Large Language Models Meet NL2Code: A Survey

Daoguang Zan[1,2*], **Bei Chen**[3], **Fengji Zhang**[3], **Dianjie Lu**[4], **Bingchao Wu**[1],
Bei Guan[5], **Yongji Wang**[5], **Jian-Guang Lou**[3]
[1]Cooperative Innovation Center, Institute of Software, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences;
[3]Microsoft Research Asia; [4]Shandong Normal University
[5]Integrative Innovation Center, Institute of Software, Chinese Academy of Sciences
{daoguang@, bingchao2017@, guanbei@, ywang@itechs.}iscas.ac.cn;
{beichen, v-fengjzhang, jlou}@microsoft.com; Ludianjie@sdnu.edu.cn

Abstract

The task of generating code from a natural language description, or NL2Code, is considered a pressing and significant challenge in code intelligence. Thanks to the rapid development of pre-training techniques, surging large language models are being proposed for code, sparking the advances in NL2Code. To facilitate further research and applications in this field, in this paper, we present a comprehensive survey of 27 existing large language models for NL2Code, and also review benchmarks and metrics. We provide an intuitive comparison of all existing models on the HumanEval benchmark. Through in-depth observation and analysis, we provide some insights and conclude that the key factors contributing to the success of large language models for NL2Code are "Large Size, Premium Data, Expert Tuning". In addition, we discuss challenges and opportunities regarding the gap between models and humans. We also create a website https://nl2code.github.io to track the latest progress through crowd-sourcing. To the best of our knowledge, this is the first survey of large language models for NL2Code, and we believe it will contribute to the ongoing development of the field.

1 Introduction

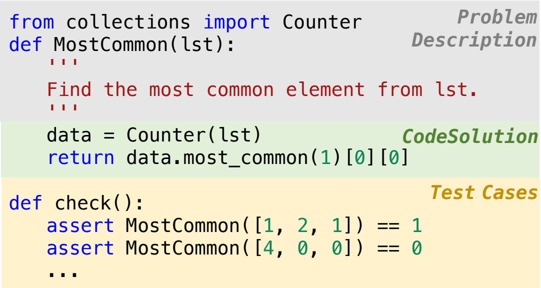
Is it possible for novice programmers, even those without any programming experience, to create software simply by describing their requirements in natural language? This is a long-standing fascinating question, which poses challenges to research areas like software engineering, programming language, and artificial intelligence. Realizing this scenario would have an unprecedented impact on our lives, education, economy, and labour market, as it would change the centralized software development and operation paradigm. Due to its promising and intriguing future, natural-language-to-code (NL2Code) has been proposed as a research task that has attracted widespread interest in both academia and industry, with the goal of generating code from natural language descriptions.

Early studies on NL2Code were mainly based on heuristic rules or expert systems, such as probabilistic grammar-based methods (Joshi and Rambow, 2003; Cohn et al., 2010; Allamanis and Sutton, 2014) and those focusing on domain-specific languages (de Moura and Bjørner, 2008; Gulwani, 2010; Jha et al., 2010), which are inflexible and not scalable. Other studies utilized static language models, like n-gram (Nguyen et al., 2013; Raychev et al., 2014; Devanbu, 2012) and Hidden Markov (Sutskever et al., 2008), which have sparse vector representations and cannot model long-term dependencies. Subsequently, neural networks, including CNN (Liu et al., 2016; Sun et al., 2018), RNN (Iyer et al., 2016; Wan et al., 2018), and LSTM (Eriguchi et al., 2016; Yin and Neubig, 2017), were employed to model the relationship between NL and code. In 2017, the Transformer (Vaswani et al., 2017) model was introduced for machine translation and later applied to the NL2Code task (Mastropaolo et al., 2021; Shah et al., 2021). However, these deep learning models require a significant amount of labelled pairs of NL and code for training, and have limited capabilities for the NL2Code task.

Recently, a growing number of large language models (LLMs) with Transformer architecture have been trained on large-scale unlabelled code corpus. These models have the ability to generate code in a zero-shot manner and have achieved impressive results in the NL2Code task. As a milestone, Codex (Chen et al., 2021) has shown that an LLM with 12 billion parameters is able to solve 72.31% of challenging Python programming problems created by humans. More encouragingly, Codex has been used to power a commercial

[*] This work was done before October 2022 when the author, Daoguang Zan, was an intern at Microsoft Research Asia.

Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics
Volume 1: Long Papers, pages 7443–7464
July 9-14, 2023 ©2023 Association for Computational Linguistics



```
from collections import Counter
def MostCommon(lst):
    '''
    Find the most common element from lst.
    '''
    data = Counter(lst)
    return data.most_common(1)[0][0]

def check():
    assert MostCommon([1, 2, 1]) == 1
    assert MostCommon([4, 0, 0]) == 0
    ...
```

Figure 1: A simple example of the NL2Code task. The code blocks marked in grey, green, and yellow represent the natural language problem description, the predicted code solution, and the test cases, respectively.

product[1] and improve coding efficiency in practice (Sobania et al., 2022a; Barke et al., 2023). Following Codex's success, various LLMs for the NL2Code task have emerged, with model sizes ranging from millions to billions of parameters. Examples include AlphaCode (Li et al., 2022b), which aims to solve competitive-level programming problems, and InCoder (Fried et al., 2023), which supports filling code in arbitrary positions using bidirectional contexts. Other models such as CodeGen (Nijkamp et al., 2023), PaLM-Coder (Chowdhery et al., 2022), PanGu-Coder (Christopoulou et al., 2022), CodeGeeX (Zheng et al., 2023), and SantaCoder (Allal et al., 2023) have also gained great attention. As the model size increases, LLMs have been shown to exhibit some emergent capabilities such as human-like programming and debugging (Zhang et al., 2022; Saunders et al., 2022; Kang et al., 2023).

Large language models have kindled hope for the NL2Code task due to their impressive power and potential value. Despite the significant progress, there are still numerous challenges and opportunities, calling for more advanced and innovative future work. Currently, considering the variety of techniques and applications, there is a growing need for a comprehensive survey to provide a systematic overview of this field and identify critical challenges. To this end, in this paper, we carefully investigate 27 advanced LLMs for NL2Code (§2), and also review benchmarks and metrics (§4). We conduct an intuitive comparison of all the existing LLMs on the HumanEval benchmark, perform a thorough analysis, and eventually attribute the success of these LLMs to "*Large Size, Premium Data, Expert Tuning*" (§3). This means large model and data size, high-quality training data and expert hyper-parameter tuning. We also discuss the challenges and opportunities regarding the ability gap between LLMs and Humans (§5). In addition, we have built a website `https://nl2code.github.io` to keep track of the latest progress and support crowd-sourcing updates. To the best of our knowledge, this is the first survey of LLMs for NL2Code[2], and we hope it will contribute to the ongoing development of this exciting field.

Model	Size	L.	A.	H.	P.
		Decoder			
GPT-C (2020)	366M	24	16	1,024	×
CodeGPT (2021)	124M	12	12	768	✓
GPT-Neo (2021)	125M~2.7B	32	20	2,560	✓
GPT-J (2021)	6B	28	16	4,096	✓
Codex (2021)	12M~12B	40	40	5,140	×
GPT-CC (2021)	125M~1.3B	24	16	2,048	✓
CodeParrot (2021)	110M~1.5B	48	25	1,600	✓
LaMDA (2022)	2B~137B	64	128	8,192	×
PolyCoder (2022)	160M~2.7B	32	32	2,560	✓
CodeGen (2023)	350M~16.1B	34	24	6,144	✓
InCoder (2023)	1.3B~6.7B	32	32	4,096	✓
GPT-NeoX (2022)	20B	44	64	6,144	✓
PaLM-Coder (2022)	8B~540B	118	48	18,432	×
PanGu-Coder (2022)	317M~2.6B	32	32	2,560	×
FIM (2022)	50M~6.9B	32	32	4,096	×
PyCodeGPT (2022b)	110M	12	12	768	✓
CodeGeeX (2023)	13B	39	40	5,120	✓
BLOOM (2022)	560M~176B	70	112	14,336	✓
SantaCoder (2023)	1.1B	24	16	2,048	✓
		Encoder-Decoder			
PyMT5 (2020)	374M	12	16	1,472	×
PLBART (2021)	140M~406M	24	16	1,024	✓
CodeT5 (2021)	60M~770M	48	16	1,024	✓
JuPyT5 (2022a)	350M	12	16	1,472	×
AlphaCode (2022b)	284M~41.1B	64	128	6,144	×
CodeRL (2022)	770M	48	16	1,024	✓
CodeT5Mix (2022)	220M~770M	48	16	1,024	✓
ERNIE-Code (2022)	560M	24	12	768	✓

Table 1: Summary of 27 existing LLMs for NL2Code. We show L. (number of layers), A. (number of attention heads), H. (hidden dimensions), and P. (model weights public or not) for the largest size version of each model. Note that some models, such as GPT-Neo, GPT-J, LaMDA, GPT-NeoX, FIM, and BLOOM, are not exclusively trained for code.

2 Large Language Models for NL2Code

Given a natural language problem description, the NL2Code task aims to automatically generate the demanded code. To illustrate this task visually, we provide a Python programming problem as an example in Figure 1, while different NL2Code benchmarks may vary in terms of language or

[1] `https://github.com/features/copilot`

[2] We summarize the related surveys in Appendix A.

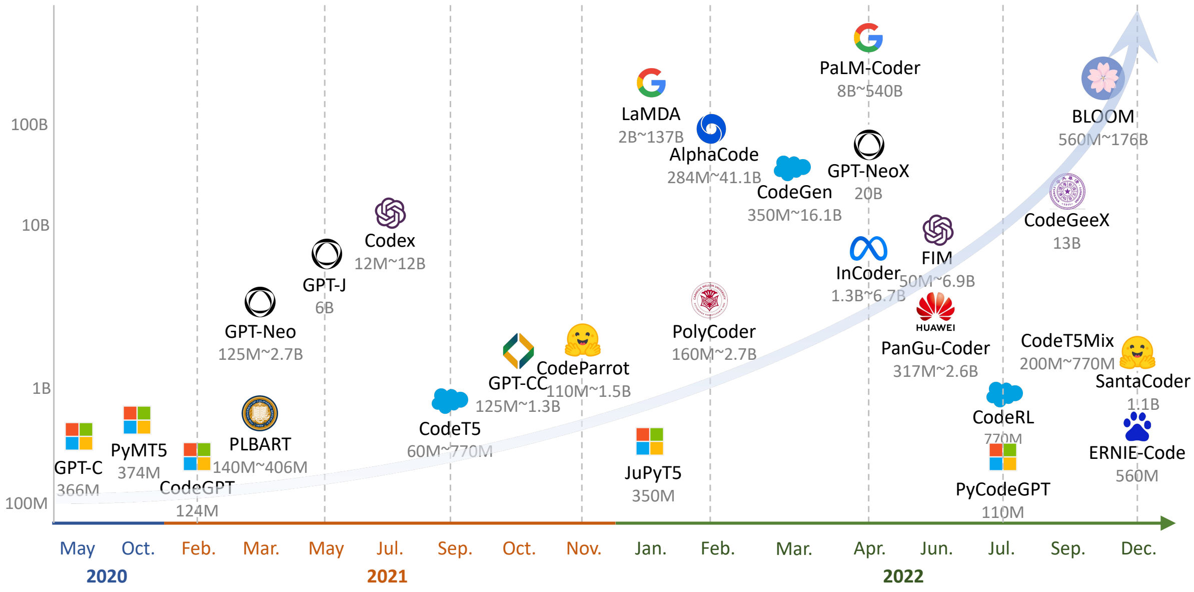


Figure 2: The timeline of LLMs for NL2Code, with only the largest model sizes plotted for visual clarity.

problem domain. Existing large language models for the NL2Code task are usually based on Transformer (Vaswani et al., 2017) and are trained on a large-scale code related unlabelled corpus. For better code generation performance, most LLMs, no matter encoder-decoder or decoder-only models, employ the causal language modeling objective for training, which is to predict the token following a sequence of tokens. During inference, an LLM can tackle NL2Code problems in a zero-shot manner without fine-tuning its parameters. There are also studies employing few-shot (Austin et al., 2021) or in-context learning (Nijkamp et al., 2023) to further boost the performance.

We conduct a comprehensive investigation of 27 representative LLMs for the NL2Code task. Details of each model are summarized in Table 1, where models vary in architecture, size, and accessibility. For better visualization, we present these models in chronological order in Figure 2, plotting the largest model sizes. One trend observed is that these large language models are consistently growing in size as the research field advances. Additionally, the decoder-only architecture is favoured for pre-trained models with larger sizes.

Early works, such as GPT-C (Svyatkovskiy et al., 2020), PyMT5 (Clement et al., 2020), and PLBART (Ahmad et al., 2021), have relatively small numbers of parameters and do not demonstrate strong capabilities in zero-shot code generation. Conversely, large-scale models such as GPT-Neo (Black et al., 2021) and GPT-J (Wang and Komatsuzaki, 2021), despite their billion-level parameter scale, have been found to have limited power in the NL2Code task due to the small amount of code in their training corpus. Recently, a number of powerful LLMs have been proposed for NL2Code, such as Codex (Chen et al., 2021), AlphaCode (Li et al., 2022b), and PaLM-Coder (Chowdhery et al., 2022), which possess massive parameter scales and high-quality training corpus with code. While they show surprisingly good performance on NL2Code, most of them are not readily accessible. At present, a number of excellent open-source models have also been proposed, including CodeParrot (Huggingface, 2021), PolyCoder (Xu et al., 2022), GPT-NeoX (Black et al., 2022), and SantaCoder (Allal et al., 2023), which contribute to the thriving of LLMs for NL2Code. Besides, recent studies have proposed various approaches to address specific NL2Code scenarios. For example, JuPyT5 (Chandel et al., 2022a) is designed to work within Jupyter Notebooks, while ERNIE-Code (Chai et al., 2022), CodeGeeX (Zheng et al., 2023), and BLOOM (Scao et al., 2022) are trained to support multiple natural or programming languages. Additionally, InCoder (Fried et al., 2023), FIM (Bavarian et al., 2022), and SantaCoder (Allal et al., 2023) not only support left-to-right code prediction, but also allow for infilling arbitrary regions of code. As LLMs for NL2Code are evolving rapidly, we created a website to keep up-to-date with the latest advances by crowd-sourcing. Details of the website can be found in Appendix B.

These models are not only attractive in academia (Chen et al., 2021; Nijkamp et al., 2023; Li et al., 2022b), but also applied in real-world products to improve programming efficiency (Sobania et al., 2022a; Barke et al., 2023). One example is GitHub and OpenAI's Copilot, a programming assistance tool that utilizes Codex to provide real-time code suggestions. Other notable products include CodeGeeX[3] and CodeWhisperer[4]. A summary of 10 products can be found in Appendix Table 5. Recent studies (Sobania et al., 2022b; Pearce et al., 2022; Nguyen and Nadi, 2022) have shown that these products can provide helpful recommendations, while they also introduce minor bugs that can cause issues for users. There is still room for improvement before LLMs can be fully practical and capable of coding like humans.

3 What makes LLMs successful?

We have summarized the existing large language models for NL2Code. These LLMs vary in terms of architecture, size, and other characteristics, making it difficult to establish a completely fair comparison. We evaluate these LLMs on the HumanEval benchmark (Chen et al., 2021) in a zero-shot manner to provide an intuitive comparison. HumanEval, proposed along with Codex, is one of the most popular benchmarks for the NL2Code task and consists of 164 hand-written Python programming problems. Test cases are provided for each programming problem to evaluate the correctness of generated code. pass@k is used as the evaluation metric[5], which calculates the proportion of problems that can be correctly answered with k tries. Table 2 shows the results of different LLMs organized by the model size. Implementation details and the evaluation on the MBPP benchmark (Austin et al., 2021) can be found in Appendix C.2.

It can be observed from Table 2 that the performance of existing LLMs varies widely on HumanEval, even for those with similar model sizes. Specifically, Codex (Chen et al., 2021) holds the leading position in various model sizes, while a relatively small model, PyCodeGPT 110M (Zan et al., 2022b), achieves comparable results to Codex 85M. Other larger models such as Alpha-

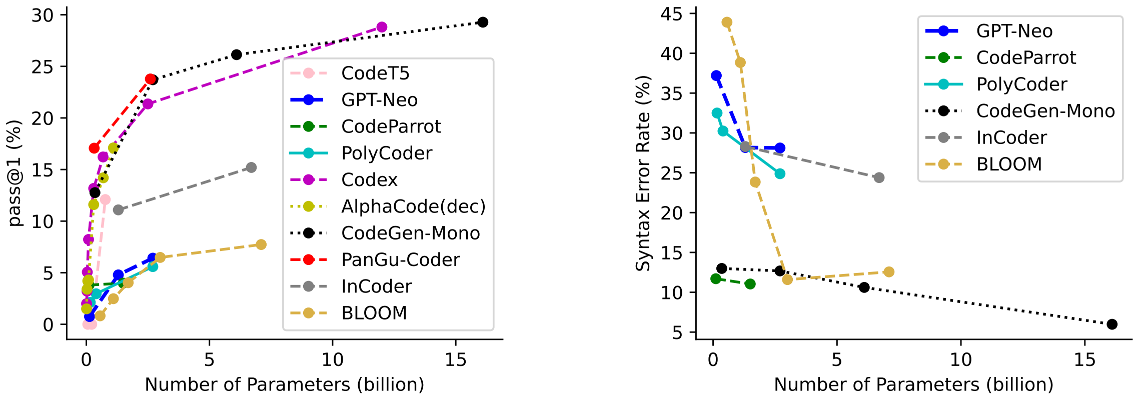
Model	**Size**	pass@k (%)		
		k=1	k=10	k=100
		Model Size:* ~100*M		
GPT-Neo	125M	0.75	1.88	2.97
CodeParrot	110M	3.80	6.57	12.78
PyCodeGPT	110M	**8.33**	**13.36**	19.13
PolyCoder	160M	2.13	3.35	4.88
Codex	12M	2.00	3.62	8.58
Codex	25M	3.21	7.1	12.89
Codex	42M	5.06	8.8	15.55
Codex	85M	8.22	12.81	**22.40**
AlphaCode(dec)	13M	1.5	3.6	8.6
AlphaCode(dec)	29M	3.4	5.8	11.2
AlphaCode(dec)	55M	4.2	8.2	16.9
AlphaCode(dec)	89M	4.3	12.2	20.0
		Model Size:* ~500*M		
CodeT5†	770M	12.09	19.24	30.93
PolyCoder	400M	2.96	5.29	11.59
JuPyT5	300M	5.40	15.46	25.60
BLOOM	560M	0.82	3.02	5.91
Codex	300M	13.17	20.37	36.27
Codex	679M	16.22	**25.70**	**40.95**
AlphaCode(dec)	302M	11.6	18.8	31.8
AlphaCode(dec)	685M	14.2	24.4	38.8
CodeGen-Mono	350M	12.76	23.11	35.19
PanGu-Coder	317M	**17.07**	24.05	34.55
		Model Size:* ~1*B		
GPT-Neo	1.3B	4.79	7.47	16.30
CodeParrot	1.5B	3.99	8.69	17.88
BLOOM	1.1B	2.48	5.93	9.62
BLOOM	1.7B	4.03	7.45	12.75
InCoder†	1.3B	11.09	16.14	24.20
AlphaCode(dec)	1.1B	17.1	28.2	45.3
SantaCoder	1.1B	**18**	**29**	**49**
		Model Size:* ~5*B		
GPT-Neo	2.7B	6.41	11.27	21.37
PolyCoder	2.7B	5.59	9.84	17.68
Codex	2.5B	21.36	35.42	59.50
PanGu-Coder	2.6B	23.78	35.36	51.24
BLOOM	3B	6.48	11.35	20.43
BLOOM	7.1B	7.73	17.38	29.47
CodeGen-Mono	2.7B	23.70	36.64	57.01
CodeGen-Mono	6.1B	**26.13**	**42.29**	**65.82**
GPT-J	6B	11.62	15.74	27.74
InCoder	6.7B	15.2	27.8	47.0
		Model Size:* >10*B		
Codex	12B	28.81	46.81	72.31
CodeGen-Mono	16.1B	29.28	49.86	75.00
GPT-NeoX	20B	15.4	25.6	41.2
LaMDA	137B	14.0	–	47.3
BLOOM	176B	15.52	32.20	55.45
PaLM-Coder	540B	36.0	–	88.4
code-cushman-001	–	33.5	54.3	77.4
code-davinci-001	–	39.0	60.6	84.1
code-davinci-002	–	**47.0**	**74.9**	**92.1**

Table 2: Performance on the HumanEval benchmark. † denotes our reproduced results, while others are cited from the original papers. AlphaCode(dec) means the decoder-only version. We also compare the Codex models (code-cushman and code-davinci) provided by OpenAI API. We exclude the models that cannot pass any problem in the benchmark.

[3] https://keg.cs.tsinghua.edu.cn/codegeex

[4] https://aws.amazon.com/cn/codewhisperer

[5] The details of pass@k can be found in Appendix C.1.



(a)

(b)

Figure 3: (a) pass@1 and (b) syntax error rates on the HumanEval benchmark with various model sizes.

Code (Li et al., 2022b), CodeGen-Mono (Nijkamp et al., 2023), and PanGu-Coder (Christopoulou et al., 2022) also exhibit impressive performance. Notably, InCoder (Fried et al., 2023) and SantaCoder (Allal et al., 2023), which use the FIM training method (Bavarian et al., 2022), also obtain remarkably decent results in the left-to-right generation setting. The significant variation in performance leads us to the question: *What makes LLMs successful in NL2Code?* Given the diversity of these models in terms of design choices, we perform a thorough analysis and conclude the answer: ***Large Size, Premium Data, Expert Tuning***. That is, large model and data size, high-quality data and expert hyper-parameter tuning are the key factors for the success of LLMs in the NL2Code task. In this section, we detail our observations and insights from the perspectives of model, data and tuning.

3.1 Large Model Size

As shown in Figure 2 and Table 2, recent LLMs for NL2Code exhibit larger sizes and superior performance. This is consistent with prior findings that an increased number of model parameters can enhance model capabilities (Radford et al., 2019; Thoppilan et al., 2022; Chowdhery et al., 2022). We further demonstrate the correlation between model size and performance in Figure 3a, which compares the pass@1 results of 10 representative models on the HumanEval benchmark. It is clear that larger models generally result in better performance. Furthermore, we also find that current models, regardless of size, still have the potential for improvement through further increases in size. Additional results on the HumanEval and MBPP benchmarks can be found in Appendix Figure 7, which also support this conclusion.

Additionally, we conduct an experiment on the HumanEval benchmark to examine the syntax error rates of the code generated by different models of varying sizes. Specifically, we make the models predict 10 code samples for each programming problem, and then calculate the percentage of code samples that have syntax errors. As shown in Figure 3b, results indicate that larger models tend to have lower syntax error rates. It is noteworthy that the largest version of the CodeGen-Mono model exhibits a remarkably low rate of syntax errors, i.e., 6%. However, as evidenced by Figure 3a and Table 2, the CodeGen-Mono model with 16 billion parameters still has unsatisfactory performance in terms of pass@k , e.g., pass@1 to be 29%. This highlights the fact that the current limitation for large pre-trained models is the generation of semantically correct code.

3.2 Large and Premium Data

As the sizes of LLMs increase in the field of NL2Code, the scale of the corpus used for training also increases. This highlights the importance of selecting and pre-processing high-quality data. In this section, we will discuss various commonly used data sources and pre-processing strategies that are essential for training LLMs.

Early models were trained using manually annotated data pairs of NL and code, and the data sources include CodeSearchNet (Husain et al., 2019), CoST (Zhu et al., 2022b), and XL-CoST (Zhu et al., 2022a). However, manual annotation is labour-intensive and time-consuming. There are also models like GPT-3 (Brown et al., 2020), GPT-Neo (Black et al., 2021), and GPT-

J (Wang and Komatsuzaki, 2021) that are trained on the Pile (Gao et al., 2020), a large-scale unsupervised dataset. However, these models have not yet demonstrated exceptional code generation capabilities due to the limited number of code files in the training corpus. More recently, with the emergence of more powerful LLMs for NL2Code, larger-scale unlabelled code datasets have been proposed, including BigQuery (Google, 2016), CodeParrot's corpus (HuggingFace, 2021a), GitHub-Code (HuggingFace, 2021b), and the Stack (HuggingFace, 2022), which are collected from general domain open-source websites like GitHub[6] and Stack Overflow[7]. Furthermore, there are also specialized datasets proposed for different scenarios, for example, using Jupyter Notebooks or competition programming problems as a training corpus. Released datasets include Jupyter (HuggingFace, 2021c), JuICe (Agashe et al., 2019), APPS (Hendrycks et al., 2021), and CodeNet (IBM, 2021).

In order to ensure the quality of the training corpus, it is common for LLMs to perform data pre-processing on the significant amount of code in the collected data. We carefully review the data pre-processing methods of five powerful LLMs, including Codex (Chen et al., 2021), AlphaCode (Li et al., 2022b), CodeGen (Nijkamp et al., 2023), InCoder (Fried et al., 2023), and PyCodeGPT (Zan et al., 2022b), and identify several commonalities. One is the removal of likely auto-generated or unfinished code files, as they are deemed to be meaningless. Additionally, specific rules are employed to filter out uncommon code files. These rules include factors such as the repository star rating, the file size, the line length, and the alphanumeric rate. In summary, the goal of these pre-processing strategies is to achieve a code corpus that is unduplicated, complete, correct, clean, and general in nature.

3.3 Expert Tuning

Training an excellent model requires careful consideration of various design choices and hyperparameters. After reviewing the existing 27 LLMs (summary in Appendix Table 6), we have the following findings. Firstly, these LLMs share some common settings. For example, we observe that the optimizer of the current models is almost all Adam (Kingma and Ba, 2014) or its variants (Loshchilov and Hutter, 2017). We also find that initializing with other natural language models yields no noticeable gain compared to training from scratch, except for accelerating convergence (Chen et al., 2021). Furthermore, there are several hyper-parameters that require expert tuning, such as learning rate, batch size, window size, warmup steps, gradient accumulation steps, and sampling temperature. For the learning rate, we analyze its correlation with model size using six powerful LLMs, as shown in Figure 4. We observe that the learning rate becomes smaller as the model gets larger. To explore the effects of temperature, in Figure 5, we report the performance of two models using multiple temperatures on HumanEval. One observation is that higher temperature leads to lower $\mathrm{pass}@1$ and higher $\mathrm{pass}@100$, which suggests that a higher temperature makes LLMs generate more diverse predictions and vice versa. Besides, some studies (erman Arsenovich Arutyunov and Avdoshin, 2022) have shown that window size is a key factor. An interesting finding is that the

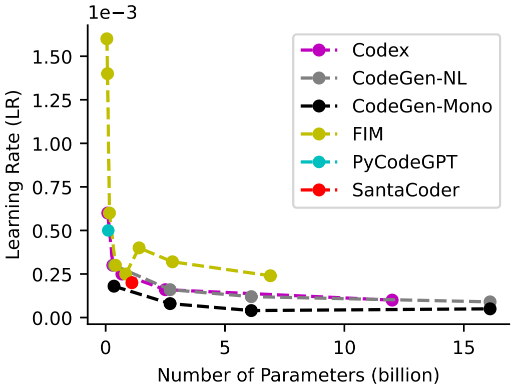


Figure 4: Learning rate of six advanced LLMs in terms of various model sizes.

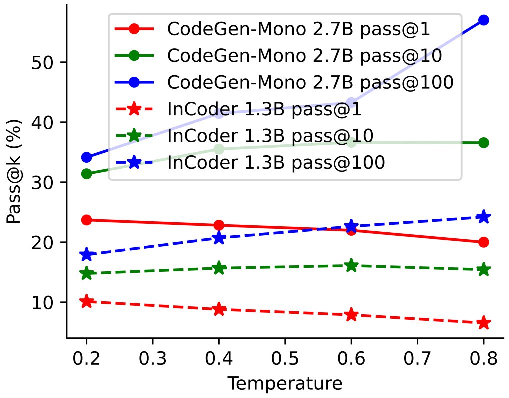


Figure 5: $\mathrm{pass}@k$ on the HumanEval benchmark with different temperatures during model inference.

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

Benchmark	Num.	P. NL	S. PL	Data Statistics					Scenario
				T.N.	*P.C.*	*P.L.*	*S.C.*	*S.L.*	
HumanEval (2021)	164	English	Python	7.8	450.6	13.7	180.9	6.8	Code Exercise
MBPP (2021)	974	English	Python	3.1	78.6	1.0	181.1	6.7	Code Exercise
APPS (2021)	5,000	English	Python	21.0	1743.4	41.6	473.8	21.4	Competitions
CodeContests (2022b)	165	English	Multi.	203.7	1989.2	66.4	2239.3	92.1	Competitions
DS-1000 (2022)	1,000	English	Python	1.6	879.1	31.6	137.4	5.0	Data Science
DSP (2022b)	1,119	English	Python	2.1	756.9	17.8	226.3	7.6	Data Science
MBXP (2022)	974*	English	Multi.	3.1	419.9	14.8	–	–	Multilingual
MBXP-HumanEval (2022)	164*	English	Multi.	7.8	825.6	30.0	–	–	Multilingual
HumanEval-X (2023)	164*	English	Multi.	7.8	468.4	15.5	264.6	12.1	Multilingual
MultiPL-HumanEval (2022)	164*	English	Multi.	7.8	453.9	13.0	–	–	Multilingual
MultiPL-MBPP (2022)	974*	English	Multi.	3.1	181.2	5.4	–	–	Multilingual
PandasEval (2022b)	101	English	Python	6.5	244.5	7.2	46.2	1.3	Public Library
NumpyEval (2022b)	101	English	Python	3.5	222.9	7.0	29.9	1.1	Public Library
TorchDataEval (2022a)	50	English	Python	1.1	329.0	8.6	50.7	1.3	Private Library
MTPB (2023)	115	English	Python	–	72.7	1.0	–	–	Multi-Turn
ODEX (2022c)	945	Multi.	Python	1.8	26.6	2.0	50.4	1.9	Open-Domain
BIG-Bench (2022)	32	English	Python	4.7	341.8	3.0	–	–	Code Exercise

Table 3: Summary of 17 benchmarks for NL2Code. Num. denotes the number of instances in the benchmark, P.NL denotes Problem description's Natural Language, S.PL denotes code Solution's Programming Language, and T.N. denotes the average Number of Test cases. P.C. and P.L. (S.C. and S.L.) stand for the average number of Characters and Lines in Problem description (code Solution). * denotes the number of instances per programming language.

small model with a large window size sometimes outperforms the large model with a small window size (details in Appendix D). In addition, powerful LLMs usually train a new tokenizer on code corpus primarily using two techniques: Byte-level Byte-Pair-Encoding (Radford et al., 2019) and SentencePiece (Kudo and Richardson, 2018). A new tokenizer can be more effective and accurate in splitting code content into tokens. These proven tuning techniques will serve as valuable references for training more powerful LLMs.

4 Benchmarks and Metrics

To evaluate the NL2Code task, high-quality benchmarks and reliable metrics are fundamental and essential. In this section, we provide a brief overview of current benchmarks and metrics, as well as our observations and the open challenges.

We summarize 17 well-studied NL2Code benchmarks in Table 3, where we can find that each of these benchmarks has its own characteristics regarding size, language, complexity, and scenario. We observe that most benchmarks contain a limited number of instances. For example, the widely used HumanEval and MBPP have 164 and 974 instances, respectively. This is because these benchmarks are typically hand-written to ensure that LLMs have not seen them during training. In the era of large language models, it is crucial to avoid data leakage when creating new benchmarks. Additionally, most current benchmarks have their problem descriptions in English and code solutions in Python. Recently, several multi-lingual benchmarks have been proposed, such as MBXP (Athiwaratkun et al., 2022), HumanEvalX (Zheng et al., 2023), and MultiPL (Cassano et al., 2022), which cover multiple programming languages, and ODEX (Wang et al., 2022c), which covers multiple natural languages. Details of multi-lingual benchmarks are listed in Appendix Table 7. Furthermore, benchmarks have been proposed for other practical scenarios, such as data science (Lai et al., 2022), public library (Zan et al., 2022b), private library (Zan et al., 2022a), multi-turn program synthesis (Nijkamp et al., 2023), and code security (Siddiq and msiddiq, 2022). For execution-based benchmarks, comprehensive test cases with complete coverage of the generated program can ensure the trustworthiness of evaluation results. As a reference, the average number of test cases for each benchmark, as well as the length statistics of the problem descriptions and solutions are also provided in Table 3.

Manually evaluating the generated code is impractical, which calls for the need for automatic metrics. The above mentioned benchmarks all provide test cases for execution-based evaluation, where metrics such as pass@k (Chen et al., 2021), n@k (Li et al., 2022b), test case average (Hendrycks et al., 2021), and execution accuracy (Rajkumar et al., 2022) can be used. However, this approach has stringent requirements for the quality of test cases and can only evaluate executable code. For non-executable code, metrics like BLEU (Papineni et al., 2002), ROUGE (Lin, 2004), and CodeBLEU (Ren et al., 2020) are used, while they can not precisely evaluate the correctness of the code. So far, there are many open challenges in designing metrics to evaluate various aspects of code, such as vulnerability, maintainability, clarity, execution complexity, and stability.

5 Challenges and Opportunities

Our investigations have revealed that advances in LLMs for NL2Code have a considerable impact on both academia and industry. Despite this progress, there are still numerous challenges that need to be addressed, offering ample opportunities for further research and applications. In this section, we explore the challenges and opportunities in terms of the ability gap between LLMs and humans.

Understanding Ability The inherent flexibility of natural language allows for a variety of expressions to convey functional requirements. Humans are able to understand various descriptions at different levels of abstraction. In contrast, current LLMs tend to be sensitive to the given context, which may cause unexpected performance degradation (Wang et al., 2022a). In addition, LLMs may struggle when faced with complex problems that have numerous conditions and requirements (Barke et al., 2022; Imai, 2022). We believe exploring the understanding abilities of LLMs is a crucial research direction. One potential solution is to break down complex problems into multiple steps, as is commonly done in reasoning tasks (Wei et al., 2022).

Judgement Ability Humans have the ability to determine whether they can solve a programming problem or not. While current models will always return a solution even if there is no answer to the problem, due to the fact that they are trained by unsupervised causal language modeling objective. This can cause problems in practical applications. To improve the judgment ability of LLMs, researchers have employed reinforcement learning to leverage user feedback, as seen in models like InstructGPT (Ouyang et al., 2022) and ChatGPT[8]. However, collecting high-quality feedback for code is costly and challenging. There are also ongoing studies (Chen et al., 2023; Key et al., 2022) exploring the possibility of self-validation for LLMs, which is also a promising research direction.

Explanation Ability It is widely acknowledged that human developers possess the ability to interpret the meaning of the code they write, which is crucial for educational purposes and software maintenance. Recent studies showed that LLMs have the potential to automatically generate code explanations. MacNeil et al. (2022a) proposed using LLMs to generate code explanations for students during their learning process, and MacNeil et al. (2022b) proposed explaining numerous aspects of a given code snippet using Copilot. Further research and explorations are necessary to fully realize the potential of LLMs in this regard.

Adaptive Learning Ability A fundamental difference between current large language models and humans is their ability to adapt to new and updated knowledge. Human developers possess a unique ability to quickly search and learn new materials, such as programming documentation, and adapt to changes in APIs with relative ease. However, re-training or fine-tuning LLMs requires significant effort and resources. This issue has inspired a number of recent studies, such as DocCoder (Zhou et al., 2023) and APICoder (Zan et al., 2022a), which utilize retrieval-based methods to provide extra or updated knowledge during model inference. Despite these advancements, it remains an open challenge to endow LLMs with the powerful learning capabilities humans possess.

Multi-tasking Ability Large language models have been applied to a variety of code-related tasks, such as code repair (Joshi et al., 2022; Prenner and Robbes, 2021), code search (Neelakantan et al., 2022), and code review (Li et al., 2022c) as well as non-code tasks that can be formatted in a code-like manner, such as mathematics (Drori and Verma, 2021; Drori et al., 2021) and chemistry (Krenn et al., 2022; Hocky and White, 2022). However, there are differences between LLMs and human abilities in terms of multi-tasking. Humans can seamlessly switch between tasks, while LLMs may require sophisticated prompt engineering (Liu et al., 2023). Another evidence is that LLMs lack the ability to quickly master multiple programming languages (Zheng et al., 2023) as humans do. These limitations highlight areas for future research.

[8] https://chat.openai.com

6 Conclusion

In this paper, we survey 27 existing large language models for NL2Code, and draw a thorough analysis of the underlying reasons for their success. We also provide a detailed review of benchmarks and metrics. Regarding the gap between models and humans, we present ongoing challenges and opportunities. In addition, we have developed a website to track the latest findings in this field. We hope this survey can contribute to a comprehensive overview of the field and promote its thriving evolution.

Limitations

In this paper, we thoroughly investigate the existing large language models for NL2Code, and summarize them from diverse perspectives with our own thinking. However, as this field is evolving so rapidly, there may be aspects that we have overlooked, or some new works that we have not covered. To mitigate this issue, we have created a website to track the latest progress through crowdsourcing, hoping that it will continually contribute to the development of the field. Besides, the existing LLMs possess their own characteristics in terms of model size, architecture, corpus, pre-processing, tokenizer, hyper-parameters, and training platforms. Also, some of them are currently not publicly available, such as AlphaCode (Li et al., 2022b) and PaLM-Coder (Chowdhery et al., 2022). Therefore, it is almost impractical to conduct a completely fair comparison. We tried our best to show a kind of comparison on the popular HumanEval and MBPP benchmarks, hoping that it can provide clues to the differences in performance of different LLMs. In addition, evaluating LLMs has a high cost in computational resources. We thus have made all files generated by the LLMs publicly available on `https://nl2code.github.io`.

References

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

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668.

aiXcoder. 2018. aiXcoder. `https://aixcoder.com`.

Alibaba. 2022. Alibaba. `https://github.com/alibaba-cloud-toolkit/cosy`.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alexander Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, J. Poirier, Hailey Schoelkopf, Sergey Mikhailovich Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Franz Lappert, Francesco De Toni, Bernardo Garc'ia del R'io, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luisa Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Christopher Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! *ArXiv*, abs/2301.03988.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.

Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, pages 472–483.

Amazon. 2022. CodeWhisperer. `https://aws.amazon.com/cn/codewhisperer`.

Anonymous. 2022. CodeT5Mix: A pretrained mixture of encoder-decoder transformers for code understanding and generation. In *Submitted to The Eleventh International Conference on Learning Representations*. Under review.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddharth Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2022. Multi-lingual evaluation of code generation models. *ArXiv*, abs/2210.14868.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *ArXiv*, abs/2108.07732.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2022. Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7:85 – 111.

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

Mohammad Bavarian, Heewoo Jun, Nikolas A. Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *ArXiv*, abs/2207.14255.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An open-source autoregressive language model. In *Proceedings of the ACL Workshop on Challenges & Perspectives in Creating Large Language Models*.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Neural Information Processing Systems*, 33:1877–1901.

Federico Cassano, John Gouwar, Daniel Nguyen, Sy Duy Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. A scalable and extensible approach to benchmarking nl2code for 18 programming languages. *ArXiv*, abs/2208.08227.

Yekun Chai, Shuohuan Wang, Chao Pang, Yu Sun, Hao Tian, and Hua Wu. 2022. ERNIE-Code: Beyond english-centric cross-lingual pretraining for programming languages. *arXiv preprint arXiv:2212.06742*.

Shubham Chandel, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. 2022a. Training and evaluating a jupyter notebook data science assistant. *ArXiv*, abs/2201.12901.

Shubham Chandel, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2022b. Training and evaluating a jupyter notebook data science assistant. *arXiv preprint arXiv:2201.12901*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

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

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C. Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhong-Yi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li yu Yan, Pingyi Zhou, Xin Wang, Yu Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jia Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. 2022. PanGu-Coder: Program synthesis with function-level language modeling. *ArXiv*, abs/2207.11280.

Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: Multi-mode translation of natural language and python code with transformers. In *Conference on Empirical Methods in Natural Language Processing*.

CodedotAl. 2021. GPT Code Clippy: The Open Source version of GitHub Copilot. `https://github.com/CodedotAl/gpt-code-clippy`.

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

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*.

DeepGenX. 2022. CodeGenX. https://docs.deepgenx.com.

Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. 2022. Code generation using machine learning: A systematic review. *IEEE Access*.

Premkumar T. Devanbu. 2012. On the naturalness of software. *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847.

Iddo Drori and Nakul Verma. 2021. Solving linear algebra by program synthesis. *arXiv preprint arXiv:2111.08171*.

Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, Roman Wang, Nikhil Singh, Taylor Lee Patti, J. Lynch, Avi Shporer, Nakul Verma, Eugene Wu, and Gilbert Strang. 2021. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences of the United States of America*, 119.

Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 823–833.

erman Arsenovich Arutyunov and Sergey Avdoshin. 2022. Big transformers for code generation. *Proceedings of the Institute for System Programming of the RAS*.

FauxPilot. 2022. FauxPilot. https://github.com/moyix/fauxpilot.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

GitHub. 2021. GitHub Copilot. https://github.com/features/copilot.

Google. 2016. GitHub on BigQuery: Analyze all the open source code. https://cloud.google.com/bigquery.

Google. 2022. Big-bench. https://github.com/google/BIG-bench.

Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Xiaodong Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. In *Neural Information Processing Systems*.

Glen M Hocky and Andrew D White. 2022. Natural language processing models that automate programming will transform chemistry research and teaching. *Digital discovery*, 1(2):79–83.

HuggingFace. 2021a. CodeParrot Dataset. https://huggingface.co/datasets/transformersbook/codeparrot.

HuggingFace. 2021b. Github-Code. https://huggingface.co/datasets/codeparrot/github-code.

HuggingFace. 2021c. GitHub-Jupyter. https://huggingface.co/datasets/codeparrot/github-jupyter.

Huggingface. 2021. Training CodeParrot from Scratch. https://huggingface.co/blog/codeparrot.

HuggingFace. 2022. The Stack. https://huggingface.co/datasets/bigcode/the-stack.

Hamel Husain, Hongqi Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the state of semantic code search. *ArXiv*, abs/1909.09436.

IBM. 2021. CodeNet. https://github.com/IBM/Project_CodeNet.

Saki Imai. 2022. Is github copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 319–321.

Srini Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 1:215–224.

Aravind Joshi and Owen Rambow. 2003. A formalism for dependency grammar based on tree adjoining grammar. In *Proceedings of the Conference on Meaning-text Theory*, pages 207–216. MTT Paris, France.

Harshit Joshi, José Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. 2022. Repair is nearly generation: Multilingual program repair with llms. *arXiv preprint arXiv:2208.11640*.

Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2023. Explainable automated debugging via large language model-driven scientific debugging. *arXiv preprint arXiv:2304.02195*.

Darren Key, Wen-Ding Li, and Kevin Ellis. 2022. I Speak, You Verify: Toward trustworthy neural program synthesis. *arXiv preprint arXiv:2210.00848*.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Mario Krenn, Qianxiang Ai, Senja Barthel, Nessa Carson, Angelo Frei, Nathan C Frey, Pascal Friederich, Théophile Gaudin, Alberto Alexander Gayle, Kevin Maik Jablonka, et al. 2022. Selfies and the future of molecular string representations. *Patterns*, 3(10):100588.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Yih, Daniel Fried, Si yi Wang, and Tao Yu. 2022. DS-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven CH Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv preprint arXiv:2207.01780*, abs/2207.01780.

Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)*, 53(3):1–38.

Yaoxian Li, Shiyi Qi, Cuiyun Gao, Yun Peng, David Lo, Zenglin Xu, and Michael R Lyu. 2022a. A closer look into transformer-based code intelligence through code transformation: Challenges and opportunities. *arXiv preprint arXiv:2207.04285*.

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

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022c. Automating code review activities by large-scale pretraining. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

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

Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. 2016. Automatic code generation of convolutional neural networks in fpga implementation. In *2016 International conference on field-programmable technology (FPT)*, pages 61–68. IEEE.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *ArXiv*, abs/2102.04664.

Lechanceux Luhunu and Eugene Syriani. 2017. Survey on template-based code generation. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*.

Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2022a. Experiences from using code explanations generated by large language models in a web software development e-book. *arXiv preprint arXiv:2211.02265*.

Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. 2022b. Generating diverse code explanations using the gpt-3 large language model. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2*, pages 37–39.

Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347. IEEE.

Microsoft. 2019. IntelliCode. https://github.com/MicrosoftDocs/intellicode.

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*.

Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of github copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 1–5.

Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

University of Oxford. 2020. Diffblue Cover. https://www.diffblue.com.

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

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Dipti Pawade, Avani Sakhapara, Sanyogita Parab, Divya Raikar, Ruchita Bhojane, and Henali Mamania. 2018. Literature survey on automatic code generation techniques. *i-Manager's Journal on Computer Science*, 6(2):34.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE.

Julian Aron Prenner and Romain Robbes. 2021. Automatic program repair with openai's codex: Evaluating quixbugs. *arXiv preprint arXiv:2111.03922*.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498*.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. 2022. Self-critiquing models for assisting human evaluators. *arXiv preprint arXiv:2206.05802*.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. BLOOM: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Meet Shah, Rajat Shenoy, and Radha Shankarmani. 2021. Natural language to python source code using transformers. In *2021 International Conference on Intelligent Technologies (CONIT)*, pages 1–4. IEEE.

Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, and Federica Sarro. 2021. A survey on machine learning techniques for source code analysis. *arXiv preprint arXiv:2110.09610*.

Jiho Shin and Jaechang Nam. 2021. A survey of automatic code generation from natural language. *Journal of Information Processing Systems*, 17(3):537–555.

Mohammed Latif Siddiq and msiddiq. 2022. SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*.

Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022a. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1019–1027.

Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022b. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1019–1027.

Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2018. A grammar-based structural cnn decoder for code generation. In *AAAI Conference on Artificial Intelligence*.

Ilya Sutskever, Geoffrey E Hinton, and Graham W Taylor. 2008. The recurrent temporal restricted boltzmann machine. *Neural Information Processing Systems*, 21.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. 2018. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures*, 52:43–62.

tabnine. 2018. TabNine. https://www.tabnine.com.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. LAMDA: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Neural Information Processing Systems*, 30.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 397–407.

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

Shiqi Wang, Zheng Li, Haifeng Qian, Cheng Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2022a. ReCode: Robustness evaluation of code generation models. *arXiv preprint arXiv:2212.10264*.

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

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F Xu, and Graham Neubig. 2022b. MCoNaLa: a benchmark for code generation from multiple natural languages. *arXiv preprint arXiv:2203.08388*.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022c. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A systematic evaluation of large language models of code. *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*.

Yichen Xu and Yanqiao Zhu. 2022. A survey on pretrained language models for neural code intelligence. *arXiv preprint arXiv:2212.10079*.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022a. When language model meets private library. In *Conference on Empirical Methods in Natural Language Processing*.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022b. CERT: Continual pretraining on sketches for library-oriented code generation. In *International Joint Conference on Artificial Intelligence*.

Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shanshan Wang, Yufei Xue, Zi-Yuan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *ArXiv*, abs/2303.17568.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao JIang, and Graham Neubig. 2023. DocCoder: Generating code by retrieving and reading docs. In *The Eleventh International Conference on Learning Representations*.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022a. XLCoST: A benchmark dataset for cross-lingual code intelligence.

Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022b. Multilingual code snippets training for program translation.

A Related Surveys

Previous surveys on the topic of code intelligence (Allamanis et al., 2018; Le et al., 2020; Li et al., 2022a; Xu and Zhu, 2022) and code generation (Pawade et al., 2018; Shin and Nam, 2021; Dehaerne et al., 2022) have primarily focused on early methodologies such as the use of programming templates (Syriani et al., 2018; Luhunu and Syriani, 2017), neural models based on CNN, RNN, and LSTM architectures (Allamanis et al., 2018; Sharma et al., 2021), and small-scale Transformer models that require labelled data for training (Mastropaolo et al., 2021; Shah et al., 2021). However, with the advancement of model size, Transformer-based models have demonstrated exceptional performance in NL2Code tasks and have given rise to the development of more capable code generation models. In light of this, there exists a clear need for a comprehensive survey of large language models for NL2Code tasks to bridge this gap in knowledge. This study endeavours to fulfill this need by providing a thorough analysis of the successful LLMs and a detailed review of NL2Code benchmarks and metrics. We also present the ongoing challenges and opportunities regarding the ability gap between LLMs and humans.

Finally, we would like to highlight some criteria for our survey. First, we only refer to official papers to investigate the size of the models. For example, Codex reported the model with a maximum size of 12B in the paper, but later trained larger ones. In this case, we only consider the 12B model as the largest one. In addition, the publication dates of the models in Figure 2 are taken from official papers or blogs.

B An Online Website

To keep tracking the latest progress of LLMs for NL2Code, we have developed an online real-time update website at `https://nl2code.github.io`. We have collected as many of the latest research works as possible on this website. Everyone is allowed to contribute to the website by pulling requests on GitHub. This website also includes features such as fuzzy search and custom tag categories, which will facilitate researchers to find the papers they want quickly. We hope this website can assist researchers and developers in related fields and contribute to its advancement.

Model	**Size**	pass@k		
		k=1	k=10	k=100
Model Size: ~100M				
GPT-Neo†	125M	0.26	2.15	7.96
CodeParrot†	110M	0.48	3.89	15.93
PyCodeGPT†	110M	**9.39**	**28.37**	**48.71**
PolyCoder†	160M	1.08	6.67	18.97
Model Size: ~500M				
CodeT5†	770M	**15.78**	38.63	50.35
PolyCoder†	400M	1.31	7.98	21.55
BLOOM†	560M	0.26	2.04	8.90
CodeGen-Mono†	350M	15.44	**42.50**	**64.40**
Model Size: ~1B				
GPT-Neo†	1.3B	3.77	16.26	29.51
CodeParrot†	1.5B	1.29	8.66	27.17
BLOOM†	1.1B	1.90	9.20	23.42
BLOOM†	1.7B	3.16	14.23	31.38
InCoder†	1.3B	**10.00**	**34.02**	**55.50**
SantaCoder†	1.1B	3.65	21.33	41.92
Model Size: ~5B				
GPT-Neo†	2.7B	5.89	23.09	44.26
PolyCoder†	2.7B	4.39	17.99	38.17
BLOOM†	3B	2.25	13.58	32.08
BLOOM†	7.1B	1.01	7.91	24.12
CodeGen-Mono†	2.7B	28.80	60.73	**75.41**
CodeGen-Mono†	6.1B	**33.70**	**62.70**	70.25
GPT-J†	6B	11.30	35.62	53.63
InCoder	6.7B	21.3	46.5	66.2
Model Size: >10B				
CodeGen-Mono	16.1B	42.4	65.8	79.1
cushman-001	–	45.9	66.9	79.9
davinci-001	–	51.8	72.8	84.1
davinci-002	–	**58.1**	**76.7**	**84.5**

Table 4: The performance of LLMs on the MBPP benchmark. † denotes our reproduced results, while others are taken from Chen et al. (2023). We omit CodeGPT, GPT-CC, and PLBART as their numbers are zero.

C Experimental Setup

In this section, we will first present the definition of pass@k , followed by the details of the experiments conducted on two benchmarks, namely HumanEval (Chen et al., 2021) (results in Table 2) and MBPP (Austin et al., 2021) (results in Table 4).

C.1 Definition of pass@k

We use pass@k as our metric for evaluation. For each programming problem, we sample n candidate code solutions and then randomly pick k of them. If any of the k code solutions pass the given test cases, the problem can be regarded as solved. So pass@k is the proportion of solved problems in the benchmark (Chen et al., 2021). Formally, assuming that the number of correct ones in k samples is c, pass@k $= 1$ if $n - c < k$; otherwise, pass@k $= 1 - \prod_{i=n-c+1}^{n}(1 - k/i)$. We chose pass@$k$ as our primary evaluation metric because it offers a completely precise evaluation of code accuracy by executing test cases, while other metrics mentioned in Section 4 either originate from pass@k or have lower precision.

C.2 Implementation Details

For HumanEval, we use the original benchmark[9]. Most results in Table 2 are taken from the original papers, while we reproduce the results of GPT-CC, PLBART, CodeT5, and InCoder 1.3B by strictly following the same experimental setup as the other models. In detail, we set the sample number to 200, the maximum length of newly generated tokens to 200, and `top_p` to 0.95. We set the temperature from 0.1 to 1.0 with an interval of 0.1, and report the best performance across these temperatures.

For MBPP, we use the version from Chen et al. (2023)[10]. In Table 4, the results of InCoder 6.7B and models larger than 10B are taken from Chen et al. (2023), while we reproduced other results. Specifically, we set the sample number to 100, the maximum length of newly generated tokens to 200, `top_p` to 0.95, and the temperature to 0.8.

For the two benchmarks above, we employ the same post-processing strategy. Following Codex (Chen et al., 2021), we terminate the sampling process when one of the following sequences is encountered in the generated code: '`\nclass`', '`\ndef`', '`\n#`', '`\n@`', '`\nif`', and '`\nprint`'. In our experiments, CodeT5 770M refers to the version[11] with the causal language modeling objective. For good reproducibility and further research, we have made our code and the generated results of the LLMs on HumanEval and MBPP publicly available on our website.

D Context Window vs. Performance

Recent work (erman Arsenovich Arutyunov and Avdoshin, 2022) claimed that the size of the context window plays a vital role in enhancing the performance of LLMs for NL2Code. Specifically, experiments are conducted on the APPS benchmark (Hendrycks et al., 2021) with GPT-NeoX (Black et al., 2022), and we visualize the results in Figure 6. It is found that the 165M version model with an $8,000$ context window is comparable to the 20B version model with a $2,000$ context window. This observation illustrates that the context window also needs to be considered when training the model.

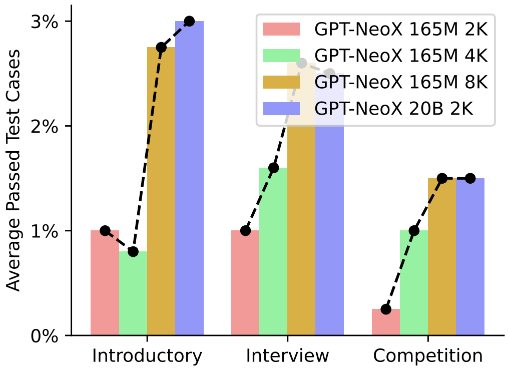


Figure 6: Performance of GPT-NeoX with different model sizes (165M and 20B) and context windows (2K, 4K, and 8K) on the APPS benchmark.

[9] https://github.com/openai/human-eval/blob/master/data/HumanEval.jsonl.gz

[10] https://github.com/microsoft/CodeT/blob/main/CodeT/data/dataset/mbpp_sanitized_for_code_generation.jsonl

[11] https://huggingface.co/Salesforce/codet5-large-ntp-py

Products	**Model**	**Supported PLs**	**Supported IDEs**
tabnine (2018)	–	Python, Java, Javascript, TypeScript, Go, Ruby, PHP, C#, C, C++, Swift, Perl, Rust, CSS, Angular, Dart, React, Haskell, HTML, Kotlin, Matlab, Sass, NodeJS, Objective C, Scala,	VS Code, Visual Studio, IntelliJ IDE, Neovim, Sublime, PyCharm, Rider, WebStorm, Android Studio, Emacs, Vim, PhpStorm, RubyMine, DataGrip, Jupyter Notebook, JupyterLab, Clion, AppCode, Eclipse, GoLand
aiXcoder (2018)	–	Python, Java, JavaScript, Typescript, Go, PHP, C, C++	VS Code, IntelliJ IDEA, PyCharm, STS3, WebStorm, Rider, Clion, STS4 Android Studio, PhpStorm, Eclipse, GoLand
IntelliCode (2019)	–	Python, Java, JavaScript, TypeScript, C#, C++, SQL Server, XAML	VS Code, Visual Studio
Diffblue Cover (2020)	–	Java	IntelliJ IDEA, CLI Tool
Copilot (2021)	Codex	Python, Java, JavaScript, TypeScript, Go, Ruby, Julia, PHP, C#, C++, Swift, Perl, PowerShell, R, Rust, CSS, SQL, JSON, HTML, SCSS, Less, .NET, Markdown, T-SQL	VS Code, Visual Studio, Neovim, JetBrains IDE
Cosy (2022)	–	Java	IntelliJ IDEA
CodeWhisperer (2022)	–	Python, Java, JavaScript, TypeScript, C#	VS Code, JetBrains IDE, AWS Cloud9, AWS Lambda
CodeGenX (2022)	GPT-J	Python	VS Code
CodeGeeX (2023)	CodeGeeX	Python, Java, JavaScript, TypeScript, Go, PHP, C#, C, C++, Perl, Rust, CSS, SQL, HTML, Kotlin, Shell, R, Cuda, Objective C, Objective C++, Pascal, Tex, Fortran, Lean, Scala	VS Code, IntelliJ IDEA, PyCharm, WebStorm, Android Studio, Rider, RubyMine, Clion, AppCode, Aqua, DataGrip, GoLand, DataSpell
FauPilot (2022)	CodeGen	Python, Java, Javascript, Go, C, C++	–

Table 5: Summary of products powered by LLMs. PLs and IDEs refer to programming languages and integrated development environments, respectively. The information for these products was recorded on December 27, 2022.

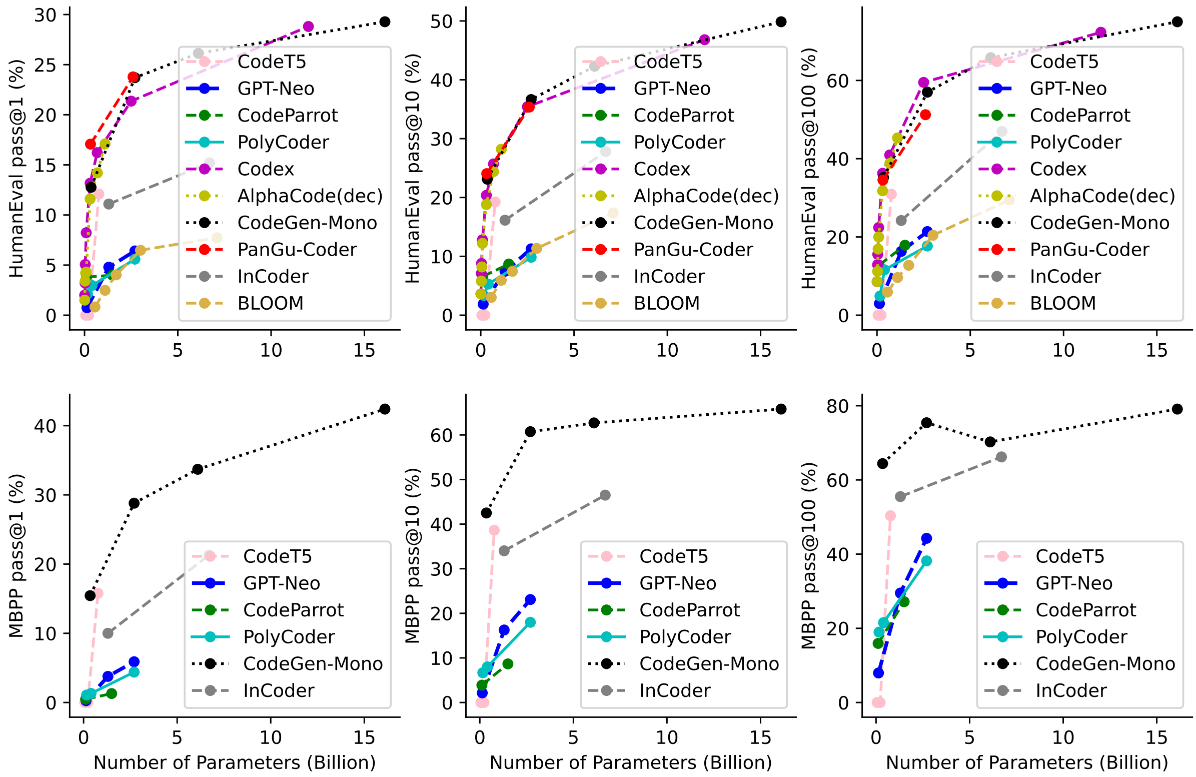


Figure 7: Performance of LLMs with varying parameter sizes on the HumanEval and MBPP benchmarks.

Model	*Data*		*Model Hyper-parameters*											*Training*	
	de.	*token.*	*opti.*	*betas*	*eps*	*bs*	*ws*	*gss*	*wp*	*lr*	*wd*	*decay*	*pr*	*init.*	*m.*
Decoder															
GPT-C 366M	×	BBPE	Adam	–	–	–	1,024	–	–	6.25e-5	–	Cosine	–	Scratch	→
CodeGPT 124M	×	BBPE	Adam	–	–	–	768	–	–	5e-5	–	–	–	GPT-2	→
GPT-Neo 2.7B	–	BBPE	Adam	0.9,0.95	1e-8	–	2,048	–	3,000	–	0.1	Cosine	–	Scratch	→
GPT-J 6B	–	BBPE	Adam	–	–	–	2,048	16	3,000	–	0.1	–	BF16	–	→
Codex 12B	✓	BBPE	Adam	0.9,0.95	1e-8	2M	4,096	–	175	1e-4	0.1	Cosine	–	GPT-3	→
GPT-CC 1.3B	✓	BBPE	AdaFa	–	–	–	1,024	–	5,000	2e-5	0.1	Linear	–	GPT-Neo	→
CodeParrot 1.5B	✓	BBPE	AdamW	0.9,0.999	1e-8	524K	1,024	16	750	5e-5	0.1	Cosine	–	Scratch	→
LaMDA 137B	–	SP	–	–	–	256K	–	–	–	–	–	–	–	–	→
PolyCoder 2.7B	✓	BBPE	AdamW	0.9,0.999	1e-8	262K	2,048	–	1,600	1.6e-4	–	Cosine	–	Scratch	→
CodeGen 16.1B	✓	BBPE	Adam	0.9,0.999	1e-8	2M	2,048	–	3,000	0.5e-4	0.1	Cosine	–	–	→
InCoder 6.7B	✓	BBPE	Adam	0.9,0.98	–	–	2,048	–	1,500	–	–	PN	–	Scratch	↔
GPT-NeoX 20B	✓	BBPE	ZeRo	0.9,0.95	1e-8	3.15M	2,048	32	–	9.7e-5	0.01	Cosine	FP16	Scratch	→
PaLM-Coder 540B	✓	SP	Adafa.	–	–	–	2,048	–	–	1e-2	–	–	–	PaLM	→
PanGu-Coder 2.6B	✓	SP	Adam	0.9,0.95	–	–	1,024	–	–	–	0.01	Cosine	–	Scratch	→
FIM 6.9B	✓	BBPE	Adam	–	–	2M	2,048	–	–	2.4e-4	–	–	–	Scratch	↔
PyCodeGPT 110M	✓	BBPE	AdamW	0.9,0.95	1e-8	480K	1,024	4	1,000	5e-4	0.1	Cosine	FP16	Scratch	→
CodeGeeX 13B	–	BBPE	ZeRo	0.9,0.95	–	–	2,048	–	–	–	0.1	Cosine	FP16	–	→
BLOOM 176B	✓	BBPE	Adam	0.9,0.95	–	–	2,048	–	–	6e-5	0.1	Cosine	BF16	Scratch	→
SantaCoder 1.1B	✓	BBPE	Adam	0.9,0.95	1e-8	–	–	–	–	2e-4	0.1	Cosine	FP16	Scratch	↔
Encoder-Decoder															
PyMT5 374M	✓	BBPE	Adam	0.9,0.98	1e-6	–	2,200	–	5,000	9.1875e-5	0.01	IS	FP16	–	→
PLBART 406M	×	SP	Adam	–,0.98	1e-6	–	768	–	–	5e-5	–	Linear	FP16	–	→
CodeT5 770M	×	BBPE	AdamW	–	–	–	–	–	1,000	2e-4	0.05	Linear	FP16	Scratch	→
JuPyT5 350M	✓	BBPE	Adam	0.9,0.98	1e-6	–	2,200	–	5,000	9.1875e-5	0.01	IS	FP16	PyMT5	→
AlphaCode 41.1B	✓	SP	AdamW	0.9,0.95	–	–	6,144	–	1,000	1e-4	0.1	Cosine	BF16	–	→
CodeRL 770M	–	BBPE	AdamW	–	–	–	–	–	–	–	–	PN	–	CodeT5	→
CodeT5Mix 770M	✓	BBPE	AdamW	–	–	–	–	–	–	–	0.1	Linear	FP16	Scratch	→
ERNIE-Code 560M	×	SP	AdaFa	–	–	–	1,024	15	1,000	1e-4	–	Linear	BF16	mT5	→

Table 6: The details of LLMs for NL2Code. We list the full names of these abbreviations: de-duplication (***de.***), tokenizer (***token.***), optimizer (***opti.***), batch size (***bs***), window size (***ws***), gradient accumulation steps (***gss***), warmup steps (***wp***), learning rate (***lr***), weight decay (***wd***), decay schedule (***decay***), precision floating point (***pr***), model initialization (***init.***), left-to-right (→), fill-in-the-middle (↔), byte-level byte-pair-encoding (BBPE), SentencePiece (SP), polynomial (PN), and inverse square (IS).

Benchmark	Originate From	Multilingual
MCoNaLa (2022b)	CoNaLa (2018)	English, Spanish, Japanese, Russian
ODEX (2022c)	CoNaLa (2018) MCoNaLa (2022b)	English, Spanish, Japanese, Russian
MBXP (2022)	MBPP (2021)	Python, Java, JavaScript, TypeScript, Go, Ruby, Kotlin, PHP, C#, Scala, C++, Swift, Perl
MBXP-HumanEval (2022)	HumanEval (2021)	Python, Java, JavaScript, Ruby, Kotlin, PHP, Scala, Swift, Perl,
MultiPL-MBPP (2022)	MBPP (2021)	Python, Java, JavaScrpt, TypeScript, Go, Ruby, Julia, PHP, C#, Scala, C++, Swift, Perl, D, Bash, Racket, Lua, R, Rust
MultiPL-HumanEval (2022)	HumanEval (2021)	Python, Java, JavaScrpt, TypeScript, Go, Ruby, Julia, PHP, C#, Scala, C++, Swift, Perl, D, Bash, Racket, Lua, R, Rust
HumanEval-X (2023)	HumanEval (2021)	Python, Java, JavaScript, Go, C++

Table 7: Details of multilingual NL2Code benchmarks. Here we also list MCoNaLa and CoNaLa, which have no test case for evaluation.

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
7: Limitations

□ A2. Did you discuss any potential risks of your work?
Not applicable. Left blank.

☑ A3. Do the abstract and introduction summarize the paper's main claims?
0: Abstract 6: Conclusion

☒ A4. Have you used AI writing assistants when working on this paper?
Left blank.

B ☑ Did you use or create scientific artifacts?

Appendix C: Experimental Setup

☑ B1. Did you cite the creators of artifacts you used?
Appendix C: Experimental Setup

□ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
Not applicable. Left blank.

□ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
Not applicable. Left blank.

□ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
Not applicable. Left blank.

□ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
Not applicable. Left blank.

☑ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.
4: Benchmarks and Metrics

C ☑ Did you run computational experiments?

3

☑ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
3

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

☑ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
Appendix C: Experimental Setup

☑ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
Appendix C: Experimental Setup

□ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
Not applicable. Left blank.

D ☒ Did you use human annotators (e.g., crowdworkers) or research with human participants?

Left blank.

□ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
Not applicable. Left blank.

□ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
Not applicable. Left blank.

□ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
Not applicable. Left blank.

□ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
Not applicable. Left blank.

□ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
Not applicable. Left blank.