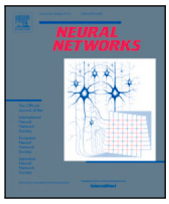
Neural Networks 184 (2025) 107067

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet



Review

Promises and perils of using Transformer-based models for SE research

Yan Xiao [a,*,1], Xinyue Zuo [b,1], Xiaoyue Lu [a], Jin Song Dong [b], Xiaochun Cao [a], Ivan Beschastnikh [c]

[a] Shenzhen Campus of Sun Yat-sen University, No. 66, Gongchang Road, Guangming District, Shenzhen, 518107, Guangdong, China
[b] National University of Singapore, Computing 1, 13 Computing Drive, 117417, Singapore
[c] University of British Columbia, ICICS/CS Building 201-2366 Main Mall, Vancouver, BC, Canada

ARTICLE INFO

Keywords:
Transformer-based pre-trained models
CodeBERT
CodeGPT
CodeT5

ABSTRACT

Many Transformer-based pre-trained models for code have been developed and applied to code-related tasks. In this paper, we analyze 519 papers published on this topic during 2017–2023, examine the suitability of model architectures for different tasks, summarize their resource consumption, and look at the generalization ability of models on different datasets.

We examine three representative pre-trained models for code: CodeBERT, CodeGPT, and CodeT5, and conduct experiments on the four topmost targeted software engineering tasks from the literature: Bug Fixing, Bug Detection, Code Summarization, and Code Search.

We make four important empirical contributions to the field. First, we demonstrate that encoder-only models (CodeBERT) can outperform encoder–decoder models for general-purpose coding tasks, and showcase the capability of decoder-only models (CodeGPT) for certain generation tasks. Second, we study the most frequently used model-task combinations in the literature and find that less popular models can provide higher performance. Third, we find that CodeBERT is efficient in understanding tasks while CodeT5's efficiency is unreliable on generation tasks due to its high resource consumption. Fourth, we report on poor model generalization for the most popular benchmarks and datasets on Bug Fixing and Code Summarization tasks.

We frame our contributions in terms of promises and perils, and document the numerous practical issues in advancing future research on transformer-based models for code-related tasks.

Contents

* Corresponding author.
E-mail address: xiaoy367@mail.sysu.edu.cn (Y. Xiao).
[1] Equal contribution.

https://doi.org/10.1016/j.neunet.2024.107067
Received 3 July 2024; Received in revised form 7 December 2024; Accepted 16 December 2024
Available online 24 December 2024

1. Introduction

The availability of large natural language corpora and advances in ML have led recent models to achieve extraordinary performance on Natural Language Processing (NLP) tasks. Transformer-based architectures (Vaswani et al., 2017) are among the most successful model variants in this field. Transformer-based models, like BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer), have revolutionized NLP tasks, including text classification, sentiment analysis, and language generation.

Given a large number of software code corpora available, Transformer-based models have also rapidly gained traction in software engineering (SE) research (Le-Cong, Kang, Nguyen, Haryono, Lo, Le, & Huynh, 2022), with hundreds of transformer-related papers published in top-tier SE conferences and journals in the past five years. In many instances, these works have reported state-of-the-art performance on a variety of SE tasks. Some example applications of transformer-based techniques include automated program repair (Fu, Tantithamthavorn, Le, Nguyen, & Phung, 2022; Xia & Zhang, 2022; Zhang, Panthaplackel, Nie, Li & Jessy and Gligoric, 2022), merge conflict resolution (Svyatkovskiy et al., 2022; Zhang, Mytkowicz, Kaufman, Piskac, & Lahiri, 2022), requirements engineering (Anish, Lawhatre, Chatterjee, Joshi, & Ghaisas, 2022; Devine, 2022; Ezzini, Abualhaija, Arora, & Sabetzadeh, 2022), and more (Huang et al., 2022; Le-Cong et al., 2022). Model structures, like encoder-only, decoder-only, and encoder–decoder (Chakraborty, Ahmed, Ding, Devanbu, & Ray, 2022), together with the different pre-training objectives, such as generative objectives and denoising objectives, also add to the diversity of work in this space.

The excitement around these transformer-based models, however, must be tempered with a careful assessment of their advantages and pitfalls. This is our focus in this paper.

In this paper, we take a step back and reflect on the copious amount of work that has been published in this area thus far. We study 519 papers published at 27 top conferences and journals during 2017–2023, which cover 101 different applications. We consider the models used in these papers, which SE applications they target, benchmarks used in evaluation, and other key characteristics of this quickly growing body of work. We then consider the performance of the top models from the literature on the most popular applications and review the corresponding model generalizability and computational efficiency. Throughout, we present each of our findings as either a *promise* or a *peril* to help position SE research that relies on transformer-based models on a firmer footing.

The closest related empirical studies of this rich research space have conducted performance evaluation of different pre-trained models across various applications (Niu et al., 2023; Zeng et al., 2022). Additionally, Zhou, Sha, and Peng (2024) investigated the calibration effectiveness of pre-trained models for code understanding tasks, while Liu, Tantithamthavorn, Liu, and Li (2024) explored the reliability and explainability of language models in program generation. Unlike these studies, our work is more comprehensive. We select applications based on extensive literature from 2017–2023 and we explore more aspects of pre-trained models of code, such as their resource consumption and generalizability. Our study, therefore, provides a more holistic view of the capabilities and limitations of transformer-based models in SE research.

We consider three representative pre-trained transformer-based models for code and four most popular applications. We study model architecture trade-offs, model performance, time consumption, and generalizability. We support our findings with statistical tests. We have made all of our related code and data open-source (Zuo, 2023).

In summary, our work makes the following contributions:

- We perform a comprehensive review of transformer-based research published during 2017–2023 and report on key characteristics. For example, we find that the four most popular applications of transformer-based models in SE are Bug Fixing, Bug Detection, Code Summarization, and Code Search, with 54, 53, 51, and 33 papers respectively. However, we also find that existing work overlooks many applications that developers need.
- We find that CodeBERT is best suited for understanding tasks and demonstrate that both CodeBERT and CodeGPT deliver promising results in certain generation tasks when evaluated with more pertinent metrics. This highlights the power of encoder-only and decoder-only models, which contrasts with prior work. Additionally, we provide guidance for software engineering researchers on how to effectively choose models for specific tasks.
- We further consider model resource consumption and find that CodeBERT is highly efficient for understanding tasks, achieving the highest performance with the lowest resource use. We question CodeT5's efficiency for generation tasks, as its high resource consumption does not guarantee consistently better performance. Therefore, when selecting transformer-based models, researchers and practitioners should carefully consider the performance and time complexity trade-offs for their specific application.
- We fill a gap in our understanding of the generalizability of transformer-based models for SE. We find that models trained on the popular benchmarks and datasets for Code Summarization and Bug Fixing generalize poorly to other datasets. This highlights a need for improvements in dataset quality.

2. Background

The background section is divided into two parts: an introduction to the overall research landscape of Transformer-based models, and a detailed explanation of the specific pipelines and knowledge of the Transformer-based models used in, or relevant to, our study.

2.1. Overview of research in transformer-based methods

In the past seven years, there has been extensive research on Transformer-based pre-trained models. These models are large-scale Transformer architectures trained on vast amounts of unlabeled data using self-supervised learning objectives. The goal of developing such models is to obtain general, transferable knowledge within a specific domain, such as programming languages Chakraborty et al. (2022).

In this section, we provide a brief overview of the foundational architectures of Transformer-based models, the key models that have been proposed, the tasks they address, and the techniques used to enhance their performance.

Model Architectures: The most important components of Transformer architecture are the encoder–decoder structure and attention mechanism, which resides in the Transformer blocks. The encoder aims to extract important information from the input, and outputs the encoded representation. The encoded representation is then taken in as input by the decoder to generate output in an autoregressive manner (Vaswani et al., 2017). Some variants of the Transformer model may contain only an encoder or only a decoder.

A Transformer has multiple layers, which are called Transformer blocks, and they serve as the building blocks for the encoder and decoder. The core component of a Transformer block is the attention mechanism, which is used to process the input to each Transformer block. Through the attention mechanism, a Transformer provides context for different tokens in the input sequence.

Models: Several foundational models have shaped the field of Transformer-based pre-trained models. BERT Kenton and Toutanova (2019) revolutionized tasks like classification and question answering. RoBERTa Liu et al. (2019) refined BERT's pre-training strategies, achieving better performance. GPT models Radford et al. (2019) advanced autoregressive generation, excelling in text generation and few-shot learning. T5 Raffel et al. (2020) unified NLP tasks in a text-to-text framework, simplifying multi-task learning. CodeBERT Feng et al. (2020) and Codex Chen et al. (2021) specialized in code-related tasks, with Codex powering tools like GitHub Copilot. XLNet Yang (2019) improved on BERT by introducing permutation-based language modeling to better capture bidirectional context. ALBERT Lan (2019) reduced the number of parameters to improve efficiency, while DistilBERT Sanh (2019) provided a smaller, faster model by distilling BERT's knowledge. These models represent foundational advancements in Transformer-based models, significantly impacting the field and inspiring ongoing research into expanding their capabilities across a wide range of applications.

Applications: Pre-trained models have demonstrated exceptional performance across a range of tasks in natural language processing (NLP), such as machine translation, question answering, and sentiment analysis (Lan, 2019; Vaswani et al., 2017; Yang, 2019). Building on their success in the NLP domain, these models have also been applied to various tasks in other fields, including software engineering (Bommasani et al., 2021; Wei et al., 2022; Zhao et al., 2023). For instance, in code search, GraphCodeBERT enhances query-matching accuracy by incorporating structural information from code Shi et al. (2023). For code generation, CodeT5 excels in tasks like cross-language code translation, ensuring functional consistency across different programming languages Shi et al. (2023). In code clone detection, pre-trained models have been highly effective at identifying functionally equivalent code fragments with different implementations by learning deep semantic representations Shi et al. (2023). Similarly, in equivalent mutant detection (EMD), traditional approaches struggle with capturing complex semantic nuances, but models like UniXcoder have significantly improved detection accuracy in this challenging task Guo et al. (2022).

Tuning Techniques: Different types of tuning techniques have been employed in the literature to improve the performance of Transformer-based models on specific downstream tasks. Fine-tuning is a widely used technique that tunes a pre-trained model on a labeled dataset to achieve high performance on downstream tasks. Benefiting from the general knowledge learned during the pre-training phase of model training, fine-tuning requires a much smaller data size than training a model from scratch.

Despite the prevalence of fine-tuning, there exist some variants. Zero-shot learning directly applies pre-trained models without tuning and few-shot learning simulates data scarcity by providing limited data examples. Prompt tuning transforms the downstream tasks into a similar format as the pre-training tasks using prompts. Closely related is in-context learning, which capitalizes on the ability of large language models to generate responses based on a small selection of examples included within the prompt. We summarize the characteristics and respective advantages of these different tuning methods in Table 1. Some recent studies focus on reducing the computational costs of fine-tuning. For example, Shi et al. (2023) proposed a layer-freezing technique that fine-tunes only the upper layers of the model, thereby reducing training time and resource consumption while maintaining performance.

2.2. Transformer-based pre-trained models in this study

Following the introduction to the broader research landscape, this section shifts focus to the specific pipelines and knowledge of the Transformer-based models that are used in or directly relevant to our study.

Input: The inputs of Transformers vary across different models. For natural language models like BERT (Kenton & Toutanova, 2019), the input starts with a special token [CLS], and sentences are separated by the special separator token [SEP]. As illustrated in Fig. 1, $w_1, w_2, \ldots, w_i$ represents the first sentence and $w_j, w_{j+1}, \ldots, w_k$ represents the second sentence.

Language models for code (the focus of this paper), such as CodeBERT (Feng et al., 2020), CodeGPT (Lu et al., 2021), CodeT5 (Wang, Wang, Joty, & Hoi, 2021), accept bimodal data instead, *i.e.*, both natural language and code. $w_1, w_2, \ldots, w_n$ represents the natural language, and $c_1, c_2, \ldots, c_m$ represents the corresponding code. The two segments are separated by the [SEP] token, and [EOS] token denotes the end of input.

There exist variants of language models for code which contain more information in their input, *e.g.*, GraphCodeBERT (Guo et al., 2020) additionally includes variables in the input program, denoted by $x_1, x_2, \ldots, x_k$. Additional input combined with corresponding pre-training objectives will enhance models' knowledge regarding a certain aspect, *e.g.*, code structure in the case of GraphCodeBERT.

Input Embedding: Inputs with correct formats are used to generate input embeddings that are fed into the Transformer blocks. An input embedding typically consists of three parts: token embedding, segment embedding, and position embedding. For token embedding, there exists three different options, word embedding, subword embedding, and character embedding. Among the three options, subword embedding is the most frequently used technique in Transformer-based models since it can deal with out-of-vocabulary (OOV) issues. There are two common subword tokenization methods, WordPiece (Schuster & Nakajima, 2012) and Byte Pair Encoding (Shibata et al., 1999).

Segment embedding is used to distinguish which segment of input a specific token belongs to.

Lastly, position embedding encodes the positional information of words. There are different options available. A traditional one is sequential embedding (Chirkova & Troshin, 2021). To capture structure-related positional information, tree positional embedding (Shiv & Quirk, 2019) can be used. Sequential relative positional embedding (Shaw, Uszkoreit, & Vaswani, 2018) and tree relative positional embedding (Kim, Zhao, Tian, & Chandra, 2021) are more effective in sequence-based tasks as they capture the relative position of the input elements.

Model Architecture: There are three different model structures for Transformer-based models: encoder-only model, decoder-only model,

Table 1
Fine-tuning and variations.

Tuning methods	Characteristics	Advantage
Fine-tuning	Require high-quality labeled dataset	Task-specific & Flexible
Zero/Few-shot Learning	Simulates data scarcity	Better generalization ability Chakraborty et al. (2022), Palatucci, Pomerleau, Hinton, and Mitchell (2009)
Prompt Tuning	Augment input with prompts	Fully utilizes pre-trained models (Huang et al., 2022; Wang et al., 2022)
In-context Learning	Include examples in model inputs	No updates to the model weights (Brown et al., 2020)

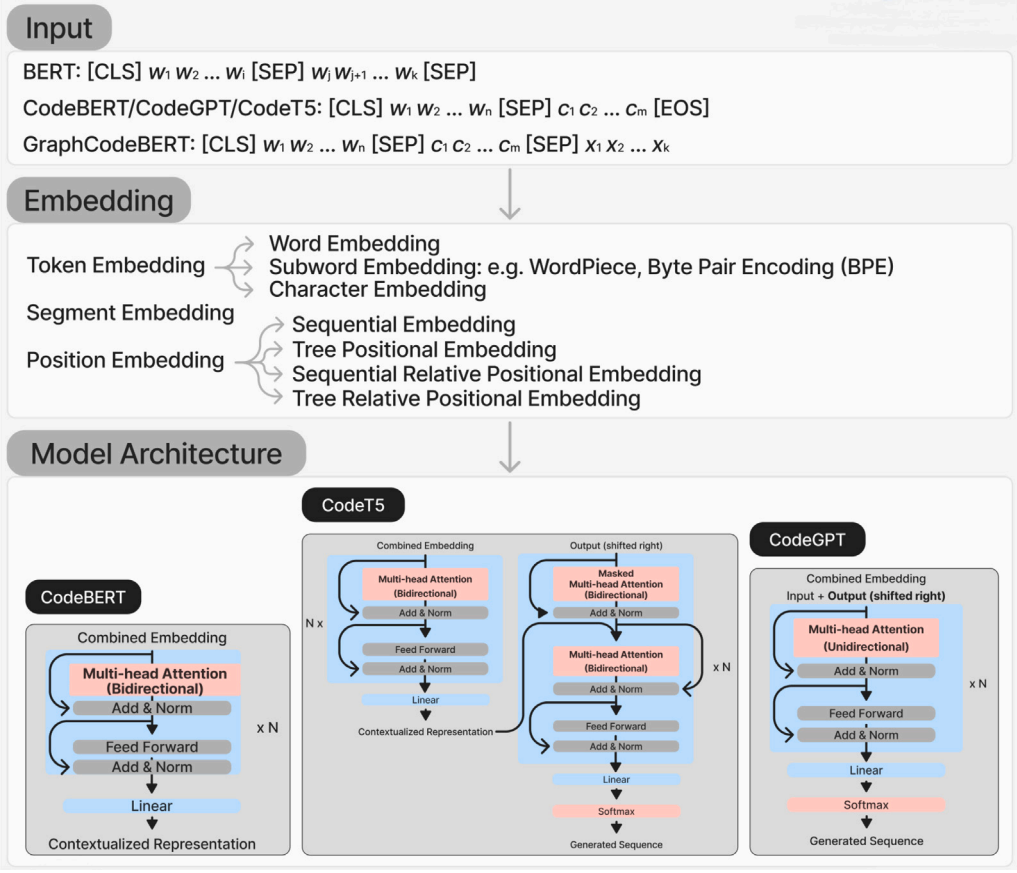


Fig. 1. Pipeline and variations for transformer-based pre-trained models.

and encoder–decoder model. The most representative pre-trained language models with such structures are BERT, GPT, and T5 for natural language, and CodeBERT, CodeGPT, and CodeT5 for programming languages.

BERT (Kenton & Toutanova, 2019), Bidirectional Encoder Representations from Transformers, is an encoder-only model pre-trained on BookCorpus and Wikipedia, with the objective of masked language modeling and next sentence prediction (Kenton & Toutanova, 2019). There are multiple variants of BERT used for SE-related tasks, such as CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020).

GPT (Radford et al., 2019), Generative Pre-trained Transformer, is a decoder-only autoregressive language model pre-trained on BookCorpus, with the generative objective of predicting the next word given some previous words (Radford et al., 2019). GPT variants that are applied to SE-related tasks include GPT-C (Svyatkovskiy, Deng, Fu, & Sundaresan, 2020) and CodeGPT (Lu et al., 2021).

T5 (Raffel et al., 2020) is an encoder–decoder model pre-trained on the C4 (Colossal Clean Crawled Corpus) dataset with a "span corruption" objective (Raffel et al., 2020). T5's code variants include CodeT5 (Wang et al., 2021) and CoditT5 (Zhang, Panthaplackel,et al., 2022).

CodeBERT, CodeGPT, and CodeT5 have the same model architectures as BERT, GPT, and T5 (see Fig. 1). Apart from the encoder and decoder components, the differences between the model architectures exist in the multi-head attention mechanism and input/output. For CodeBERT and CodeT5, the attention mechanism is bidirectional, allowing the model to capture the context from both directions and to better capture long-range dependencies. Whereas the attention mechanism in CodeGPT is unidirectional, which means that the model only attends to the past inputs in the sequence, avoiding potential future information leakage.

The output of CodeBERT is a contextualized representation of input, and can be utilized to perform, *e.g.*, classification tasks. For CodeT5, after the encoder has generated the contextualized representation of an input, the decoder takes it in to combine with the output generated at previous steps to form the input to the decoder. CodeGPT generates output using the input combined with the output generated at previous steps, and the output of CodeT5 and CodeGPT are both probabilities of tokens that can be converted to the corresponding generated sequence.

```
# Calculates the area of a rectangle
def calculate_area(length, width):
    area = length * width
    return area
```

Fig. 2. Original program.

```
# Calculates the area of a rectangle
def calculate_area(length, [MASK]):
    area = length * width
    return area
```

(a) Masked Language Modeling

```
# Calculates the area of a rectangle
def calculate_area(length, width):
    area = length * length
    return area
```

(b) Replaced Token Detection

Fig. 3. Objectives of CodeBERT.

```
# Calculates the area of a rectangle
def calculate_area(length, width):
    area = length * [PRED]
```

Fig. 4. Objectives of CodeGPT.

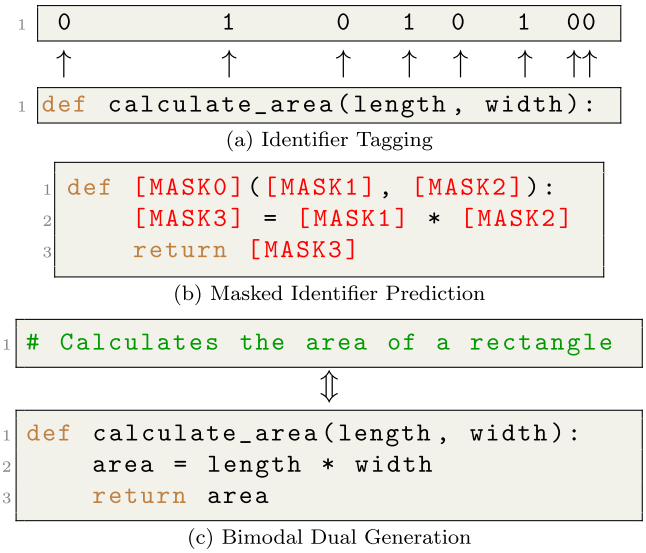
Pre-training Objectives: A pre-trained model for code may have multiple objectives, which constitute a hybrid objective function and contribute to better code understanding (Feng et al., 2020; Wang et al., 2021). The three pre-trained models that we investigate in this paper have different pre-training objectives. However, Transformer-based pre-trained models for code are mostly pre-trained on different subsets of the same dataset, CodeSearchNet (Husain, Wu, Gazit, Allamanis, & Brockschmidt, 2019).

CodeBERT: CodeBERT is pre-trained with the objectives of Masked Language Modeling (MLM) and Replaced Token Detection (RTD) (Feng et al., 2020). MLM aims to predict the masked out token in both NL and PL sections of the program. Fig. 3(a) is an example of MLM. The objective is to predict the original token for [MASK]. RTD aims to determine whether a token is the original one or a replaced one. For example, if the generator mutates the original program (Fig. 2), to Fig. 3(b), the discriminator should recognize that "length" is a replaced token.

CodeGPT: CodeGPT is pre-trained with the objective to predict the next token, given previous context. Fig. 4 is the illustration of the pre-training objective.

CodeT5: CodeT5 has four pre-training objectives. The first one is Masked Span Prediction (MSP). It can be viewed as a variation of MLM, and it allows masking of multiple consecutive tokens. Besides MSP, CodeT5 introduced two additional tasks: Identifier Tagging (IT) and Masked Identifier Prediction (MIP), to enable the model to learn code-specific structural information. IT aims to determine whether a token is an identifier and MIP performs obfuscation on the PL part of the program and aims to predict the masked-out identifiers, as shown in Figs. 5(a) and 5(b). The last pre-training objective of CodeT5 is bimodal dual generation, which aims to perform NL→PL generation and PL→NL generation simultaneously, as illustrated in Fig. 5(c).

The different pre-training objectives, together with different model architectures, enable the models to be suitable for different tasks.



(a) Identifier Tagging

(b) Masked Identifier Prediction

(c) Bimodal Dual Generation

Fig. 5. Objectives of CodeT5.

3. Methodology

The objective of this paper is twofold: first, to provide a comprehensive review of transformer-based techniques for tackling SE problems, and second, to design and address research questions that explore both the promises and potential pitfalls of these techniques. Through a combination of literature review and empirical study, we aim to offer researchers and practitioners a clearer understanding of the capabilities and limitations of transformer-based models. In the following section, we introduce the methodology for the literature review, as well as the research questions designed to guide the empirical study.

3.1. Literature review

Keywords for Literature Review. To generate keywords for our comprehensive literature review, we first searched for the top four software engineering conferences from 2019-2023: (1) ESEC/FSE (2) ICSE (3) ASE (4) ISSTA.[2] There are multiple types of pre-trained models mentioned in the papers, including Vanilla Transformer, BERT, GPT, T5, and their variants. We recorded all the models mentioned in the papers, as well as their popular variants, resulting in a comprehensive list of 17 keywords, including Transformer, BERT, GPT, T5, CodeBERT, CodeBERTa, GraphCodeBERT, RoBERTa, CuBERT, C-BERT, BERTOverflow, GPT-C, CodeGPT, PLBART, BART, IntelliCode, and CodeT5. We believe these keywords are sufficient in supporting us to identify Transformer-based papers in SE research, which is the focus of this study.

Identify Related Literature. We identify 27 relevant SE conferences and journals with core ranking A* or A, as shown in Table 2. Using the keywords mentioned above, we conducted an extensive search for Transformer-based papers in those 27 conferences and journals published between 2017 and 2023.

We locate the keywords to confirm that the papers are relevant to applying Transformer-based pre-trained models to the SE domain, and exclude papers in which the keywords appear by coincidence: *e.g.*, a mathematical transformer. Note that transformer-related papers are unlikely to be missed out using the keyword list, as papers usually mention or reference the models in the list, even if the paper uses

[2] Corresponding full names are provided in the GitHub link (Zuo, 2023).

Table 2
Statistics for papers published in top-tier venues.

Venues	2019	2020	2021	2022	2023	Sum
ESEC/FSE	0	6	14	16	24	60
ICSE	0	4	15	21	48	88
ASE	1	4	13	15	52	85
ISSTA	0	3	2	7	9	21
TSE	0	1	9	12	14	36
TOSEM	0	0	1	8	14	23
ESE	0	0	2	6	6	14
PLDI	0	0	7	0	1	8
OOPSLA	0	1	0	0	0	1
ISSRE	0	2	7	0	8	17
ESEM	1	1	0	0	2	4
SANER	0	0	4	8	10	22
EASE	0	0	1	1	1	3
IST	0	0	4	13	12	29
JSS	0	1	1	8	10	20
ICPC	0	0	3	7	9	19
RE	0	4	13	2	9	28
CAiSE	0	1	2	1	1	5
ICSME	1	4	11	0	3	19
ICST	0	0	1	1	1	3
MSR	0	1	4	4	3	12
ICSA	0	0	0	1	0	1
ECSA	0	1	0	0	0	1
Sum	3	34	114	131	237	519

Note that, POPL, SEAMS, TOPLAS, and FM had 0 papers for all years.

variants of them. Finally, we identified 519 relevant papers and 101 different applications.

To extract information from the selected papers, we primarily focused on the pipelines used by the authors to incorporate transformers into their research. This involved studying the applications, datasets, pre-processing, input, architecture, training, and output of the transformers used. We also searched through all 519 papers using the 101 summarized application names and recorded the number of papers for each application.

3.2. Research questions

RQ1. Literature, Popular Applications, and Developers' Needs: *What are the characteristics of papers utilizing Transformer models, such as the yearly publication trends and the extent to which different applications have been explored?*

To summarize the publication characteristics, we conducted a thorough review of 519 papers mentioned in Section 3. Based on these papers, we summarized all the applications related to transformer-based models. We have examined why certain applications are more popular than others and whether they are relevant to the needs of SE developers. Through this analysis, we aim to shed light on the most important applications for SE research, and help the community focus on the areas that are most relevant to developers' needs.

RQ2. Applications' Performance: *How do the three base models with varying architectures perform across the top four popular applications?*

To investigate the performance of various models for different tasks, we conducted a comparative analysis of three representative models: CodeBERT, CodeGPT, and CodeT5. Specifically, we evaluated their effectiveness on the top four most popular applications: Bug Fixing, Bug Detection, Code Summarization, and Code Search. By fine-tuning these models on benchmark datasets for each application, we found that previous claims about model performance and architectures have been misleading without the use of up-to-date metrics for specific tasks. Our findings are reinforced by statistical testing, enhancing the validity of our conclusions regarding model suitability. We also identified the most commonly-used models for each task by reviewing the literature, and checked for consistency with our concluded best-performing models.

RQ3. Resource Consumption: *What is the resource consumption of inference for each base model and application?*

We answer this question by comparing and analyzing the average inference time and memory usage of various models on all 4 applications across different datasets.

RQ4. Generalization: *How well do the base models trained on commonly-used benchmark datasets for each application generalize to frequent datasets, and conversely, how well do models trained on frequent datasets perform on the benchmark datasets?*

To evaluate the generalization ability of the models across datasets within the same domain, we conducted a search for the datasets used in all papers related to the top four applications. For these applications, we adopt the datasets from CodeXGLUE (Lu et al., 2021) as the benchmark datasets, which are widely used in the study of Transformer-based techniques for SE applications. The frequent dataset, defined as the dataset with the highest *frequency* of use other than the benchmark dataset, implies high quality and reliability due to its repeated selection by the research community. We follow the common train test split used in the literature. Our review revealed that the benchmark datasets were most commonly used across all applications, except for Bug Fixing, where datasets such as Defects4J (Just, Jalali, & Ernst, 2014) were used more frequently than the benchmark datasets (BFPsmall & BFPmedium (Tufano et al., 2019)). However, for Code Search and Bug Detection, only a few researchers used non-benchmark datasets, which makes it less meaningful to investigate generalization on those datasets. Therefore, our focus was primarily on the benchmark and frequent datasets for Code Summarization and Bug Fixing. We then evaluated the models trained on the benchmark dataset on the test set of the frequent dataset and vice versa to examine their generalization capabilities. Statistical testing is performed to safeguard our conclusions.

4. Experimental setup

In this section, we outline the experimental setup, which includes the pre-trained models used, the datasets for the four most popular SE tasks, and the evaluation metrics employed to assess model performance.

4.1. Pre-trained models

We choose CodeBERT, CodeGPT, and CodeT5 as the pre-trained models with the consideration of representativeness in model architectures and their wide presence in the literature.

CodeBERT is an encoder-only model which has the same model architecture as RoBERTa (Liu et al., 2019). It is pre-trained on CodeSearchNet and is capable of processing both source code and natural language text. The model we use is CodeBERT-base, which has 125M parameters.[3]

CodeGPT is a decoder-only model with the same model architecture as GPT-2 (Radford et al., 2019). We use CodeGPT-small-java-adaptedGPT2[4] with 124M parameters, which is pre-trained on the Java corpora from the CodeSearchNet dataset and uses the GPT-2 model as the starting point.

CodeT5 is a variant of T5 (Raffel et al., 2020), and achieves state-of-the-art performance for many code intelligence tasks. It views all tasks through a sequence-to-sequence paradigm. CodeT5 is pre-trained on CodeSearchNet and an additional C/C# dataset (Wang et al., 2021). We use CodeT5-base[5] that has 220M parameters.

4.2. Datasets

Table 3 shows the benchmark datasets we use in the experiments for the four applications, following the widely used CodeXGLUE benchmarks in Transformer-based techniques.

[3] https://huggingface.co/microsoft/codebert-base
[4] https://huggingface.co/microsoft/CodeGPT-small-java-adaptedGPT2
[5] https://huggingface.co/Salesforce/codet5-base

Table 3
Datasets.

Task	Fine-tuning dataset	Train/Valid/Test
Bug fixing	BFPsmall (Tufano et al., 2019)	46,680/5,835/5,835
	BFPmedium (Tufano et al., 2019)	52,364/6,545/6,545
Bug detection	Zhou et al. Zhou, Liu, Siow, Du, and Liu (2019)	21,854/2,732/2,732
Code summarization	Java Subset in CodeSearchNet (Husain et al., 2019)	164,923/5,183/10,955
Code search	Python Subset in CodeSearchNet (Husain et al., 2019)	251,820/9,604/19,210

Bug Fixing: This dataset is provided by Tufano et al. (2019). It contains method-level pairs of the buggy and fixed code from thousands of GitHub Java repositories. The pairs are called bug-fix pairs, BFPs in short. Based on the code length, Tufano et al. provided two datasets: BFPsmall, which has a code length below 50; and BFPmedium, which has a length between 50 and 100.

Bug Detection: This dataset is provided by Zhou et al. (2019). It contains 27k+ C code snippets from two open-source projects, FFmpeg and QEMU, of which 45% are defective.

Code Summarization: CodeSearchNet is a dataset which consists of $< code, comment >$ pairs from open source projects (Husain et al., 2019). Code refers to the code snippet for a method, and comment refers to the description of the code, for example in Javadoc format.

Code Search: This dataset is the Python version of the CodeSearchNet (Husain et al., 2019) dataset.

4.3. Evaluation metrics

A variety of evaluation metrics are applied to assess model performance across the four applications, and statistical tests are conducted to validate the findings.

Bug Fixing: We use BLEU (Bilingual Evaluation Understudy) (Papineni, Roukos, Ward, & Zhu, 2002), Accuracy, and CodeBLEU (Ren et al., 2020) to measure the quality of the repaired code, where accuracy considers only exact matches, and CodeBLEU additionally considers code structure, etc., and involves n-gram, weighted n-gram, AST, and data-flow matches:

$$CodeBLEU = \alpha * BLEU + \beta * BLEU_{weight} + \gamma * Match_{ast} + \delta * Match_{df} \tag{1}$$

BLEU is defined below under Code Summarization. The definition of accuracy adopted in this paper is the same as below except that y_i and $\hat{y}_i$ refer to the buggy and fixed code instead.

Bug Detection: We use Accuracy as the evaluation metric for bug detection, following the work of CodeT5 (Wang et al., 2021). Accuracy helps to measure the ability of the model to distinguish buggy code from normal code:

$$Accuracy = \frac{\sum_{i=1}^{|D|} 1(y_i == \hat{y}_i)}{|D|} \tag{2}$$

where $|D|$ refers to the dataset size, y_i and $\hat{y}_i$ refer to the ground truth label and predicted label, respectively. The function in the numerator is *1* if the two labels are equal, and 0 otherwise.

Code Summarization: We use BLEU (Papineni et al., 2002), METEOR (Banerjee & Lavie, 2005), and ROUGE-L (Lin, 2004) to measure the quality of the summary generated in the Code Summarization task. Each of them considers different aspects.

BLEU is based on the n-gram precision between the generated summary and the Papineni et al. (2002):

$$BLEU = BP * exp(\sum_{n=1}^{N} w_n log p_n) \tag{3}$$

where BP penalizes short summary. p_n refers to n-gram precision and w_n is the weight.

METEOR focuses on the harmonic mean of unigram precision and recall (Banerjee & Lavie, 2005):

$$METEOR = (1 - P) * F_{mean} \tag{4}$$

where P is the penalty for difference between the word order in the summary generated and the reference, and more weight is put on recall when calculating the harmonic mean F_{mean}. METEOR allows exact, stem, and synonym matches.

ROUGE-L is based on the longest common sub-sequence (LCS) between the generated summary and the Lin (2004):

$$ROUGE - L = \frac{(1 + \beta^2) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \tag{5}$$

where R_{lcs} measures the proportion of the LCS length relative to the length of the reference and P_{lcs} measures that to the length of the generated summary, and ROUGE-L calculates the harmonic mean of them.

Code Search: We use Mean Reciprocal Rank (MRR) to measure the ability of the model to retrieve relevant code given a natural language query. It calculates the multiplicative inverse of the rank of the correctly retrieved code snippet and is defined as below (Hu et al., 2022):

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{6}$$

where $rank_i$ refers to the rank of the first correctly retrieved code snippet and $|Q|$ represents the number of queries.

Statistical Testing: To evaluate the significance of the differences in model performance across tasks, we employ the Wilcoxon signed-rank test, a non-parametric method suitable for comparing paired data without assuming a normal distribution. This approach ensures a robust analysis of performance variations across models. For all tests, we set the significance level at $\alpha = 0.05$.

4.4. Configurations

We conducted all experiments on an NVIDIA RTX A4000 GPU with CUDA version 11.6 and 16 GB of VRAM. The implementation of all models, as described in Section 4.1, was carried out using the PyTorch framework. Detailed hyperparameter settings, including learning rates, batch sizes, and other parameters for fine-tuning across various tasks, are available in the GitHub repository (Zuo, 2023).

5. Results

In this section, we aim to answer the research questions mentioned in Section 3.2 by discussing the promises and perils of using transformer-based models for SE research. Our findings through the literature review help answer RQ1, and we conduct our own empirical study to answer RQ2-4.

5.1. RQ1. Literature, popular applications, and developers' needs

On Papers Published.

Table 2 presents the number of papers on Transformer-based techniques published in top-tier SE conferences or journals during 2019–2023. Through our search described in Section 3, we found a total of

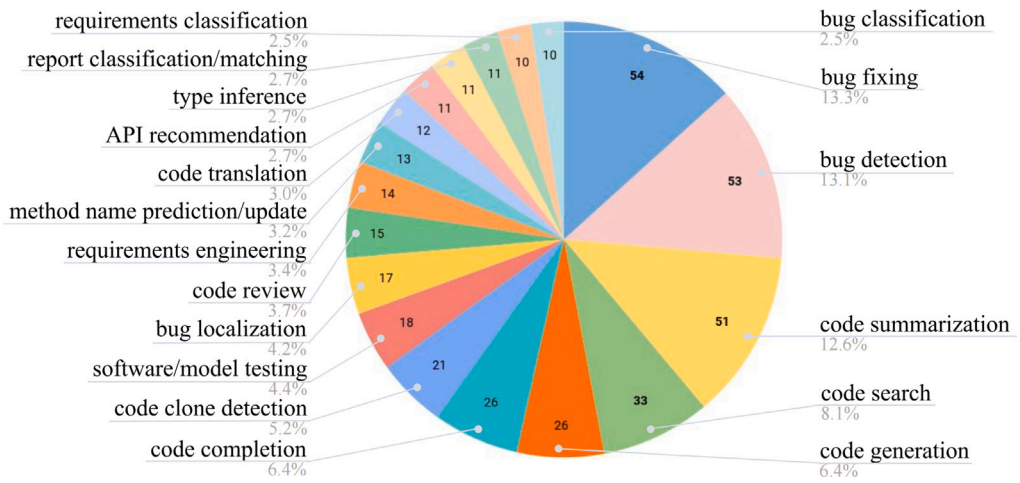


Fig. 6. Frequency of Applications. We only list applications with frequency ≥ 10.

3 papers published in 2019, 34 papers in 2020, 114 papers in 2021, 131 papers in 2022, and 237 papers in 2023. The increasing trend in the numbers indicates a rapidly growing interest in using Transformer-based techniques to solve SE-related problems, demonstrating their promising future with more application domains and outstanding performance. The emergence of large language models with unprecedented scale and effectiveness, such as ChatGPT (OpenAI, 2022), has led to a remarkable escalation in relevant academic publications in 2023.

Although the Transformer architecture was proposed in 2017 (Vaswani et al., 2017), it was rarely used by SE researchers until 2020. Besides, even though transformer models are more widely studied over time, there is still room for improvement. For example, some SE studies (Cheng, Hu, Wei, & Mo, 2022; Li et al., 2022; Zhou et al., 2022; Zhu, Sha, & Niu, 2022) only base their implementations on or compare their models to the Vanilla Transformer, introduced in 2017 by Vaswani et al. instead of using or including additional and more advanced models such as T5, nearly three years after the proposal of these new models. This shows that the SE research community needs to explore breakthrough techniques from other domains and investigate their application to SE-related tasks. Additionally, researchers should stay informed and attentive to the latest methodologies, such as T5, CodeT5, and so on.

> ***Peril 1:*** *SE researchers using transformer-based models should pay attention to recent techniques in domains like NLP and be more attentive to the latest methodologies.*

On Applications.

We examined the frequency of the 101 different applications explored by papers counted in Table 2. Parts of the application statistics are shown in Fig. 6.[6]

Out of the 519 papers, we found that bug fixing appeared 54 times, bug detection appeared 53 times, code summarization appeared 51 times, and code search appeared 33 times, making them the top four popular applications. They are the only four applications with a count larger than 30, and due to their representation of the community's research focus and diversity in the task nature, we chose to conduct experiments on these four applications for our research questions.

Bug Fixing (Tufano et al., 2019): This task aims to fix buggy programs automatically.

Bug Detection (Zhou et al., 2019): This task performs binary classification - determining if a program is buggy or not.

[6] Full list of applications can be found in GitHub (Zuo, 2023).

Table 4
Top-4 popular applications.

Task name	Nature	Category
Bug fixing	Code → Code	Generation
Bug detection	Code → Class (0/1)	Understanding
Code summarization	Code → NL	Generation
Code search	NL → Code	Understanding

Code Summarization (Iyer, Konstas, Cheung, & Zettlemoyer, 2016): This task generates a natural language summary for a code snippet to aid programmers' understanding.

Code Search (Husain et al., 2019): This task searches for relevant code snippets given their natural language descriptions.

The four tasks belong to different categories (Table 4), making them representative of SE tasks researchers study and appropriate for our empirical study. Furthermore, due to their specific task implementation, bug detection and code search focus more on understanding, while code summarization and bug fixing are more generation-oriented. Thus, optimal performance on different tasks is achieved by different types of models, as discussed in RQ2.

> ***Promise 1:*** *The four topmost targeted tasks, in descending order of popularity, are Bug Fixing, Bug Detection, Code Summarization, and Code Search. The first and third are generation tasks and the others are understanding tasks. Transformer-based techniques demonstrate a promising future with increasing attention from the SE research community, more diverse application domains, and outstanding performance.*

Apart from experimenting with the most representative and popular applications, we have also identified the least-studied applications. Some applications are too specific, *e.g.*, taint propagation detection, leading to fewer researchers studying them. In this context, "too specific" refers to topics that are highly specialized or narrow in scope, focusing on particular issues that may not have broad applicability across different fields, thereby attracting less research activity. Additionally, some applications are derivatives of more popular ones, *e.g.*, algorithm classification can be considered a specialized offshoot of code summarization, as both seek to understand intent. Another potential reason other applications are less well-studied is that no commonly used benchmarks exist. For example, developers' dialog analysis has been studied in three distinct works (Pan et al., 2021; Shi et al., 2021; Silva, Galster, & Gilson, 2022), but each study used a different dataset, complicating efforts to build upon previous research.

Table 5
Performance of three representative models on top-4 applications.

		CodeBERT	CodeGPT	CodeT5
Bug fixing (BLEU-4/ Accuracy/ CodeBLEU)	Small (S)	74.41/17.58/**79.04**	72.84/19.78/77.31	**78.04**/**21.54**/77.70
	Medium (M)	88.60/10.10/**88.40**	86.56/12.07/86.09	**88.86**/**13.60**/86.42
Bug detection (Accuracy)		**63.54**	63.25	62.99
Code summarization (BLEU-4/METEOR/ ROUGE-L)		18.48/13.07/35.02	14.63/**16.36**/34.18	**20.25**/15.07/**38.29**
Code search (MRR)		**28.45**	23.93	27.41

Peril 2: *The least-studied applications are typically too specific, related to the more popular applications, or lack high-quality benchmarks.*

Developers' Needs. Existing studies have emphasized the importance of communication and collaboration in the software development process (Aniche et al., 2018; Gonçalves, de Souza, & González, 2011). However, the potential of transformer-based techniques, such as ChatGPT, to facilitate these processes is yet to be extensively explored. Additionally, the time developers spend seeking information is a critical factor in their daily work (Gonçalves et al., 2011; Ko, DeLine, & Venolia, 2007). While code search has been identified as one of the top-4 most popular applications, there are still many detailed issues that developers frequently encounter that remain unexplored. For example, how to efficiently transfer deprecated features, functions, or methods to new ones (Sawant, Aniche, van Deursen, & Bacchelli, 2018), or how to solve module dependencies (Bogart, Kästner, & Herbsleb, 2015). Furthermore, despite code summarization being the top-1 most popular application, it is rated much lower than commit conflict resolution by developers (Treude, Figueira Filho, & Kulesza, 2015).

In addition to these needs, there is a significant gap in tools and practices for Efficient Debugging, Security Practices Integration, and Accessible Documentation. Efficient debugging tools are essential as they directly impact the developers' productivity and software quality, yet advancements in this area have been limited (Ceccato, Marchetto, Mariani, Nguyen, & Tonella, 2015). Similarly, integrating security practices throughout the development lifecycle remains underexplored, despite the increasing importance of software security (Valdés-Rodríguez, Hochstetter-Diez, Díaz-Arancibia, & Cadena-Martínez, 2023). Accessible and comprehensive documentation is also critical, yet often neglected, impacting developers' ability to understand and use software effectively (Dagenais & Robillard, 2010). These underdeveloped areas highlight the discrepancy between developers' actual needs and the current research and development focus, suggesting that more attention should be directed towards these critical aspects to better support the software development process.

Peril 3: *Current research neglects applications actually demanded by developers. SE researchers should prioritize applications that are most relevant to developers' needs, such as improving communication and collaboration, efficiently transferring deprecated features/functions/methods to new ones, solving module dependencies, and dealing with commit conflicts.*

5.2. RQ2. Applications' performance

Suitability for Different Tasks. The performance of three representative models - CodeBERT, CodeGPT, and CodeT5, on the four different tasks - Bug Fixing, Bug Detection, Code Summarization, and Code Search is summarized in Table 5. Our implementation demonstrates comparable performance to those reported in the literature (Chakraborty et al., 2022; Lu et al., 2021; Zeng et al., 2022), according to the metrics used in existing papers, therefore validating our implementation's correctness. Moreover, to provide a thorough evaluation of model capabilities, we add more up-to-date metrics to our experiments. We can observe from Table 5 that the highest performance on each of the four applications is achieved by different models.

For Bug Detection and Code Search, CodeBERT achieves the best performance, whereas, the best performance for Bug Fixing and Code Summarization is achieved by different models under different metrics.

Bug Detection is an understanding task, as once the model understands the code, it will be able to predict whether the code is defective or not. Our implementation of Code Search is to compare the vector representations of code candidates and the natural language query, which makes the task fall into the category of understanding as well. With $\alpha = 0.05$, CodeBERT significantly outperforms CodeGPT and CodeT5 in Code Search. Although the advantage in Bug Detection is not statistically significant, CodeBERT still remains the best choice with the lowest resource consumption, as discussed in RQ4. Thus, consistent with the literature (Zeng et al., 2022), we conclude that an encoder-only model like CodeBERT is suitable for understanding tasks.

For Bug Fixing, CodeT5 achieves the highest BLEU-4 and Accuracy, whereas CodeBERT achieves the highest CodeBLEU. CodeBLEU is a metric that considers aspects such as code structure and is arguably more important than BLEU-4 and Accuracy, which focus on term matching in the Bug Fixing task. Moreover, CodeT5's better performance on BLEU-4 on the BFPmedium dataset is insignificant, whereas CodeBERT significantly outperforms CodeT5 in terms of CodeBLEU. This indicates that encoder-only models, such as CodeBERT, can not only compete with but also outperform encoder–decoder models in specific program generation tasks when evaluated under more relevant, task-specific metrics, while also achieving the highest performance in code-understanding tasks. Previous studies have often favored encoder–decoder architectures for their supposed versatility in handling both understanding and generation tasks (Ahmad, Chakraborty, Ray, & Chang, 2021; Lewis et al., 2019; Raffel et al., 2020), however, our findings challenge this assumption. Additionally, the superior performance of CodeT5 in BLEU-4 and Accuracy may not solely be attributed to its architecture; rather, it could be influenced by its pre-training objectives, which are specifically tailored to code-related tasks, potentially enhancing its term matching capabilities. Future research could explore whether encoder-only models might further excel in code generation tasks with the introduction of more sophisticated, code-focused generative pre-training objectives.

For Code Summarization, CodeT5 achieves the highest scores in BLEU-4 and ROUGE-L metrics. However, CodeGPT notably excels in METEOR, a metric highly correlated with human judgment (Banerjee & Lavie, 2005) due to its consideration of synonyms during evaluation. Previous studies (Roy, Fakhoury, & Arnaoudova, 2021) have also demonstrated that METEOR is a more reliable metric than commonly used metrics such as BLEU-4, and it has become the state-of-the-art evaluation metric for Code Summarization tasks. This finding underscores the effectiveness of decoder-only models, like CodeGPT, in managing specific generative tasks. It offers a more nuanced understanding of the capabilities of these models using a more reliable metric than BLEU-4, which has been predominantly used in prior research. Our finding extends the claim (Zeng et al., 2022) that decoder-only models (such as CodeGPT) fail to enable optimal performance on any task. This result not only broadens the scope of evaluation for decoder-only models but also highlights the importance of using diverse metrics to fully appreciate the capabilities of different models.

Table 6
Most frequently used model vs. Best performing model.

Task name	Most frequent model	Best performing model[a]
Bug fixing	CodeBERT (26/54)	CodeBERT/CodeT5
Bug detection	CodeBERT (32/53)	CodeBERT
Code summarization	Vanilla Transformer (26/51)	CodeGPT/CodeT5
Code search	CodeBERT (20/33)	CodeBERT

[a] Best Performing Model refers to the model with the highest performance in Table 5.

Most Frequently Used Pre-trained Model for Each Task. We extracted the information from every paper with any one of the four applications. Table 6 is the summary of the most frequently used pre-trained model, as well as the best-performing model for each task.

We can see that the most frequently used model is also the best-performing model for Bug Detection and Code Search, and partially for Bug Fixing. However, this is not the case for Code Summarization.

For Bug Fixing, it is worth noting that the literature commonly believes that the encoder–decoder models are more suitable for code generation tasks (Ahmad et al., 2021; Lewis et al., 2019; Raffel et al., 2020), yet many papers have opted for encoder-only models (CodeBERT) without proper justification or comparison to encoder–decoder models.

For Code Summarization, the best-performing models are CodeT5 and CodeGPT, whereas the most frequently used model is the Vanilla Transformer. CodeT5 is a more advanced pre-trained model with an encoder–decoder architecture, just like Vanilla Transformer. CodeGPT is a decoder-only model with a different model architecture. Previous studies (Lu et al., 2021; Wang et al., 2021) collectively demonstrate that CodeT5 can outperform the Vanilla Transformer significantly in this task, and our result shows that CodeGPT has competitive performance over CodeT5 under state-of-the-art evaluation metric. Thus, the community should watch for ML and SE advancements and integrate advanced models to achieve optimal results, instead of using the earliest or maybe the most well-known models.

Peril 4: *Previous claims regarding the optimality of the encoder–decoder architecture and the low performance of decoder-only models do not hold. The SE research community should put more care into selecting the most suitable model for the task; for example, the most frequently used model for Code Summarization is Vanilla Transformer (26 out of 51 papers).*

5.3. RQ3. Resource consumption

In this research question, we investigate the resource consumption of different models across tasks and datasets. Resource consumption becomes important if a model is deployed and concurrently used by many users. We focus on resource consumption during the inference phase for this reason. Table 7 presents the average inference time and memory consumption for the three models across tasks and datasets.

We conclude that CodeBERT is highly efficient for understanding tasks like Bug Detection and Code Search, achieving the highest performance while consuming the least resources. Additionally, significantly more resources are required for CodeBERT to perform generation tasks than understanding tasks.

For generation tasks like Bug Fixing and Code Summarization, while CodeT5 demonstrates superior performance in certain metrics, it also exhibits an increase in resource consumption attributable to the model's complexity. This raises questions regarding the efficiency of CodeT5 in generation tasks, especially given its higher resource consumption and subpar performance observed in some experiments employing more targeted and contemporary metrics, such as CodeBLEU and METEOR. Additionally, CodeT5 consistently utilizes more memory compared to CodeBERT and CodeGPT across all tasks, a consequence of its larger parameter size, standing at 220M.

Table 7
Inference time and memory consumption.

Time (seconds)/Memory (GB)		CodeBERT	CodeGPT	CodeT5
Bug fixing	Benchmark (S)[a]	0.55/0.71	0.55/0.51	0.87/0.89
	Benchmark (M)[b]	1.55/0.71	1.07/0.51	1.70/0.89
	Frequent	0.78/0.71	0.68/0.51	1.00/0.89
Bug detection	Benchmark	0.13/0.51	0.14/0.52	0.17/0.90
Code summarization	Benchmark	0.27/0.72	0.34/0.52	0.26/0.90
	Frequent	0.20/0.72	0.19/0.52	0.25/0.90
	Mix	0.22/0.72	0.26/0.52	0.26/0.90
Code search	Benchmark	0.14/0.51	0.14/0.52	0.15/0.90

[a] Stand for BFPsmall.
[b] Stand for BFPmedium.

Besides, with the contrast between the time consumption for Benchmark (S) and Benchmark (M) for Bug Fixing, which differ in the length of code, we can clearly see that the time complexity of models for a certain task increases when the input complexity increases.

Promise 2: *CodeBERT is the most efficient model for code understanding tasks, achieving the highest performance with the least resources.*

Peril 5: *CodeT5 is not efficient for generation tasks: this complex model requires high resource consumption, but does not guarantee a consistently better performance.*

5.4. RQ4. Generalization

In the last research question, we explore how well the base models trained on commonly-used benchmark datasets for each application generalize to frequent datasets, and conversely, how well models trained on frequent datasets perform on the benchmark datasets. Both the benchmark and frequent datasets are similar in nature and utilize the same programming language, suggesting that models should theoretically generalize well to both. Table 8 presents the benchmark and frequent datasets used in our study. For Bug Fixing, the frequent dataset was used 18 times (compared to 10 times for the benchmark dataset), while for Code Summarization, the frequent dataset was used 6 times (compared to 22 times for the benchmark dataset). To ensure the validity of the comparison, we pre-processed the frequent datasets to be in the same format as benchmark datasets. For example, patches in Defects4J were processed into bug-fix pairs.

To assess the generalization ability of the models trained on the benchmark and frequent datasets, we conducted experiments where we trained each model on the benchmark dataset and then evaluated it on the test set of the frequent dataset. This enabled us to evaluate whether the knowledge that the models learned from the benchmark dataset is transferable to other datasets or if it is only useful for the benchmark dataset. We also evaluated the performance of the model trained on the frequent dataset on the test set of the benchmark dataset and compared it to the performance of the model trained on the benchmark dataset itself.

Bug Fixing. Table 9 presents the results of evaluating models trained on Benchmark/Frequent datasets on the Frequent/Benchmark test sets for Bug Fixing. Through statistical testing, we found that the frequent model (trained on the frequent dataset) significantly outperforms the benchmark model (trained on the benchmark dataset) when tested on the frequent dataset. Similarly, the benchmark model's performance on the benchmark dataset is significantly higher than the frequent model's

Table 8
Frequency of datasets.

Task name	Benchmark dataset	Frequent dataset
Bug fixing	BFPmedium (Tufano et al., 2019) (10)	Defects4J (Just et al., 2014), etc.[a] (18)
Code summarization	CodeSearchNet (Husain et al., 2019) (22)	LeClair et al. LeClair and McMillan (2019) (11)

[a] The frequent datasets for Bug Fixing include Defects4J Just et al. (2014), Bugs.jar Saha, Lyu, Lam, Yoshida, and Prasad (2018), Bears Madeiral, Urli, Maia, and Monperrus (2019), QuixBugs Lin, Koppel, Chen, and Solar-Lezama (2017), and ManySStuBs4J Karampatsis and Sutton (2020).
Note: The numbers in parentheses indicate the frequency of dataset usage. Code Search and Bug Detection (excluded) are less meaningful as only a few researchers used non-benchmark datasets.

Table 9
Generalization performance of models on bug fixing (BLEU-4/Accuracy/CodeBLEU).

Bug Fixing	CodeBERT	CodeGPT	CodeT5
B model on B	88.60/10.10/88.40	86.56/12.07/86.09	88.86/13.60/86.42
F model on B	88.04/0.05/81.45	87.41/0.08/84.91	67.10/0.00/76.35
F model on F	97.61/92.87/92.73	97.16/92.20/92.24	90.04/90.37/92.21
B model on F	13.29/0.00/18.96	14.55/0.00/16.99	8.16/0.00/32.84

Note: the zeroes in the table indicate that there are no exact matches, however, partial matches exist and are reflected by the BLEU-4/CodeBLEU scores.

performance on the benchmark dataset, except for CodeGPT on BLEU-4, where the frequent model is able to generalize onto the benchmark dataset and significantly outperform the benchmark model.

These findings indicate that while there is potential for models to generalize across datasets of the same nature, as evidenced by CodeGPT's performance, both the benchmark and frequent datasets used in Bug Fixing tasks require enhancements to better support model generalization across different datasets.

Code Summarization. Table 10 presents the performance of various models on different datasets and settings. Note that "B" refers to the benchmark dataset, "F" refers to the frequent dataset, "Mix-bench" consists of the benchmark dataset combined with randomly sampled data from the frequent dataset (the sampled dataset has roughly equal size as the benchmark dataset), and "Mix-Frequent" consists of the frequent dataset combined with equal size of data from oversampling benchmark dataset. "B model on B" refers to the performance of the model trained on the benchmark dataset evaluated on the corresponding testing set. "F model on B" refers to the performance of the model trained on the frequent dataset evaluated on the testing set of the benchmark dataset, etc.

When evaluating the models trained on either the benchmark or the frequent dataset on the test set of the other dataset, we found that they both do not generalize well (refer to columns "B model on B", "F model on B", "F model on F", and "B model on F" in Table 10). For instance, CodeBERT, when trained on the benchmark dataset and tested on the frequent dataset, achieved a BLEU-4 score of 17.23—significantly lower than its score of 31.48 when trained and tested on the frequent dataset itself. Across all models and evaluation metrics, B model on B is able to significantly outperform F model on B, and F model on F significantly outperforms B model on F. Thus, both the benchmark and frequent datasets fail to enable models to generalize onto other datasets, indicating sub-optimal dataset quality.

Therefore, we look into improving dataset quality, subsequently enhancing model capabilities, by including more diverse knowledge, with the simplest implementation of combining data from both datasets. While our experiment results have shown promising signs, we recognize the opportunity for refining our experimental approach and incorporating dataset selection methods in future research.

The performance of the models trained on the Mix-bench dataset and evaluated on the benchmark dataset is shown in the "Mix-bench model on B" column of Table 10. Comparing with "B model on B", we found that the performance drop is only significant for CodeGPT on ROUGE-L and CodeT5 on METEOR. The performance gain by including frequent data is significant for CodeBERT across all metrics and CodeT5 on BLEU-4. When comparing "Mix-frequent model on F" to "F model on F", we found significant improvements in performance for CodeBERT across all metrics, and CodeGPT on BLEU-4 and ROUGE-L. The only drop in performance is on CodeT5's METEOR.

These findings suggest that incorporating more diverse data can potentially enhance model performance and generalization capabilities, despite the introduction of potential noise. We recommend future research to explore dataset pruning or selection techniques to further refine dataset quality and thus improve model generalization.

> ***Peril 6:*** *For Code Summarization and Bug Fixing, both benchmarks and frequent datasets largely fail to provide effective model generalization to other dataset. However, there is potential for generalization across similar datasets. Therefore, the research community should prioritize improving dataset quality to enhance generalization capabilities.*

> ***Promise 3:*** *Integrating data from additional sources enhances model performance and generalization, despite the potential introduction of noise. We recommend investigating dataset pruning and selection techniques to further refine data quality and improve model efficacy.*

6. Discussion and threats to validity

6.1. Discussion on additional statistics

In this section, we explore the statistical data collected in 2024 and its implications for future trends. Survey results show that the number of research papers published in 2024 has reached 301, a significant increase compared to 2023. This growth is evident despite the fact that not all conferences and journals have released their proceedings, indicating a notable rise in research activity related to Transformer-based models. The surge in publications likely reflects the strong performance of Transformer-based models in software engineering tasks, signaling a promising future for their continued application and development.

Applications. Although the total number of applications in 2024 did not increase significantly, we observed notable shifts in the popularity of certain tasks. To capture these changes, we summarize the updated statistics of applications, now including data for 2024, in Fig. 7. For instance, Code Generation has seen a substantial rise in interest, now surpassing Code Search in popularity. This trend can be attributed to the widespread adoption of generative large language models, such as ChatGPT, which excel at producing coherent and contextually relevant code. The success of these models has enhanced the efficiency and flexibility of Code Generation tasks, encouraging an increase in research. Additionally, the growing demand from users for open-source generative tools has further encouraged development in this field.

Despite the increasing prominence of Code Generation, it is important to highlight that our research still focuses on four specific tasks: the two most popular generation tasks (Bug Fixing and Code Summarization) and the two most popular understanding tasks (Bug Detection and Code Search). To maintain a balanced exploration of generation and understanding tasks, we have chosen not to include

Table 10
Generalization performance of models on code summarization (BLUE-4/METEOR/ROUGE-L).

Code summarization	CodeBERT	CodeGPT	CodeT5
B model on B	18.48/13.07/35.02	14.63/16.36/34.18	20.25/15.07/38.29
F model on B	9.02/5.34/14.93	2.19/1.73/6.01	11.08/8.11/19.34
Mix-bench model on B	19.36/13.25/35.74	14.46/16.32/33.74	20.68/14.74/38.39
F model on F	31.48/20.16/41.65	31.43/19.78/41.02	32.40/22.08/44.00
B model on F	17.23/12.26/24.66	12.61/13.37/22.94	19.21/14.34/28.32
Mix-frequent model on F	32.17/20.46/42.32	31.62/19.91/41.57	32.47/21.86/44.11

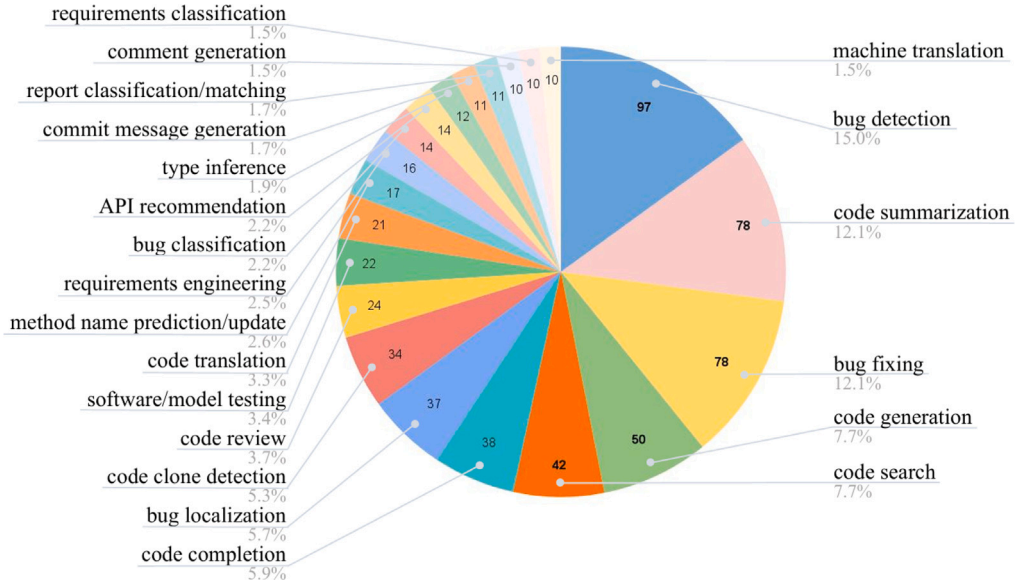


Fig. 7. Frequency of Applications (2017–2024). We only list applications with frequency ≥ 10.

Code Generation, now the third most popular generation task, in our empirical study. However, we recognize its rising importance and leave the exploration to future research.

Most Frequently Used Pre-trained Models. Table 11 summarizes the most commonly used pre-trained models before and in 2024, as well as the best-performing models for each task. We observe that the most commonly used models are also the top-performing models for Bug Detection and Code Search, and partially for Bug Fixing, preserving the conclusions drawn in RQ2.

For Code Summarization, although CodeGPT and CodeT5 demonstrate superior performance, CodeBERT remains the most widely used model in 2024. As with our previous conclusion, we suggest that the research community should closely monitor advancements in the field and consider integrating more advanced models to optimize outcomes, rather than continuing to rely on widely recognized models.

Frequency of Datasets. After including the statistics from 2024, the datasets listed in Table 8 continue to be the most frequently used.

6.2. Threats to validity

Internal Validity. The experimental settings in this paper are influenced by literature published before 2024. To minimize the risk of bias, we have ensured broad coverage across a long period and extensive software engineering venues, aiming to provide a comprehensive literature review and a representative empirical study.

External Validity. The conclusions drawn in this research may not be directly applicable to other tasks or models. Further consideration is needed to evaluate their relevance across different settings. We encourage future work to explore additional advanced models and tasks to gain broader insights into the field of Transformer-based models.

Table 11
Most frequently used model before and in 2024 vs. Best performing model.

Task name	Most frequent model before 2024	Most frequent model in 2024	Best performing model[a]
Bug fixing	CodeBERT (26/54)	CodeT5(10/19)	CodeBERT/CodeT5
Bug detection	CodeBERT (32/53)	Code-BERT(27/44)	CodeBERT
Code summarization	Vanilla Transformer (26/51)	Code-BERT(21/27)	CodeGPT/CodeT5
Code search	CodeBERT (20/33)	CodeBERT(9/9)	CodeBERT

[a] Best Performing Model refers to the model with the highest performance in Table 5.

7. Related work

For our related work, we focus on the pre-trained language models, the four applications - bug fixing, bug detection, code summarization, and code search, and related empirical studies of transformers.

7.1. Pre-trained language models

Different pre-trained language models are developed and demonstrated to have high performance in many NLP tasks (Kenton & Toutanova, 2019; Lewis et al., 2019; Liu et al., 2019; Radford et al., 2019). With the success of pre-trained language models in the NLP domain, researchers have been exploring and applying these models to code-related tasks (Kanade, Maniatis, Balakrishnan, & Shi, 2020; Liu, Li, Zhao, & Jin, 2020; Roziere, Lachaux, Chanussot, & Lample, 2020). Many pre-trained models for code have been developed.

CodeBERT (Feng et al., 2020) is one of the earliest models that has been specifically trained for code-related tasks. Subsequently, models like GraphCodeBERT (Guo et al., 2020) were proposed to improve over CodeBERT by incorporating additional information, such as data flow. Similarly, CodeGPT (Lu et al., 2021) and CodeT5 (Wang et al., 2021) are built based on GPT and T5 architectures, but pre-trained on a code-related corpus with additional pre-training objectives to better understand code. Our experiments are conducted on these three representative pre-trained models for code - CodeBERT, CodeGPT, and CodeT5. Currently, many revolutionary large language models are being developed and applied to different domains, *e.g.*, GPT-4 (OpenAI, 2023) and LLaMA (Touvron et al., 2023). These models, however, are not included in our study due to their commercial nature and substantial size, which demand extensive resources. Furthermore, comparing them with the models we studied in this paper could introduce unfairness, as they have vastly more parameters and are trained on significantly larger datasets.

7.2. Applications

We study four applications in this paper - bug fixing, bug detection, code summarization, and code search.

7.2.1. Bug fixing

There has been a lot of work in the domain of Transformer-based models that focuses on bug fixing. A majority of this work fine-tunes a pre-trained model for code. For example, CURE (Jiang, Lutellier, & Tan, 2021) fine-tuned a GPT model to generate patches for buggy code. SPT-Code (Niu et al., 2022), which has similar model structure as CodeBERT, also used fine-tuning to perform code refinement. In addition, Fu, Nguyen, Tantithamthavorn, Phung, and Le (2024) enhanced bug fixing by incorporating vulnerability queries and vulnerability masks into the Transformer model. AIBugHunter Fu, Tantithamthavorn, et al. (2024) fine-tuned CodeBERT to predict vulnerability severity scores and achieve real-time repairs.

7.2.2. Bug detection

Many approaches have been explored in the field of bug detection. Over the past decades, developer information has been utilized to predict bugs (Meneely, Williams, Snipes, & Osborne, 2008; Pinzger, Nagappan, & Murphy, 2008; Weyuker, Ostrand, & Bell, 2007). With the development of deep learning techniques, Yang, Lo, Xia, Zhang, and Sun (2015) leveraged deep learning to generate new features from traditional features using a Deep Belief Network (DBN). Later, Li, He, Zhu, and Lyu (2017) generated new features from Abstract Syntax Trees (ASTs) and combined them with hand-crafted features to perform bug prediction. There are also deep learning algorithms that specialize in bug detection, *e.g.*, DeepJIT (Hoang, Dam, Kamei, Lo, & Ubayashi, 2019) and CC2Vec (Hoang, Kang, Lo, & Lawall, 2020). Furthermore, Du and Yu (2023) employ Semantic Flow Graph (SFG), while Steenhoek, Gao, and Le (2024) and Rahman et al. (2024) leverage causal analysis to enhance the efficiency of bug detection.

7.2.3. Code summarization

Code summarization is one of the most popular tasks in deep learning. Fernandes, Allamanis, and Brockschmidt (2018) combined RNN/Transformers with GGNN. Ahmad, Chakraborty, Ray, and Chang (2020) applied Transformer to code summarization, and showed the advantage of sequential relative attention over positional encoding. EyeTrans Zhang, Li, et al. (2024) integrates human attention with machine attention to improve the effectiveness of neural code summarization, whereas Esale Fang et al. (2024) enhances code summarization by employing a multi-task learning approach, training encoders on three summary-focused tasks to better capture code-summary alignment.

7.2.4. Code search

In deep learning domain for code search, Sachdev et al. (2018) developed the tool NCS to learn embeddings of code without supervision. Gu, Zhang, and Kim (2018) proposed CODEnn to learn code representations through three encoded individual channels. With the outstanding performance of Transformer-based pre-trained models, many works (Ahmad et al., 2021; Feng et al., 2020; Mastropaolo et al., 2021; Phan et al., 2021) have looked into their application to code search and achieved satisfying results. Oracle4CS Phan and Jannesari (2024) improves code search by integrating a novel code representation technique, ASTSum, with classical machine translation models. HFEDR Zhang, Peng, Shen and Wu (2024) enhances code retrieval by increasing training data through hierarchical feature extraction and data reorganization.

7.3. Empirical study

There are some empirical studies on transformers in the literature. For example, Ma et al. (2024) analyzed various code models across different code comprehension tasks, while Zeng et al. (2022) and Niu et al. (2023) examined the suitability of different pre-trained models for a range of SE tasks. In Hou et al. (2023), Hou et al. surveyed existing pre-trained models, methods, and SE tasks. Minaee et al. (2024) expanded the scope of research beyond merely comparing coding abilities of pre-trained models to include diverse areas such as arithmetic reasoning and hallucination evaluation. Furthermore, there are studies that delve into the impacts of various pre-training (Tufano, Pascarella, & Bavota, 2023) and tuning techniques (Shi et al., 2023) on the performance of these models. For example, Hu et al. (2024) investigated the effectiveness of active learning on code models.

Compared to these previous works, our work additionally reviews the literature, summarizes the most widely studied applications, suggests on developers' needs, contests certain beliefs, investigates the resource consumption of different models, and concludes on model generalization ability trained on different datasets.

8. Conclusion

In this empirical study, we comprehensively review the literature on transformer-based pre-trained models used in SE research published during 2017–2023. We focus on the three widely used models – CodeBERT, CodeGPT, CodeT5 – and evaluate their performance on the four most popular code-related tasks: Bug Fixing, Bug Detection, Code Summarization, and Code Search. We examine existing literature and developers' needs, contest current beliefs for model architectures, consider model resource consumption, and evaluate model generalization abilities. We frame our findings as promises and perils of using transformer-based models for SE research.

Our study highlights several practical and important concerns for researchers working in this field. First, it is important to investigate approaches to support applications that are most relevant to developers' needs, such as communication and collaboration. Second, the encoder–decoder architecture's optimality for general-purpose coding tasks should be re-examined, given its non-dominant performance and high resource consumption. Third, it is important to carefully select the most suitable model for each specific task. Finally, the commonly used benchmark datasets should be improved to enhance model generalization, potentially with data selection strategies.

CRediT authorship contribution statement

Yan Xiao: Writing – original draft. **Xinyue Zuo:** Writing – original draft. **Xiaoyue Lu:** Data curation. **Jin Song Dong:** Validation. **Xiaochun Cao:** Validation, Supervision, Conceptualization. **Ivan Beschastnikh:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work was supported by the Fundamental Research Funds for the Central Universities, China, Sun Yat-sen University, China, under Grant 24qnpy153.

Data availability

We have provided the link of code and dataset.

References

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2020). A transformer-based approach for source code summarization. arXiv preprint arXiv:2005.00653.

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). Unified pre-training for program understanding and generation. arXiv preprint arXiv:2103.06333.

Aniche, M., Treude, C., Steinmacher, I., Wiese, I., Pinto, G., Storey, M.-A., et al. (2018). How modern news aggregators help development communities shape and share knowledge. In *Proceedings of the 40th international conference on software engineering* (pp. 499–510).

Anish, P. R., Lawhatre, P., Chatterjee, R., Joshi, V., & Ghaisas, S. (2022). Automated labeling and classification of business rules from software requirement specifications. In *Proceedings of the 44th international conference on software engineering: Software engineering in practice* (pp. 53–54).

Banerjee, S., & Lavie, A. (2005). METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization* (pp. 65–72).

Bogart, C., Kästner, C., & Herbsleb, J. (2015). When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *2015 30th IEEE/ACM international conference on automated software engineering workshop* (pp. 86–89). IEEE.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., et al. (2021). On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, *33*, 1877–1901.

Ceccato, M., Marchetto, A., Mariani, L., Nguyen, C. D., & Tonella, P. (2015). Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *25*(1), 1–38.

Chakraborty, S., Ahmed, T., Ding, Y., Devanbu, P. T., & Ray, B. (2022). NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering* (pp. 18–30).

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., et al. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

Cheng, W., Hu, P., Wei, S., & Mo, R. (2022). Keyword-guided abstractive code summarization via incorporating structural and contextual information. *Information and Software Technology*, *150*, Article 106987.

Chirkova, N., & Troshin, S. (2021). Empirical study of transformers for source code. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (pp. 703–715).

Dagenais, B., & Robillard, M. P. (2010). Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on foundations of software engineering* (pp. 127–136).

Devine, P. (2022). Finding appropriate user feedback analysis techniques for multiple data domains. In *Proceedings of the ACM/IEEE 44th international conference on software engineering: Companion proceedings* (pp. 316–318).

Du, Y., & Yu, Z. (2023). Pre-training code representation with semantic flow graph for effective bug localization. In *Proceedings of the 31st ACM joint European software engineering conference and symposium on the foundations of software engineering* (pp. 579–591).

Ezzini, S., Abualhaija, S., Arora, C., & Sabetzadeh, M. (2022). Automated handling of anaphoric ambiguity in requirements: a multi-solution study. In *Proceedings of the 44th international conference on software engineering* (pp. 187–199).

Fang, C., Sun, W., Chen, Y., Chen, X., Wei, Z., Zhang, Q., et al. (2024). Esale: Enhancing code-summary alignment learning for source code summarization. *IEEE Transactions on Software Engineering*.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., et al. (2020). Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (pp. 1536–1547).

Fernandes, P., Allamanis, M., & Brockschmidt, M. (2018). Structured neural summarization. arXiv preprint arXiv:1811.01824.

Fu, M., Nguyen, V., Tantithamthavorn, C., Phung, D., & Le, T. (2024). Vision transformer inspired automated vulnerability repair. *ACM Transactions on Software Engineering and Methodology*, *33*(3), 1–29.

Fu, M., Tantithamthavorn, C., Le, T., Kume, Y., Nguyen, V., Phung, D., et al. (2024). Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering*, *29*(1), 4.

Fu, M., Tantithamthavorn, C., Le, T., Nguyen, V., & Phung, D. (2022). VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering* (pp. 935–947).

Gonçalves, M. K., de Souza, C. R., & González, V. M. (2011). Collaboration, information seeking and communication: An observational study of software developers' work practices. *Journal of Universal Computer Science*, *17*(14), 1913–1930.

Gu, X., Zhang, H., & Kim, S. (2018). Deep code search. In *Proceedings of the 40th international conference on software engineering* (pp. 933–944).

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022). Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Shujie, L., et al. (2020). GraphCodeBERT: Pre-training code representations with data flow. In *International conference on learning representations*.

Hoang, T., Dam, H. K., Kamei, Y., Lo, D., & Ubayashi, N. (2019). DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th international conference on mining software repositories* (pp. 34–45). IEEE.

Hoang, T., Kang, H. J., Lo, D., & Lawall, J. (2020). CC2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering* (pp. 518–529).

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., et al. (2023). Large language models for software engineering: A systematic literature review. arXiv preprint arXiv:2308.10620.

Hu, X., Guo, Y., Lu, J., Zhu, Z., Li, C., Ge, J., et al. (2022). Lighting up supervised learning in user review-based code localization: dataset and benchmark. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering* (pp. 533–545).

Hu, Q., Guo, Y., Xie, X., Cordy, M., Ma, L., Papadakis, M., et al. (2024). Active code learning: Benchmarking sample-efficient training of code models. *IEEE Transactions on Software Engineering*.

Huang, Q., Yuan, Z., Xing, Z., Xu, X., Zhu, L., & Lu, Q. (2022). Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code. In *37th IEEE/ACM international conference on automated software engineering* (pp. 1–13).

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436.

Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. In *Proceedings of the 54th annual meeting of the Association for Computational Linguistics (Volume 1: Long papers)* (pp. 2073–2083).

Jiang, N., Lutellier, T., & Tan, L. (2021). Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd international conference on software engineering* (pp. 1161–1173). IEEE.

Just, R., Jalali, D., & Ernst, M. D. (2014). Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis* (pp. 437–440).

Kanade, A., Maniatis, P., Balakrishnan, G., & Shi, K. (2020). Learning and evaluating contextual embedding of source code. In *International conference on machine learning* (pp. 5110–5121). PMLR.

Karampatsis, R.-M., & Sutton, C. (2020). How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th international conference on mining software repositories* (pp. 573–577).

Kenton, J. D. M.-W. C., & Toutanova, L. K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT* (pp. 4171–4186).

Kim, S., Zhao, J., Tian, Y., & Chandra, S. (2021). Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd international conference on software engineering* (pp. 150–162). IEEE.

Ko, A. J., DeLine, R., & Venolia, G. (2007). Information needs in collocated software development teams. In *29th international conference on software engineering* (pp. 344–353). IEEE.

Lan, Z. (2019). Albert: A lite bert for self-supervised learning of language representations. arXiv preprint arXiv:1909.11942.

Le-Cong, T., Kang, H. J., Nguyen, T. G., Haryono, S. A., Lo, D., Le, X.-B. D., et al. (2022). AutoPruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering* (pp. 520–532).

LeClair, A., & McMillan, C. (2019). Recommendations for datasets for source code summarization. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: Human language technologies, volume 1 (Long and short papers)* (pp. 3931–3937).

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., et al. (2019). Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461.

Li, J., He, P., Zhu, J., & Lyu, M. R. (2017). Software defect prediction via convolutional neural network. In *2017 IEEE international conference on software quality, reliability and security* (pp. 318–328). IEEE.

Li, X., Liu, S., Feng, R., Meng, G., Xie, X., Chen, K., et al. (2022). Transrepair: Context-aware program repair for compilation errors. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering* (pp. 1–13).

Lin, C.-Y. (2004). Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out* (pp. 74–81).

Lin, D., Koppel, J., Chen, A., & Solar-Lezama, A. (2017). QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: Software for humanity* (pp. 55–56).

Liu, F., Li, G., Zhao, Y., & Jin, Z. (2020). Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering* (pp. 473–485).

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., et al. (2019). Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.

Liu, Y., Tantithamthavorn, C., Liu, Y., & Li, L. (2024). On the reliability and explainability of language models for program generation. *ACM Transactions on Software Engineering and Methodology*.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., et al. (2021). Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664.

Ma, W., Liu, S., Zhao, M., Xie, X., Wang, W., Hu, Q., et al. (2024). Unveiling code pre-trained models: Investigating syntax and semantics capacities. *ACM Transactions on Software Engineering and Methodology*, *33*(7), 1–29.

Madeiral, F., Urli, S., Maia, M., & Monperrus, M. (2019). Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering* (pp. 468–478). IEEE.

Mastropaolo, A., Scalabrino, S., Cooper, N., Palacio, D. N., Poshyvanyk, D., Oliveto, R., et al. (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* (pp. 336–347). IEEE.

Meneely, A., Williams, L., Snipes, W., & Osborne, J. (2008). Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT international symposium on foundations of software engineering* (pp. 13–23).

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., et al. (2024). Large language models: A survey. arXiv preprint arXiv:2402.06196.

Niu, C., Li, C., Ng, V., Chen, D., Ge, J., & Luo, B. (2023). An empirical comparison of pre-trained models of source code. arXiv preprint arXiv:2302.04026.

Niu, C., Li, C., Ng, V., Ge, J., Huang, L., & Luo, B. (2022). SPT-code: sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th international conference on software engineering* (pp. 2006–2018).

OpenAI (2022). Chatgpt: Optimizing language models for dialogue. URL https://chat.openai.com.

OpenAI (2023). GPT-4 technical report. arXiv preprint arXiv:2303.08774.

Palatucci, M., Pomerleau, D., Hinton, G. E., & Mitchell, T. M. (2009). Zero-shot learning with semantic output codes. *Advances in Neural Information Processing Systems*, *22*.

Pan, S., Bao, L., Ren, X., Xia, X., Lo, D., & Li, S. (2021). Automating developer chat mining. In *2021 36th IEEE/ACM international conference on automated software engineering* (pp. 854–866). IEEE.

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (pp. 311–318).

Phan, H., & Jannesari, A. (2024). Leveraging statistical machine translation for code search. In *Proceedings of the 28th international conference on evaluation and assessment in software engineering* (pp. 191–200).

Phan, L., Tran, H., Le, D., Nguyen, H., Anibal, J., Peltekian, A., et al. (2021). Cotext: Multi-task learning with code-text transformer. arXiv preprint arXiv:2105.08645.

Pinzger, M., Nagappan, N., & Murphy, B. (2008). Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT international symposium on foundations of software engineering* (pp. 2–12).

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, *1*(8), 9.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., et al. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, *21*(1), 5485–5551.

Rahman, M. M., Ceka, I., Mao, C., Chakraborty, S., Ray, B., & Le, W. (2024). Towards causal deep learning for vulnerability detection. In *Proceedings of the IEEE/ACM 46th international conference on software engineering* (pp. 1–11).

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., et al. (2020). Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297.

Roy, D., Fakhoury, S., & Arnaoudova, V. (2021). Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (pp. 1105–1116).

Roziere, B., Lachaux, M.-A., Chanussot, L., & Lample, G. (2020). Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, *33*, 20601–20611.

Sachdev, S., Li, H., Luan, S., Kim, S., Sen, K., & Chandra, S. (2018). Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages* (pp. 31–41).

Saha, R. K., Lyu, Y., Lam, W., Yoshida, H., & Prasad, M. R. (2018). Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories* (pp. 10–13).

Sanh, V. (2019). DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108.

Sawant, A. A., Aniche, M., van Deursen, A., & Bacchelli, A. (2018). Understanding developers' needs on deprecation as a language feature. In *Proceedings of the 40th international conference on software engineering* (pp. 561–571).

Schuster, M., & Nakajima, K. (2012). Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing* (pp. 5149–5152). IEEE.

Shaw, P., Uszkoreit, J., & Vaswani, A. (2018). Self-attention with relative position representations. arXiv preprint arXiv:1803.02155.

Shi, L., Chen, X., Yang, Y., Jiang, H., Jiang, Z., Niu, N., et al. (2021). A first look at developers' live chat on gitter. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (pp. 391–403).

Shi, E., Wang, Y., Zhang, H., Du, L., Han, S., Zhang, D., et al. (2023). Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond. arXiv preprint arXiv:2304.05216.

Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T., et al. (1999). Byte Pair encoding: A text compression scheme that accelerates pattern matching.

Shiv, V., & Quirk, C. (2019). Novel positional encodings to enable tree-based transformers. *Advances in Neural Information Processing Systems*, *32*.

Silva, C. C., Galster, M., & Gilson, F. (2022). A qualitative analysis of themes in instant messaging communication of software developers. *Journal of Systems and Software*, *192*, Article 111397.

Steenhoek, B., Gao, H., & Le, W. (2024). Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM international conference on software engineering* (pp. 1–13).

Svyatkovskiy, A., Deng, S. K., Fu, S., & Sundaresan, N. (2020). Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (pp. 1433–1443).

Svyatkovskiy, A., Fakhoury, S., Ghorbani, N., Mytkowicz, T., Dinella, E., Bird, C., et al. (2022). Program merge conflict resolution via neural transformers. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering* (pp. 822–833).

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., et al. (2023). Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971.

Treude, C., Figueira Filho, F., & Kulesza, U. (2015). Summarizing and measuring development activity. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 625–636).

Tufano, R., Pascarella, L., & Bavota, G. (2023). Automating code-related tasks through transformers: The impact of pre-training. arXiv preprint arXiv:2302.04048.

Tufano, M., Watson, C., Bavota, G., Penta, M. D., White, M., & Poshyvanyk, D. (2019). An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *28*(4), 1–29.

Valdés-Rodríguez, Y., Hochstetter-Diez, J., Díaz-Arancibia, J., & Cadena-Martínez, R. (2023). Towards the integration of security practices in agile software development: a systematic mapping review. *Applied Sciences*, *13*(7), 4578.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., et al. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*.

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859.

Wang, C., Yang, Y., Gao, C., Peng, Y., Zhang, H., & Lyu, M. R. (2022). No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering* (pp. 382–394).

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., et al. (2022). Emergent abilities of large language models. arXiv preprint arXiv:2206.07682.

Weyuker, E. J., Ostrand, T. J., & Bell, R. M. (2007). Using developer information as a factor for fault prediction. In *Third international workshop on predictor models in software engineering* (p. 8). IEEE.

Xia, C. S., & Zhang, L. (2022). Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering* (pp. 959–971).

Yang, Z. (2019). XLNet: Generalized autoregressive pretraining for language understanding. arXiv preprint arXiv:1906.08237.

Yang, X., Lo, D., Xia, X., Zhang, Y., & Sun, J. (2015). Deep learning for just-in-time defect prediction. In *2015 IEEE international conference on software quality, reliability and security* (pp. 17–26). IEEE.

Zeng, Z., Tan, H., Zhang, H., Li, J., Zhang, Y., & Zhang, L. (2022). An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis* (pp. 39–51).

Zhang, Y., Li, J., Karas, Z., Bansal, A., Li, T. J.-J., McMillan, C., et al. (2024). Eyetrans: Merging human and machine attention for neural code summarization. *Proceedings of the ACM on Software Engineering*, *1*(FSE), 115–136.

Zhang, J., Mytkowicz, T., Kaufman, M., Piskac, R., & Lahiri, S. K. (2022). Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper). In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis* (pp. 77–88).

Zhang, J., Panthaplackel, S., Nie, P., Li, J. J., & Gligoric, M. (2022). CoditT5: Pretraining for source code and natural language editing. In *37th IEEE/ACM international conference on automated software engineering* (pp. 1–12).

Zhang, F., Peng, M., Shen, Y., & Wu, Q. (2024). Hierarchical features extraction and data reorganization for code search. *Journal of Systems and Software*, *208*, Article 111896.

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., et al. (2023). A survey of large language models. arXiv preprint arXiv:2303.18223.

Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems*, *32*.

Zhou, Z., Sha, C., & Peng, X. (2024). On calibration of pre-trained code models. In *Proceedings of the IEEE/ACM 46th international conference on software engineering* (pp. 1–13).

Zhou, Y., Shen, J., Zhang, X., Yang, W., Han, T., & Chen, T. (2022). Automatic source code summarization with graph attention networks. *Journal of Systems and Software*, *188*, Article 111257.

Zhu, X., Sha, C., & Niu, J. (2022). A simple retrieval-based method for code comment generation. In *2022 IEEE international conference on software analysis, evolution and reengineering* (pp. 1089–1100). IEEE.

Zuo, X. (2023). GitHub. URL https://github.com/ZuoXinyue/Transformer-Empirical-SE.