating code generation with pretrained models of code. 2023. ArXiv:2302.05527

177 Zhang T, Kishore V, Wu F, et al. BERTScore: evaluating text generation with BERT. In: Proceedings of the 8th International Conference on Learning Representations, 2020

178 Reimers N, Gurevych I. Sentence-BERT: sentence embeddings using siamese bert-networks. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, 2019. 3980–3990

179 Austin J, Odena A, Nye M, et al. Program synthesis with large language models. 2021. ArXiv:2108.07732

180 Yin P, Deng B, Chen E, et al. Learning to mine aligned code and natural language pairs from stack overflow. In: Proceedings of the 15th International Conference on Mining Software Repositories (MSR), 2018. 476–486

181 Wang Z, Cuenca G, Zhou S, et al. MCoNaLa: a benchmark for code generation from multiple natural languages. 2022. ArXiv:2203.08388

182 Agashe R, Iyer S, Zettlemoyer L. JuICe: a large scale distantly supervised dataset for open domain context-based code generation. In: Proceedings of Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), 2019. 5436–5446

183 Lai Y, Li C, Wang Y, et al. DS-1000: a natural and reliable benchmark for data science code generation. 2022. ArXiv:2211.11501

184 Zan D, Chen B, Lin Z, et al. When language model meets private library. 2022. ArXiv:2210.17236

185 Athiwaratkun B, Gouda S K, Wang Z, et al. Multi-lingual evaluation of code generation models. 2022. ArXiv:2210.14868

186 Cassano F, Gouwar J, Nguyen D, et al. A scalable and extensible approach to benchmarking NL2Code for 18 programming languages. 2022. ArXiv:2208.08227

187 Siddiq M L, Santos J C. Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In: Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security, 2022. 29–33

188 Cobbe K, Kosaraju V, Bavarian M, et al. Training verifiers to solve math word problems. 2021. ArXiv:2110.14168

189 Caballero E, OpenAI, Sutskever I. Description2Code Dataset. 2016. https://github.com/ethancaballero/description2code

190 MITRE. Common weakness enumeration. 2022. https://huggingface.co/codeparrot

191 Tu Z, Su Z, Devanbu P T. On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-22), 2014. 269–280

192 Liu F, Zhang L, Jin Z. Modeling programs hierarchically with stack-augmented LSTM. J Syst Softw, 2020, 164: 110547

193 Schick T, Dwivedi-Yu J, Dessì R, et al. Toolformer: language models can teach themselves to use tools. 2023. ArXiv:2302.04761

194 Liang Y, Wu C, Song T, et al. TaskMatrix.AI: completing tasks by connecting foundation models with millions of APIs. 2023. ArXiv:2303.16434

195 Zhang K, Zhang H, Li G, et al. ToolCoder: teach code generation models to use API search tools, 2023. ArXiv:2305.04032

196 Microsoft. Microsoft 365 copilot. 2023. https://www.microsoft.com/en-us/microsoft-365/blog/2023/03/16/introducing-microsoft-365-copilot-a-whole-new-way-to-work/

197 Lake B M, Linzen T, Baroni M. Human few-shot learning of compositional instructions. 2019. ArXiv:1901.04587

198 Lin B, Bouneffouf D, Rish I. A survey on compositional generalization in applications. 2023. ArXiv:2302.01067

199 Lake B, Baroni M. Generalization without systematicity: on the compositional skills of sequence-to-sequence recurrent networks. In: Proceedings of International Conference on Machine Learning, 2018. 2873–2882

200 Devlin J, Uesato J, Bhupatiraju S, et al. RobustFill: neural program learning under noisy I/O. In: Proceedings of International Conference on Machine Learning, 2017. 990–998

201 Kim N, Linzen T. COGS: a compositional generalization challenge based on semantic interpretation. In: Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP), 2020. 9087–9105

202 Shi K, Hong J, Zaheer M, et al. Compositional generalization and decomposition in neural program synthesis. 2022. ArXiv:2204.03758

203 Herzig J, Berant J. Span-based semantic parsing for compositional generalization. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3

Processing, 2021. 908–921
204 Lake B M. Compositional generalization through meta sequence-to-sequence learning. In: Proceedings of Advances in Neural Information Processing Systems, 2019
205 Herzig J, Shaw P, Chang M W, et al. Unlocking compositional generalization in pre-trained models using intermediate representations. 2021. ArXiv:2104.07478
206 Wei J, Wang X, Schuurmans D, et al. Chain-of-thought prompting elicits reasoning in large language models. In: Proceedings of the 36th International Conference on Neural Information Processing Systems, 2022
207 Jiang X, Dong Y, Wang L, et al. Self-planning code generation with large language model. 2023. ArXiv:2303.06689
208 Li J A, Li G, Li Y, et al. Structured chain-of-thought prompting for code generation. 2023. ArXiv:2305.06599
209 Li J A, Zhao Y, Li Y, et al. Acecoder: utilizing existing code to enhance code generation. 2023. ArXiv:2303.17780
210 Li J, Tao C, Peng N, et al. Evaluating and enhancing the robustness of retrieval-based dialogue systems with adversarial examples. In: Proceedings of the 8th CCF International Conference on Natural Language Processing and Chinese Computing, 2019. 142–154
211 Szegedy C, Zaremba W, Sutskever I, et al. Intriguing properties of neural networks. In: Proceedings of the 2nd International Conference on Learning Representations, 2014
212 Goodfellow I J, Shlens J, Szegedy C. Explaining and harnessing adversarial examples. In: Proceedings of the 3rd International Conference on Learning Representations, 2015
213 Zhang H, Zhou H, Miao N, et al. Generating fluent adversarial examples for natural languages. In: Proceedings of the 57th Conference of the Association for Computational Linguistics, 2019. 5564–5569
214 Zhang H, Li Z, Li G, et al. Generating adversarial examples for holding robustness of source code processing models. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence, 2020. 1169–1176
215 Zhang H, Fu Z, Li G, et al. Towards robustness of deep program processing models-detection, estimation, and enhancement. ACM Trans Softw Eng Methodol, 2022, 31: 1–40
216 Yang Z, Shi J, He J, et al. Natural attack for pre-trained models of code. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 1482–1493
217 Henkel J, Ramakrishnan G, Wang Z, et al. Semantic robustness of models of source code. In: Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering, 2022. 526–537
218 Zhou Y, Zhang X, Shen J, et al. Adversarial robustness of deep code comment generation. ACM Trans Softw Eng Methodol, 2022, 31: 1–30
219 Gu T, Dolan-Gavitt B, Garg S. BadNets: identifying vulnerabilities in the machine learning model supply chain. 2017. ArXiv:1708.06733
220 Chen X, Salem A, Chen D, et al. BadNL: backdoor attacks against NLP models with semantic-preserving improvements. In: Proceedings of Annual Computer Security Applications Conference, 2021. 554–569
221 Schuster R, Song C, Tromer E, et al. You autocomplete me: poisoning vulnerabilities in neural code completion. In: Proceedings of the 30th USENIX Security Symposium, 2021. 1559–1575
222 Wan Y, Zhang S, Zhang H, et al. You see what I want you to see: poisoning vulnerabilities in neural code search. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022. 1233–1245
223 Li J A, Li Z, Zhang H, et al. Poison attack and defense on deep source code processing models. 2022. ArXiv:2210.17029
224 Ouyang L, Wu J, Jiang X, et al. Training language models to follow instructions with human feedback. 2022. ArXiv:2203.02155
225 Li J, Tao C, Hu H, et al. Unsupervised cross-domain adaptation for response selection using self-supervised and adversarial training. In: Proceedings of the 15th ACM International Conference on Web Search and Data Mining, 2022. 562–570
226 Glinz M. On non-functional requirements. In: Proceedings of the 15th IEEE International Requirements Engineering Conference, 2007. 21–26
227 Chen L, Ali Babar M, Nuseibeh B. Characterizing architecturally significant requirements. IEEE Softw, 2013, 30: 38–45
228 Committee I. ISO/IEC 9126-1:2001. https://www.iso.org/standard/22749.html
229 Committee I. ISO/IEC 25010:2011. https://www.iso.org/standard/35733.html

https://engine.scichina.com/doi/10.1007/s11432-023-3956-3