

# Pitfalls in Language Models for Code Intelligence: A Taxonomy and Survey

XINYU SHE\*, Huazhong University of Science and Technology, China

YUE LIU\*, Monash University, Australia

YANJIE ZHAO, Ohio State University, USA

YILING HE, Zhejiang University, China

LI LI, Beihang University, China

CHAKKRIT TANTITHAMTHAVORN, Monash University, Australia

ZHAN QIN, Zhejiang University, China

HAOYU WANG<sup>†</sup>, Huazhong University of Science and Technology, China

Modern language models (LMs) have been successfully employed in source code generation and understanding, leading to a significant increase in research focused on learning-based code intelligence, such as automated bug repair, and test case generation. Despite their great potential, **language models for code intelligence (LM4Code) are susceptible to potential pitfalls, which hinder realistic performance and further impact their reliability and applicability in real-world deployment.** Such challenges drive the need for a comprehensive understanding - not just identifying these issues but delving into their possible implications and existing solutions to build more reliable language models tailored to code intelligence. Based on a well-defined systematic research approach, we conducted an extensive literature review to uncover the pitfalls inherent in LM4Code. Finally, 67 primary studies from top-tier venues have been identified. After carefully examining these studies, we designed a taxonomy of pitfalls in LM4Code research and conducted a systematic study to summarize the issues, implications, current solutions, and challenges of different pitfalls for LM4Code systems. We developed a comprehensive classification scheme that dissects pitfalls across four crucial aspects: data collection and labeling, system design and learning, performance evaluation, and deployment and maintenance. Through this study, we aim to provide a roadmap for researchers and practitioners, facilitating their understanding and utilization of LM4Code in reliable and trustworthy ways.

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

Additional Key Words and Phrases: Language models for Code, Software engineering, Code generation, Code intelligence, Trustworthiness

\*Co-first authors who contributed equally to this work.

<sup>†</sup>Haoyu Wang is the corresponding author (haoyuwang@hust.edu.cn).

Authors' addresses: Xinyu She, xinyushe@hust.edu.cn, Huazhong University of Science and Technology, Wuhan, China; Yue Liu, yue.liu1@monash.edu, Monash University, Melbourne, Australia; Yanjie Zhao, carolzhao233@gmail.com, Ohio State University, Columbus, USA; Yiling He, yilinghe@zju.edu.cn, Zhejiang University, Hangzhou, China; Li Li, lilicoding@ieee.org, Beihang University, Beijing, China; Chakkrit Tantithamthavorn, chakkrit@monash.edu, Monash University, Melbourne, Australia; Zhan Qin, qinzhao@zju.edu.cn, Zhejiang University, Hangzhou, China; Haoyu Wang, haoyuwang@hust.edu.cn, Huazhong University of Science and Technology, Wuhan, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2023 Association for Computing Machinery.

0004-5411/2023/10-ART1 \$15.00

<https://doi.org/10.1145/xxxxxxx>

**ACM Reference Format:**

Xinyu She, Yue Liu, Yanjie Zhao, Yiling He, Li Li, Chakkrit Tantithamthavorn, Zhan Qin, and Haoyu Wang. 2023. Pitfalls in Language Models for Code Intelligence: A Taxonomy and Survey. *J. ACM* 37, 4, Article 1 (October 2023), 41 pages. <https://doi.org/10.1145/xxxxxxx>

**1 INTRODUCTION**

With every upgrade, language models (LMs) seem to redefine future boundaries. Language models have achieved remarkable successes in natural language understanding and generation [97, 148], underlined by the significant contributions from state-of-the-art models such as T5 [35, 160], BERT [38, 39, 85], and GPT [77, 165]. Due to the format similarity between source code and natural language, language models have been widely applied in the domain of software engineering [18, 103]. They are now extensively researched and employed for source code understanding and generation, such as code completion [114, 114, 125], code summarization [39], code generation [83, 128], code search [138], program repair [30, 165], and test case generation [180]. With powerful learning capabilities, language models have shown superior performance against traditional code intelligence approaches, such as template-based, heuristic-based, and machine learning-based approaches [58, 111, 125]. Their superior performance stems from the fact that many LMs are trained on vast and diverse code repositories, enabling LMs to discern complex syntax, comprehend semantic context, and effectively predict code sequences [152].

However, the lack of transparency, often termed “black-box”, poses significant challenges and concerns [141, 143]. In other words, while language models for code intelligence (LM4Code) approaches offer powerful capabilities, they often lack transparency in their underlying reasoning and decision-making process. Tantithamthavorn *et al.* also raised concerns that such a lack of transparency often leads to a lack of adoption of LM4Code in practice [61, 143]. Consequently, hidden or neglected pitfalls in data or algorithms may persist, leading to unrealistic performance evaluation and unreliable code recommendations [52, 142]. For example, Shi *et al.* [133] found that noisy data (e.g., empty methods or duplicated code) was prevalent in widely-used benchmark datasets for code summarization, with contamination levels ranging between 31% to 66%. By filtering out this noisy data, performance metrics like the BLEU-4 score witnessed a substantial increase (e.g., from 11.36% to 16.48%). Similarly, Sun *et al.* [138] highlighted a substantial amount of noise in user queries across various code search benchmark datasets. Such instances underscore the hidden data noise that might undermine the trustworthiness of code produced or recommended by LMs. What’s more concerning is when these pitfalls go unnoticed, which raises significant questions about the reliability and integrity of the LM4Code systems built on them, thereby preventing the adoption of research advances in academia and industry.

As LMs become increasingly prevalent in code intelligence despite increasing obstacles, there emerges an urgent need for a comprehensive understanding of potential pitfalls within LM4Code systems. This isn’t limited to pitfall identification; it demands a deeper exploration into the understanding of the implications of these pitfalls, current solutions, and possible challenges. Although there is a growing body of research concerning or addressing pitfalls in LM4Code [90, 133, 138, 175], the domain lacks a comprehensive and systematic overview of these efforts. Without such an overview, researchers, developers, and practitioners potentially overlook significant pitfalls identified in previous studies. In this study, we conducted a systematic literature review, adhering to a well-defined approach that identifies, evaluates, and interprets the relevant literature that focuses on the pitfalls within LM4Code. Our contributions of this paper are as follows:

- *Paper Collection of Pitfalls in LM4Code.* Through a rigorous systematic literature review (SLR) protocol as outlined by [66, 68] and after an in-depth analysis of the primary studies, we collected 67 primary papers (spanning 2018 to 2023) closely related to evaluating or addressing

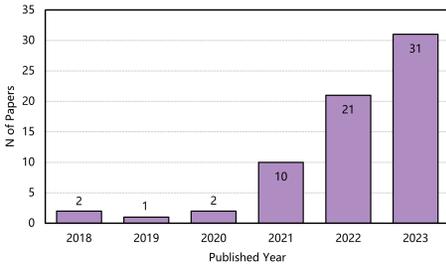


Figure 1. Distribution of papers over years

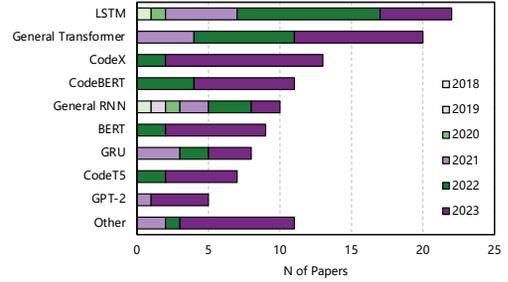


Figure 2. Distribution of papers across LMs

LM4Code pitfalls. Comprehensive details on our review process and the collected papers are available online <sup>1</sup>.

- *Comprehensive Taxonomy.* We conducted a qualitative and quantitative synthesis of the collected studies. We present a taxonomy of the collected studies according to the LM4Code lifecycle, including data collection and labeling, system design and learning, performance evaluation, deployment, and maintenance. Our synthesis investigates the pitfalls present in LM4Code, summarizes the implications of these pitfalls, investigates how these issues are addressed, and outlines future challenges in this field.
- *Insightful Findings and Recommendations.* In addition to identifying and analyzing pitfalls, we distilled practical insights and recommendations for researchers and practitioners in the field of LM4Code. These findings pave the way for developing more robust and reliable language models tailored for code intelligence, mitigating potential challenges and maximizing their utility of such models in real-world applications.

## 2 STUDY DESIGN

### 2.1 Research Questions and Motivations

In recent research, language models trained for code intelligence have shown promising performance [51, 155, 175]. However, an increasing number of literature [101, 133, 138] has highlighted the existence of pitfalls in LM4Code that can skew their realistic performance, leading to either substantial overestimation or underestimation of their effectiveness. The aim of conducting this systematic review is to gain an in-depth understanding of the pitfalls present in language models tailored for code intelligence. Ensuring the robustness, reliability, and trustworthy deployment of LMs is important for their effective integration into the software development lifecycle. Consequently, it is crucial to discern the nature of these pitfalls, comprehend their implications, and examine existing solutions. Thus, we aim to answer the following research questions in this study:

- **RQ1: What types of pitfalls are prevalent in language models for code intelligence?** This research question aims to identify the prevalent pitfalls in LM4Code systems, exploring how they could affect various stages of the learning-based system lifecycle.
- **RQ2: What are the implications of these pitfalls?** This research question investigates the implications of the identified pitfalls, specifically focusing on their impacts on the effectiveness, reliability, and ethical considerations of LM4Code systems.
- **RQ3: What solutions have been proposed to address these pitfalls?** This research question reviews the existing body of literature to identify proposed approaches for solving the identified pitfalls.

<sup>1</sup><https://github.com/yueyueL/ReliableLM4Code>

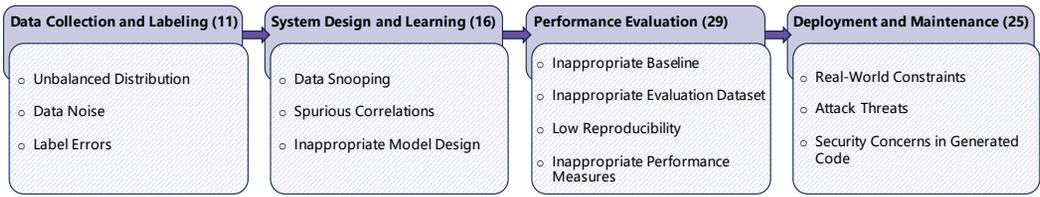


Figure 3. The overview of pitfalls of LMs for code intelligence

## 2.2 Paper Collection and Selection

To systematically identify relevant studies on pitfalls of LM4Code, we followed a rigorous methodology proposed by Kitchenham *et al.* [66, 68] and Zhang *et al.* [178] to perform a lightweight Systematic Literature Review (SLR). We utilized the “Quasi-Gold Standard” (QGS) [178] approach, combining manual and automated search strategies across major academic databases. This ensured comprehensive coverage while maintaining a focus on high-quality studies. In total, we obtained over 100,000 papers from major academic databases including ACM Digital Library, IEEE Xplore, Springer, ScienceDirect, Web of Science, and DBLP. Specifically, through QGS we obtained over 100,000 candidate papers from ACM Digital Library, IEEE Xplore, Springer, ScienceDirect, Web of Science, and DBLP. To filter these results, we defined robust inclusion/exclusion criteria and performed the quality assessment of full texts. Additionally, we also conducted backward and forward snowballing [67] to complement the database searches and avoid excluding important works. Through snowballing, we evaluated over 1,000 additional papers. By systematically combining these search strategies, selection criteria, and quality checks, we identified 67 high-quality studies investigating pitfalls and challenges in LM4Code. Due to page limits, we make the review protocol details available in the supplementary report and our online repository.

Figure 1 displays the distribution of the collected research studies across the published year. From Figure 1, we have noted that there is a significant increase in the number of relevant research studies published annually from 2021, indicating a rising interest in investigating potential LM4Code pitfalls. Figure 2 further presents the distribution of language modes used in the collected studies. It is important to note that while both LSTM and GRU are types of RNN, papers that only specify the use of RNN without further detail are categorized under “General RNN” in this study. Similarly, despite observing the utilization of several popular transformer-based architectures such as CodeBERT, Codex, and CodeT5, papers that merely claim the use of a self-defined or custom-designed transformer are classified as “General Transformer” in subsequent sections. We find that LSTM models exhibit a higher prevalence than other types. However, over the past two years, studies utilizing transformer-based LMs, particularly pre-trained models like CodeBERT and Codex, have substantially increased. Overall, these findings indicate growing attention toward identifying and evaluating challenges with the reliability and effectiveness of LM4Code. The community appears to be moving towards a comprehensive exploration of the realistic performance of LM4Code. Our systematic collection provides an opportunity to thoroughly analyze LM4Code pitfalls.

## 2.3 Paper Organization

Through our study collection process, we identified 67 research papers that specifically discuss or address the pitfalls in LM4Code. In the following content, we aim to answer our initial research questions based on these papers. Similar to prior research [6], our answers are organized following the typical workflow of LMs for code intelligence, which ensures the topics discussed in collected studies can be covered. To be specific, we segment the pitfalls into four key aspects of the LM

Bias Type	Paper	SE Tasks	Description	Implications
Unbalanced Distribution	[14], [168], [134]	Vulnerability Detection	The ratio of vulnerable and non-vulnerable cases in real-world projects is extremely unbalanced	F1-score drops by 73%
	[173]	Defect Prediction	Many datasets are imbalanced with target classes under 30%	Imbalanced data results in an F1 score below 0.2
	[76]	Bug Report Classification, Defect Prediction	Class imbalance	/
Data Noise	[89], [177]	Commit Message Generation	Commit messages mix bot-generated content with human-written trivial messages containing redundant or easily inferred information.	BLEU-4 drops from 31.92% to 14.19%
	[138]	Code Search	Over one-third of queries of code search datasets contain noises that make them deviate from natural user queries (e.g., HTML tags, interrogation)	MRR improves from 0.407 to 0.512 after data cleaning
	[133]	Code Summarization	Noisy code-comment pairs, including non-literal and duplicated code, are prevalent in four benchmark datasets (31% to 66%)	Training three models with the cleaned datasets improves the BLEU-4 by 27%, 21%, and 24%
Labeling Errors	[76]	Bug Report Classification, Defect Prediction	Mislabelled samples - issue reports that describe defects but are not classified as such.	/
	[79]	Code Translation, Clone Detection, Code Search	Most of the collected code snippets are unlabeled	/
	[101]	Vulnerability Detection	Error labels are common in many vulnerability datasets	F1-score drops by 20.7%

Table 1. Summary of Common Biases in Data Collection and Labeling from Reviewed Research Studies

pipeline: data collection and labeling (Section 3), system design and learning (Section 4), performance evaluation (Section 5), and deployment and maintenance (Section 6). This framework is depicted in Figure 3. For each aspect, we first summarize the types of prevalent pitfalls discussed in the collected studies (RQ1), then introduce the implications of these pitfalls (RQ2), and finally explore potential solutions and best practices recommended in the literature (RQ3). In Section 7, we further discuss open challenges and promising research directions. This organized structure enables a comprehensive analysis of pitfalls and considerations across the entire LM4Code pipeline. Our taxonomy aims to provide crucial insights for developing more robust, reliable, and practical LM systems for code intelligence tasks.

### 3 DATA COLLECTION AND LABELING

The data-hungry language models require large-scale and high-quality training datasets. According to a survey by Hou *et al.* [51], the majority of LMs for code intelligence are trained using data from open-source platforms, with GitHub and StackOverflow being the most popular options. However, the data in these platforms are user-contributed, varying significantly in the level of quality and reliability. It leads to non-negligible noises, bias, and errors in the training dataset and further affects the behavior of the models, which brings significant pitfalls in LMs for code intelligence. In this section, we provide a brief description of related studies and discuss the implications and potential solutions during the data collection and labeling stages.

#### 3.1 RQ1-Pitfalls

From the collected papers, we identified 11 research studies focusing on pitfalls during the data collection and labeling process. Table 1 presents the statistics of literature on this topic, where the pitfalls can be grouped into three main categories.

**Unbalanced Distribution:** Unbalanced distribution arises when there is a lack of proper randomization in the selection of samples, leading to certain populations being underrepresented or overrepresented [127]. In code-related scenarios, it usually refers to the gap between the sample distribution of real-world practices and training datasets. For example, as emphasized by [14, 134, 168], vulnerable instances in vulnerability detection studies are overwhelming while neutral code instances in real-world environments considerably outnumber their vulnerable counterparts. This imbalance extends to other code-based tasks. In software defect prediction, where defective modules

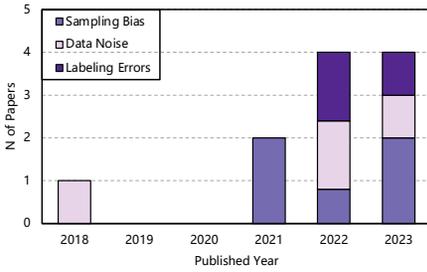


Figure 4. Paper distribution across time (Section 3)

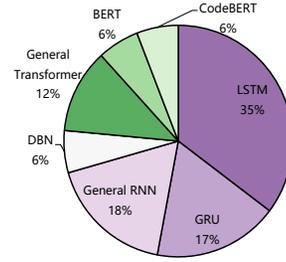


Figure 5. Distribution of LMs (Section 3)

are scarce compared with non-defective cases in real-world environments [76, 173]. Similarly, bug report classification suffers from underrepresentation of the minority bug class [76].

**Data Noise:** Data noises are the samples that are meaningless or even harmful for the models to learn, such as samples with deprecated coding conventions, multi-lingual comments, and auto-generated code snippets [133]. Such noises widely exist in code datasets. For example, as investigated by Sun *et al.* [138] and Shi *et al.* [133], over one-third of the popular code dataset, CodeSearchNet [54], are noises that are hardly seen in neural code search. Their analysis results show that the examined datasets contain a multitude of noise categories, including unrelated comments, non-literal characters, and issues like empty or duplicated code. Liu *et al.* [89] and Zhang *et al.* [177] specifically investigated data noise in commit message generations, where approximately 16% of the commit messages of benchmark dataset by Jiang *et al.* [59] were identified as noises.

**Labeling Errors:** Labeling errors arise when ground-truth labels are inaccurate, unstable, or erroneous [6, 144]. In some code-related tasks, such as vulnerability detection, the raw code datasets need to be labeled by human annotators. Nie *et al.* [101] explored the labeling error problem in vulnerability detection wherein a vulnerable code sample is mislabeled as non-vulnerable, and vice versa. They found that mislabeling a non-vulnerable sample as vulnerable was a more pervasive issue. The research further assessed three prominent datasets, D2A [183], Big-Vul [29], and Cross-Vul [102], discovering that in the worst cases, nearly 30% of the labels in these datasets may cause noisy labels. Similarly, Li *et al.* [76] examined the datasets from Herzig *et al.* [49] for Bug Report Classification (BRC) and from Yatish *et al.* [172] for Software Defect Prediction (SDP). Their findings revealed that these datasets possess mislabel rates ranging between 2% and 29%.

Overall, Figure 4 and Figure 5 present the distribution of the 11 papers concerning bias in data collection and labeling. Figure 4 shows that the issue of data noise has continuously attracted the attention of researchers, and this attention has intensified significantly in recent years. Figure 5 reveals that while the emergence and widespread use of sophisticated models such as BERT and GPT [51], the majority of the examined articles on data bias mostly concentrate on conventional language models, specifically LSTM and GRU. This observation indicates a potential avenue for further investigation into the data biases present in modern language models.

### 3.2 RQ2-Implications

Our review results highlight the pervasive presence of pitfalls during the data collection and labeling processes in various automated code-related tasks. However, the deeper implications of these pitfalls remain to be fully discerned. Thus, in this section, we aim to summarize the reviewed papers and provide insights into how these pitfalls influence the overall efficacy and performance of language models in code intelligence.

**Performance Overestimation:** Train-test-split is a common practice for evaluating the neural models [6]. However, derived from the same datasets, the test set contains the same bias, such as imbalanced data, as in the train set. It leads to an overestimation of the model's performance since there is a gap between the testing dataset and real-world practices. For example, Chakraborty *et al.* [14] presented that vulnerability detection techniques are implemented on balanced datasets, and the models can achieve more than 90% F1 scores. However, the scenario drastically shifts when the same models are evaluated with a realistic dataset (where only 6% of the examples are vulnerable). In such cases, while the model could boast a recall as high as 91.24%, this seemingly perfect metric can be deceptive. A deeper dive reveals a mere 18.47% F1 score, leading to a significant number of false positives. A similar scenario arises with the introduction of data noise. Zhang *et al.* [177] demonstrated this by training Transformer-based methods on a noisy dataset that included both bot-generated and trivial messages for code commit generation. Remarkably, the model achieved a 42.4% BLEU-4 score under these conditions. Yet, once the data noise was removed, the BLEU-4 score dropped sharply to 26.2%. Overall, pitfalls in data, whether from sampling imbalances or the presence of data noise, can lead to exaggerated performance metrics in language models for code tasks.

**Compromised Model Efficacy:** High-quality training data serves as the foundation for the trustworthy training of models. When the training data introduces inherent noises and errors, language models might inadvertently learn irrelevant patterns or establish spurious correlations. This not only distorts the model's understanding but can also undermine its performance. For instance, Sun *et al.* [138] proved that code search models, when trained on carefully cleaned data without data noise, achieve a significant improvement in the number of answered queries and the rank of ground truth in search results (e.g., MRR improves from 0.407 to 0.512). Similarly, Nie *et al.* [101] showed that labeling errors severely compromise the performance of prevalent vulnerability detection models, with the worst instances seeing an average F1 score plummet of 20.7%.

### 3.3 RQ3-Solutions

Recognizing pitfalls in data collection and labeling has emphasized the need for robust solutions to address and mitigate these issues. Several solutions have been proposed in the literature that we've reviewed. These solutions have been organized into distinct categories based on their underlying principles and methodologies.

**Data Cleaning/Denoising:** Data cleaning/denoising is the process of improving a dataset by removing or correcting abnormalities, inconsistencies, and inaccuracies. This phase is critical to ensuring that the training data is accurate and does not contain any misleading or irrelevant information. Many pitfalls develop as a result of noisy or erroneous data, and data cleaning is the major method for dealing with such difficulties. Shi *et al.* [133] introduced a rule-based cleaning tool, named CAT (Code-comment cleAning Tool), that employs configurable heuristics rules to automatically scan and filter out comments and code with syntactic anomalies, thereby detecting the occurrences and distribution of data noises. Similarly, Sun *et al.* [138] presented a data cleaning framework tailored for code search. It begins with a rule-based syntactic filter configured with heuristic rules to identify syntactically inconsistent comments. This is followed by a model-based semantic filter, which focuses on comments with the fewest reconstruction discrepancies using a Variational Auto-Encoder model trained on a pre-established bootstrap query corpus. Their evaluation results demonstrate that this hybrid filter approach not only significantly save computational resources but also enhances model accuracy. Nie *et al.* [101] introduced confident learning [107] and differential training [167] for denoising-based noisy label detection, aiming to enhance the label quality of vulnerability datasets. They found that the effectiveness of denoising

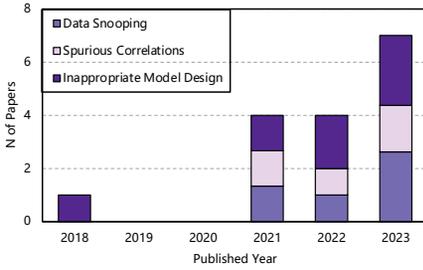


Figure 6. Paper distribution across time (Section 4)

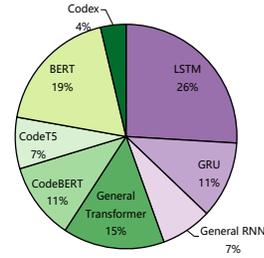


Figure 7. Distribution of LMs (Section 4)

methods heavily relies on the vulnerability detection models' fitting ability to the datasets, and these denoising methods show considerable promise in boosting vulnerability detection performance.

**Real-world Benchmarks:** To precisely evaluate the models, researchers propose to use real-world benchmarks, instead of train-test-split datasets. Many benchmarks are thus constructed. For example, many benchmarks, such as HumanEval [17], DS-1000 [70], and MBPP [7], are constructed using human-written tasks and test cases to evaluate the code generation LMs. Compared with the code snippets in open-source repositories, such human-written tasks are closer to the real user requests in practice, which better reflects the performance of the model. In addition, the models can be evaluated directly using the production data accumulated during the operation of LM-based systems. For example, Hellendoorn *et al.* [47] and Aye *et al.* [8] adopt the production data of code completion systems as the evaluation dataset to better measure the model's performance. Similarly, Mozannar *et al.* [98, 99] use the user behavior data to demonstrate the effectiveness of their proposed methods. Apart from that, Lin *et al.* [79] examine their approach on a real-world dataset composed of programming exercises with multiple solutions.

#### Summary - Data Collection and Labeling

Based on 11 relevant studies, our literature review reveals three prevalent pitfalls (i.e., unbalanced distribution, data noise, and labeling errors) in the data collection and labeling process. These pitfalls propagate, causing overestimated performance and compromised model efficacy. Though initial solutions like data cleaning/denoising and real-world benchmarks have been proposed, the field is far from reaching a comprehensive resolution. The implications underscore the need for automated and scalable techniques to ensure high-quality data for LM4Code.

## 4 SYSTEM DESIGN AND LEARNING

This section examines pitfalls in the system design and learning process for LM4Code. The training of these LM4Code models directly impacts their quality and efficacy for empowering code intelligence. However, several challenges arise in crafting optimal model architectures, formulating strategic training-testing approaches, refining data preprocessing techniques, and selecting suitable learning algorithms. Each design decision risks introducing pitfalls that can undermine model robustness and effectiveness.

### 4.1 RQ1-Pitfalls

We have identified 16 research studies dedicated to the exploration of pitfalls introduced in the system design and learning process. These pitfalls can be broadly categorized into three categories:

data snooping, spurious correlations, and inappropriate model design. In the following, we provide comprehensive descriptions of these three pitfalls.

**Data Snooping:** Data snooping arises when LMs are inadvertently exposed during training to information that should be inaccessible. Such unintentional exposure primarily stems from improper data handling. A common source is improper train-test split, where evaluation/testing data leaks into training. For example, research has identified instances where training data incorporates bugs or vulnerabilities that closely mirror those encountered in testing [25, 164]. This concept of overlap is further supported by Liu *et al.* [82] highlighting the risks associated with functionality similarities between train and test data. Another dimension to consider is the origin of the data. For instance, as emphasized by Steenhoek *et al.* [134] and Wu *et al.* [164], using samples from the same software projects for both training and testing can influence the models. Even data structure and representation choices enable snooping. Liu *et al.* [88] and Shi *et al.* [129] emphasize that pitfalls can emerge from the data processing processes such as test prefix generation or concurrent use of same-class data.

**Spurious Correlations:** Spurious correlations arise when language models mistakenly depend on irrelevant artifacts rather than the code's intrinsic logic or intent for decision-making, leading to misleading associations. The artifacts vary across SE tasks. For instance, in vulnerability detection, artifacts may manifest as recurring code patterns or reliance on specific function names that LMs incorrectly associate with vulnerabilities [14, 134]. In code summarization, models might focus more on strings or certain code structures while overlooking elements key for developers [112]. When generating commit messages in the context of code review, models often produce outputs that adhere to a few simple patterns, potentially failing to capture the nuances of the actual code changes [26]. Actually, introducing advanced pre-trained models like CodeBERT has not eliminated these pitfalls. Specifically, if not fine-tuned appropriately for downstream tasks, these models might still overemphasize basic elements like keywords over richer code semantics [182].

**Inappropriate Model Design:** Inappropriate model design in LM4Code arises when the underlying architecture fails to capture critical characteristics of code, such as hierarchy and composition. The inability to construct robust semantic representations of code's intricate structural and logical attributes hinders model efficacy on downstream code intelligence tasks. Such design shortcomings can manifest in several ways. For instance, in vulnerability detection, models may exhibit a significant overlap in the feature space between classes, hindering precise vulnerability identification [14]. Code search models might lean on coarse-grained representations, capturing merely lexical or structural elements, often overlooking the true functionality of the code [153]. Similarly, the encoder-decoder framework used in code summarization might neglect the hierarchical nature of code or struggle with sequence generation, leading to inadequate summaries [151, 159]. This issue is not limited to traditional architectures. Even modern program repair models, adapted from neural machine translation, face design-related challenges that affect their translation accuracy and diversity [25, 96]. While there are innovative attempts such as leveraging deep reinforcement learning or shared encoder-decoder architectures [79], these approaches still exhibit shortcomings in addressing the diverse needs of various LM4Code applications.

To summarize, Figure 6 and 7 display the distribution of papers that discuss LM4Code pitfalls in the system design and learning stage. Figure 6 indicates that while inappropriate model design was first identified in 2018, research efforts on addressing key pitfalls have increased over the past three years. Among these, data snooping has garnered increasing research attention. Meanwhile, spurious correlations have become more prominent with the advent of explainable artificial intelligence (XAI) techniques for elucidating model reasoning [20, 141, 143]. Discussions around inappropriate model design remain ongoing as new frameworks and learning strategies continue to emerge. Contrary to our observations regarding the data collection and labeling process, Figure 7 reveals a greater

emphasis on modern Transformer-based language models compared to conventional architectures. Specifically, 19% of relevant studies employ BERT, 15% leverage general Transformer models, 11% utilize CodeBERT, and 7% investigate CodeT5. This distribution highlights a shift towards examining potential pitfalls in sophisticated language models for code intelligence tasks, setting the stage for continued research focused on enhancing model transparency, interpretability, and reliability.

## 4.2 RQ2-Implications

In the system design and learning phase, pitfalls emerge that can distort LM4Code outcomes. Following the pitfalls identified in the previous phase, these design-related pitfalls similarly lead to performance overestimation and compromised model efficacy. We will now elaborate on the specific impacts of these pitfalls and how they manifest in various software engineering tasks.

**Performance Overestimation:** Pitfalls in system design and learning can lead to over-optimistic performance metrics for LM4Code models. Data snooping is a major contributor to this overestimation. For example, the presence of overlapping functionality between training and test sets can elevate the F1 score of a clone detection model from 0.42 to 0.96 [82]. In vulnerability detection, a mix of projects in both training and testing phases can introduce discrepancies in F1 scores as large as 0.32 [134]. Spurious correlations represent a subtler and often more elusive challenge. These pitfalls cause models to make correct predictions, but often for the wrong reasons, leading them to rely on irrelevant code patterns or unrelated artifacts. This not only misleads performance interpretation but also makes the models unreliable in varied scenarios [134]. Models might also give undue attention to superficial code constructs, leading to inefficiencies that don't necessarily enhance outcomes [182].

**Compromised Model Efficacy:** While overestimation impacts perceived performance, inappropriate model design directly compromises the efficacy of models in practical scenarios. For example, token sequence-based vulnerability detection models might fail to capture the underlying causes of vulnerabilities and instead focus on surface-level patterns, leaving a significant margin for improvement. Chakraborty *et al.* [14] highlighted this limitation and showed that the use of gated graph neural networks can improve the baselines by 33.57% in precision and 128.38% in recall. Similarly, the design limitations in code search models lead them to generate coarse-grained representations that may overlook core functionalities [153]. Program repair models, based on neural machine translation models, have shown slow learning curves, achieving a mere 4.5% repair prediction accuracy even after 10 epochs [25]. This indicates that the structural design of the model hinders its ability to learn and adapt efficiently.

## 4.3 RQ3-Solutions

To address the three pitfalls related to the system design and learning process, researchers have employed a variety of approaches which we describe as follows.

**Refined Data Handling:** To mitigate the challenges posed by data snooping, a multifaceted approach is essential. Emphasizing rigorous data partitioning is foundational, as exemplified by Steenhoek *et al.* [134], who advocate for cross-project validation to prevent inadvertent overlaps between training and testing sets. Additionally, techniques like data augmentation and time-based splits can further insulate models from over-relying on specific pattern [42, 115]. Incorporating regularization techniques, such as batch normalization, can curb overfitting and deter models from exploiting inadvertent data correlations [45]. Finally, the importance of external validation, underscored by Liu *et al.* [82], ensures that performance assessments are unbiased and reflective of real-world scenarios. By adopting these strategies, researchers can foster more reliable and generalizable outcomes.

**Model Interpretability:** To address the biases inherent in LM4Code, especially spurious correlations, improving model interpretability has emerged as a crucial solution [141]. By examining the decision-making process of LM4Code models, researchers are better positioned to pinpoint and mitigate pitfalls, leading to more reliable predictions [46]. Within vulnerability detection, Fu *et al.* [36], Li *et al.* [75], and Zou *et al.* [187] proposed methods to enhance explanation accuracy, leveraging sophisticated visualization tools to correlate the internal dynamics of neural models with code structures, thus providing a comprehensive understanding of model reasoning. Cito *et al.* [21] offers a distinctive perspective, centering on elucidating mispredictions. Their approach, which integrates neural predictions with symbolic logic, allows for precise error detection accompanied by rule-based explanations. Additionally, attention mechanisms to explain pre-trained models have also been analyzed. While Shi *et al.* [130] unravels how transformer-based models allocate attention for code summarization, Wan *et al.* [150] probes into the nuances of attention during code-to-code translation. These XAI approaches can serve to identify and rectify model pitfalls, ensuring the reliability of LM4Code applications.

**Model Optimization Strategies:** In light of the pitfalls introduced by inappropriate model design, researchers have turned to model optimization strategies to address and minimize their effects. These strategies encompass several techniques designed to enhance a model's structure, training process, and generalization capabilities. Firstly, Model Design Adjustments involve refining the architecture to better capture data intricacies. Studies like that by Wan *et al.* [151] have demonstrated the benefits of introducing novel layers or structures to better understand the tree structure of code, yielding improved performance. Secondly, Model Ensembling is gaining traction, where the strengths of multiple models are leveraged to offset individual biases, as seen in the work by Zhang *et al.* [182] which employs multiple views of the same data for more robust predictions. Lastly, Regularization and Fine-tuning techniques play a pivotal role. Regularization, such as dropout or L2 regularization, helps in preventing overfitting, while fine-tuning allows pre-trained models, like CodeBERT, to adapt to specific dataset nuances, as demonstrated by Fang *et al.* [31]. By integrating these strategies, models can be better positioned to achieve superior outcomes.

#### Summary - System Design and Learning

In this study, we uncover 16 research studies related to pitfalls in system design and learning. These pitfalls can be categorized into three main categories: data snooping, spurious correlations, and inappropriate model design. These pitfalls lead to overestimated performance and compromised efficacy of LMs. Proposed solutions encompass refined data handling, model explainability, and optimization strategies like architecture adjustments, ensembling, and regularization.

## 5 PERFORMANCE EVALUATION

The performance evaluation stage focuses on precisely assessing and analyzing the model's performance using predefined test sets and evaluation metrics. Additionally, comparative performance evaluation against benchmarks provides insights into a model's strengths and weaknesses on specific code-related tasks. However, potential pitfalls can emerge from factors such as improper baselines, test sets, and performance metrics. These challenges must be thoroughly examined and addressed to ensure that the evaluation is unbiased, comprehensive, and representative of a model's true capabilities. Thus, this section provides a brief description of related studies and discusses the implications and potential solutions during the performance evaluation stages.

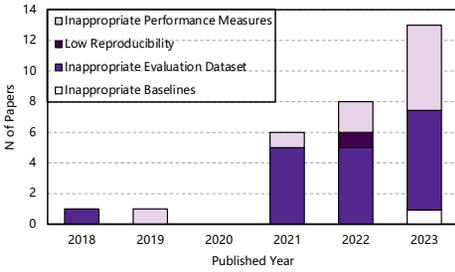


Figure 8. Paper distribution across time (Section 5)

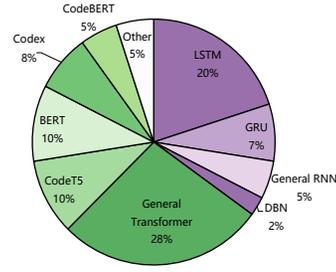


Figure 9. Distribution of LMs (Section 5)

### 5.1 RQ1-Pitfalls

From the collected studies, we identified 29 research studies focusing on pitfalls during the performance evaluation phase. We have methodically categorized the collected literature into four categories, as shown in Figure 8.

**Inappropriate Baseline:** Inappropriate baselines arise when the performance evaluation for LM4Code is carried out without, with limited, or with skewed baseline approaches. Such poor comparisons fail to convincingly demonstrate the improvements or strengths of newly proposed LM4Code approaches, potentially leading to misguided experimental findings or exaggerated efficacy claims. For instance, comparing only against basic rule-based approaches would inflate capabilities, while limiting comparisons to other advanced LM-based approaches masks potential weaknesses. Liu *et al.* [88] highlight this issue in state-of-the-art Neural Test Oracle Generation (NTOG) evaluation, where the lack of a straightforward baseline causes considerable gaps between reported and real-world performance.

**Inappropriate Evaluation Datasets:** Inappropriate evaluation datasets refer to the use of unsuitable, non-representative, or limited test sets that fail to adequately represent the true complexities of the studied tasks. Such test sets can provide misleading results, and offer an obscured, over-optimistic, or pessimistic view of a model's realistic capabilities. Liu *et al.* [83] highlight the frequent use of small, non-representative, and non-diverse datasets in code generation evaluations that fail to capture real-world software complexities. Specifically, the widely-used HS dataset [80] comprises code from a single project, exhibiting poor diversity. The average code length in the Django dataset [108] is a mere 33 characters, indicating limited program complexity. The CoNaLa dataset [174] utilizes automatically extracted Stack Overflow questions instead of real software requirements. In short, common benchmarks poorly approximate the complexities of realistic code generation scenarios. In vulnerability detection, Liu *et al.* [84] highlight that most evaluations make the assumption that training and test datasets are drawn from the same distribution. However, they argue this assumption overlooks the continuous evolution of software vulnerabilities and projects, leading to the Cross-Domain issue where test sets should contain novel vulnerabilities or projects. Furthermore, Nong *et al.* [106] further reveal that vulnerability datasets like SARD [12] comprise unrealistic synthetic examples, which exhibit smaller vocabulary, smaller program length, and higher pattern frequency compared to real-world code. Zeng *et al.* [176] claim that the state-of-the-art Just-in-Time (JIT) defect prediction tool, CC2Vec [50], was only evaluated on a limited dataset with marginal improvements to demonstrate generalizability and scalability. Overall, common benchmarks in code-based research utilize inappropriate test sets that fail to capture real-world complexities, leading to unrealistic performance evaluation.

**Low Reproducibility:** Reproducibility is a fundamental requirement of scientific research to verify the validity and generalizability of findings through consistent replication across settings [56,

71]. However, LM-based research introduces specific challenges around efficiently storing big data, communication between distributed clusters, algorithm implementation, and appropriate software/hardware environments [60, 71]. Thus, low reproducibility in LM4Code significantly impacts performance evaluation, affecting the generalizability and trustworthiness of findings. Zeng *et al.* [175] conducted an extensive empirical study revealing reproducibility issues with pre-trained LMs for program understanding and generation. As described in their study, multiple performance comparison results can be reversed compared to original publications. For example, CodeGPT incurred the largest variance across 5 clone detection runs under the benchmark BigCloneBench dataset, able to outperform or underperform other models depending on the run. Although PLBART reported a 0.7 F1 advantage over CodeBERT in the original paper, its min-max F1 variations of 0.84 make this finding questionable. Additionally, CodeBERT and PLBART's defect detection accuracy reversed between the original PLBART paper and the subsequent research by Zeng *et al.* [175]. Such statistically significant variations could invert performance comparisons, thereby undermining LM4Code's trustworthiness.

**Inappropriate Performance Measures:** Inappropriate performance measures arise from a mismatch between standardized metrics and the distinct challenges inherent to software engineering tasks. Prior survey studies [51, 157] have shown widespread use of generic metrics like accuracy, precision, BLEU, and Pass@k in LM4Code research. However, these metrics often provide an incomplete view of model capabilities on code-based tasks, owing to the complex and multifaceted nature of software engineering scenarios. For instance, Roy *et al.* [123] conducted an empirical study with 226 human annotators and showed that popular metrics like BLEU and ROUGE are poor reliable indicators of human judgment. This demonstrates how common metrics may not fully align with code quality assessments. Additionally, when evaluating classification tasks like vulnerability detection, reporting solely accuracy is insufficient, as true-positive and false-positive decisions are not observable [14, 88]. Moreover, popular metrics like BLEU derived from natural language processing overlook critical attributes of code. As a textual similarity metric, BLEU calculates n-gram precision between generated and reference sentences. However, programming languages contain many “trivially shared n-grams”, rendering BLEU ineffective at distinguishing actually similar code from coincidental similarities [28]. As we discussed before, code-based tasks exhibit multiple distinct challenges such as data imbalance, data snooping, execution correctness, and exception handling. For example, despite high BLEU scores, CodeT5 generated code with only 6.4% compilation rate, indicating poor execution correctness [164]. Overall, it is critical to utilize appropriate evaluation metrics in LM4Code research to provide a comprehensive and accurate understanding of model performance.

Figure 8 presents the distribution of relevant literature across years, while Figure 9 summarizes the distribution across different LMs. Between 2018 and 2023, there was a noticeable rise in the number of papers, suggesting a growing focus by researchers on pitfalls in LM4Code during the performance evaluation phase. Among the identified studies, inappropriate evaluation datasets have the most mentions, with 19 studies, showing the difficulty in establishing representative datasets of LM4Code for reliable model performance. Additionally, inappropriate performance measures also garnered considerable research attention, with 6 identified studies. From Figure 9, we can observe that Transformer-based models were most discussed, evidencing their increasing adoption for SE tasks. In summary, these observations emphasize critical areas for improvement to enable robust evaluation and unbiased analysis of capabilities as LM4Code evolves. Careful consideration of evaluation datasets, metrics, and baselines will be integral to advancing progress in the field.

## 5.2 RQ2-Implications

Through a thorough examination of 29 relevant studies, our review has identified several pitfalls present in current approaches to evaluating the performance of LM4Code. If left unaddressed, these pitfalls can produce misleading or unrealistic results, skewing perceptions of how these models might perform in real-world settings. In this section, we discuss the broader implications of the identified evaluation pitfalls and their potential impacts on LM4Code research and practical applications.

**Performance Overestimation:** Pitfalls during performance evaluation can lead to a concerning overestimation of model capabilities, creating a misleading gap between reported metrics and real-world performance. Key contributors include the use of inappropriate and unrealistic evaluation datasets and improper performance measures misaligned with practical objectives. Small and synthetic evaluation datasets often fail to adequately represent the true complexities that models face in real-world code intelligence tasks. Yet these limited datasets yield seemingly “perfect” metrics during evaluation. As Liu *et al.* [83] empirically showed, the capabilities of code generation models are often overestimated due to the use of limited datasets. When evaluated on the small Django [108] and HS [80] datasets, code generation approaches yielded strong BLEU scores of 0.811 and 0.646 respectively. However, simply switching to a new and more practical dataset led to a drastic BLEU score drop to 0.167, revealing how exaggerated initial metrics can be. Similarly, performance metrics misaligned with practical objectives tend to provide an unrealistic overestimate of capabilities. These inflated metrics collapse when models face the true complexities of real-world deployment. As Ahmed *et al.* [3] showed, as the length of input tokens increases for program repair tasks, the accuracy of language models can decrease substantially from 82.88% to 55%. While overall accuracy may seem high during evaluation, metrics fail to reflect significant drops in certain practical scenarios. Furthermore, popular metrics like BLEU can fail to accurately assess model capabilities, as demonstrated by Eghbali *et al.* [28]. Their analysis showed BLEU’s shortcomings in distinguishing between a neural code generation model and a dummy model that simply exploited common n-grams. Despite the dummy model producing low-quality code, BLEU scored it equivalent to the neural model. This indicates BLEU’s inability to differentiate between models genuinely solving complex tasks and those exploiting superficial patterns. Overall, inappropriate evaluation practices systematically and significantly overestimate the capabilities of LM4Code models, obscuring major gaps between reported metrics and real-world effectiveness. More rigorous and realistic benchmarking is critical.

**Compromised Reproducibility:** The reproducibility of research findings through independent verification is a fundamental requirement of the scientific process that allows reported improvements to be validated and incrementally built upon [56]. However, our analysis reveals that performance evaluation in LM4Code research may lack sufficient reproducibility. The proposed approaches frequently bypass consistent evaluation protocols, datasets, and implementation details, making it difficult for others to replicate the experiments described in the literature and validate their accuracy [175]. Furthermore, inconsistent performance across different runs or implementations obscures the reliability of findings, as initial results that appear promising frequently fail to fully replicate in subsequent studies [175]. This phenomenon indicates that minor variances in experimental conditions, which are rarely comprehensively reported, can significantly influence outcomes. Limited methodological transparency through selective reporting of details further inhibits reproducibility [63].

**Misleading Benchmarks:** Misleading benchmarks and exaggerated claims lead research communities astray and substantially hinder actual progress [147]. For instance, evaluating models on limited datasets or with improper metrics often amplifies perceived improvements beyond actual

capabilities. However, identifying possible pitfalls remains an open challenge for future LM4Code researchers to address. As Liu *et al.* [88] highlighted, the lack of appropriate baselines frequently causes considerable gaps between reported and real-world performance. Promoting such flawed evaluations through publications and conferences propagates unreliable techniques built on shaky foundations. This questionable practice squanders precious community resources as researchers may pursue exaggerated claims rather than meaningful progress. More alarmingly, it obscures the path forward for meaningful innovation that solves real-world needs, as progress is measured on inflated claims instead [147]. Ultimately, systemic misleading benchmarks threaten to steer LM4Code research astray, leading the community away from impactful advancements. Establishing rigorous and peer-validated standards for benchmarking to guide productive research directions is an urgent need.

### 5.3 RQ3-Solutions

Given the pitfalls identified during the performance evaluation phase, it is essential to propose solutions to address these challenges and optimize the evaluation process. We summarize the existing solutions as follows.

**Standardized and Realistic Benchmarks:** Establishing robust and realistic benchmarks is critical for rigorous performance evaluation in LM4Code research. Researchers should carefully select appropriate baselines based on thorough literature analysis, opting for well-established approaches tailored to specific code intelligence tasks. This enables comprehensive capability assessment. Moreover, the community must collaboratively institute standardized benchmarks and protocols, fostering consistent evaluation. For example, Liu *et al.* [88] highlight the need for establishing straightforward and realistic baselines to ensure truthful evaluation outcomes. For specific tasks, standardized task-specific benchmarks have emerged, such as VJBench proposed by Wu *et al.* [164] for automated program repair. Such domain-specific benchmarks facilitate targeted capability assessment. Furthermore, diverse benchmark datasets play a significant role in comprehensively evaluating and advancing LM4Code. For instance, Lu *et al.* [92] introduced CodeXGLUE, a comprehensive collection that encompasses 10 SE tasks across 14 datasets. It provides a platform for structured model evaluation and comparison across tasks using standardized baselines.

**Enhancing Reproducibility:** Inconsistent replication remains a key pitfall during performance evaluation in LM4Code [60]. Some solutions to enhance reproducibility have been proposed. At its core, it's critical that code, datasets, and evaluation scripts are made publicly accessible, echoing the call for open-source initiative and providing continuously maintained links to high-quality replication packages [81]. Furthermore, reporting key statistical measures like variance, confidence intervals, and significance quantifies stability across runs. Such transparency fosters community collaboration, facilitating the validation of research outcomes and paving the way for future scholarly advancements. Carefully controlling and reporting experimental conditions like hardware specifications, software versions, random seeds, and tuning details significantly influences outcomes [55]. Chen *et al.* [16] introduced a systematic approach to train reproducible AI systems, with general criteria to evaluate reproducibility, and a unified framework to decrease randomness.

**Refined Performance Measures:** Appropriate and comprehensive performance metrics are critical for accurately evaluating model performance on complex code-related tasks. To address the limitations of common generic metrics like BLEU and accuracy, researchers have proposed more refined measures tailored to LM4Code challenges. Ren *et al.* [121] introduced CodeBLEU to improve upon BLEU for assessing deeper semantic similarities in code generation tasks, while Eghbali *et al.* [28] introduced CrystalBLEU for minimizing the noise from commonly shared n-grams in programming languages. Given the runtime nature of programming languages, metrics

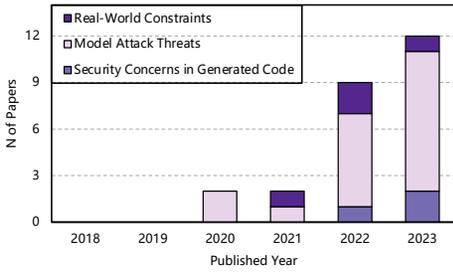


Figure 10. Paper distribution across time (Section 6)

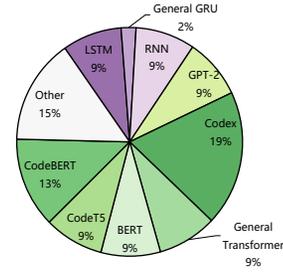


Figure 11. Distribution of LMs (Section 6)

like compilation rate and execution accuracy on test cases have also been used [164]. For specific LM4Code tasks, domain-specific metrics can prove more reliable. For example, for test oracle generation, Liu *et al.* [88] presented an evaluation metric named Found@K, which counts how many bugs can be found if developers only check the top-K recommended test cases per bug. To better capture complex code quality attributes, direct human assessment can complement automated metrics, evaluating aspects like readability, conciseness, and correctness [24, 112]. Overall, more refined, code-aware, and multifaceted performance measures aligned with practical goals provide a more accurate and reliable model capability assessment.

### Summary - Performance Evaluation

In the performance evaluation of LM4Code, we identify four pitfalls related to performance evaluation, including inappropriate baseline, inappropriate evaluation datasets, low reproducibility, and inappropriate performance measures. From 29 relevant research studies, inappropriate datasets and metrics receive much attention. Such pitfalls can lead to overestimated evaluation and compromised reproducibility, misleading benchmarks for the future. To address this, tailored solutions like standardized benchmarks, transparency, realistic assessments, and community coordination are needed.

## 6 DEPLOYMENT AND MAINTENANCE

Systems based on advanced LM4Code, such as GitHub Copilot [40], Amazon CodeWhisperer [4], and Microsoft IntelliCode [139], have already been deployed in real-world IDEs and have garnered a large number of users. There exist various challenges when such LM4Code systems are deployed in practice, like security threats and how LM4Code systems should be updated to adapt the rapidly changing software practices. This section discusses pitfalls, implications, and current solutions when deploying and maintaining LM4Code.

### 6.1 RQ1-Pitfalls

In our literature review, we identified 25 research papers involving the pitfalls of LM4Code deployment and maintenance. This number of identified publications surpasses that related to other stages, highlighting the fact that deploying and maintaining LM4Code presents a complex set of challenges that have gained considerable attention from the research community.

Figure 10 and Figure 11 present the distribution of these 25 research studies over time and different types of LLM4Code. From Figure 10, we can observe that there has been a significant increase in the number of research papers focusing on the pitfalls in the deployment and maintenance of LM4Code, especially in the past two years. The data presented in Figure 11 demonstrates a significant preference for utilizing the Codex model for analyzing biases related to deployment and

maintenance, which may be due to the fact that the widely used Github Copilot is based on Codex. Moreover, large pre-trained models such as CodeT5, BERT, Codex, and GPT-2 appear to surpass traditional LMs such as RNN and LSTM in terms of widespread use and popularity. These observed patterns indicate a shift in research focus towards investigating the complexities of advanced models when deployed in real-world contexts. In the subsequent sections, we will examine the particular biases and issues that arise in regard to the development and maintenance of LM4Code.

**Real-World Constraints:** Evaluating LM4Code systems solely in controlled lab settings often overlooks practical constraints and complexities of real-world deployment [6]. Although controlled evaluations provide useful insights into the effectiveness of a model within specific settings, they sometimes overlook the complexity and diversity of real-world deployments. It is important to consider runtime and storage constraints when deploying an LM4Code system in the wild. Svyatkovskiy *et al.* [140] mentioned that a reasonable upper bound of model size for an IDE plugin is 50 MB. However, the size of LM4Code keeps increasing. For example, the popular LM4Code, CodeBERT [34], has 125 million parameters and is 425 MB in size, which is way larger than the suggestion by Svyatkovskiy *et al.* [140]. Recently proposed models are even larger. Considering LLaMa, developed by Meta, its smallest version has 6.7B parameters [91]. Models that appear to have good performance in a controlled setting may be resource-intensive or excessively large for practical applications, especially in environments with limited computational power or storage capacity [33].

**Attack Threats:** LM4Code systems face various threats from malicious attackers. Despite the breakthrough capabilities exhibited by LMs, prior works have noted that the state-of-the-art LMs are vulnerable to a variety of attack threats such as evasion attacks [95, 154, 185], backdoor attacks, privacy attacks, etc. These threats can appear across the entire lifecycle of the model, from data collection to model deployment. We identified a total of 17 research papers that specifically focus on the analysis of security threats within the LM4Code system. Figure 10 shows that the number of these studies has increased over the recent years. Our study employs the classification scheme mentioned in Battita *et al.*'s work [11], which has identified a diverse range of attack types in machine learning.

- (1) *Evasion Attacks:* Evasion attacks (a.k.a. adversarial attacks) leverage adversarial samples [85, 185], which are carefully perturbed instances while appearing as regular and benign inputs to the human observer. Yet, these small perturbations can mislead the trained LM4Code model to produce incorrect predictions. As shown in Table 2, evasion attacks are a major concern, with 9 out of the 17 examined papers investigating them. Zeng *et al.* [175] thoroughly evaluated eight different adversarial attack approaches (i.e., word importance rank, genetic algorithm, random substitution) against widely-used LM4Code, including CodeBERT, GraphCodeBERT, and CodeT5. Their findings emphasized the vulnerability of these models to semantics-preserving adversarial samples. Interestingly, even simple random attack techniques showed significant effectiveness in degrading pre-trained LM4Code.
- (2) *Poisoning Attacks:* Poisoning attacks inject malicious examples into the training data, aiming to manipulate the model's behavior [11, 137]. This can lead the model to produce incorrect or attacker-chosen outputs when certain triggers appear in the inputs [125]. Poisoning attacks can be categorized into two main classes: untargeted poisoning attack and targeted poisoning attack [146]. One special case of the target poisoning attack is the backdoor attack, where adversaries carefully insert a distinct pattern into a subset of training samples to embed a backdoor. When the pattern appears, the model produces pre-defined outputs (e.g., recommending vulnerable APIs). Otherwise, the model behaves normally. Table 2 lists six

	Category	Percentage	References & Details
<b>Attack Objectives</b>	Evasion attacks	59%	[38, 48, 85, 158, 170, 175, 179, 180, 185]
	Poisoning attacks	35%	[74, 116, 125, 135, 137, 149]
	Privacy/confidential attacks	6%	[93, 105]
<b>Attack Models</b>	RNN-based (e.g., LSTM)	41%	[38, 48, 125, 149, 179, 180, 185]
	General Transformer	29%	[38, 149, 158, 175, 185]
	LLM (e.g., Codex and T5)	41%	[74, 93, 105, 116, 125, 135, 137, 175]
<b>Countermeasure-inclusive</b>	Include	71%	[38, 48, 85, 93, 125, 135, 137, 149, 170, 179, 180, 185]
	Non-include	29%	[74, 105, 116, 158, 175]

Table 2. Summary of Attack Threats in Reviewed LM4Code Papers

papers related to poisoning attacks, with all of them specifically focusing on backdoor attacks. Among various code scenarios, three studies [135, 137, 149] target LMs-based code search models, demonstrating that backdoor samples within code search tasks closely resemble clean code and are not easily differentiated. Four studies [74, 116, 125, 137] focus on code generation scenarios, demonstrating that attackers can manipulate code recommendations.

- (3) *Privacy Attacks*: Privacy attacks refer to attacks aiming to infer the private information of LM4Code, e.g., parameters of models that are hosted remotely and the model training data. One such example is model stealing (a.k.a. model extraction) involves extracting knowledge (e.g., hyperparameters, model architecture, and training data) from a trained model without direct access to its parameters or training data [13, 110]. Adversaries use this technique to ‘copy’ the functionality of a model, often by querying it repeatedly and analyzing the outputs [110]. For example, Lukas *et al.* [93] delved into the risk of language models, like GPT-2, leaking personally identifiable information. Yang *et al.* [171] found that simply extracting 20,000 outputs (each having 512 tokens) from CodeParrot [1] can produce over 40,125 code snippets that are memorized from its training data. In addition, Niu *et al.* [105] introduced and evaluated a semi-automated pipeline that employs a membership inference approach on various code generation models like CodeParrot [1] and Polycoder [166]. By leveraging the GitHub Search API’s hit rate as a distinguishing heuristic and incorporating human-in-the-loop evaluations, they found that approximately 8% (43) of the prompts in the Codex model, used in GitHub Copilot, resulted in privacy leaks.

**Security Concerns in Generated Code:** The outputs from LM4Code, i.e., generated code, will be further used in other software systems. Consequently, the safety and robustness of the generated code come under scrutiny. Several recent investigations have shown that LM-generated code can contain vulnerabilities, emphasizing the need for rigorous validation and enhancement of their outputs. Pearce *et al.* [113] analyzed the generated code of GitHub Copilot and identified security vulnerabilities. The authors produced 89 different scenarios for Copilot to complete, resulting in 1,689 programs. Alarmingly, they found approximately 40% of these to be vulnerable. Such vulnerabilities arise because code often contains bugs, and given the vast quantity of unvetted code that Copilot processes, the language model is prone to learning from exploitable and buggy code.

## 6.2 RQ2-Implications

According to the reviewed papers, the identified pitfalls and security concerns related to the deployment and maintenance of LM4Code have significant implications for both researchers and practitioners in artificial intelligence and software engineering. Next, we will describe the implications in detail.

**Untrustworthy Results:** Both the pitfalls from external attackers and internal drawbacks can lead to untrustworthy outputs, requiring the developers to carefully review and test the generated code. External attacks like evasion, poisoning, and backdoor attacks, can induce LM4Code systems to demonstrate manipulated behavior, i.e., generating outputs specifically chosen by attackers. For example, Schuster *et al.* [125] demonstrate that code completion models can be manipulated to use unsafe encryption algorithms and deprecated security protocol so that the system built based on these suggestions is vulnerable to further attacks. Aghakhani *et al.* [2] reveal that the backdoors in code generation models can inject various insecure code, including Cross-Site Scripting (CWE-79), Path Traversal (CWE-22), and Deserialization of Untrusted Data (CWE-502). Wan *et al.* [149] can mislead the code search models to retrieve the code snippets containing malicious actions, such as deleting specific files. Attackers can also provoke systems into producing unexpected predictions so that the application behaves in an unintended way. For example, the adversarial attacks [170, 181] against LM4Code Systems can lead to completely wrong predictions. Even without external attacks, the model itself may also naturally generate untrustworthy results, which has been largely reported in various studies. For instance, Pearce *et al.* [113] revealed vulnerabilities frequently present in GitHub Copilot’s generated code. Furthermore, Lukas *et al.* [93] suggested LMs may inadvertently retain or even leak portions of their training data, which raises concerns regarding unintentional memorization of sensitive information. Niu *et al.* [105] investigated code completion models like the popular Codex model leaking private information (e.g., passwords and addresses) through generated outputs.

**Copyright Infringement:** Pitfalls in LM4Code deployment raise significant concerns regarding copyright infringement and loss of intellectual property. Attackers might resort to model stealing, effectively replicating the functionality of proprietary models without authorization [62, 78]. Li *et al.* [78] demonstrate that imitation models can even exceed the performance of the victim model. Additionally, when models are deployed on user clients, they face the potential threat of reverse engineering. Zhou *et al.* [184] highlight that the on-device models may leak their confidential information, such as hyper-parameters and weights. The risk of copyright infringement and intellectual property theft not only undermines incentives to develop innovative models, but also threatens the commercialization prospects and trustworthiness of the LM4Code industry.

**Compromised Model Efficacy:** In real-world practice, many constraints, such as latency, device memory, and computational costs, need to be satisfied. On the one hand, it prevents some powerful models from being deployed. For instance, Feng *et al.* [33] noted multiple malware detection models are infeasible on mobile devices due to limitations in computational power, memory, and energy consumption. Zhang *et al.* [182] highlighted that even standard CodeBERT requires extensive resources for pre-training and fine-tuning before use, which may not be affordable for individual developers. On the other hand, the performance of deployed systems may be compromised as a trade-off. For example, to be deployed on the user devices, the models have to be pruned or distilled, where considerable performance degradation can be observed [131, 132]. Furthermore, Ganesh *et al.* [37] showed that while quantization and unstructured pruning can reduce model size, these techniques alone do not improve runtime inference speed or memory usage during execution, unless paired with specialized hardware or processing libraries. Therefore, while constraints exist in real-world deployment that compels simpler models, such compromises often degrade model performance, highlighting the need for techniques that can provide comparable accuracy with greater efficiency.

## 6.3 RQ3-Solutions

**6.3.1 Improving Model Robustness.** Improving model robustness is essential to defend against evolving security threats targeting LM4Code systems. Here, we summarize some of the available solutions that have been proposed to enhance the robustness of LM4Code models:

**Adversarial Training:** Adversarial training incorporates adversarial examples into training data to improve model robustness against adversarial attacks. Regarding code comment generation, Zhou *et al.* [185] demonstrated masked adversarial training can significantly enhance robustness while retaining performance on normal test data. Zhang *et al.* [180] illustrated that adversarially trained language models exhibit markedly reduced attack success rates (approximately 30%-40%). Moreover, Henkel *et al.* [48] implemented robust optimization employing a semantics-preserving adversary, and this approach outperformed standard data augmentation, optimally balancing accuracy on clean samples with robustness to perturbations.

**Inference with Self-repair:** Self-repair is to let the model introspect and correct mistakes or vulnerabilities in its own code. Delving into this, Olausson *et al.* [109] investigated to what extent the GPT models can provide accurate feedback on the causes of code errors, validating the efficacy of GPT-4's self-repair capabilities. Moreover, Chen *et al.* [15] enhanced the self-repair process by incorporating knowledge from human-written natural language feedback.

**Domain Knowledge:** Beyond the internal capability of the model, external aids with domain knowledge can also be employed to address the challenges. For instance, Jain *et al.* [57] enhances these LLMs by utilizing a post-processing step based on program analysis and synthesis techniques, thereby improving the correctness of the syntax and semantics in code. Wei *et al.* [162] introduced Repilot, a framework designed to further assist AI "copilots" (i.e., LLMs) by synthesizing more valid patches during the repair process. Moreover, Johnson *et al.* [62] proposed Random Utility-Driven Synthesis Under Uncertain Regions (R-U-SURE), a method that builds uncertainty-aware suggestions based on a decision theory model of objective conditional utility, using random samples from generative models as proxies for unobserved potential intents of end-users. Poesia *et al.* [117] propose a framework to apply constraints on partial outputs to produce complete correct programs without re-training or fine-tuning of the language model.

**Train with Real-world Dataset:** Directly learning from the real-world datasets helps the model to better align with the user intention. For instance, Aye *et al.* [9] observed a significant decline in the accuracy of Transformer sequence models when tested using real-world data from production logs. Moreover, they demonstrated that training on real-world examples yields a more robust model. However, there is a significant shortage of large-scale, real-world datasets, especially for vulnerability. Nong *et al.* [106] suggest a promising alternative solution: using deep learning to generate real-world samples.

**6.3.2 Enhancing Computational Efficiency.** To make full use of the limited resources, various measures have been proposed.

**Model Compression and Specialization:** To enable the deployment of large language models in resource-constrained environments, model compression techniques have been widely explored [44, 118]. Shi *et al.* [132] introduced Compressor, which uses a genetic algorithm to guide the simplification of pre-trained code models. This compressed models to significantly smaller sizes with acceptable accuracy loss. In addition to compression, methods to specialize large language models (LLMs) for specific tasks show promise. Parameter-Efficient Fine-Tuning (PEFT) effectively adapts LLMs using limited task data. As Weyssow *et al.* [163] demonstrated, PEFT outperforms approaches like Incremental Curriculum Learning in reducing computational overhead and boosting performance when specializing broad LLMs.

**Efficient Inference:** Numerous inference optimization strategies have also been proposed. For instance, Zhang *et al.* [182] adopted input simplification strategies like word dropout and frequency filtering to simplify input programs. By focusing on the most informative tokens, computational costs are significantly reduced. Additionally, Chirkova *et al.* [19] introduced a tokenization method that reduces the average token length by 17% without any downstream performance loss. They further demonstrated that a properly chosen tokenization can even enhance the model's performance by 0.5-2%. Furthermore, Svyatkovskiy *et al.* [140] unveiled an innovative neural completion model by combining static analysis with granular token encoding. This model boasts a lean memory footprint, consuming just 6 MB of RAM — a significant 19x reduction compared to previous models. It can generate a single piece of code completion in a mere 8 ms and delivers an impressive 90% accuracy rate for its top five suggestions.

**6.3.3 Privacy and Copyright Protection.** The ability of large language models to memorize and reproduce training data raises critical privacy and copyright concerns. Thus, developing effective protection for privacy and copyright has been widely investigated.

**Privacy-preserving Techniques:** The memorization and regeneration capabilities of large language models raise critical privacy concerns that demand research attention. Some state-of-the-art models like StarCoder employed human annotators to mask any personal information such as keys and addresses present in the training data, in an effort to mitigate privacy risks [72]. In addition, differential privacy techniques have emerged as promising strategies for mitigating privacy risks. Lukas *et al.* [93] introduce a sentence-level differential privacy approach, which provides guarantees under the assumption that records are unlikely to be duplicated. Their results show that while helpful in reducing privacy leakage to a large extent, differential privacy alone cannot completely eliminate risks. Further research into complementary privacy-preserving mechanisms is needed to develop LM4Code that generates high-quality outputs while provably protecting user privacy.

**Model Obfuscation:** Obfuscating models by hiding their structure and parameters has been proposed as a technique to protect against extraction attacks. Prior Research has shown that attackers can easily craft white-box-like attacks against models on devices, even to the extent of reversing their training data [53]. For example, Zhou *et al.* [184] developed ModelObfuscator to apply techniques like model file obfuscation and model structure obfuscation. In model file obfuscation, they utilized renaming, parameter encapsulation, and neural structure obfuscation approaches, effectively obfuscating the data and structures of on-device models. model structure obfuscation utilizes shortcut and extra layer injection, making reverse engineering harder. Although model obfuscation has shown promise by Zhou *et al.* [184], it can increase library size, introducing extra memory overhead and computation time. Further research is needed to balance security, efficiency, and accuracy as model obfuscation techniques are applied.

**Watermarking:** Watermarking techniques can help protect model intellectual property. For instance, Sun *et al.* [137] introduced CoProtector, which uses data poisoning to embed watermarks into source code repositories. This ensures open-source code can't be exploited by models while providing a way to reveal watermark backdoors. However, CoProtector notably diminishes model performance, making it hard to adopt broadly. In neural code completion, Sun *et al.* [136] proposed CodeMark which embeds imperceptible user-defined watermarks into code. This traces code utilization while meeting key watermark properties like harmlessness, verifiability, and robustness.

### Summary - Deployment and Maintenance

Based on 25 relevant studies, our literature review reveals three main pitfalls in deploying and maintaining LM4Code systems: real-world constraints, attack threats, and generated code containing security concerns. These pitfalls can lead to untrustworthy results, copyright infringement, and reduced model efficacy. Proposed solutions involve improving robustness through adversarial training and error detection, enhancing efficiency via compression and optimized inference, and protecting privacy and copyright through obfuscation and watermarking. However, there remains a need for robust evaluation frameworks and techniques that balance security, efficiency, and accuracy. The implications highlight that real-world deployment introduces complex challenges for LM4Code systems.

## 7 DISCUSSION

### 7.1 Recommendations for LM4Code Research

Our systematic literature review reveals numerous pitfalls that can undermine the realistic performance and real-world effectiveness of LM4Code systems. These pitfalls span the data, models, evaluation, and deployment phases of the LM4Code lifecycle. LM4Code has become an increasingly prominent research area, evidenced by the rapid increase in publications. A prior survey by Hou *et al.* [51] uncovered 229 papers on large language models for software engineering between 2020-2023, while Wang *et al.* [157] uncovered 350 papers on deep learning for software engineering between 2015-2020. However, our focused study on LM4Code pitfalls only identified 67 relevant papers. This indicates that research attention to pitfalls in LM4Code is still insufficient, compared to the overall research volume. Thus, future LM4Code research must not overlook the pitfalls when applying LM4Code models to software engineering tasks.

**Recognizing Existing Pitfalls.** In this study, we summarized multiple common pitfalls associated with LM4Code. These pitfalls, each with distinct implications, highlight the inherent complexities in applying LM4Code to real-world software engineering problems. As our review results demonstrate, pitfalls can introduce unrealistic performance evaluation, compromise model efficacy, and raise security concerns. Thus, it becomes important for future LM4Code research to recognize and avoid potential pitfalls when building LM4Code systems for SE tasks. To ensure trustworthy findings of LM4Code research, it is essential to demonstrate effectiveness through a rigorous and reliable experimental design that reflects real-world scenarios. Furthermore, although our review has summarized common pitfalls, as previous surveys like Hou *et al.* [51] present, there exist more than 50 specific large language models tailored to over 55 software engineering scenarios. So while we report general implications in some common settings using prevalent models, further investigation is required to discern more specific implications in specific or unconventional scenarios.

**Addressing Existing Pitfalls.** Addressing the identified pitfalls is vital for advancing robust and reliable LM4Code techniques. As our study reveals, solutions like data cleaning, model explainability, optimized model design, and rigorous benchmarking have shown promise in mitigating certain pitfalls. However, these solutions, although effective in specific contexts, may not be universally applicable due to the complexity and ever-evolving nature of the software engineering landscape. A coordinated effort by the community is required to establish guidelines and best practices that enable mitigating pitfalls in data construction, model design, performance evaluation, and deployment.

**Uncovering New Pitfalls.** The dynamic nature of the LM4Code field means novel pitfalls will likely emerge as techniques rapidly evolve. Specifically, the ever-evolving software engineering landscape, increasingly complex codebases, and new LM4Code techniques provide fertile ground for novel pitfalls to emerge. Thus, identifying emerging pitfalls is critical. The community needs to

continuously analyze model reasoning, behaviors, and performance under realistic experimental settings to uncover new pitfalls in a timely manner. For example, testing on diverse unexplored situations or probing model decisions via XAI techniques yield valuable insights. Through periodically updating benchmarks, refining evaluation approaches, and incorporating real-world deployment scenarios, we can ensure that we not only keep pace with the ever-evolving landscape, but also recognize and mitigate new challenges.

## 7.2 Open Challenges and Research Directions

**7.2.1 Improving Data Quality for LM4Code.** The processes of data collection and labeling play critical roles in the model construction, determining their efficiency, trustworthiness, and overall performance [90]. With the emergence of large language models such as GPT-4 [111], the reliance on massive data has increased significantly. However, as we have reviewed, there are numerous obstacles and unanswered questions associated with these phases.

**Volume vs. Quality.** The increasing number and widespread use of large language models such as GPT-4, underscore the inherent conflict between the quantity and quality of data. These models, with their vast number of parameters, heavily rely on extensive datasets to achieve their remarkable performance. For example, Codex [17], which is a variant of the GPT-3.5 framework introduced in 2021, conducted training using a dataset that was sourced from 54 million publicly available software repositories on the GitHub platform, resulting in a total data size of 159 GB. Yet, collecting vast amounts of data, a process that is both time-consuming and labor-intensive, is not a “silver bullet”; data quality is also important. Even with advanced large models, “garbage in - garbage out” [86, 113]. Notably, even widely-used benchmark datasets like CodeSearchNet [54] have been observed to contain considerable noise and errors, as highlighted in prior research [133, 138]. Over-reliance on volume can lead to models that are prone to biases, noise, and even adversarial attacks. On the contrary, an excessive focus on data quality can inadvertently decrease the diversity and richness of the dataset. Finding the ideal compromise between these conflicting requirements offers an appealing opportunity for future investigation.

**Automated Data Quality Assurance.** As the implications of low-quality data, it is essential to utilize high-quality datasets to develop LM4Code approaches. In the dynamic evolution environment of software and language models, manually searching and examining extensive datasets for data noise or errors is neither practical and efficient. As a result, there is an urgent need for automated tools and frameworks that can assure and maintain data trustworthiness and quality, particularly for LM4Code models. We can systematically discover and correct data noise, labeling errors, and other anomalies, allowing models to train on robust and reliable datasets and ensuring their reliability and efficiency in real-world applications.

**7.2.2 Strengthening Robustness and Trustworthiness in LM4Code.** LM4Code models are becoming increasingly integrated into the software development lifecycle, influencing everything from code generation to vulnerability detection. Ensuring that these models are robust and trustworthy is essential. This does not just relate to their prediction accuracy but extends to the reliability, interpretability, and generalization capacity of the model, especially in diverse and evolving coding environments [104, 156].

**Building Interpretable LM4Code.** The black-box nature of language models has been a long-term concern, especially when LM4Code applications directly influence software development outcomes [20, 141, 156, 161]. Transparency in LM4Code requires an in-depth examination of the correlations and reasoning processes that models depend on, instead of just knowing the model’s predictions. Our review results show that pitfalls can exist throughout the entire LM4Code lifecycle, potentially resulting in spurious correlations. These misleading correlations are based on

wrong artifacts for generating predictions, presenting significant challenges for practical real-world applications. Although prior studies [21, 130, 150, 187] have introduced various explainable AI (XAI) techniques into LM4Code, the current solutions lag behind the rapid evolution of language models [94, 186]. Existing XAI techniques for LMs, in particular, only provide explanations as either the contribution of individual words to the decision or the layer/neuron at which syntax or semantics are encoded [186]. Although helpful, these explanations only offer a fragmented picture of the model's decision-making process, ignoring a considerable amount of its intricate reasoning. As the complexity of language models grows, there is an imperative need to develop more comprehensive and accessible XAI approaches for LM4Code.

**Improving Robustness Against Errors and Threats.** Recent literature [105, 116, 137] reveals that state-of-the-art LM4Code models like Codex [17], GPT-3, and Starcoder [72] are susceptible to inadvertent data errors and malicious threats. Such pitfalls not only degrade model performance, but raise concerns about the security and trustworthiness of LM4Code systems. Improving model robustness is therefore an urgent need to enable reliable LM4Code adoption. While prior studies [135, 137, 138, 179] have proven that techniques like adversarial training and data augmentation can enhance robustness, more efforts should be spent by our research community to holistically defend against newly emerging issues and threats. Specifically, possible solutions like domain-specific preprocessing and learning, continuous evaluation, hybrid models, and anomaly detection should be explored. Overall, a multilayered defense-in-depth strategy is essential to ensure LM4Code reliability and trustworthiness against growing pitfalls or issues.

**Adapting to Ever-evolving Code Environments.** Within the ever-evolving field of software development, new programming languages emerge, old ones get updated, and coding techniques and habits are constantly transforming. Against this backdrop, a primary challenge for LM4Code is ensuring the broad applicability and robust generalizability of LM4Code systems. Prior studies [106, 120, 145, 156] have highlighted the superior generalizability of advanced LMs over traditional ML/DL techniques, particularly when it comes to previously unseen distributions. However, there remain significant research gaps. Specifically, while the improvements in models like the advanced Transformer are promising, they may not always translate to practical efficiency. For instance, Thongtanunam *et al.* [145] indicate that while the Transformer demonstrates an improvement of 490% to 567% on new tokens, its accuracy remains around 10%, indicating a significant room for improvement. Thus, the incorporation of domain-specific knowledge, continuous model updating, and feedback loops with developers could pave the way for more adaptable and reliable LM4Code solutions in an ever-evolving coding landscape.

*7.2.3 Towards Reliable Performance Evaluation of LM4Code.* Performance evaluation plays a key role in demonstrating the efficiency and capability of the proposed LM4Code approaches. However, as discussed in Section 5, there are several pitfalls that can undermine realistic performance, including inappropriate baselines, inappropriate test sets, reproducibility issues, and inappropriate performance measures. Therefore, it is important to focus future research efforts on solid benchmarks and rigorous evaluation methodology.

**Towards Reliable and Standardized Benchmarks.** It is important to establish a rigorous evaluation methodology supported by trustworthy and standardized benchmarks. These benchmarks provide consistent frameworks for comparing different LM4Code approaches and set baselines for new techniques [10]. However, our review results reveal that multiple pitfalls related to performance evaluation like inappropriate baselines, limited test sets, and reproducibility issues distort understanding of actual LM4Code capabilities, questioning research correctness [83, 88, 175]. For example, Laaber *et al.* [69] emphasize the inherent benchmark instability, stressing the importance of identifying and rectifying these instabilities to ensure accurate evaluations. Without addressing

pitfalls, perceived model performance can become inflated, leading to potentially misleading conclusions. Related benchmarking challenges have been noted across various research domains like computer security [147], and cloud computing [22, 126]. Hence, there is an urgent need to prioritize the development of standardized, stable, and reliable benchmarks in LM4Code research. Future research should focus on creating benchmarks that are both comprehensive and representative of real-world scenarios. To achieve this, the community should coordinate on curating representative test sets, establishing strong baselines, quantifying uncertainty, and promoting reproducible experiments, ensuring that advancements in the field of LM4Code are grounded in robust and reliable evaluations. This will not only foster trustworthiness within the research community but also drive meaningful progress in LM4Code.

**Towards Reliable Evaluation Metrics.** As language models like GPT-4 grow in complexity, accurately assessing their capabilities and limitations becomes imperative. However, as discussed, traditional evaluation metrics, often borrowed from the NLP domain, may fail to capture the intricacies and details that are crucial for code generation and understanding tasks [121, 124]. For example, metrics like accuracy and BLEU focusing solely on syntactic correctness fail to account for potential semantic errors with practical consequences, as noted by Fan *et al.* [30]. In addition, most studies rely heavily on automated metrics, but human evaluation remains indispensable for assessing the quality and utility of generated code [24, 112]. Additionally, the variability and uncertainty inherent in generative models like GPT-4 need to be measured and analyzed in order to understand reliability in real-world settings [111]. Therefore, developing holistic and standardized metrics tailored to code intelligence tasks is urgently needed. These should measure syntactic validity, semantic consistency, coherence, human quality judgments, and variability. For LM4Code to move from impressing lab-only evaluation to transforming real-world applications, reliable evaluation metrics are crucial.

*7.2.4 Optimizing LM4Code Deployment for Real-World Scenarios.* Realizing LM4Code's potential requires addressing deployment challenges in transitioning from controlled research to practice. The integration, security, and scalability of LM4Code techniques become important considerations.

**Integrating LM4Code into Developer Workflows and Tools.** While modern LM4Code models like GitHub Copilot show potential for integrating AI-driven code suggestions into developers' workflows, realizing this in practice presents unique challenges. First, it is crucial to ensure the correctness and human comprehensibility of suggestions, especially since the generated code can contain errors and vulnerabilities that may be problematic for real-world usage [100, 113, 116]. Additionally, customization and personalization are also important, since every developer has a unique coding style and preferences. Modern LM4Code tools should be adaptable and learn from individual developer behaviors to provide personalized code suggestions. Finally, integration with developer tools like version control and debugging is critical for comprehensive functionality. Future research should focus on addressing these challenges, ensuring that the integration of LM4Code models into developer workflows is smooth, efficient, and beneficial.

**Towards LM4Code Security.** As discussed in Section 6, LM4Code faces several security concerns that need to be addressed before responsible and ethical deployment can be achieved. These include risks of evasion attacks, data poisoning, privacy leakage, and generated code potentially containing vulnerabilities. Implementing comprehensive solutions is thus critical to safeguard models, data, and users. Advancing the security of LM4Code requires continued research across multiple domains: (1) Adversarial training techniques [185] can potentially increase model robustness against adversarial inputs; (2) Employing formal verification approaches to evaluate the security and correctness of generated code [113]; (3) Enhanced data construction and quality assurance processes are needed to systematically identify and eliminate data poisoning, preventing the propagation of risks [125];

(4) Approaches such as differential privacy [93] and federated learning [73] may strengthen privacy preservation in model training; (5) Watermarking and provenance tracking mechanisms can enable authentication of model ownership and detect plagiarism or unauthorized use [65, 137]; (6) Manual examination approaches including code reviews and human-AI collaborative interactions are important. Substantial multidisciplinary efforts are critical to ensure the secure and ethical development and deployment of powerful generative AI systems like LM4Code. This remains an open research challenge requiring continued progress across communities.

**Scalability and Latency Concerns.** While powerful general large language models like GPT-4 [111], PaLM [41], and Claude [5] offer public API access, privately deployed specialized LM4Code models can face greater scalability challenges. Developing private models is important to customize LM4Code systems to specific domains and tasks. However, large language models often comprise billions of parameters, requiring massive computational resources [51]. This poses challenges for real-time efficient integration into developer workflows. Reducing inference latency through methods like knowledge distillation [132] and efficient attention [64] is crucial for reasonable responsiveness. Additionally, scaling up throughput for concurrent users via cloud-native architectures is also essential. Optimizing memory utilization via compression and caching helps in deploying large LM4Code models [132]. Energy-efficient deployment through quantization and lightweight model distillation is likewise critical [43]. Tackling these scalability and latency issues will be critical to realize LM4Code's full potential through deployments that smoothly integrate LM4Code into practical developer environments in a sustainable and user-friendly manner.

### 7.3 Threats to Validity

This systematic literature review was conducted according to the established guidelines [66, 178] to mitigate potential threats to validity. However, there are still certain limitations primarily associated with our search strategy and the data extraction process used for constructing our paper taxonomy.

One primary threat is selection bias, wherein the selection process may miss some relevant research studies. First, one possible cause may be that some search engines may provide some irrelevant studies or overlook some studies. Another reason could be that our keywords don't cover all the relevant research studies. With an emerging field like LM4Code, important ongoing work may not yet be indexed in the primary digital libraries we searched. In addition, as we stated before, pitfalls do not have consistent keywords or terminology, so our manual checking of the pitfalls of LM4Code papers also has the potential for omissions. To minimize this risk, we systematically performed the paper searching across six major digital libraries in computer science, manually searched top venues, iteratively refined search strings based on a quasi-gold standard approach defined by Zhang *et al.* [178], and conducted backward/forward snowballing. Moreover, each manuscript that an individual author expressed uncertainty about regarding its including/excluding underwent thorough discussion between the first two authors before making a final decision.

Another threat is internal validity in constructing our taxonomy of LM4Code pitfalls. A significant contribution of this paper is developing a taxonomy to categorize and synthesize key pitfalls across the LM4Code field. To mitigate subjectivity in our taxonomy, we adapted a framework from Arp *et al.* [6] previously applied across computer security, which was collaboratively validated by the first six authors with LM4Code expertise. At the same time, to enhance accuracy, each primary study classification was reviewed by at least three authors, with disagreements resolved through discussion. Despite the fact that multiple evaluators reduce the possibility of bias, subjective factors persist. To improve the integrity of our taxonomy and offer transparency into our workings, all of our collected research studies and their classification are available in our online repository.

## 8 CONCLUSION

In this research study, we conducted a comprehensive and rigorous systematic literature review to examine the pitfalls present in LM4Code. We utilized a well-defined systematic literature review approach and finally obtained 67 relevant research studies from top-tier venues. We first provided a taxonomy and we classified the existing pitfalls in LM4Code based on the various stages of the LM4Code lifecycle, including data collection and labeling, system design and learning, performance evaluation, and deployment and maintenance. For each stage, we provided a thorough review of the relevant studies based on the pitfall types, implications, and existing solutions. Finally, we described the current challenges and discussed the open opportunities that demand more study in this area. We hope that our work will motivate other researchers, making language models enhanced for code intelligence more reliable and trustworthy, thereby ensuring their effective deployment into real-world applications.

## ACKNOWLEDGEMENTS

We sincerely thank Zhou Yang and Zhensu Sun, PhD students at Singapore Management University, for their invaluable insights, feedback, and assistance in clearly defining the taxonomy of pitfalls in language models for code. Their contributions greatly helped shape and improve this work.

## REFERENCES

- [1] [n.d.]. codeparrot (CodeParrot). <https://huggingface.co/codeparrot>
- [2] Zohreh Aghababaeyan, Manel Abdellatif, Lionel Briand, Ramesh S, and Mojtaba Bagherzadeh. 2023. Black-Box Testing of Deep Neural Networks through Test Case Diversity. *IEEE Trans. Software Eng.* 49, 5 (May 2023), 3182–3204. <https://doi.org/10.1109/TSE.2023.3243522>
- [3] Toufique Ahmed, Noah Rose Ledesma, and Premkumar Devanbu. 2023. SynShine: Improved Fixing of Syntax Errors. *IEEE Trans. Software Eng.* 49, 4 (April 2023), 2169–2181. <https://doi.org/10.1109/TSE.2022.3212635>
- [4] Amazon. 2023. Amazon CodeWhisperer. <https://aws.amazon.com/codewhisperer/>.
- [5] Anthropic. 2023. Getting started with Claude. <https://docs.anthropic.com/claude/docs>.
- [6] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*. 3971–3988.
- [7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [8] Gareth Ari Aye, Seohyun Kim, and Hongyu Li. 2021. Learning autocompletion from real-world datasets. (2021), 131–139.
- [9] Gareth Ari Aye, Seohyun Kim, and Hongyu Li. 2021. Learning autocompletion from real-world datasets. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 131–139.
- [10] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2019. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer* 21 (2019), 1–29.
- [11] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2154–2156.
- [12] Paul E Black. 2017. Sard: A software assurance reference dataset. (2017).
- [13] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. 2021. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*. 2633–2650.
- [14] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).
- [15] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749* (2023).
- [16] Boyuan Chen, Mingzhi Wen, Yong Shi, Dayi Lin, Gopi Krishnan Rajbahadur, and Zhen Ming Jiang. 2022. Towards training reproducible deep learning models. In *Proceedings of the 44th International Conference on Software Engineering*.

2202–2214.

- [17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [18] Nadezhda Chirkova and Sergey Troshin. 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*. 703–715.
- [19] Nadezhda Chirkova and Sergey Troshin. 2023. CodeBPE: Investigating Subtokenization Options for Large Language Model Pretraining on Source Code. *arXiv preprint arXiv:2308.00683* (2023).
- [20] Jürgen Cito, Satish Chandra, Chakkrit Tantithamthavorn, and Hadi Hemmati. 2023. Expert Perspectives on Explainability. *IEEE Software* 40, 3 (2023), 84–88. <https://doi.org/10.1109/MS.2023.3255663>
- [21] Jürgen Cito, Isil Dillig, Seohyun Kim, Vijayaraghavan Murali, and Satish Chandra. 2021. Explaining mispredictions of machine learning models using rule induction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 716–727. <https://doi.org/10.1145/3468264.3468614>
- [22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
- [23] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. 2022. Data preparation for software vulnerability prediction: A systematic literature review. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1044–1063.
- [24] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734.
- [25] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J. Hellendoorn. 2021. Patching as translation: the data and the metaphor. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, Virtual Event Australia, 275–286. <https://doi.org/10.1145/3324884.3416587>
- [26] Jinhao Dong, Yiling Lou, Dan Hao, and Lin Tan. 2023. Revisiting Learning-based Commit Message Generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 794–805.
- [27] Vinicius HS Durelli, Rafael S Durelli, Simone S Borges, Andre T Endo, Marcelo M Eler, Diego RC Dias, and Marcelo P Guimarães. 2019. Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability* 68, 3 (2019), 1189–1212.
- [28] Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
- [29] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.
- [30] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.
- [31] Sen Fang, Tao Zhang, Youshuai Tan, He Jiang, Xin Xia, and Xiaobing Sun. 2023. RepresentThemAll: A Universal Learning Representation of Bug Reports. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 602–614.
- [32] China Computer Federation. 2023. CCF Recommended List of International Conferences and Periodicals. <https://www.ccf.org.cn/en/Bulletin/2019-05-13/663884.shtml>.
- [33] Ruitao Feng, Sen Chen, Xiaofei Xie, Guozhu Meng, Shang-Wei Lin, and Yang Liu. 2021. A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices. *IEEE Trans.Inform.Forensic Secur.* 16 (2021), 1563–1578. <https://doi.org/10.1109/TIFS.2020.3025436>
- [34] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
- [35] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair (*ESEC/FSE 2022*). Association for Computing Machinery, New York, NY, USA, 935–947. <https://doi.org/10.1145/3540250.3549098>
- [36] Michael Fu, Chakkrit Tantithamthavorn Van Nguyen, Trung Le, and Dinh Phung. 2023. VulExplainer: A Transformer-based Hierarchical Distillation for Explaining Vulnerability Types. *IEEE Transactions on Software Engineering (TSE)* (2023).
- [37] Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Hassan Sajjad, Preslav Nakov, Deming Chen, and Marianne Winslett. 2021. Compressing large-scale transformer-based models: A case study on bert. *Transactions*

of the Association for Computational Linguistics 9 (2021), 1061–1080.

- [38] Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, Jun Sun, David Lo, and Yue Yu. 2023. Two sides of the same coin: Exploiting the impact of identifiers in neural code comprehension. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1933–1945.
- [39] Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. 2023. Keeping Pace with Ever-Increasing Data: Towards Continual Learning of Code Intelligence Models. *arXiv preprint arXiv:2302.03482* (2023).
- [40] GitHub. 2023. Github copilot. <https://copilot.github.com>.
- [41] Google. 2023. Build generative AI applications with Google. <https://developers.generativeai.google/>.
- [42] Alejandro Guerra-Manzanares and Hayretin Bahsi. 2022. On the relativity of time: Implications and challenges of data drift on long-term effective android malware detection. *Computers & Security* 122 (2022), 102835.
- [43] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 810–822.
- [44] Manish Gupta and Puneet Agrawal. 2022. Compression of deep learning models for text: A survey. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 16, 4 (2022), 1–55.
- [45] Yiling He, Yiping Liu, Lei Wu, Ziqi Yang, Kui Ren, and Zhan Qin. 2022. MsDroid: Identifying malicious snippets for android malware detection. *IEEE Transactions on Dependable and Secure Computing* (2022).
- [46] Yiling He, Jian Lou, Zhan Qin, and Kui Ren. 2023. FINER: Enhancing State-of-the-art Classifiers with Feature Attribution to Facilitate Security Analysis. *arXiv preprint arXiv:2308.05362* (2023).
- [47] Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When Code Completion Fails: A Case Study on Real-World Completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 960–970. <https://doi.org/10.1109/ICSE.2019.00101>
- [48] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2022. Semantic Robustness of Models of Source Code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Honolulu, HI, USA, 526–537. <https://doi.org/10.1109/SANER53432.2022.00070>
- [49] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It’s not a bug, it’s a feature: how misclassification impacts bug prediction. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 392–401.
- [50] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529.
- [51] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *arXiv preprint arXiv:2308.10620* (2023).
- [52] Haonan Hu, Yue Liu, Yanjie Zhao, Yonghui Liu, Xiaoyu Sun, Chakkrit Tantithamthavorn, and Li Li. 2023. Detecting Temporal Inconsistency in Biased Datasets for Android Malware Detection. In *The 6th International Workshop on Advances in Mobile App Analysis (A-Mobile)*. 0–0.
- [53] Yujin Huang, Han Hu, and Chunyang Chen. 2021. Robustness of on-device models: Adversarial attack to deep learning models on android apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 101–110.
- [54] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).
- [55] Matthew Hutson. 2018. Artificial intelligence faces reproducibility crisis.
- [56] Peter Ivie and Douglas Thain. 2018. Reproducibility in scientific computing. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–36.
- [57] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.
- [58] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *Proceedings of the 45th International Conference on Software Engineering* (2023).
- [59] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 135–146.
- [60] Wenxin Jiang, Nicholas Synovic, Matt Hyatt, Taylor R. Schorlemmer, Rohan Sethi, Yung-Hsiang Lu, George K. Thiruvathukal, and James C. Davis. 2023. An Empirical Study of Pre-Trained Model Reuse in the Hugging Face Deep Learning Model Registry. (2023), 2463–2475. <https://doi.org/10.1109/ICSE48619.2023.00206>
- [61] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John C. Grundy. 2021. Practitioners’ Perceptions of the Goals and Visual Explanations of Defect Prediction Models. In *18th IEEE/ACM International Conference on Mining Software*

- Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021. IEEE, 432–443. <https://doi.org/10.1109/MSR52588.2021.00055>
- [62] Daniel D Johnson, Daniel Tarlow, and Christian Walder. 2023. RU-SURE? Uncertainty-Aware Code Suggestions By Maximizing Utility Across Random User Intents. *arXiv preprint arXiv:2303.00732* (2023).
- [63] Magne Jørgensen, Tore Dybå, Knut Liestøl, and Dag IK Sjøberg. 2016. Incorrect results in software engineering experiments: How to improve research practices. *Journal of Systems and Software* 116 (2016), 133–145.
- [64] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169* (2023).
- [65] John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. 2023. A watermark for large language models. *arXiv preprint arXiv:2301.10226* (2023).
- [66] Barbara Kitchenham and Stuart Charters. 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering. 2 (Jan. 2007).
- [67] Barbara Kitchenham and Stuart Charters. 2007. Guidelines for performing systematic literature reviews in software engineering. (2007).
- [68] Barbara Kitchenham, Lech Madeyski, and David Budgen. 2023. SEGRESS: Software Engineering Guidelines for REporting Secondary Studies. *IEEE Transactions on Software Engineering* 49 (March 2023), 1273–1298. <https://doi.org/10.1109/TSE.2022.3174092> Conference Name: IEEE Transactions on Software Engineering.
- [69] Christoph Laaber, Mikael Basmaci, and Pasquale Salza. 2021. Predicting unstable software benchmarks using static source code features. *Empirical Software Engineering* 26, 6 (2021), 114.
- [70] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Yih, Daniel Fried, Si yi Wang, and Tao Yu. 2022. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. *ArXiv abs/2211.11501* (2022). <https://api.semanticscholar.org/CorpusID:253734939>
- [71] Bo Li, Peng Qi, Bo Liu, Shuai Di, Jingen Liu, Jiquan Pei, Jinfeng Yi, and Bowen Zhou. 2023. Trustworthy AI: From principles to practices. *Comput. Surveys* 55, 9 (2023), 1–46.
- [72] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
- [73] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. 2020. Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine* 37, 3 (2020), 50–60.
- [74] Yanzhou Li, Shangqing Liu, Kangjie Chen, Xiaofei Xie, Tianwei Zhang, and Yang Liu. 2023. Multi-target Backdoor Attacks for Code Pre-trained Models. *arXiv preprint arXiv:2306.08350* (2023).
- [75] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 292–303. <https://doi.org/10.1145/3468264.3468597>
- [76] Zhong Li, Minxue Pan, Yu Pei, Tian Zhang, Linzhang Wang, and Xuandong Li. 2022. Robust Learning of Deep Predictive Models from Noisy and Imbalanced Software Engineering Datasets. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.
- [77] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1238–1250.
- [78] Zongjie Li, Chaozheng Wang, Pingchuan Ma, Chaowei Liu, Shuai Wang, Daoyuan Wu, and Cuiyun Gao. 2023. On the feasibility of specialized ability stealing for large language code models. *arXiv preprint arXiv:2303.03012* (2023).
- [79] Zehao Lin, Guodun Li, Jingfeng Zhang, Yue Deng, Xiangji Zeng, Yin Zhang, and Yao Wan. 2022. XCODE: Towards Cross-Language Code Representation with Large-Scale Pre-Training. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–44.
- [80] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 599–609.
- [81] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. On the Reproducibility and Replicability of Deep Learning in Software Engineering. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 15 (oct 2021), 46 pages. <https://doi.org/10.1145/3477535>
- [82] Chenyao Liu, Zeqi Lin, Jian-Guang Lou, Lijie Wen, and Dongmei Zhang. 2021. Can Neural Clone Detection Generalize to Unseen Functionalities. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Melbourne, Australia, 617–629. <https://doi.org/10.1109/ASE51524.2021.9678907>
- [83] Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. 2022. Deep Learning Based Program Generation From Requirements Text: Are We There Yet? *IEEE Trans. Software Eng.* 48, 4 (April 2022), 1268–1289. <https://doi.org/10.1109/TSE.2020.3018481>

- [84] Shigang Liu, Guanjun Lin, Lizhen Qu, Jun Zhang, Olivier De Vel, Paul Montague, and Yang Xiang. 2022. CD-VulD: Cross-Domain Vulnerability Discovery Based on Deep Domain Adaptation. *IEEE Trans. Dependable and Secure Comput.* 19, 1 (Jan. 2022), 438–451. <https://doi.org/10.1109/TDSC.2020.2984505>
- [85] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. *arXiv preprint arXiv:2301.09072* (2023).
- [86] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2023. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *arXiv preprint arXiv:2307.12596* (2023).
- [87] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep learning for android malware defenses: a systematic literature review. *Comput. Surveys* 55, 8 (2022), 1–36.
- [88] Zhongxin Liu, Kui Liu, Xin Xia, and Xiaohu Yang. 2023. Towards More Realistic Evaluation for Neural Test Oracle Generation. *arXiv preprint arXiv:2305.17047* (2023).
- [89] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 373–384.
- [90] David Lo. 2023. Trustworthy and Synergistic Artificial Intelligence for Software Engineering: Vision and Roadmaps. *arXiv preprint arXiv:2309.04142* (2023).
- [91] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning (Practical Experience Report). *arXiv preprint arXiv:2308.11148* (2023).
- [92] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.
- [93] Nils Lukas, Ahmed Salem, Robert Sim, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. 2023. Analyzing Leakage of Personally Identifiable Information in Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 346–363.
- [94] Andreas Madsen, Siva Reddy, and Sarath Chandar. 2022. Post-hoc interpretability for neural nlp: A survey. *Comput. Surveys* 55, 8 (2022), 1–42.
- [95] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the robustness of code generation techniques: An empirical study on github copilot. *arXiv preprint arXiv:2302.00438* (2023).
- [96] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based Neural Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1456–1468.
- [97] Bonan Min, Hayley Ross, Elior Sulem, Amir Poursan Ben Veyshe, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2021. Recent advances in natural language processing via large pre-trained language models: A survey. *Comput. Surveys* (2021).
- [98] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. *ArXiv abs/2210.14306* (2022). <https://api.semanticscholar.org/CorpusID:253117056>
- [99] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2023. When to Show a Suggestion? Integrating Human Feedback in AI-Assisted Programming. *ArXiv abs/2306.04930* (2023). <https://api.semanticscholar.org/CorpusID:259108906>
- [100] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot’s code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5.
- [101] Xu Nie, Ningke Li, Kailong Wang, Shangguang Wang, Xiapu Luo, and Haoyu Wang. 2023. Understanding and Tackling Label Errors in Deep Learning-Based Vulnerability Detection (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 52–63.
- [102] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1565–1569.
- [103] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. (2023), 2136–2148. <https://doi.org/10.1109/ICSE48619.2023.00180>
- [104] Changan Niu, Chuanyi Li, Vincent Ng, and Bin Luo. 2023. CrossCodeBench: Benchmarking Cross-Task Generalization of Source Code Models. *arXiv preprint arXiv:2302.04030* (2023).
- [105] Liang Niu, Shujaat Mirza, Zayd Maradni, and Christina Pöpper. 2023. {CodexLeaks}: Privacy Leaks from Code Generation Language Models in {GitHub} Copilot. In *32nd USENIX Security Symposium (USENIX Security 23)*.

- 2133–2150.
- [106] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating realistic vulnerabilities via neural code editing: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Singapore Singapore, 1097–1109. <https://doi.org/10.1145/3540250.3549128>
  - [107] Curtis Northcutt, Lu Jiang, and Isaac Chuang. 2021. Confident learning: Estimating uncertainty in dataset labels. *Journal of Artificial Intelligence Research* 70 (2021), 1373–1411.
  - [108] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.
  - [109] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT Self-Repair for Code Generation. *arXiv preprint arXiv:2306.09896* (2023).
  - [110] Daryna Olynyk, Rudolf Mayer, and Andreas Rauber. 2023. I know what you trained last summer: A survey on stealing machine learning models and defences. *Comput. Surveys* (2023).
  - [111] OpenAI. 2023. GPT-4 Technical Report. *arXiv:2303.08774* [cs.CL]
  - [112] Matteo Paltenghi and Michael Pradel. 2021. Thinking Like a Developer? Comparing the Attention of Humans with Neural Models of Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Melbourne, Australia, 867–879. <https://doi.org/10.1109/ASE51524.2021.9678712>
  - [113] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot’s code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.
  - [114] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2339–2356. <https://doi.org/10.1109/SP46215.2023.10179324>
  - [115] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium (USENIX Security 19)*. 729–746.
  - [116] Xutan Peng, Yipeng Zhang, Jingfeng Yang, and Mark Stevenson. 2022. On the Security Vulnerabilities of Text-to-SQL Models. *arXiv preprint arXiv:2211.15363* (2022).
  - [117] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchronesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227* (2022).
  - [118] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. In *International Conference on Learning Representations*.
  - [119] Mikko Raatikainen, Juha Tiihonen, and Tomi Männistö. 2019. Software product lines and variability modeling: A tertiary study. *Journal of Systems and Software* 149 (2019), 485–510.
  - [120] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.
  - [121] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
  - [122] The Computing Research and Education Association of Australasia. 2023. CORE Rankings Portal. <https://www.core.edu.au/conference-portal>.
  - [123] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 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*. 1105–1116.
  - [124] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: How far are we?. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 417–427.
  - [125] Roi Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)*. 1559–1575.
  - [126] Malte Schwarzkopf, Derek G Murray, and Steven Hand. 2012. The seven deadly sins of cloud computing research. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*.
  - [127] Nima Shahbazi, Yin Lin, Abolfazl Asudeh, and HV Jagadish. 2023. Representation Bias in Data: A Survey on Identification and Resolution Techniques. *Comput. Surveys* (2023).

- [128] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating domain knowledge through task augmentation for front-end JavaScript code generation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, Singapore Singapore, 1533–1543. <https://doi.org/10.1145/3540250.3558965>
- [129] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*. 1597–1608.
- [130] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards Efficient Fine-tuning of Pre-trained Code Models: An Experimental Study and Beyond. *arXiv preprint arXiv:2304.05216* (2023).
- [131] Jieke Shi, Zhou Yang, Hong Jin Kang, Bowen Xu, Junda He, and David Lo. 2023. Smaller, Faster, Greener: Compressing Pre-trained Code Models via Surrogate-Assisted Optimization. *arXiv preprint arXiv:2309.04076* (2023).
- [132] Jieke Shi, Zhou Yang, Bowen Xu, Hong Jin Kang, and David Lo. 2022. Compressing pre-trained models of code into 3 mb. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
- [133] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 107–119.
- [134] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2237–2248.
- [135] Weisong Sun, Yuchen Chen, Guan hong Tao, Chunrong Fang, Xiangyu Zhang, Quanjun Zhang, and Bin Luo. 2023. Backdooring Neural Code Search. *arXiv preprint arXiv:2305.17506* (2023).
- [136] Zhensu Sun, Xiaoning Du, Fu Song, and Li Li. 2023. CodeMark: Imperceptible Watermarking for Code Datasets against Neural Code Completion Models. *arXiv preprint arXiv:2308.14401* (2023).
- [137] Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022. Coprotector: Protect open-source code against unauthorized training usage with data poisoning. In *Proceedings of the ACM Web Conference 2022*. 652–660.
- [138] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 1609–1620. <https://doi.org/10.1145/3510003.3510160>
- [139] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.
- [140] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 329–340.
- [141] Chakkrit Tantithamthavorn, Jürgen Cito, Hadi Hemmati, and Satish Chandra. 2023. Explainable AI for SE: Challenges and Future Directions. *IEEE Software* 40, 3 (2023), 29–33. <https://doi.org/10.1109/MS.2023.3246686>
- [142] Chakkrit Tantithamthavorn and Ahmed E. Hassan. 2018. An experience report on defect modelling in practice: pitfalls and challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Frances Paulisch and Jan Bosch (Eds.). ACM, 286–295. <https://doi.org/10.1145/3183519.3183547>
- [143] Chakkrit Tantithamthavorn and Jirayus Jiarpakdee. 2021. Explainable AI for Software Engineering. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1–2. <https://doi.org/10.1109/ASE51524.2021.9678580>
- [144] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. 2015. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 812–823.
- [145] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. AutoTransform: automated code transformation to support modern code review process. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, Pittsburgh Pennsylvania, 237–248. <https://doi.org/10.1145/3510003.3510067>
- [146] Zhiyi Tian, Lei Cui, Jie Liang, and Shui Yu. 2022. A comprehensive survey on poisoning attacks and countermeasures in machine learning. *Comput. Surveys* 55, 8 (2022), 1–35.
- [147] Erik van der Kouwe, Gernot Heiser, Dennis Andriess, Herbert Bos, and Cristiano Giuffrida. 2019. SoK: Benchmarking flaws in systems security. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 310–325.

- [148] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
- [149] Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. You see what I want you to see: poisoning vulnerabilities in neural code search. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, Singapore Singapore, 1233–1245. <https://doi.org/10.1145/3540250.3549153>
- [150] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture?: a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, Pittsburgh Pennsylvania, 2377–2388. <https://doi.org/10.1145/3510003.3510050>
- [151] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. Association for Computing Machinery, Montpellier France, 397–407. <https://doi.org/10.1145/3238147.3238206>
- [152] Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th International Conference on Software Engineering*. 287–298.
- [153] Huaijin Wang, Pingchuan Ma, Yuanyuan Yuan, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2023. Enhancing DNN-Based Binary Code Function Search With Low-Cost Equivalence Checking. *IEEE Trans. Software Eng.* 49, 1 (Jan. 2023), 226–250. <https://doi.org/10.1109/TSE.2022.3149240>
- [154] Jindong Wang, Xixu Hu, Wenxin Hou, Hao Chen, Runkai Zheng, Yidong Wang, Linyi Yang, Haojun Huang, Wei Ye, Xiubo Geng, et al. 2023. On the robustness of chatgpt: An adversarial and out-of-distribution perspective. *arXiv preprint arXiv:2302.12095* (2023).
- [155] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software Testing with Large Language Model: Survey, Landscape, and Vision. *arXiv preprint arXiv:2307.07221* (2023).
- [156] Simin Wang, Liguang Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng. 2022. Machine/deep learning for software engineering: A systematic literature review. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1188–1231.
- [157] Simin Wang, Liguang Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng. 2023. Machine/Deep Learning for Software Engineering: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 49, 3 (March 2023), 1188–1231. <https://doi.org/10.1109/TSE.2022.3173346>
- [158] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2022. ReCode: Robustness Evaluation of Code Generation Models. *arXiv preprint arXiv:2212.10264* (2022).
- [159] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip S. Yu, and Guandong Xu. 2022. Reinforcement-Learning-Guided Source Code Summarization Using Hierarchical Attention. *IEEE Trans. Software Eng.* 48, 1 (Jan. 2022), 102–119. <https://doi.org/10.1109/TSE.2020.2979701>
- [160] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- [161] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–58.
- [162] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. *arXiv preprint arXiv:2309.00608* (2023).
- [163] Martin Weysow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2023. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. *arXiv preprint arXiv:2308.10462* (2023).
- [164] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. *arXiv preprint arXiv:2305.18607* (2023).
- [165] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery.
- [166] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.
- [167] Jiayun Xu, Yingjiu Li, and Robert H. Deng. 2021. Differential Training: A Generic Framework to Reduce Label Noises for Android Malware Detection. In *Network and Distributed System Security Symposium*. <https://api.semanticscholar.org/CorpusID:231879075>

- [168] Xu Yang, Shaowei Wang, Yi Li, and Shaohua Wang. 2023. Does data sampling improve deep learning-based vulnerability detection? Yeas! and Nays!. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2287–2298.
- [169] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)* 54, 10s (2022), 1–73.
- [170] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*. 1482–1493.
- [171] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2023. What Do Code Models Memorize? An Empirical Study on Large Language Models of Code. arXiv:2308.09932 [cs.SE]
- [172] Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2019. Mining software defects: Should we consider affected releases?. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 654–665.
- [173] Rahul Yedida and Tim Menzies. 2021. On the value of oversampling for deep learning in software defect prediction. *IEEE Transactions on Software Engineering* 48, 8 (2021), 3103–3116.
- [174] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*. 476–486.
- [175] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 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 (ISSTA 2022)*. Association for Computing Machinery, Virtual South Korea, 39–51. <https://doi.org/10.1145/3533767.3534390>
- [176] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 427–438. <https://doi.org/10.1145/3460319.3464819>
- [177] Fengyi Zhang, Bihuan Chen, Yufei Zhao, and Xin Peng. 2023. Slice-Based Code Change Representation Learning. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 319–330.
- [178] He Zhang, Muhammad Ali Babar, and Paolo Tell. 2011. Identifying relevant studies in software engineering. *Information and Software Technology* 53, 6 (2011), 625–637.
- [179] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua’an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards Robustness of Deep Program Processing Models Detection, Estimation, and Enhancement. *ACM Trans. Softw. Eng. Methodol.* 31, 3 (July 2022), 50:1–50:40. <https://doi.org/10.1145/3511887>
- [180] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. *AAAI* 34, 01 (April 2020), 1169–1176. <https://doi.org/10.1609/aaai.v34i01.5469>
- [181] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. 2023. Challenging Machine Learning-Based Clone Detectors via Semantic-Preserving Code Transformations. *IEEE Trans. Software Eng.* 49, 5 (May 2023), 3052–3070. <https://doi.org/10.1109/TSE.2023.3240118>
- [182] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, Singapore Singapore, 1073–1084. <https://doi.org/10.1145/3540250.3549094>
- [183] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 111–120.
- [184] Mingyi Zhou, Xiang Gao, Jing Wu, John Grundy, Xiao Chen, Chunyang Chen, and Li Li. 2023. ModelObfuscator: Obfuscating Model Information to Protect Deployed ML-Based Systems. (2023), 1005–1017. <https://doi.org/10.1145/3597926.3598113>
- [185] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald Gall. 2022. Adversarial Robustness of Deep Code Comment Generation. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (July 2022), 60:1–60:30. <https://doi.org/10.1145/3501256>
- [186] Julia El Zini and Mariette Awad. 2022. On the explainability of natural language processing deep models. *Comput. Surveys* 55, 5 (2022), 1–31.
- [187] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. 2021. Interpreting Deep Learning-based Vulnerability Detector Predictions Based on Heuristic Searching. *ACM Trans. Softw. Eng. Methodol.* 30, 2 (March 2021), 23:1–23:31. <https://doi.org/10.1145/3429444>

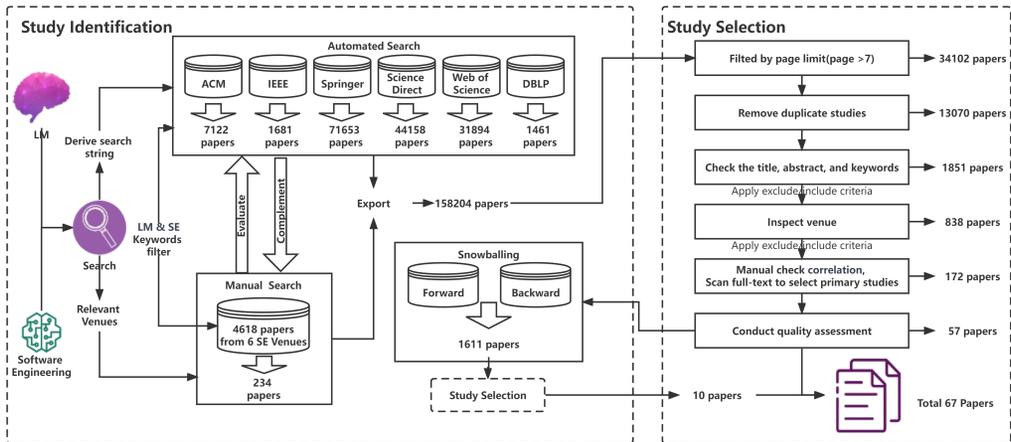


Figure 12. The overview of the review process.

## A DETAILS FOR PAPER COLLECTION AND SELECTION

Since we explore the state-of-the-art research of the pitfalls within LM4Code, we first outline our methodology for identifying relevant research studies, and then provide an overview of our collected literature. We follow the rigorous methodology proposed by Kitchenham *et al.* [66, 68] and Zhang *et al.* [178] to perform our lightweight Systematic Literature Review (SLR). The primary steps in our systematic literature review can be summarized as follows: (1) planning the review and formulating a review protocol, (2) proposing research questions, (3) designing search strategies and proposing inclusion/exclusion criteria, (4) conducting a lightweight snowballing, (5) data extraction, and (6) data synthesis. The outline for our review can be represented as presented in Figure 12.

### A.1 Research Questions and Motivations

In recent research, language models trained for code intelligence have shown promising performance [51, 155, 175]. However, an increasing number of literature [101, 133, 138] has highlighted the existence of pitfalls in LM4Code that can skew their realistic performance, leading to either substantial overestimation or underestimation of their effectiveness. The aim of conducting this systematic review is to gain an in-depth understanding of the pitfalls present in language models tailored for code intelligence. Ensuring the robustness, reliability, and ethical deployment of such models is important for their effective integration into the software development lifecycle. Consequently, it is crucial to discern the nature of these pitfalls, comprehend their implications, and examine existing solutions. Thus, we aim to answer the following research questions:

- **RQ1: What types of pitfalls are prevalent in language models for code intelligence?** This research question aims to identify the prevalent pitfalls in LM4Code systems, exploring how they could affect various stages of the learning-based system lifecycle.
- **RQ2: What are the implications of these pitfalls?** This research question investigates the implications of the identified pitfalls, specifically focusing on their impacts on the effectiveness, reliability, and ethical considerations of automated code intelligence systems.
- **RQ3: What solutions have been proposed to address these biases and pitfalls?** This research question reviews the existing body of literature to pinpoint proposed approaches for mitigating the identified pitfalls.

Acronym	Venues
ASE	International Conference on Automated Software Engineering
ESEC/FSE	Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering
ICSE	International Conference on Software Engineering
ISSTA	International Symposium on Software Testing and Analysis
TOSEM	Transactions on Software Engineering and Methodology
TSE	Transactions on Software Engineering

Table 3. Publication venues for manual search

## A.2 Search Strategy

To find out all potentially relevant research papers, we utilized the “Quasi-Gold Standard” (QGS) [178] approach, which combines both manual and automated search strategies. Using QGS offers an optimal balance between efficiency and research coverage, as evidenced in several previous studies [27, 156]. As illustrated in Figure 12, our search strategy involved the following sequential steps:

- (1) Select appropriate publication venues for manual search and select digital databases for automated search that encompass all the chosen venues.
- (2) Establish QGS: Screen all papers for manual search and filter by inclusion/exclusion criteria (defined in Table 3).
- (3) Define search string based on domain knowledge from Language Models (LM) and Software Engineering (SE).
- (4) Conduct an automated search using the search keywords defined in Step 3.
- (5) Evaluate the quality of included studies through QGS.

In our research approach, we integrated both manual and keyword-based search methodologies to identify relevant papers. During the manual approach, we focused on six top-tier SE conferences and journals, namely ICSE, ESEC/FSE, ASE, ISSTA, TOSEM, and TSE, all of which are CCF [32] A-ranked venues in the software engineering domains (as indicated in Table 3). We systematically crawled a list comprising 4,618 published papers from the top venues. After automating scanning with scripts, we thoroughly verified and discovered 234 papers relating to LM4Code. These 234 relevant papers served as the foundation for developing the Quasi-Gold Standard (QGS) for the following automated search. Our search string should include two sets of keywords: one for LMs and another for code intelligence. Only if the paper contains both types of keywords does it have a higher probability of being the one we require. The full list of search keywords is as follows:

- Keywords related to LMs: "LLM" OR "Large Language Model\*" OR "Language Model\*" OR "LM" OR "PLM" OR "Pre-trained" OR "Pre-training" OR "Natural Language Processing" OR "NLP" OR "Machine Learning" OR "ML" OR "Deep Learning" OR "DL" OR "Artificial Intelligence" OR "AI" OR "Transformer" OR "BERT" OR "CODEX" OR "GPT" OR "T5" OR "Sequence Model\*" OR "Attention Model\*" OR "Transfer Learning" OR "Neural Network\*" OR "ChatGPT" OR "GPT-\*" OR "Deep neural network\*" OR "DNN\*"
- Keywords related to code intelligence tasks: "Software Engineering" OR "Software Development" OR "Program\*" OR "Software Testing" OR "Software Mainten\*" OR "SE" OR "Software Lifecycle" OR "Software Design\*" OR "Code representation" OR "Code generation" OR "Code comment generation" OR "Code search" OR "Code localization" OR "Code completion" OR "Code summarization" OR "Method name generation" OR "Bug detection" OR "Bug localization" OR "Vulnerability detection" OR "Testing techniques" OR "Test case generation" OR "Program analysis" OR "Bug classification" OR "Defect prediction" OR "Program repair" OR "Code clone detection" OR "Bug report" OR "Software quality evaluation" OR "SATD"

<b>Inclusion Criteria</b>	
I1)	Studies explicitly utilizing LMs.
I2)	Studies claim that the study involves code-related tasks.
I3)	Studies highlighting pitfalls, particularly emphasizing unrealistic performance evaluation or factors that negatively influence performance in LM4Code.
<b>Exclusion Criteria</b>	
E1)	Studies whose full-text is inaccessible.
E2)	The study whose number of pages is less than 8.
E3)	Redundant or nearly identical studies from the same authors.
E4)	Papers not written in English.
E5)	Systematic literature reviews, reviews, or surveys.
E6)	Studies from workshops, doctoral symposiums, books, theses, monographs, keynotes, or panels.
E7)	Non peer-reviewed academic literature.
E8)	Studies not related to language models or code intelligence.
E9)	Studies emphasizing LM4Code without discussing pitfalls related to realistic performance.
E10)	Studies with a primary focus on cyberattacks or Operating Systems.
E11)	Studies that are published in a journal or conference with a CORE [122] ranking of less than A.

Table 4. Inclusion and Exclusion Criteria

detection" OR "Code smell detection" OR "Compiled-related" OR "Code review" OR "Software classification" OR "Code classification" OR "Code change" OR "Incident detection" OR "Requirement extraction" OR "Requirement traceability" OR "Requirement validation" OR "Effort cost prediction" OR "Mining GitHub/Github mining" OR "Mining SO (StackOverflow)/SO mining" OR "Mining app/App mining" OR "Mining tag/Tag mining" OR "Developer-based mining"

It's important to highlight that our list of keywords is specific to LM4Code, and we intentionally omitted keywords that are related to pitfalls during the paper search process. The reason behind this derives from the ambiguity around the term "pitfalls", which is open to various interpretations. We decided to rely on our rigorous inclusion/exclusion criteria because it is difficult to precisely categorize the types of pitfalls that exist within LM4Code, which is also the main motivation behind this taxonomy study. We were able to include relevant papers with this methodology, even if they didn't explicitly mention "pitfalls" in their content.

After establishing the search string, we proceeded to conduct an automated search across six widely used databases to ensure comprehensive coverage of all relevant published papers. Specifically, our search spanned four major academic publishers: ACM Digital Library, IEEE Xplorer, Springer, and Science Direct. Additionally, we included two renowned indexing databases: DBLP and Web of Science, which indexes several other smaller academic databases. Similarly to previous studies [23, 156], we did not use other search engines like Google Scholar due to the existence of excessive irrelevant information in the search results and the requirement for subjective criteria to choose when to stop the search process. Finally, we obtained 7,122 papers from the ACM Digital Library, 1,681 papers from IEEE Xplore, 71,653 papers from Springer, 44,158 papers from ScienceDirect, 44,158 papers from Web of Science, and 1,461 papers from DBLP.

### A.3 Study Selection

From among the papers collected by the paper search process, we attempted to select any research study that focused on the pitfalls of LM4Code. As we discussed before, we define these “pitfalls” as any significant issues or constraints present within the datasets, model architectures, experimental designs, or even model deployment that could potentially undermine the reliability or realistic performance of the proposed LM4Code systems. The inclusion/exclusion criteria we adopted, shown in Table 4, were inspired by similar studies [23, 87, 119]. It is noted that to maintain a reliable taxonomy of the pitfalls of LM4Code based on high-quality research studies, we removed the studies published in low-quality venues: venue ranking below A using the CORE ranking system [122].

To determine whether the studies met the inclusion requirements, we combined thorough manual assessment with automated script filtering. The study selection process involves six distinct stages, as illustrated in Figure 12. The first two stages (filtering and deduplication) used automated scripts, substantially reducing the initial set to 13,070 papers. Subsequently, in stages three and four, we applied the inclusion/exclusion criteria, delving into the titles, abstracts, keywords, and publication venues of each paper. It led to a sharp reduction in the number of remaining papers, leaving us with 838. The primary reason for exclusion was the lack of keywords correlating with both language models and code intelligence. Furthermore, we dismissed 890 papers classified as grey literature or misaligned with our main focus, as well as 105 systematic literature review articles to maintain our emphasis on the pitfalls associated with LM4Code tasks. In the fifth and sixth stages, every paper underwent a manual evaluation for relevance and Quality Assessment (QA), which led to the final selection of 57 papers.

For SLRs, it is important to analyze the quality of collected studies to ensure that we form an accurate and unbiased representation of the actual research [66]. Thus, we undertook the quality assessment process using a pre-defined quality checklist. We established four quality assurance criteria (QA1 to QA4) to reassess all selected papers.

- **QA1.** Are the pitfalls of LM4Code clearly described?
- **QA2.** Are the implications of the LM4Code pitfalls clearly stated or demonstrated?
- **QA3.** Is there a robust evaluation or solution for the identified LM4Code pitfalls in the proposed methodology?
- **QA4.** Is the contribution of the research clearly stated?

We critically evaluate and rate each QA on a scale of 0 to 4 (with 4 denoting “high”, while 0 denoting “low”). Then, we calculate the average quality score based on four quality criteria. We set the threshold equal to 2.5 (50 percent of the percentage score), which means if the average quality score of the research isn’t larger than 2.5, the study would be excluded.

### A.4 Snowballing

In order not to miss some important work, we conducted a lightweight snowballing [67], which is a commonly used search approach to complement automated queries. We executed both forward and backward snowballing, incorporating references and citations into consideration. Specifically, each primary study was examined for its references (backward snowballing) and its subsequent citations using Google Scholar (forward snowballing). The set of 57 papers from the prior step served as the initial set. From the forward and backward snowballing, we garnered 1003 and 1819 papers respectively. Following the processes of merging, deduplication, and the removal of articles already uncovered in our automated and manual searching, we ended up with a pool of 1611 papers. These papers underwent the same study selection process, culminating in the identification of an extra 10 papers. As a result, we collected a total of 67 papers focusing on LM4Code pitfalls.



SE activities	SE tasks	Total	
<b>Software development</b>	Code Generation(14)	Code Classification(3)	52
	Code Summarization(12)	Code Representation(2)	
	Code Search(9)	Code Comment Generation(1)	
	Code Completion(5)	Authorship Attribution(1)	
	Code Translation(4)	Named Entity Recognition(1)	
<b>Software quality assurance</b>	Vulnerability Detection(12)	Test generation(1)	13
<b>Software maintainance</b>	Clone Detection(10)	Duplicate Bug Report Detection(1)	28
	Program Repair(7)	Bug Report Summarization(1)	
	Defect Prediction(5)	Bug-Fix Commit Identification(1)	
	Commit Message Generation(2)	Bug Report Classification(1)	

Table 5. Distribution of papers across SE activities

Figure 14 shows a word cloud generated from the abstracts of the collected papers, highlighting that most of the prominent terms are associated with LM4Code. Figure 15 further presents the distribution of language modes used in the collected paper. It is important to note that while both LSTM and GRU are types of RNN, in this study, papers that only specify the use of RNN without further detail are categorized under “General RNN”. Similarly, despite observing the utilization of several popular transformer-based architectures such as CodeBERT, CodeX, and CodeT5, papers that merely claim the use of a self-defined or custom-designed transformer are classified as “General Transformer” in subsequent sections. As depicted in Figure 15, it is evident that the LSTM model has a higher prevalence compared to other types. In the past two years, there has been a significant increase in research inquiries focused on transformer-based language models, specifically targeting pre-trained models like CodeBERT and Codex. These observations are consistent with previous survey studies related to learning-based software engineering [51, 155, 161]. Table 5 presents a summary of the distribution of SE tasks that the collated papers address. Notably, tasks such as code generation, code summarization, code search, vulnerability detection, and clone detection emerge as the dominant scenarios for investigating the pitfalls in LM4Code. These primarily encompass classification tasks, as well as code or text generation challenges in LM4Code.

Overall, our analysis underscores that LM4Code has been an increasing area of interest. Particularly, since 2021, there has been an increasing amount of literature emphasizing the pitfalls of LM4Code. These pitfalls may hinder realistic performance and can affect how reliable and practical LM4Code systems are in real-world situations. Our collected papers cover a wide range of language models and many different software engineering tasks. As we delve deeper into our primary topic in the subsequent sections, this preliminary overview sets the foundation for a comprehensive analysis of the challenges and opportunities in LM4Code.