https://doi.org/10.1007/s11704-025-41376-3

REVIEW ARTICLE

Data preparation and quality for code-centric generative software engineering tasks: a systematic literature review

Shihao WENG, Yang FENG✉, Yining YIN, Zhenlun ZHANG, Baowen XU✉

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

Received December 17, 2024; accepted June 16, 2025

E-mail: fengyang@nju.edu.cn; bwxu@nju.edu.cn

Abstract

The rapid advancements in Deep Neural Networks (DNNs) have revolutionized generative software engineering tasks, including code summarization, program repair, code generation, and code translation. However, the performance of DNN models in these tasks heavily depends on the quality of their training and evaluation datasets. This systematic literature review examines 70 primary studies to comprehensively analyze dataset construction methodologies, prevalent data quality challenges, and solutions proposed to address these challenges. Our findings reveal that dataset construction processes significantly influence quality, with common issues such as noise, redundancy, imbalance, and insufficient granularity undermining model effectiveness. We identify key strategies to mitigate these problems, including data augmentation, automated cleaning techniques, and standardized validation frameworks. Furthermore, we highlight the critical role of dataset diversity and timeliness in improving model generalization. This study provides actionable insights for researchers and practitioners in the era of generative AI, where high-quality datasets are essential for developing reliable language models as software engineering tools. By emphasizing rigorous dataset curation and innovative quality assurance methods, our work bridges the gap between theoretical advancements and practical applications, enabling the creation of robust, generalizable models for real-world code-related tasks. The synthesized recommendations aim to guide future research in optimizing dataset design, fostering reproducibility, and addressing evolving challenges in data-driven software engineering.

Keywords

systematic literature review; data quality; deep learning for software engineering

1 Introduction

The rapid advancements in DNNs have significantly transformed the landscape of software engineering, particularly through the emergence of generative capabilities. In recent years, code-centric generative software engineering tasks [1,2] have garnered substantial attention from both academia and industry. These tasks, such as code summarization [3–6], program repair [7–9], code generation [10–13], and code translation [14], leverage the powerful capabilities of DNNs to automate and enhance various aspects of software development. Throughout this paper, we use the term GSET to specifically refer to these four code-related tasks.

The relationship between DNNs and generative AI in software engineering represents a paradigm shift in how software artifacts are created and maintained. Unlike traditional approaches that generate software artifacts from predefined templates and models, DNN-driven generative software engineering utilizes neural networks to learn patterns from existing code repositories and generate new software artifacts that maintain semantic correctness while exhibiting creativity and adaptability. These neural models process vast amounts of code data, capturing complex relationships between code structure, functionality, and natural language descriptions, thereby enabling tasks that were previously considered to require significant human expertise. Code summarization automates the creation of concise, human-readable descriptions of code snippets, aiding developers in understanding code functionality without examining implementation details. Program repair leverages neural models to automatically identify and fix bugs, enhancing software reliability and reducing debugging time. Code generation produces functional code from natural language specifications, enabling non-experts to develop software and expediting routine coding tasks. Code translation converts code between programming languages while preserving functionality, facilitating cross-platform compatibility and legacy code migration.

These advancements hold substantial promise for improving productivity, reducing errors, and facilitating the maintenance and evolution of software systems [15]. However, the efficacy of DNN models in these tasks is heavily contingent upon the quality of the data used for training and evaluation. High-quality datasets are

paramount as they directly influence the model's ability to generalize, perform accurately, and produce reliable outputs [16,17] across diverse programming contexts. Despite the growing interest in applying DNNs to GSETs, there is a conspicuous gap in the literature concerning the quality of data used in these endeavors. While numerous studies have proposed [6,12] novel models and techniques for various GSETs, the underlying data quality issues often receive insufficient attention [18]. Furthermore, there is an absence of a comprehensive literature review that consolidates existing knowledge on dataset construction processes, identifies prevalent data quality issues, and evaluates the solutions proposed to mitigate these issues.

This systematic review addresses this gap by examining the processes involved in dataset construction for GSETs, the common quality issues that arise in these datasets, and the effectiveness of proposed solutions to these challenges. Through a rigorous analysis of 70 research papers, our review provides insights into the intricate relationship between dataset construction methodologies and data quality in generative software engineering.

Our key findings reveal that dataset construction processes vary widely across different GSETs, often influencing the quality and performance of the resulting models. Common quality issues include lack of diversity, noise, imbalance, and insufficient granularity, which significantly hinder the effectiveness of DNN models. We identify various strategies employed to mitigate these problems, such as data augmentation, cleaning techniques, and novel training methods, and assess their impact on model performance.

Based on our analysis, we propose several recommendations for researchers in this field. These recommendations emphasize the importance of prioritizing data quality in generative software engineering research, exploring robust methodologies for dataset construction, and developing comprehensive frameworks for data quality assessment and enhancement. By adopting these recommendations, researchers can improve the reliability and performance of DNN models in GSETs, driving both scientifically sound and practically relevant advancements.

The main contributions of this study include:

1) To the best of our knowledge, this paper is the first literature review to analyze and summarize the processes of dataset construction and their quality issues in code-centric GSETs.

2) This paper provides a detailed analysis of data quality issues and their impact on model performance, evaluates solutions proposed in the literature, and guides the direction for future research.

3) This paper offers practical recommendations for subsequent researchers on building high-quality datasets, including how to select and clean data, and how to annotate and validate datasets, to enhance the performance of DNN models in GSETs.

The remainder of this paper is structured as follows. Section 2 presents the background and related works. Section 3 provides the methodology used for conducting this review, including our specific research questions. Sections 4–6 present our findings and analysis. Section 7 provides recommendations for subsequent researchers based on our analysis. Section 8 analyzes the threats to validity of this study. Section 9 concludes the paper.

2 Background and related works

This section first introduces each of the GSETs discussed in this paper, then presents some similar reviews, and explains their shortcomings and the necessity of this study.

2.1 The DNN-driven generative software engineering tasks

DNNs have revolutionized numerous fields, including software engineering, by enabling machines to learn complex patterns and generate human-like outputs. DNNs are a subset of machine learning approaches characterized by multiple layers of interconnected neural units that learn hierarchical representations of data. Unlike traditional software engineering methods that rely on explicit programming rules, DNNs excel at automatically discovering intricate patterns in large volumes of code, making them particularly suitable for generative tasks that require understanding both the syntax and semantics of programming languages. Figure 1 illustrates the relationship between DNNs and GSETs. The diagram shows how data flows from various sources through preparation and quality assurance processes before being used to train DNNs. These networks then power the four main GSETs with data quality issues potentially affecting all stages of this pipeline. The feedback loops in the diagram emphasize how data quality impacts the performance of these models and, consequently, the quality of their generated outputs.

The application of DNNs to software engineering has evolved significantly in recent years. Early approaches primarily utilized Recurrent Neural Networks (RNNs) [19] and their variants such as Long Short-Term Memory (LSTM) [20] networks to model code as sequences. While these architectures showed promise, they struggled with capturing long-range dependencies in code and scaling to large codebases. The introduction of transformer-based architectures [21] marked a paradigm shift in DNN applications for code. Transformers use self-attention mechanisms that enable them to consider the entire context simultaneously, rather than sequentially processing code. This architectural innovation has led to a new generation of pre-trained models specifically designed for code, including CodeBERT [10], PLBART [22], and CodeT5 [12]. These foundation models are pre-trained on vast amounts of code data and can be fine-tuned for specific GSETs.

In this paper, we focus on four main GSETs that have been significantly transformed by DNNs:

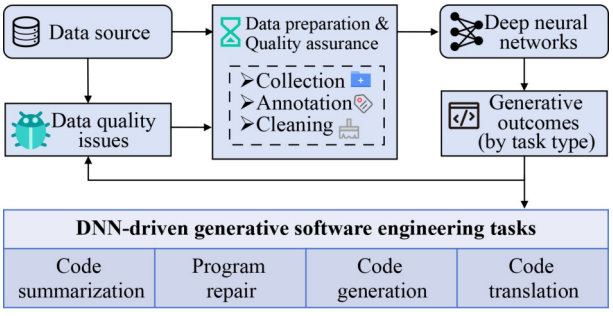

Fig. 1 Conceptual framework of DNN-driven generative software engineering tasks

1) Code summarization [23] is the task of generating concise, human-readable descriptions of code snippets, functions, classes, or entire programs. This task primarily focuses on translating the functionality of code into natural language, aiding developers in understanding the code's purpose without delving into the actual implementation. The importance of code summarization cannot be overstated, as it contributes significantly to software documentation and maintenance. Automated summarization tools can assist in keeping documentation up to date, especially in large, evolving codebases where manual documentation is labor-intensive and often neglected [24].

2) Program repair [8] focuses on the automatic identification and correction of program errors and bugs. This task is crucial for enhancing software reliability, reducing the time spent on debugging, and improving software quality. Through automated program repair, DNN models attempt to understand patterns of common mistakes and propose corrections in real-time, which can be directly applied by developers.

3) Code generation [25] is the task of automatically generating code based on a given input, which can be a natural language description, a formal specification, or a high-level intention. Code generation systems aim to automate the programming process, making it easier for non-experts to develop software or for developers to expedite routine coding tasks.

4) Code translation, also referred to as code transpilation, is the process of converting code from one programming language to another while preserving the program's logic and behavior. This task is critical in scenarios such as migrating legacy codebases, enabling cross-platform compatibility, and facilitating multi-language software development.

These four tasks represent core challenges in software engineering that can be addressed through generative approaches powered by DNNs. By automating these tasks, developers can focus on higher-level design decisions and more complex programming challenges, ultimately increasing productivity and code quality.

The performance of DNNs for these GSETs is inextricably linked to the quality of data used during training. Several characteristics of high-quality datasets are particularly important: volume (DNNs typically require large amounts of data to learn generalizable patterns), diversity (coverage of various programming languages, coding styles, and application domains), correctness (syntactically valid and semantically meaningful code examples), representativeness (reflection of real-world code distribution), and timeliness (inclusion of contemporary programming practices and API usage). Data quality issues can propagate through models and significantly impact their performance. For instance, a model trained on outdated code patterns may generate code that follows deprecated practices, while models trained on noisy data may produce syntactically incorrect solutions. Addressing these data quality challenges is therefore essential for reliable and effective DNN applications in software engineering.

2.2 Related data quality reviews

The importance of data quality in machine learning for software engineering tasks has gained increasing attention in recent years. Several survey studies have explored various aspects of this topic, though few have specifically focused on the preparation and quality assurance of datasets for GSETs.

One related work is the review conducted by Zhang et al. [26], which thoroughly analyzes the data preparation processes for deep learning (DL)-based code smell detection (CSD) [27]. The authors focused on understanding how data is collected, labeled, and cleaned for training DL models. They emphasized the crucial role of high-quality training data in ensuring the models' reliability and scalability. However, despite the comprehensive nature of this work, a significant limitation lies in its narrow focus on code smell detection within the domain of software engineering. While they emphasize the importance of quality data in DL-based CSD, they do not extend their analysis to generative software engineering tasks like code summarization, repair, or generation, which differ significantly from classification tasks due to their open-ended outputs and the need for more complex data preparation and evaluation methods. This limits the generalizability of their findings to broader applications of deep learning in software engineering.

Besides, Croft et al. [28] review 61 papers to identify the challenges of preparing datasets for Software Vulnerability Prediction (SVP). They develop a taxonomy of 16 challenges and map existing solutions, emphasizing the importance of data quality in SVP model performance. The review highlights issues such as data scarcity, inconsistent reporting, and the reluctance of organizations to share sensitive data, offering recommendations for future research. However, while the review offers a valuable synthesis of data preparation challenges and existing solutions in SVP, it lacks a deeper exploration of the broader implications of data quality beyond vulnerability prediction. The review is narrowly focused on vulnerability prediction, which limits its generalizability to other domains of software engineering that also rely on data-driven models.

Another significant review in this domain is Watson et al.'s comprehensive systematic literature review [29] that examines the use of deep learning in software engineering research. This study analyzes 128 papers across 23 unique SE tasks, including generative tasks like code generation, program repair, code summarization, program synthesis, and code translation. Their analysis is structured around the "components of learning" framework, examining how data is extracted, preprocessed, and used for various SE tasks. While they discuss aspects of data preparation and highlight factors affecting model reproducibility, their review doesn't specifically focus on dataset construction processes or quality issues for generative SE tasks. They note that approximately half of the examined studies don't perform exploratory data analysis to combat issues like data snooping or bias, indicating potential concerns about dataset quality. However, their work lacks a detailed analysis of dataset construction methodologies specific to generative tasks, focusing more broadly on DL applications across the SE field.

Ghaisas and Singhal [30] investigated data challenges in integrating Generative AI (GenAI) and NLP into Requirements Engineering (RE). Their work focuses on data quality issues arising

during the lifecycle of AI-driven software systems, particularly in tasks like automated requirements generation and question-answering over specifications. The authors highlight challenges such as data scarcity, annotation subjectivity, and validation complexities in RE-specific contexts, proposing mitigation strategies like transfer learning and rationale-augmented prompting. While their study offers valuable insights into data-centricity in GenAI for SE, it primarily addresses RE tasks (e.g., requirement classification, contract analysis) rather than generative software engineering tasks like code summarization, repair, or translation. This narrow focus limits its applicability to broader code-related generative tasks, which often require distinct dataset construction processes and quality assurance techniques. Besides, Wang et al. [31] conducted a 12-year systematic literature review of 1,428 studies on ML/DL applications in software engineering (2009–2020), offering critical insights into data quality challenges. The review highlighted pervasive issues in data preprocessing, such as noise filtering, imbalanced datasets, and feature engineering, which directly impact model reliability and reproducibility. However, they lack discussion on solutions to data quality issues.

Table 1 provides a comparative analysis of our study against five related survey-based works, highlighting the distinct contributions of our research. Unlike Zhang et al. [26] and Croft et al. [28], which focus on classification tasks rather than generative tasks, our study specifically addresses GSETs. While Watson et al. [29] and Wang et al. [31] do cover some generative tasks in their broader reviews of deep learning applications in software engineering, they lack comprehensive analysis of dataset construction processes or quality solutions specific to these tasks. Ghaisas and Singhal's work [30], though focused on data challenges, is limited to requirements engineering rather than code-related generative tasks. Our study stands out by comprehensively examining the four major GSETs while providing an in-depth analysis of the entire data preparation pipeline, from construction processes to quality issues and their solutions. This approach fills a significant gap in the literature, as no previous review has systematically analyzed all these aspects specifically for generative software engineering tasks, making our contribution particularly valuable for researchers and practitioners working in this rapidly evolving field.

3 Research methodology

Our review follows an established methodology [32,33] adapted from prior review guidelines [26,28] in the software engineering field to ensure a comprehensive and unbiased review.

As shown in Fig. 2, the process of this review involved several steps to ensure a comprehensive and high-quality selection of papers. The process began with a *Predefined Search String*, which was applied across multiple digital databases to retrieve an initial set of papers. After this, *Duplicate Removal* was performed, reducing the number of papers from 4,953 to 3,751. Next, We defined the *Inclusion/Exclusion Criteria* to manually evaluate these papers one by one in order to select those that meet the theme of our review. It consists of two phases, In Phase I, a title and abstract screening was conducted, narrowing the pool to 85 papers. These papers underwent a more detailed examination in Phase II, where the full text was reviewed, resulting in 73 papers that met the preliminary inclusion criteria. After applying these criteria, the papers were assessed for quality through a *Quality Assessment*, which led to 64 papers being

Table 1 Comparison of related survey studies on data quality in software engineering

Study	Focus	GSETs covered	Analysis
[26]	Data preparation process for code smell detection	None	Data construction processes, quality issues, and solutions
[28]	Data preparation process for software vulnerability prediction	None	Data construction processes, quality issues, and solutions
[29]	The application of deep learning in software engineering	Code Generation, Program Repair, Code Summarization, Program Synthesis, Code Translation	Lack analysis of the dataset construction processes
[30]	Data challenges of GenAI in software requirements engineering	Software Requirements Generation, Automated Question and Answering	Data construction processes, quality issues, and solutions
[31]	The application of deep learning in software engineering	Code Summarization, Code Generation, Test Case Generation, Defect Fixing	Lack analysis of the solutions
Ours	Data preparation process for GSETs	Code Summarization, Code Generation, Program Repair, Code Translation	Data construction processes, quality issues, and solutions

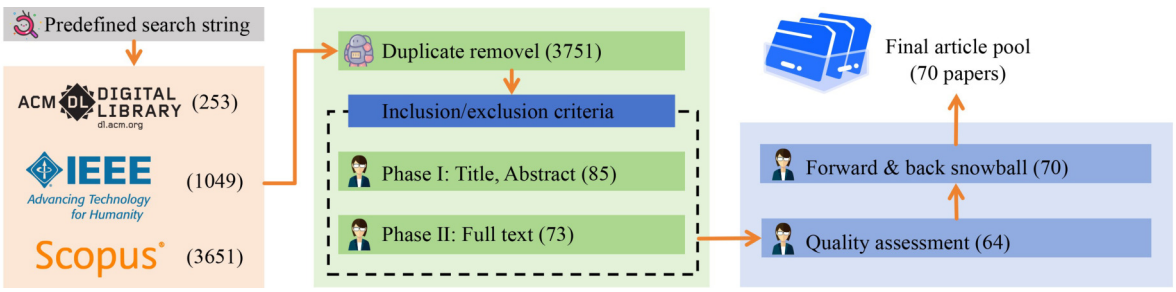

Fig. 2 Process of paper search and selection

retained. To ensure the comprehensiveness of the review, *Forward and Backward Snowballing* was conducted, adding 6 additional papers to the pool. The final set of studies included in the review consisted of 70 papers.

3.1 Research questions

This review studies 70 research papers, providing a thorough examination of the dataset construction processes, the data quality issues encountered, and the solutions proposed to address these issues in the context of DNN-driven GSETs. The review is guided by three primary research questions (RQs):

1) **RQ1**: *What are the common processes and methodologies used in constructing datasets for DNN-driven GSETs?* The main purpose of this RQ is to understand the key processes involved in building datasets for GSETs. The analysis covers three primary aspects: dataset requirements, dataset collection, and dataset annotation methods. For dataset requirements, we analyze the type of task, targeted programming languages, data volume & format, and the timeliness of the data. For dataset collection, we analyze whether the process is fully-automated, semi-automated, or manually reviewed, and identify data sources. For annotation methods, we analyze the specific annotation process, the level of automation in the process, and the annotation granularity of the dataset.

2) **RQ2**: *What are the prevalent data quality issues associated with these datasets?* The main purpose of this RQ is to identify common data quality issues in GSETs and to explore their causes, i.e., to explore the relationship between the dataset construction process and data quality, thereby assisting researchers in constructing GSETs datasets in the future. We analyze the three primary aspects: details of quality issues, their root cause to specific stages of the dataset construction process, and the impacts of these quality issues.

3) **RQ3**: *What solutions have been proposed to address these data quality issues?* The main purpose of this RQ is to explore how existing solutions address data quality issues and analyze their effectiveness, thereby helping researchers optimize dataset construction and processing. We analyze the two primary aspects: the methods used to address data quality issues, their effectiveness, and future directions.

These three research questions form a cohesive framework that contributes to our overall research objective. RQ1 establishes the foundation by mapping out current practices in dataset construction, providing context for understanding where and how quality issues might arise. RQ2 builds upon this foundation by identifying specific quality problems within these datasets, examining their root causes, and assessing their impact on model performance. Finally, RQ3 completes the framework by evaluating existing solutions to these issues, their effectiveness, and areas where further research is needed.

By systematically exploring these questions, this review aims to offer a detailed understanding of the current landscape and to identify areas that require further investigation and improvement. Together, these RQs enable us to provide comprehensive insights into dataset quality in GSETs, ultimately supporting our goal of enhancing the reliability and effectiveness of DNN models in software engineering applications.

3.2 Search strategy

The search strategy is based on the PICO framework [34], a widely accepted method for formulating research questions and structuring search terms in review. The PICO framework ensures the comprehensiveness and relevance of the literature search by breaking the research questions down into four key components: Population (P), Intervention (I), Comparison (C), and Outcome (O). Each component of the framework is aligned with the objectives of this review. Below, we outline how each element of the PICO framework is applied:

1) **Population (P)**: The population of interest in this review refers to the subject matter in the field of software engineering, specifically "Code" or "Programs".

2) **Intervention (I)**: In this review, the intervention refers to the use of deep learning, such as some well-known deep learning techniques applied in the field of software engineering. Additionally, this part should also include keywords related to datasets and dataset quality, which can ensure the inclusion of papers discussing the construction, preparation, and quality of datasets.

3) **Comparison (C)**: In this review, the comparison component is not applicable.

4) **Outcomes (O)**: The outcome of interest in this review is centered around specific GSETs, namely code summarization, program repair, code generation, and code translation.

As shown in Table 2, the search string for this review was constructed based on the above PICO framework. By combining

Table 2 Search terms of our study

Category	Subject	Search terms
Population	Code / Program	"Code" OR "Program"
Intervention	DL Technique /	"Learning" OR "Transformer" OR "Encod*" OR "Deep neural network" OR
	Data Construction /	"Benchmark" OR "Data*" OR "Data Quality" OR "High Quality" OR
	Data Quality Study	"Label Noise" OR "Data Noise" OR "Noise"
Comparison	–	–
Outcomes	GSETs	"Code Generat*" OR "Text to Code" OR "Natural Language to Code" OR
		"Code Summar*" OR "Code Repair" OR "Program Repair" OR "Code Translat*"

these three parts using the boolean operator "AND", we can make the search results as consistent as possible with the research content of this review. The boolean operator "OR" was used within each part to combine synonyms and related terms. We also employed wildcards like "*" to capture variations of key terms. For example, "Code Summar*" was used to include both "Code Summary" and "Code Summarization." This approach helped ensure that the search string would capture papers even if the terminology varied slightly between papers, thus broadening the scope of relevant results. Note that we deliberately excluded the broader term "generative AI" from our search string as preliminary searches showed it introduced numerous irrelevant results while the task-specific terms already effectively captured the relevant literature.

This search string was applied to three major academic databases: ACM Digital Library, IEEE Xplore, and Scopus. These databases were selected due to their extensive and authoritative coverage of high-quality literature in computer science, software engineering, and deep learning. ACM Digital Library is a central repository for computing and AI-related research. IEEE Xplore is a leading source of technical literature for engineering and computing research. Scopus, being a multidisciplinary database, offers broad coverage of high-quality papers in technology, engineering, and computer science, covering a wide range of relevant research. When applying the search string to these databases, we focused on searching within the titles, abstracts, and keywords of the papers. After applying the search string, we initially retrieved 4953 papers: 253 papers from ACM Digital Library, 1049 papers from IEEE Xplore, and 3651 papers from Scopus. We downloaded all of these papers and used a deduplication tool [35] to remove duplicates, which reduced the total number of papers to 3751. We selected these three specialized databases rather than broader search engines like Google Scholar as they collectively provide comprehensive coverage of high-quality software engineering literature while maintaining domain-specific focus and minimizing irrelevant results.

3.3 Paper selection

Our aim is to meticulously select qualified papers to facilitate subsequent discussions on the construction and quality issues of datasets for GSETs. Therefore, this screening process is divided into the following three steps. It is worth noting that our selection process did not distinguish between preprints and formally published papers. Given the rapidly evolving nature of research in generative software engineering tasks.

3.3.1 Inclusion/exclusion criteria

Table 3 outlines the inclusion and exclusion criteria used to screen the papers for our systematic literature review on GSETs. As shown in Fig. 2, the screening process was conducted in two phases: First, we manually assessed the titles and abstracts of the 3,751 papers retrieved in the previous step. Then, we review the full texts of the shortlisted papers. Our aim was to identify papers that contribute meaningfully to the discussion on dataset construction, data quality issues, and solutions for GSETs.

The inclusion criteria aimed to capture works that introduced new benchmarks or datasets relevant to the field analyzed the quality of existing datasets, or proposed solutions to identified data quality issues. Specifically, papers were included if they proposed a new benchmark or dataset (IC1), conducted a detailed analysis of data quality issues within existing datasets (IC2), or put forward novel approaches to address these data quality issues (IC3). This allowed us to focus on works that directly contribute to the development and improvement of datasets used in GSETs, which is central to our review.

Conversely, the exclusion criteria were designed to filter out papers that did not meet the necessary scope. Papers were excluded if they were not written in English (EC1), as non-English papers would introduce language barriers that could hinder consistent interpretation and evaluation. Additionally, papers that did not provide detailed information about the datasets they used were excluded (EC2). This exclusion criterion ensured that each selected paper had sufficient detail on the datasets, making it possible to thoroughly analyze the dataset construction process and any associated quality issues. Finally, literature review papers were also excluded (EC3), as our focus was on original research contributions rather than secondary analyses. The result of this two-phase screening process was a final set of 73 papers.

3.3.2 Quality assessment

We conducted a rigorous quality assessment of the 73 papers identified during the previous screening process. The quality assessment was guided by a quality criteria checklist shown in Table 4 that evaluated the clarity of dataset construction processes, the identification of data quality issues, and the proposed solutions for addressing such issues.

For papers focusing on constructing new datasets, the assessment considered three key aspects. First, we examined whether the data collection method was clearly explained (QC1). This involved determining whether the authors provided a comprehensive description of where the data was sourced and how it was gathered, including any automated or manual processes. Second, we evaluated the clarity of the data annotation process (QC2), which involved reviewing whether the paper explained how the data was labeled,

Table 3 Inclusion and exclusion criteria for manually screening papers

Inclusion criteria	Exclusion criteria
IC1: The paper proposes a new related benchmark / dataset.	**EC1:** The paper is not in English.
IC2: The paper analyzes the data quality of existing datasets and identifies some data quality issues.	**EC2:** The paper does not provide detailed information about the dataset used.
IC3: The paper proposes solutions to the data quality issues.	**EC3:** The paper is already a review article.

Table 4 Quality criteria checklist for screening papers

Paper on constructing a new dataset
QC1: Is the data collection method explained clearly?
QC2: Are the details of data annotation explained clearly?
QC3: Is the data format of the dataset clearly described?
Paper on discussing data quality issues
QC4: Are the data quality issues explained clearly?
QC5: Is there an argument presented regarding the impact of these quality issues?
Paper on proposing solutions
QC6: Are the proposed solutions described clearly?
QC7: Is there evidence showing the impact on the model after resolving these issues?

who conducted the labeling (e.g., experts or crowdsourcing), and any tools used. Finally, we assessed whether the data format of the dataset was adequately described (QC3), ensuring that readers could understand how the dataset was structured and could verify its usability for future research.

For papers discussing data quality issues, we focused on two additional criteria. The first was whether data quality problems were clearly explained (QC4), which required a detailed account of the specific problems encountered, such as noise, imbalance, or redundancy in the dataset. The second criterion was whether the paper provided a substantive argument about the impact of these quality issues on the performance and reliability of GSETs models (QC5). This step was crucial in understanding how data issues might contribute to reduced model accuracy or generalizability.

Finally, for papers proposing solutions to data quality issues, we evaluated the clarity of the proposed solutions (QC6) and whether there was evidence showing the impact of these solutions on model performance (QC7). This involved reviewing whether the paper demonstrated improvements in model accuracy, robustness, or scalability after addressing the identified data quality issues. The inclusion of empirical results or comparative studies provided further validation of the proposed methods. After careful manual evaluation, the final set of papers was narrowed down to 64.

3.3.3 Forward and backward snowballing

In the forward and backward snowballing process, we aimed to ensure a comprehensive and exhaustive review of the relevant literature, particularly focusing on the dataset preparation and data quality issues in DNN-driven GSETs. We employed the snowballing method to capture significant papers that may have been missed during the initial database search. This process involves reviewing the references (backward snowballing) and citations (forward snowballing) of the previously selected papers to identify additional relevant studies.

In this step, we applied snowballing to the 64 papers that had successfully passed the inclusion and quality assessment phases. By examining the reference lists of these papers, we identified key studies that were frequently cited in discussions related to dataset construction and data quality issues in GSETs. These referenced papers often provided foundational insights or proposed novel methods that were critical to our research questions. Through backward snowballing, we were able to locate several studies that had been instrumental in shaping the current understanding of data quality challenges in deep learning for software engineering but were not captured in our initial search string results. Similarly, forward snowballing was employed by tracking which of the 64 papers had been cited in subsequent research. This strategy allowed us to identify emerging trends and more recent solutions that addressed the evolving challenges of data quality in GSETs.

As a result of the forward and backward snowballing process, we identified an additional 6 papers that directly contributed to the topics of dataset preparation, data quality issues, and solutions in the context of GSETs. This brought the total number of papers included in our review to 70, ensuring a more comprehensive and robust analysis of the existing literature.

3.4 Data extraction

The data extraction process in our review followed a structured approach to collect information from the 70 selected papers, enabling us to address our research questions effectively. Table 5 was designed to systematically capture key information related to the dataset construction processes, data quality issues, and the proposed solutions, in alignment with our research questions (RQ1, RQ2, and RQ3). The extracted data are classified into categories such as metadata, RQ1-data, RQ2-data, and RQ3-data, ensuring comprehensive coverage of each paper's contributions.

The Metadata section in our data extraction table includes basic but essential information, such as the title of the paper, the relevant research questions (RQ Type), the specific research focus of the paper (Paper Type), and the GSETs it pertains to. Additionally, the metadata captures details about the proposed or researched dataset, including its name, the programming languages it targets, and any other distinguishing features. This categorization provides a foundation for understanding each paper's scope and relevance to our study.

For RQ1, which focuses on the dataset construction processes, we extracted information about the data sources, annotation methods, collection methods, dataset volume, format, granularity, and timeliness. This information is critical for understanding how GSETs datasets are built and how the construction process influences data quality. For instance, the data sources refer to whether the datasets are derived from open-source repositories, industry projects, or other sources. The annotation methods explore whether the datasets were manually labeled, semi-automatically annotated, or fully automated. Collection methods and volume provide insights into the scale and comprehensiveness of the datasets, and the granularity (file, function, or line level) and timeliness (time period of collected data) offer further details about the dataset's structure and relevance.

For RQ2, which investigates data quality issues, we extracted data concerning the specific issues identified in the papers, their root cause to particular stages in the dataset construction process, and the impacts these issues have on model performance. Common issues

Table 5 Data extraction forms used for collecting information from the papers in our study

	Item	Description
Metadata	Title	Title of the paper.
	RQ Type	Our RQs addressed by the paper.
	Paper Type	Specific research content of the paper.
	GSETs	Specific task the paper belongs to.
	Dataset name	The name of the dataset proposed or researched in the paper.
	Language	The programming languages targeted by the dataset proposed in the paper.
RQ1-data	Data source	The data sources for the dataset proposed in the paper.
	Annotation method	Annotation method of the dataset.
	Collection method	Collection method of the dataset.
	Volume	The amount of data in the dataset.
	Format	Data format of the dataset.
	Granularity	The data is at the file level, function level, or line level.
	Timeliness	The time period to which the collected data belongs.
RQ2-data	Issues	Data quality issues discussed in the paper.
	Root cause	The data construction steps attributed to data quality issues.
	Impact	The impacts caused by data quality issues.
RQ3-data	Solutions	The approaches proposed in the paper to address data quality issues.
	Effect	The role of the solution in the specific task.
Others		Other valuable findings for this study.

include data sparsity, noise, imbalance, and lack of representativeness, which can hinder the generalization and accuracy of DNN models in GSETs.

For RQ3, which explores solutions to data quality problems, we categorized the proposed solutions and evaluated their effectiveness. Papers often propose techniques such as data augmentation, noise reduction, or the development of more representative datasets to mitigate data quality issues. The "Effect" column in our data extraction table records the impact of these solutions on the performance of DNN models in the corresponding GSETs.

Additionally, we classified the 70 selected papers by RQ Type, Paper Type, Annotation Type, and GSETs in Tables 6 and 7, which provides a comprehensive overview of the focus areas of each paper. Specifically, in the above tables, we separately categorize traditional program repair and program repair using deep learning methods. The models of the latter are usually evaluated on the datasets of the former, so it is necessary to discuss both. Figure 3 shows the distribution of the publication years, illustrating trends in research attention over time.

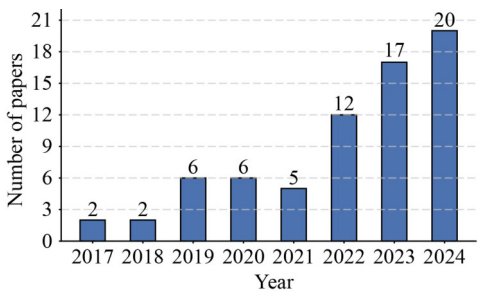

Fig. 3 Number of selected papers by year

4 RQ1: common process for constructing datasets of GSETs

This section systematically analyzes the dataset construction processes in the context of four specific GSETs. We discuss each specific task separately because each task presents unique requirements for dataset curation, including differences in the data sources, annotation methods, collection processes, and dataset formats. By discussing the papers relevant to each task, this section explores how datasets are tailored to meet the specific challenges of each GSETs and highlight the importance of these processes in ensuring high-quality training and evaluation data for DNN models.

4.1 Dataset requirement

We first uniformly examine the programming languages represented in the datasets and their relative proportions, as shown in Fig. 4. It provides statistical insight into the distribution of programming languages across the datasets reviewed, revealing that languages such as Python, Java, and C/C++ dominate the datasets, while other

Table 6 Papers we selected for our review and their classification

References	RQ type			Paper type					Annotation type			GSETs
	RQ1	RQ2	RQ3	CND	RED	MDC	DA	DC	FA	SA	MR	
[36–38]	√			√					√			CS
[39]	√			√						√		CS
[40]	√	√	√	√	√	√				√		CS
[41]	√	√	√	√	√				√			CS
[42]	√	√	√	√	√						√	CS
[43]		√	√		√			√				CS
[44]		√	√			√						CS
[45]		√	√		√		√					CS
[46]		√	√		√							CS
[47]	√			√		√				√		PR-T
[48]	√			√		√			√			PR-T
[49–52]	√			√							√	PR-T
[53,54]		√			√							PR-T
[55]		√	√		√			√				PR-T
[56–58]	√			√		√				√		PR-L
[59]	√			√					√		√	PR-L
[60]	√	√		√	√				√			PR-L
[61–64]	√			√					√			PR-L
[65]		√	√		√							PR-L
[66]	√			√		√			√			CG
[67,68]	√	√	√	√	√						√	CG
[69–72]	√			√							√	CG

[1] CND: constructing a new dataset; RED: research on existing datasets; MDC: method for dataset construction; DA: data augmentation; DC: data cleaning; FA: fully-automatic; SA: semi-automatic; MR: manual review; CS: ccode summarization; PR-T: Program repair - traditional methods; PR-L: Program repair - learning methods; CG: code generation; CT: code translation.

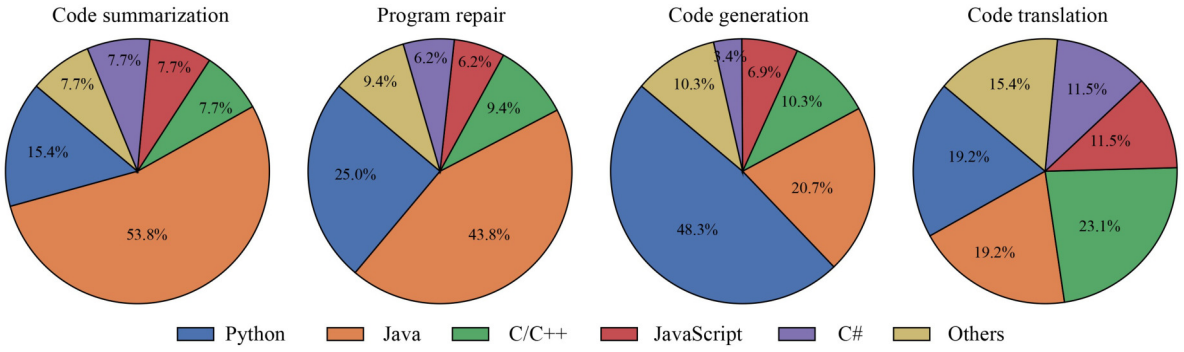

Fig. 4 Proportion of various programming languages in GSETs datasets (statistics from selected papers)

languages like JavaScript and Ruby are less commonly featured. This analysis helps to contextualize the focus of current research and highlights potential gaps in dataset diversity across different programming languages.

● **Code summarization**

Targeted programming languages: For code summarization, datasets are primarily built around Java, a popular statically-typed language with rich object-oriented features. Java's abundant open-

Table 7 Papers we selected for our review and their classification (continued)

References	RQ type			Paper type					Annotation type			GSETs
	RQ1	RQ2	RQ3	CND	RED	MDC	DA	DC	FA	SA	MR	
[73–79]	√			√						√		CG
[80]	√	√		√	√						√	CG
[81]	√					√				√		CG
[82–84]	√			√					√			CG
[85]		√	√				√					CG
[86]		√	√		√			√				CG
[87]	√	√	√	√	√				√			CT
[88]	√			√							√	CT
[89]	√	√	√	√	√					√		CT
[90,91]	√			√					√			CT
[92,93]		√	√		√		√					CT
[94]		√	√				√					CT
[95]		√			√							CT
[96]	√			√					√			CS, CG, CT
[97]		√	√		√		√					CS, CT
[98]		√			√							CS, PR-L, CG, CT
[99]	√	√	√	√	√				√			CS, PR, CG, CT
[100]	√	√	√			√				√		PR-T, PR-L
[101]	√	√	√	√	√					√		PR-T, PR-L
[102]	√			√							√	PR-T, PR-L
[103,104]	√			√					√			PR-T, PR-L
[105]	√	√	√	√	√						√	PR-T, PR-L

source repositories provide ample code-comment pairs, such as those used in [36,38].

Data volume and format: Many datasets are designed to include millions of samples, providing sufficient data to train models effectively. For example, large datasets such as those presented in [37,38,40,41] focus on pairing methods or functions with corresponding comments, capturing diverse programming scenarios to enhance model generalization. Others, like [96], adopt a multi-task approach, combining 43 million code-text pairs with additional unimodal samples and line-level comments to support a range of learning objectives. Some datasets, such as [36], incorporate additional metadata, like call dependencies, to improve a model's understanding of contextual and structural information in code. Complementing these large-scale datasets, smaller, manually annotated collections, such as the 6,645 code-comment pairs provided by [42], offer high-quality benchmarks for evaluation and ensure consistency in model performance comparisons. Together, these datasets strike a balance between scalability and precision, addressing the varying requirements of code summarization tasks.

Timeliness of the data: The relevance of datasets is crucial, as outdated codebases may lead to incorrect summaries and reduce model applicability [106,107]. Despite the impact of language and framework evolution, none of the studied datasets explicitly address data timeliness, raising concerns about outdated models and potential data leakage due to unclear timestamps.

- **Program repair**

Targeted programming languages: Java and Python are the most frequently supported languages in program repair datasets, as shown in Figure 4. Datasets like [47–49,62,101–103,105] focus on Java, while [51,58,63] target Python. Some datasets, such as [57], focus solely on C/C++, while others like [52,60] support both Java and Python. More comprehensive datasets, like [50,56,64,100,104], cover a wider range of languages, including JavaScript and C#.

Data volume and format: Datasets for classical program repair are typically smaller and include complete test cases, while deep learning models require much larger datasets. Classic datasets like [47,49–52] offer detailed explanations and annotations, often used to evaluate

deep learning models. For example, [47] contains 251 errors from 72 projects, while [49] includes 283 bug-fix pairs, and [52] provides 80 program files with single-line bugs. Larger datasets for deep learning models include [56], which contains over 921,825 bug-fix pairs, and [59], with 653,606 bug-fix pairs. Other large-scale datasets like [60,61,63,64] provide hundreds of thousands of bug-fix instances across multiple languages.

Timeliness of the data: Timeliness is crucial for program repair datasets to ensure current, relevant data. For example, [56] uses GitHub events from January to September 2020, and [59] collects data from PHP projects active after January 2021. Several datasets, such as [51,100,101], restrict bug-fix submissions to those made in 2023 to keep data up-to-date. Additionally, datasets like [50] include "Obsolete" comment-edit pairs, reflecting updates necessary to align with current technology. Overall, program repair datasets prioritize the timeliness and activity of projects to maintain practical relevance.

● Code generation

Targeted programming languages: Python is the most commonly used language for code generation datasets, as shown in Fig. 4. Datasets like [69–72,76,78,79,82,83] support only Python, while [66,67] focus on Java, [73] on C/C++, and [80] on JavaScript. Some datasets, such as [74,75,96], support more than three languages. In summary, Python dominates in code generation datasets.

Data volume and format: Code generation datasets typically pair natural language descriptions with corresponding code. Large-scale datasets such as [66,73–76,79,82–84,96] are used for training and validation. For instance, [66] provides 13,792 and 10,069 prompt-code pairs, while [76] offers 1.5 million Jupyter notebook examples. Other datasets like [82] and [96] contain hundreds of gigabytes of code in multiple languages, paired with natural language descriptions. Smaller, high-quality evaluation datasets like [67–70] focus on validating models. For example, [67] includes 1,208 Java API problems, and [69] manually collects 1,000 Python problems covering data science libraries.

The timeliness of the data: Five papers mention dataset timeliness. [68] focuses on open-source projects from the past five years. [69] uses Python 3.7.10, while [75] collects Reddit posts from Dec 2022 to Jan 2023. [76] gathers Jupyter notebooks created before May 2019, and [84] spans problems submitted from 2006 to 2022, factoring in publication date to assess problem difficulty.

● Code translation

Targeted programming languages: The distribution of programming languages in code translation datasets is relatively balanced, as shown in Fig. 4. [88] supports C/C++ and Fortran translation, [90] covers C/C++, Java, and Python, and [89] includes Python, C/C++, C#, and Java. Notably, [87] supports 45 programming languages, while [91] covers 9 languages.

Data volume and format: Code translation datasets are commonly composed of paired code samples written in different programming languages, enabling models to learn cross-lingual mappings between codebases. For instance, some datasets are designed at the program level, including tens of thousands of samples, often enriched with unit tests to ensure functional correctness [87,91]. Others focus on function-level translations, offering parallel functions across languages like C++, Java, and Python to capture syntactic and semantic equivalences [88–90]. Additionally, datasets may emphasize problem-solving contexts, where each problem is paired with multiple solutions and extensive test cases to evaluate correctness [91]. This variety in dataset design, ranging from program-level to function-level translations, reflects the need for both scalability and language diversity in code translation tasks.

The timeliness of the data: None of the collected papers addressed dataset timeliness. This is a critical issue, as outdated datasets may not reflect current programming standards or technologies. Future work should prioritize timeliness to ensure datasets remain relevant and practical for evolving code translation tasks.

4.2 Dataset collection

Collection methods: Based on our statistics of the selected papers, we categorized data collection methods into three types: fully automated, semi-automated, and manual:

1) The fully automated data collection method relies on data crawlers, API queries, and existing large-scale code repositories (such as GitHub and GitLab) to automatically acquire code data. The advantage of this approach lies in its high efficiency and large scale, allowing for the rapid collection of vast amounts of code data.

2) The semi-automated approach combines automated tools with a certain degree of manual intervention, generally involving initial data collection through automated crawling or querying, followed by manual verification, filtering, and cleaning rules to ensure data quality.

3) Manual collection refers to completely manual acquisition and organization of data, typically relying on experts to carefully select data or code snippets.

Figure 5 summarizes the number of different data collection methods in GSETs. For code summarization, most datasets were collected automatically, as seen in several studies [36–39,41,96]. A smaller number of researchers opted for semi-automated approaches, such as [40], or manual methods, like the work in [42]. Similarly, program repair datasets show a preference for automated collection,

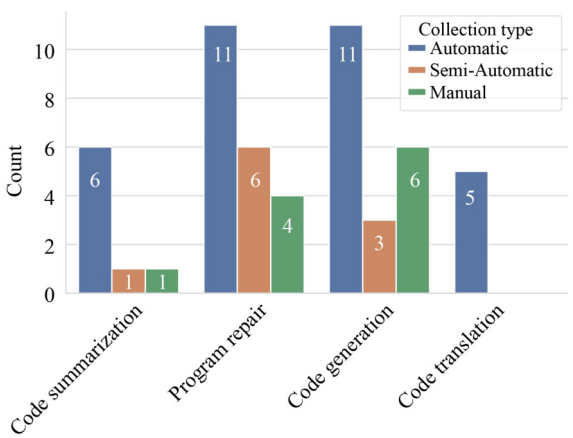

Fig. 5 Number of different collection types in GSETs (statistics from selected papers)

with numerous works [48,58–64,100,101,103] employing this method. However, a notable portion of studies [47,50,51,56,57,104] adopted semi-automated techniques to balance efficiency and quality, while others [49,52,102,105] relied on fully manual curation to ensure accuracy. In the domain of code generation, automated methods dominate [66,68,73,74,77–79,82–84,96] but are supplemented by semi-automated [75,76,81] and manual approaches [67,69–72,80] to address specific challenges. Interestingly, for code translation, all reviewed datasets [87–91] were collected automatically, with no use of semi-automated or manual methods.

In summary, the majority of datasets across tasks were collected automatically, with fewer datasets using semi-automated or manual collection methods.

Data source: As shown in Fig. 6, these datasets are derived from various data sources, each of which plays a unique role in shaping the input and output for a particular task. Six primary sources of data have been identified in our review of existing papers:

1) GitHub: As the largest platform for hosting open-source projects, GitHub is an important source of data for GSETs. It provides a vast repository of real-world code across various programming languages, offering diverse code samples that can be used to train models for code summarization, code generation, and program repair. GitHub's diverse repositories, from small personal projects to large-scale industrial applications, make it a valuable resource for capturing a wide range of programming practices, patterns, and bug fixes. Moreover, GitHub data is often used because it allows researchers to collect metadata such as commit messages, which can be linked to the actual code, making it particularly useful for tasks like code summarization [37,38,42] and program repair [47,48,51,56–59,61–64,100–103].

2) StackOverflow: It is another important data source. As an online forum, StackOverflow offers developers a wealth of authentic data on common coding issues and solutions. The question-and-answer format of the platform is particularly useful for tasks such as code generation [67,69,71,81], as the site contains numerous question-code pairs. StackOverflow data can also be used for code summarization [38,39], where natural language descriptions of code snippets in discussions serve as useful training examples. The community-edited nature of StackOverflow ensures that the data quality is relatively high, with answers often peer reviewed.

3) Specific project: Specific well-known open-source projects, such as those hosted by the Apache Foundation or Linux, represent another crucial source of data. These projects are typically high-quality, extensively tested, and widely used in the software development community. Datasets derived from these projects are often used in program repair [57,105], as they provide a stable, well-documented codebase. The high visibility and long-term maintenance of these projects ensure that the code is reliable, making these sources ideal for training models that require clean and structured inputs.

4) Programming website: Programming and competition question websites, such as LeetCode, Codeforces, and HackerRank, provide highly structured and diverse datasets. These websites host a wide variety of algorithmic problems, each accompanied by detailed problem descriptions, test cases, and often multiple correct solutions in different programming languages, thus they are particularly useful for code generation [83,84]. Besides, many of these websites feature multiple implementations of a single problem in different languages, which makes them ideal for code translation [87,90,91]. Additionally, programming and competition question websites are also suitable for constructing program repair datasets [52,60,104], as they often contain user-submitted solutions that include both correct and incorrect implementations, providing a rich source of examples where models can learn to identify and fix common errors or suboptimal code patterns.

5) Other datasets: Some datasets [40,41,49,74,77,79,82,89,96] are constructed by modifying or combining existing datasets. Researchers optimize or expand on the original dataset, improving the data quality or making it more suitable for other task scenarios.

6) Others: Other miscellaneous sources include data from proprietary projects, academic assignments, or manually curated collections. These sources are less common but offer highly specialized data for niche tasks.

Analyzing Fig. 6, the data needs for tasks like code summarization, generation, repair, and translation vary, leading to diverse sources. GitHub is the most common in code summarization and repair due to its range of real-world code. StackOverflow aids code generation with its structured Q&A format. Program repair often uses clean, well-maintained open-source projects, while code generation benefits from competition websites offering multi-language solutions. Despite these differences, many datasets are adapted or augmented from existing sources, with some niche datasets addressing specific needs.

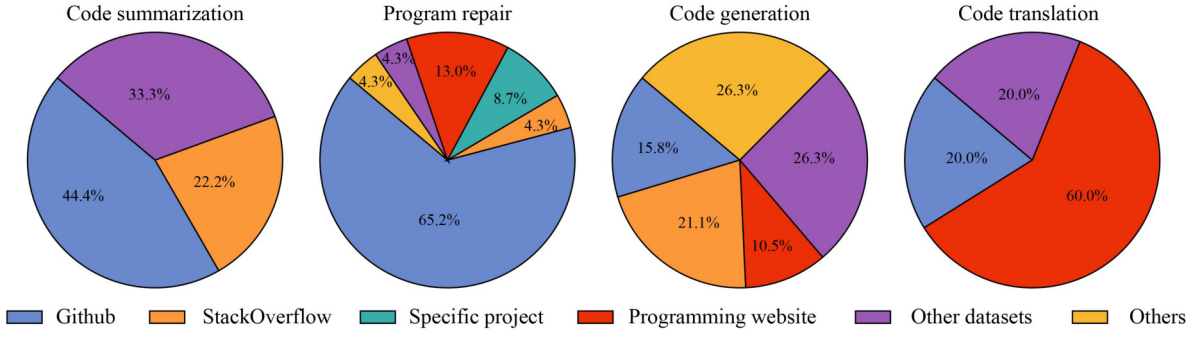

Fig. 6 Proportion of dataset sources in different GSETs (statistics from selected papers)

4.3 Dataset annotation

We categorize data annotation methods into three types: fully automated, semi-automated, and manual annotation. Below is an introduction to each type of annotation.

1) Fully automated annotation: This method typically relies on existing tools and technologies to automatically annotate training and evaluation data, such as extracting relevant source and target data using static analysis tools, Abstract Syntax Tree (AST) extraction, and code refactoring tools. The advantage of this approach is its speed and scalability, allowing it to handle large datasets, which is especially suitable for training large-scale machine learning models. However, the accuracy of fully automated annotation depends on the capabilities of the tools and may introduce errors or noise.

2) Semi-automated annotation: This approach combines automated tools with human review. After an initial automated annotation, humans intervene to check and correct errors or inaccuracies in the dataset. This method ensures data quality while maintaining the scale of the data. It reduces the workload compared to purely manual annotation, but the data quality still depends on the specific manner and extent of human involvement, and it requires a certain amount of human labor cost.

3) Manual annotation: Manual annotation relies entirely on human experts to analyze and annotate the data. Usually, domain experts in programming examine and annotate the code line by line to ensure its accuracy. The advantage of manual annotation is the extremely high data quality, which is particularly suitable for small-scale datasets or tasks that require high precision. However, the disadvantage is that it is time-consuming and labor-intensive, difficult to scale to large datasets, and may be subject to human subjectivity.

Furthermore, according to our statistics, we also classify data annotation granularity into three levels: file-level, method/function-level, and line-level.

1) File-level annotation: It is typically applied in large-scale program repair tasks, especially in scenarios where the entire file needs modification or repair. This granularity covers complete source code files, capturing all contextual information within the file, allowing the model to understand global code structure and logic. File-level datasets usually contain many lines of code and various functions, enabling the model to learn more complex code relationships and potential dependencies, which is suitable for tasks requiring modifications to the entire file.

2) Method/Function-level annotation: Annotation at the method or function level is a common granularity in tasks such as code summarization and code generation. Each sample in the dataset corresponds to an independent function or method, allowing the model to focus on analyzing local code logic without considering the context of the entire file. This granularity provides more detailed code fragments than file-level while still retaining relatively complete semantic information of units like functions or methods.

3) Line-level annotation: It is the most detailed and is often used for tasks requiring repair or modification of specific lines of code, such as program repair. Each sample in the dataset typically includes a line of code and its context, and the model needs to make predictions or generate repair suggestions based on the given code line. This annotation method is precise to the smallest unit of code, suitable for handling fine-grained error repair and detail optimization issues.

Below, we introduce the detailed annotation processes for these four task datasets separately.

- **Code summarization**

Annotation process: The task of code summarization relies heavily on accurately aligning code with natural language comments, making precise annotations essential to avoid introducing noise. To achieve this, researchers have adopted a variety of strategies. A common approach involves directly treating function comments, such as docstrings, as summaries of their corresponding functions. In some cases, automated rules have been implemented to filter out noisy or irrelevant comments, enhancing the dataset's overall quality [36–38,41]. Another strategy combines manual review with automated preprocessing. For example, some datasets involve manually identifying noise patterns in the data and applying rule-based denoising methods to improve data quality [40]. Furthermore, StackOverflow-based datasets have been constructed by automatically extracting pairs of natural language (e.g., question titles) and code snippets, followed by creating a gold-standard dataset through limited manual annotation to train classifiers that align natural language with code [39]. Advanced parsing tools, such as customized tree-sitter parsers, have also been applied to extract functions, classes, and methods from multilingual datasets, ensuring a diverse and well-structured representation [96]. For smaller-scale but highly accurate datasets, manual efforts are sometimes prioritized. For instance, one dataset involved researchers categorizing comments from thousands of Java files into multiple categories, resolving annotation conflicts through multi-reviewer arbitration, and dedicating hundreds of hours to ensure consistency and reliability [42]. These varied efforts underscore the importance of tailoring annotation methods to the specific needs of code summarization tasks, balancing scale with quality control.

Annotation type: Five papers [36–38,41,96] employed fully automated annotation methods, two papers [39,40] used semi-automated annotation methods, and one paper [42] adopted manual annotation methods.

Annotation granularity: Seven papers [36–42,96] were annotated at the method-level granularity. Moreover, the annotation granularity of [42] is at the line-level, where each comment in the dataset is precisely matched with the specific lines of code it is associated with.

- **Program repair**

Annotation process: Program repair datasets often rely on automated tools to collect bug-fix pairs, particularly from sources like GitHub pull requests. After initial automated collection, these pairs are typically filtered using bug-fix keywords or repair templates, followed by manual screening to ensure high-quality data and test cases [51,56,59,61–64,100–103]. Some methods leverage continuous integration tools, such as Travis CI, to identify build failures and corresponding repair patches, which are then stored in

public repositories for broader community use [47]. In other cases, datasets are generated by injecting defects into existing programs, allowing researchers to create controlled bug-fix pairs for analysis [57]. Filtering out irrelevant changes, such as non-bug-related refactorings, is another common approach, often employing tools to ensure that the remaining patches are both compilable and testable [48]. Annotation processes vary widely, with some focusing on clustering repair fragments using abstract syntax tree (AST) distances, while manually classifying these fragments into categories like bug fixes and refactorings [58]. Others track competitive programming submissions to identify minimally faulty solutions, annotating them with fault locations and pairing them with correct versions [60]. Line-by-line annotations of bug fix commits are also common, where disagreements between annotators are resolved through consensus [105]. Beyond these approaches, datasets like Defects4J are augmented with issue descriptions from platforms like GitHub, while others extract comment-edit pairs from StackOverflow, manually verifying their alignment to ensure accuracy [49,50]. Lastly, some datasets involve translating programs between languages with meticulous attention to preserving bug consistency, or they employ differential testing to label outputs for correctness, ensuring datasets meet high standards of validity [52,104]. These diverse methodologies highlight the importance of combining automated tools with manual verification to construct robust program repair datasets.

Annotation type: Eight papers [48,60–64,103,104] employed fully automated annotation methods, six papers [47,56–58,100,101] used semiautomated annotation methods, and six papers [49–52,102,105] adopted manual annotation methods. Specifically, [59] used a fully automated annotation method for constructing the training set and employed manual annotation for the test set.

Annotation granularity: Nine papers [47,48,51,59,60,63,100,101,104] were annotated at the filelevel granularity, five papers [49,50,57,64,102] were annotated at the method-level granularity, and four papers [52,62,103,105] were annotated at the line-level granularity. Specifically, [58] performed annotations at both the method-level and line-level granularities, while [56,61] annotated at all three granularities.

● Code generation

Annotation process: Annotation methods for code generation datasets span a range of automated and manual approaches, often tailored to the specific challenges of capturing the alignment between natural language and code. Many datasets are automatically constructed by scraping tasks and solutions from programming platforms, leveraging tools to ensure functional correctness without requiring manual intervention [79,84]. On the other hand, manual annotation techniques are frequently employed for datasets that require detailed natural language descriptions, context, or function-specific details. For instance, some datasets focus on annotating "how-to" questions from StackOverflow by pairing them with detailed code snippets and contextual information, such as imports and usage intent [71,81]. To ensure high-quality annotations, experts often review and refine questions and reference solutions, checking for code executability and adding constraints and test cases where necessary [69,77]. More specialized methods use program dependency analysis to extract meaningful metadata, as in cases where docstrings are manually annotated to align with corresponding functions [68]. Tools like topic modeling and OCR are also employed to extract and annotate code from social media posts, with manual review ensuring the accuracy of the extracted data [75].

In addition to natural language alignment, some datasets focus on generating multilingual code representations, using automated rule transformations or advanced models like GPT-4 for back-translation, with manual checks to validate translation quality [74,82]. Rule-based augmentation techniques are also used to expand datasets by generating natural language descriptions from code snippets [78]. For highly domain-specific tasks, manual efforts play a critical role, such as annotating StackOverflow questions with API usage rules or pairing JavaScript expressions with Chinese descriptions while manually filtering for accuracy [67,80]. Some datasets are constructed entirely through manual processes, with hundreds of hours spent ensuring the quality of task descriptions, solutions, and test cases [70]. By contrast, automated frameworks simplify the construction of large-scale datasets by synthesizing text-code pairs through function identifier matching, ensuring proper tokenization and alignment [66]. These diverse annotation strategies reflect the balance between automated scalability and the need for meticulous manual review to create high-quality datasets for code generation.

Annotation type: Five papers [66,82–84,96] employed fully automated annotation methods, eight papers [73–79,81] used semi-automated annotation methods, and seven papers [67–72,80] adopted manual annotation methods.

Annotation granularity: Four papers [73,77,83,84] were annotated at the file-level granularity, twelve papers [66–70,72,74,78–80,82,96] were annotated at the method-level granularity, and two papers [71,81] were annotated at the line-level granularity. Specifically, [75] performed annotations at all three granularities.

● Code translation

Annotation process: The construction and annotation of datasets for code translation tasks often rely on obtaining multilingual implementations of the same problem from programming platforms and repositories. Verified repositories such as tutorial websites and example libraries are commonly used as sources, with unit tests ensuring the functional correctness of translated code [87]. In some cases, automated tools are employed to map and standardize code pairs, as demonstrated with OpenMP Fortran and C++ translations, where expert proofreading is incorporated to handle complex cases and ensure accuracy [88]. For more diverse datasets, platforms like GeeksForGeeks and HumanEval provide multilingual code snippets that are categorized into translation tasks of varying complexity, such as token-level, syntax-level, library-level, and algorithm-level transformations. These tasks are further expanded through manual standardization and validation, with unit tests attached to confirm functional equivalence between the source and target code [89].

To ensure broader coverage, datasets also extract solutions implementing the same algorithm across languages like C++, Java,

and Python, using automated unit tests to verify correctness without manual annotation [90]. Competitive programming platforms, such as Codeforces and Atcoder, offer another avenue for dataset creation. These platforms enable automated crawling of correct submissions, which are subsequently cleaned by removing comments and standardizing formats, although manual annotation is not typically involved in these cases [91]. Together, these approaches combine automated tools with expert interventions to create high-quality datasets that address the functional correctness and alignment requirements of code translation tasks.

Annotation type: Three papers [87,90,91] employed fully automated annotation methods, eight papers [89] used semi-automated annotation methods, and one paper [88] adopted manual annotation methods.

Annotation granularity: Three papers [87,90,91] were annotated at the file-level granularity, and two papers [88,89] were annotated at the method-level granularity. No papers for this task were annotated at the line-level granularity.

4.4 Answer to RQ1

In general, dataset construction for GSETs follows three major phases: defining requirements, collecting raw data, and designing and annotating data. Despite task-specific variations, this structure remains consistent across tasks like code summarization, program repair, code generation, and code translation. Notably, dataset sizes and scales vary, with code generation often having the largest datasets, while program repair tends to focus on smaller, manually verified datasets.

One common trend is the adoption of automated or semi-automated annotation methods to handle the increasing volume of data. However, manual validation is limited, particularly in tasks like code summarization and code translation, where fully automated annotation dominates. In contrast, program repair datasets prioritize manual inspection and validation, ensuring higher data quality through extensive human intervention. Overall, while automation accelerates dataset creation, there remain concerns about the thoroughness of annotation verification in most GSETs tasks.

Takeaways. The key insight is that while automation has enabled the creation of larger, more diverse datasets, there exists a critical trade-off between scale and quality in dataset construction for GSETs. Future research should focus on developing more balanced approaches that combine the efficiency of automated methods with the reliability of manual validation to ensure both comprehensive coverage and high data quality.

5 RQ2: dataset quality of GSETs

We have aggregated the data quality issues mentioned in the collected papers. Table 8 shows all the data quality issues that appear in the four GSETs, which we will discuss one by one by specific tasks in the following text. We carefully read through the data quality issues mentioned in each paper and classified the same quality issues; for each specific issue, we collected perspectives from the three aspects of description, root cause, and impact. [1)]

Table 8 Data quality issues in GSETs and their frequency of discussion in selected studies

GSETs	Data quality issue	Freq.	Ratio/%
CS	Noisy data.	3	20.0
	Information redundancy.	1	6.7
	Insufficient data collection.	1	6.7
	Uneven data distribution.	2	13.3
	Data scarcity.	2	13.3
	Lack of diversity.	3	20.0
	Lack of standardization.	1	6.7
	Inaccurate associations.	1	6.7
PR	Unrelated commits.	4	14.8
	Outdated Datasets.	2	7.4
	The rarity of manually validated.	4	14.8
	Repeated use of benchmarks.	2	7.4
	Data leakage.	1	3.7
	Insufficient novelty and diversity.	2	7.4
	Insufficient granularity.	1	3.7
CG	Functionality correctness overemphasis.	1	4.2
	Lacking real-world coding issues.	2	8.3
	Scarcity and lack of diversity.	2	8.3
	Lack tests.	3	12.5
	Unreasonable format and style.	2	8.3
	Vague problem descriptions.	1	4.2
CT	Insufficient programming languages.	2	14.3
	Scarcity of parallel data.	4	28.6
	Insufficiency of diversity.	3	21.4
	Uneven distribution.	1	7.1
	Poor code quality and execution.	3	21.4

5.1 Code summarization datasets

As shown in Table 8, for the code summarization dataset, we have summarized the following eight common data quality issues.

5.1.1 Noisy data

Description: Noisy data is one of the most pervasive issues in code summarization datasets. Noisy data forms include improper preprocessing, incomplete comments, tampered content, non-English documentation, auto-generated code, and duplicate code. These have been reported to interfere with model training [40,43,46]. For

[1)] Some studies did not assess both the root cause and impact of dataset quality issues, hence, certain issues might lack analysis in these areas.

example, the CodeSearchNet (CSN) [108] dataset contains repetitive descriptions, non-ASCII characters, and unnecessary symbols that degrade the quality of the data [40]. Similarly, the DeepCom dataset suffers from escaped characters, HTML tags, and meaningless descriptions, which further introduce noise [43].

The root cause of noisy data can often be attributed to insufficient or inappropriate preprocessing operations during dataset construction. The automatic tools used in collecting and processing code and comments tend to amplify these issues by retaining irrelevant content such as HTML tags or overly splitting variable names [43]. Moreover, inconsistent coding styles and commenting habits from real-world open-source projects further contribute to noise [42].

Impact: Noisy data can significantly reduce the performance of code summarization models. The presence of incomplete, irrelevant, or erroneous data during model training leads to degraded quality in the generated code summaries [43]. This interference results in poorer generalization and lower accuracy, ultimately hindering the model's ability to generate precise and meaningful summaries.

5.1.2 Information redundancy

Description: Information redundancy refers to the inclusion of excessive and irrelevant details in the dataset, which can overwhelm the model during training. A large portion of the content in the source code may not relate to the task of generating summaries, causing unnecessary complexity. For instance, [44] demonstrated that using simplified code segments, such as function signatures, produced better summaries than using full code, indicating that the additional information in full code inputs is often irrelevant.

Impact: Redundant information can overload DNN models, leading to decreased performance. When the model is provided with unnecessary details, it struggles to focus on the key elements needed for generating accurate summaries. Experiments show that reducing input complexity by focusing on critical segments like function signatures improves the model's ability to generate high-quality summaries [44].

5.1.3 Insufficient data collection

Description: Insufficient data collection is a critical issue in code summarization datasets, where either the dataset size is too small to scale effectively or it contains a significant amount of noise from automated collection processes. Manually annotated datasets, while high in quality, are often too limited in size to provide sufficient training data [45].

The reason of insufficient data collection is twofold: high-quality manual annotations are difficult to scale, while automatically collected data tends to be noisy and less reliable for training purposes [45]. Balancing the trade-off between quality and quantity remains a key challenge in constructing datasets for code summarization.

Impact: Insufficient data limits the model's capacity to learn effectively, especially when the problem coexists with data noise in the dataset. As a result, the model may struggle to generate accurate summaries, particularly for rare or specialized methods that appear infrequently in the training data [45].

5.1.4 Uneven data distribution

Description: Datasets used for code summarization often suffer from uneven data distribution, where common methods such as `size`() or `count`() are overrepresented, while rarer or specialized methods appear infrequently. [45].

Impact: Uneven data distribution causes models to perform well on frequent methods but poorly on rare ones. This imbalance skews the model's learning process, leading to inaccurate predictions for less common functions. As a result, the model's ability to generalize across different types of code is compromised, reducing the overall quality and applicability of the generated summaries [45].

5.1.5 Data scarcity

Description: Data scarcity is an issue particularly prevalent in low-resource programming languages, such as Ruby and JavaScript, where the amount of available training data is limited. Compared to high-resource languages like Java and Python, these low-resource languages suffer from smaller datasets, often focused on narrow application domains [98].

The scarcity of data in certain programming languages can be attributed to the difficulty in collecting large, diverse, and high-quality datasets. Open-source code resources for some languages are fewer, and the available datasets often cover limited domains, restricting the diversity of examples available for training [98].

Impact: Data scarcity in low-resource languages leads to poor model performance, as the limited dataset size reduces the ability of models to generalize effectively. This results in reduced accuracy and coverage for these languages, impairing the model's performance in code summarization tasks [98]. Multilingual joint training has been proposed as a solution, leveraging the similarities between languages to improve performance in low-resource settings.

5.1.6 Lack of diversity

Description: Closely related to data scarcity, lack of diversity refers to the homogeneity of datasets, where the application domain or functionality of the code is too narrow. This issue limits the model's exposure to a wide range of programming constructs, reducing its ability to generalize across different tasks and languages [98].

Impact: The lack of diversity negatively impacts the model's generalization ability. When the dataset does not expose the model to a broad range of code examples, the model may perform poorly on tasks outside the specific domain or language it was trained on. This limitation is particularly apparent in low-resource languages, where diverse datasets are harder to come by [98].

5.1.7 Lack of standardization

Description: The lack of standardization in dataset construction is another significant issue in code summarization. Different datasets often employ varying segmentation strategies and preprocessing techniques, leading to inconsistencies across studies. For example,

some datasets are segmented by method-level while others by project-level, and there is no clear consensus on whether to remove auto-generated code [41].

This issue stems from the absence of universally accepted guidelines for constructing code summarization datasets. Different research groups adopt different preprocessing and segmentation strategies, leading to inconsistent data preparation and making comparisons across studies difficult [41].

Impact: The lack of standardization [41] hinders the replicability and comparability of results across studies. Inconsistent segmentation and preprocessing steps can lead to significant variations in experimental outcomes, making it difficult to perform accurate comparisons or reproduce results. Furthermore, this lack of consistency can obscure the true performance of models, as certain partitioning strategies may artificially inflate performance metrics like BLEU scores [109].

5.1.8 Inaccurate associations

Description: Inaccurate associations between code snippets and comments in datasets represent another significant quality issue. Many datasets rely on simplistic heuristics to link comments with code, which can lead to incorrect associations, particularly with non-contiguous lines of code [42].

This issue arises due to the reliance on heuristic methods during dataset construction, which may not be sophisticated enough to handle more complex code structures. As a result, comments may be incorrectly linked to code snippets, reducing the relevance and accuracy of the dataset [42].

Impact: Inaccurate associations between comments and code snippets directly impact the training of models, leading to poor-quality code summaries. Models trained on such datasets are likely to generate summaries that are misaligned with the actual functionality of the code, reducing both the accuracy and practical utility of the generated summaries [42].

5.2 Program repair

As shown in Table 8, for the program repair dataset, we have summarized the following seven common data quality issues.

5.2.1 Unrelated commits

Description: One common issue in program repair datasets is the presence of unrelated commits. Datasets built from open-source repositories such as GitHub often include mixed or loosely related code changes that are not strictly tied to bug fixes. Several studies [53,65,105] have identified this phenomenon, where commits may encompass refactoring, feature additions, or other modifications that are unrelated to repair tasks. This confounds the dataset, making it harder for models to learn the specific patterns of bug fixes.

The problem of unrelated commits can largely be attributed to the automated tools used in dataset collection. These tools are unable to distinguish between bug fixes and other types of code changes, leading to "tangled code changes" in the datasets [53,55]. As noted by [105], the inability of automated tools to isolate bug-fixing changes from other modifications is a key factor contributing to this issue.

Impact: Unrelated commits introduce noise into the dataset, contaminating the annotations and misleading models during training. This results in inaccurate models that struggle to generalize to unseen bugs, ultimately reducing the efficacy of program repair techniques [105].

5.2.2 Outdated datasets

Description: Another significant issue is that many program repair datasets contain outdated bug fixes that no longer reflect current development practices. As technology and coding practices evolve, it is essential for datasets to keep up with these changes to remain relevant. However, many benchmarks fail to include recent bug fixes and rely on older examples that may not be representative of modern bugs [100,101].

The outdated nature of these datasets is often due to the reliance on older codebases or the inclusion of third-party code dependencies that may become unavailable over time, making it difficult to reproduce results [100]. Additionally, fragile test suites and dependencies on external actors, such as package maintainers, contribute to the degradation of benchmark executability, further compounding the issue.

Impact: Using outdated datasets can result in models that perform well on historical bugs but fail to generalize to contemporary errors. This reduces the practical applicability of program repair models, as they may not be able to address the types of bugs encountered in modern software development [100].

5.2.3 Rarity of manually validated

Description: The scarcity of manually validated datasets is another critical quality issue in program repair research. Most datasets rely on automated tools with minimal or no manual validation, which can lead to low-quality data. Manually validated datasets, although more accurate, are rare and often small in scale [105].

This issue arises because manual validation is a labor-intensive process, requiring experts to examine and confirm the correctness of bug fixes and repair solutions. As a result, most datasets are either entirely automated or receive only limited manual validation, leading to potential inaccuracies in the data [105].

Impact: The lack of manually validated datasets limits the reliability of models trained on these datasets. Automated validation tools may miss certain nuances in bug fixes, leading to incorrect or incomplete data, which in turn affects the performance of program repair models. Furthermore, the small size of manually validated datasets restricts their usefulness for large-scale research.

5.2.4 Repeated use of benchmarks

Description: The repeated use of the same benchmarks in program repair research presents a threat to the external validity of findings. When the same benchmarks are used across multiple studies, improvements may appear significant but may not generalize beyond the specific dataset being evaluated [54]. This over-reliance on a limited set of benchmarks can lead to a disconnect between research outcomes and real-world applicability.

Impact: Overuse of the same benchmarks can result in models that are overfitted to these specific datasets, leading to artificially inflated performance metrics. This creates a false sense of progress in the field, as models may not perform as well on new or unseen datasets. Consequently, the external validity of research findings may be compromised, limiting the practical impact of the proposed solutions [54].

5.2.5 Data leakage

Description: Data leakage occurs when models are inadvertently exposed to data during training that they will later encounter during evaluation. This is particularly problematic in program repair datasets, where older datasets like Defects4J and ManyBugs may have been included in the training data of large language models (LLMs), leading to overly optimistic performance metrics [60].

Data leakage in program repair datasets is often due to the use of older code repositories and datasets that predate the advent of LLMs. These older datasets may have already been included in the training data of widely used LLMs, causing models to encounter familiar data during evaluation [60].

Impact: If a model has seen the same or similar data during training, its performance during evaluation may not accurately reflect its true capabilities. Such data leakage can lead to inflated performance metrics, giving the false impression that a model is more effective than it actually is. This misjudgment hampers the fair evaluation of program repair models and can lead to misleading conclusions about their generalization capabilities and real-world performance [60].

5.2.6 Insufficient novelty and diversity

Description: Two works point out that current repair datasets suffer from a lack of novelty and diversity. These datasets often rely on code from a narrow range of open-source projects or specific domains, which may not accurately reflect the diversity and complexity of real-world programming errors. This limited scope affects the generalization capabilities of models trained on such datasets, as they may perform poorly when applied to novel or diverse bugs [60,98].

The lack of novelty and diversity in program repair datasets can be traced to data collection practices that focus on a small set of well-known projects or domains. [60].

Impact: Insufficient novelty and diversity in datasets lead to models that are unable to generalize effectively to new or unseen bugs. This reduces the practical utility of program repair models, as they may perform well on the specific types of bugs present in the dataset but struggle with more varied or complex errors encountered in real-world software development. This limited scope fails to capture a broad range of programming errors and coding practices, reducing the dataset's relevance for training robust, generalizable models [60].

5.2.7 Insufficient granularity

Description: Another prevalent issue in program repair datasets is the lack of fine-grained fault localization annotations. Many datasets provide only coarse-grained annotations, such as method-level or module-level labels, which are insufficient for accurately pinpointing the location of bugs in the code. Fine-grained annotations, such as line-level labels, are often missing, which limits the usefulness of the dataset for tasks requiring precise fault localization [60,105].

This issue arises from the imprecision in the labeling step, namely, after data collection, the failure locations were not annotated with fine granularity [60].

Impact: Coarse-grained annotations hinder the ability of models to perform fine-grained analyses of bug fixes. This reduces the effectiveness of program repair models, particularly for tasks that require precise identification of the buggy code. Inaccurate fault localization also impacts the evaluation of these models, as they may miss critical parts of bug repairs, leading to incomplete or incorrect fixes [105].

5.3 Code generation

As shown in Table 8, for the code generation dataset, we have summarized the following six common data quality issues.

5.3.1 Functionality correctness overemphasis

Description: A common issue in current code generation datasets is an excessive focus on functionality correctness. Many datasets are designed to ensure that the generated code passes functional tests, often neglecting other important aspects like code stability, robustness, and long-term reliability. While functional correctness is crucial, it alone does not guarantee the code will perform well in practical settings [67].

This issue usually arises when determining the goals for data set construction. For code generation issues, test cases are generally used to evaluate the accuracy of the generated code. However, this may lead to neglecting the semantic differences or broader context in the generated code. As a result, datasets fail to capture the robustness and reliability aspects that are essential for real-world code [67].

Impact: By focusing solely on functional correctness, models trained on these datasets may produce code that passes tests but may experience issues in production environments, such as resource leaks, scalability problems, or crashes. This can lead to unreliable code that requires further intervention from developers, undermining the practical utility of these models [67].

5.3.2 Lacking real-world coding issues

Description: Another significant issue is the lack of real-world coding challenges in existing datasets. Programming in the real-world usually consists of many different types of tasks, and if only one type of task is considered when building the dataset, it would cause the entire dataset to be unable to cover all coding scenarios. In addition, the single code granularity may also exacerbate this issue. Current benchmarks, such as HumanEval, often focus on independent functions that do not require external context, making them less appropriate for evaluating models in real-world development scenarios [67,68].

This problem is largely due to the data source selection during dataset construction. Many datasets favor simplified problems from challenge websites rather than sourcing real-world development tasks from platforms like StackOverflow. Additionally, the selection of

independent functions that rely solely on standard libraries, rather than complex, context-dependent functions, further contributes to the unrealistic nature of the datasets [68].

Impact: The absence of real-world coding issues limits the practical applicability of code generation models. Models trained on such datasets may struggle to handle dependencies, complex API usage, or project-specific contexts, which are critical in real-world development. This issue in evaluation dataset also leads to overestimating the model's capabilities and hinders its effectiveness when applied in actual software development environments [67,68].

5.3.3 Scarcity and lack of diversity

Description: The scarcity and lack of diversity in code generation datasets is another major concern, particularly in low-resource programming languages such as Ruby and JavaScript. Constructing large-scale datasets with paired data (i.e., natural language descriptions and corresponding code) is challenging, especially in domain-specific code generation tasks. Additionally, many current datasets cover only specific domains, limiting their diversity [80,98].

The scarcity of datasets can be attributed to the high cost of manually constructing natural language descriptions and code pairs, which requires significant time and effort. Furthermore, the limited availability of open-source code resources and the focus on specific application domains further restrict the scale and diversity of datasets [80,98].

Impact: The scarcity and lack of diversity in datasets negatively affect the training of code generation models, limiting their ability to generalize across different programming languages and domains. Models trained on small or homogeneous datasets may perform well in specific scenarios but fail to adapt to more diverse and complex tasks. This reduces the overall effectiveness and robustness of the models [80].

5.3.4 Lack of tests

Description: A critical issue affecting code generation datasets is the insufficient number of tests per coding problem. Most existing benchmarks, such as HUMANEVAL, contain fewer than 10 tests per problem, and these tests are often overly simplistic, failing to thoroughly examine the functionality or edge cases of the generated code [85].

Impact: The lack of comprehensive testing in code generation datasets undermines the evaluation of models. Without adequate tests, it becomes difficult to assess how well a model handles edge cases or complex functional requirements. This can lead to an overestimation of the model's performance, as it may only be evaluated on basic test cases, leaving more intricate or subtle bugs undetected [85].

5.3.5 Unreasonable format and style

Description: Different developers follow different coding styles, such as varying quote types, indentation, and semicolon usage. When the programming format and style of the code are uniform or not standardized, the dataset will have this problem. Additionally, current datasets often overlook code readability and structure, focusing primarily on increasing data volume and ensuring functionality correctness [80,86].

This issue stems from insufficient preprocessing during dataset construction. The variability in coding styles among different developers is often not standardized, resulting in datasets that contain inconsistent and noisy code. Moreover, prioritizing quantity over quality in data collection results in datasets that compromise crucial aspects like code readability, structure, and modularity [80,86].

Impact: Inconsistent formatting and poor coding practices in datasets can degrade the performance of code generation models. Models trained on such datasets may generate functionally correct code, but the code may lack clarity, modularity, and maintainability. This reduces the practical utility of the generated code, as developers may need to refactor or rewrite it to meet coding standards. The presence of noise in the dataset also increases the difficulty of training models, leading to unstable performance across different coding styles [86].

5.3.6 Vague problem descriptions

Description: Many code generation datasets suffer from vague or ambiguous problem descriptions. The natural language input provided alongside function signatures in these datasets is often too unclear to fully specify the expected program behavior. As a result, models may generate code that does not accurately address the intended problem, leading to incorrect or incomplete solutions [85].

Impact: Vague problem descriptions complicate the evaluation of code generation models. When the task description is ambiguous, models may interpret the problem differently than intended, leading to suboptimal code generation. This can cause capable models to be misjudged as incompetent, as their performance may be hindered by unclear or incomplete task specifications rather than an actual lack of ability [85].

5.4 Code translation

As shown in Table 8, for the code translation dataset, we have summarized the following five common data quality issues.

5.4.1 Insufficient programming languages

Description: One of the most prominent issues in code translation datasets is the limited coverage of programming languages. Most datasets focus on a handful of popular languages, such as Java, Python, and C#, while neglecting many niche or emerging languages. This imbalance restricts the ability of code translation models to operate effectively in multilingual environments [87].

This issue stems from the data collection phase, where dataset curators tend to prioritize popular languages due to their widespread use and availability of resources. Though these minority and niche programming languages are important for specific domains and legacy systems, they are often overlooked during the dataset construction process [87].

Impact: The lack of coverage across programming languages limits the applicability of code translation models, particularly in scenarios

where multilingual support is essential, such as code migration and maintenance for niche languages. Without sufficient data from a broader range of languages, models may struggle to generalize, leading to subpar performance when translating between less common language pairs [87].

5.4.2 Scarcity of parallel data

Description: Parallel data, which consists of code pairs from different programming languages that are functionally equivalent, is essential for training supervised code translation models. However, there is a notable scarcity of such data in existing datasets. Currently, the main sources of code translation data are open-source software and projects, however, the number of code pairs with good correspondence is very limited. And some language pairs have even rarer data sources [92,93].

The lack of parallel data is primarily due to the high cost and difficulty of manual annotation. Constructing large-scale parallel datasets requires experts to ensure that the code pairs are functionally equivalent, which is a time-consuming and labor-intensive process. Additionally, open-source code resources for certain languages are relatively scarce, limiting the amount of available data [92,93,98].

Impact: The scarcity of parallel data impedes the effectiveness of supervised learning methods for code translation. Without sufficient parallel data, models struggle to accurately learn the functional equivalence between languages, resulting in poor translation accuracy and generalization. This limitation also affects the evaluation of models, as small datasets may not provide enough variety to thoroughly test the model's capabilities [92,97]. In particular, non-parallel code pairs in test datasets can introduce bias in evaluation metrics, making it difficult to assess the true performance of translation models [97].

5.4.3 Insufficiency of diversity

Description: Another major issue is the lack of diversity in code translation datasets. Many datasets contain repetitive code structures or simple token mappings, which allow models to achieve high scores by memorizing patterns rather than learning complex translation tasks. Additionally, complex challenges, such as library calls and algorithmic rewriting, are often underrepresented in these datasets [89].

This issue arises from shortcomings in the dataset construction process. Existing datasets tend to focus on simpler translation tasks, which are easier to collect and annotate, but they fail to capture the complexity of real-world code translation scenarios. As a result, the datasets are biased toward repetitive or trivial tasks, limiting their ability to test models on more challenging translation problems [89].

Impact: The lack of diversity in code translation datasets significantly limits the ability of models to generalize to more complex tasks. Models trained on such datasets may perform well on simple translations but struggle with more sophisticated code transformations, such as handling complex API calls or rewriting algorithms. This can result in overestimating the model's capabilities and under-preparing it for real-world code translation tasks [89,98].

5.4.4 Uneven distribution

Description: The data distribution in existing code translation datasets is often skewed, with certain programming languages being overrepresented while others are underrepresented. This imbalance is mainly reflected in the difference in the amount of data across different languages. Some languages have more data, while others only have a few paired data [87].

This uneven distribution is a result of imbalances during the data sampling phase. Languages with more open-source resources and larger developer communities, such as Java and Python, are more likely to be included in datasets, while niche or low-resource languages receive less attention [87].

Impact: The uneven distribution of data among languages creates performance disparities in code translation models. Models trained on datasets that heavily favor high-resource languages may perform exceptionally well on those languages but fail to translate effectively between low-resource languages. This imbalance reduces the practical utility of the models in multilingual environments, where support for a broader range of languages is often required [87].

5.4.5 Poor code quality and execution

Description: Many code translation datasets suffer from poor code quality and execution issues. Some code samples in these datasets cannot be compiled or executed, and others contain errors or inconsistencies, which can affect the model's ability to learn accurate translations. Additionally, there are reports of non-parallel code pairs, where the source and target code are not functionally equivalent, further complicating the evaluation of model performance [87,92,97].

The root cause of this issue is the lack of rigorous data verification and cleaning procedures during dataset construction. In many cases, the collected data is not thoroughly checked for correctness or execution capability, leading to the inclusion of faulty or non-executable code samples. Moreover, the manual effort required to ensure functional equivalence between code pairs is often not invested, resulting in non-parallel pairs being included in the dataset [87,97].

Impact: Poor code quality and execution issues undermine the reliability of code translation models. Models trained on such data may produce translations that are syntactically correct but fail to execute properly or maintain functional equivalence. This not only affects the quality of the generated translations but also introduces biases into model evaluation, as non-parallel code pairs can distort performance metrics and lead to incorrect conclusions about the model's capabilities [92,97].

5.5 Answer to RQ2

The prevalent data quality issues in GSETs datasets can be categorized into several common challenges. Inadequate preprocessing and inconsistent coding practices contribute significantly to noisy data, which poses a major challenge to model performance. Information redundancy, often caused by excessive or irrelevant details, can overwhelm models and reduce their effectiveness. Insufficient data collection, especially in low-resource

programming languages, limits a model's ability to generalize. Uneven data distribution and lack of diversity further exacerbate these problems, making it difficult for models to handle infrequent or specialized code patterns. Besides, the construction process of the dataset is also lacking in standardized norms, and potential data leaks and task conflicts restrict the construction of large-scale multi-task datasets.

Although many tasks face similar or identical quality issues, different coding tasks have different main quality problems and root causes. For example, inconsistencies between code and comments significantly reduce the data quality of the code summarization task. For program repair tasks, data quality issues are closely related to data obsolescence, incorrect submission information, and insufficient manual inspection. For code generation and translation tasks, the scale of the dataset and the diversity of languages remain key quality bottlenecks. These data quality issues significantly impact GSETs model performance, hindering generalization and practical applicability.

Takeaways. In summary, addressing data quality issues requires task-specific approaches that consider the unique challenges of each GSETs. Moving forward, researchers should prioritize developing standardized preprocessing pipelines, ensuring balanced and diverse data representation, and implementing rigorous validation mechanisms to mitigate these persistent quality challenges.

6 RQ3: solutions

There have been many datasets for GSETs, and researchers discovered data quality issues with raising concerns. However, few techniques are proposed to address these issues, highlighting a gap in this field. The following outlines the existing technical methods, their effectiveness and future directions.

6.1 Data cleaning and filtering methods

Methodology: Data cleaning and filtering are essential techniques for improving dataset quality in GSETs, as noisy or inconsistent data can significantly hinder model performance. For code summarization, automated approaches such as CAT have been used to systematically detect and remove noise, targeting issues like incomplete comments, fragmented variable names, and empty functions. By applying predefined rules, these methods enhance the consistency of datasets and mitigate the impact of irrelevant or erroneous data [43]. Similarly, noise removal techniques that rely on manually identified patterns have proven effective, particularly for Java code/comment pairs, where filtering out errors and expanding datasets can improve data reliability and scale [40]. Granular preprocessing steps, such as eliminating code snippets with syntax errors, overly long or short methods, and duplicates, further refine datasets by reducing biases and standardizing inputs [46].

In program repair, the challenge of "entangled changes," where bug-fixing commits include unrelated modifications, has been addressed through automated purification techniques. For example, DEPTEST leverages code coverage analysis and incremental debugging to isolate bug-related changes, filtering out irrelevant edits without requiring manual intervention [55]. This ensures cleaner, more focused datasets that better capture patterns specific to program repair tasks.

Code generation datasets also benefit from advanced cleaning approaches that prioritize readability and maintainability. Techniques such as using large language models (LLMs) to rename variables for semantic clarity, modularize complex code into smaller functions, and generate natural language plan comments have significantly improved the functional and organizational quality of datasets. These steps not only preserve the functional correctness of the code but also make it more interpretable, creating datasets that are better suited for training and evaluation [86].

Effectiveness: The impact of these cleaning methodologies is evident in the substantial improvement in model performance across various GSETs. For code summarization, reducing noise in benchmark datasets has led to increases of 21% to 27% in BLEU-4 scores, demonstrating the importance of clean data for generating precise summaries [43]. Similarly, creating a larger, cleaner dataset, such as CoDesc, has enhanced Java code translation models by providing more representative and reliable training data [40]. Granular preprocessing techniques have further reduced errors and biases, enabling more accurate comparisons across different methods [46].

In program repair, DEPTEST has significantly improved the reliability of automated program repair (APR) evaluations by producing higher-quality patches and reducing noise. By filtering out unrelated changes, datasets become more focused, allowing for fairer comparisons between APR methods and more precise evaluations [55].

For code generation, refining datasets through LLM-driven cleaning has yielded notable gains in model performance. Experimental results revealed up to a 30% improvement in functional accuracy for models trained on cleaned datasets compared to those trained on unrefined data [86]. Notably, even a smaller, high-quality dataset (15% of the original size) outperformed models trained on the full, uncleaned dataset, underscoring the pivotal role of data quality in enhancing model efficiency and generalization.

These findings highlight the critical importance of robust data cleaning and filtering methods in improving model training and evaluation, enabling advancements across tasks such as code summarization, program repair, and code generation. By ensuring cleaner, more representative datasets, these techniques lay a stronger foundation for the development of more reliable and effective GSETs models.

Future directions: Data cleaning and filtering methods should focus on developing more automated, task-specific techniques to identify and remove noise from datasets. Future advancements may incorporate LLM-driven tools that dynamically adapt to diverse dataset structures while preserving relevant information and ensuring minimal human intervention for large-scale data processing.

6.2 Data augmentation methods

Methodology: Data augmentation has emerged as a vital strategy to

tackle challenges related to dataset scale, imbalance, and quality in GSETs, enabling models to generalize better and improve performance in low-resource settings. For code summarization, augmentation techniques that cluster similar code-comment pairs have proven effective in addressing data imbalance. By targeting underrepresented categories and replacing non-keywords in comments to generate new pairs, these methods ensure semantic consistency while simultaneously expanding the dataset and filtering out potential noise [45]. Beyond these task-specific approaches, multilingual and numerical augmentation strategies have also been explored. Techniques such as back translation generate pseudo-parallel data, while multilingual augmentation leverages patterns across programming languages to improve generalization. Numerical augmentation further enhances robustness by varying numerical values in code, ensuring that models become less sensitive to specific numeric inputs [97].

In the context of code translation, augmentation methods have been designed to address limitations in parallel data and enhance diversity. For example, leveraging similar corpora —whether retrieved, naturally occurring, or generated —enables models to better understand functional equivalence between programming languages, even when syntactic structures differ. Generating multiple reference translations for the same code, filtered using unit tests, further enhances dataset diversity and mitigates overfitting [92]. Additionally, advanced transformation techniques such as rule-based and retrieval-based methods have been introduced to create new code pairs while preserving semantic correctness. Rule-based transformations systematically modify code structures, while retrieval-based methods extract high-quality parallel data from large code repositories, avoiding errors often found in machine-generated samples [93,94]. Together, these approaches aim to enrich datasets with meaningful, diverse samples that align well with real-world coding scenarios.

Effectiveness: The effectiveness of these augmentation methods is evident in their ability to boost model performance across various tasks. By addressing imbalances and expanding datasets, clustering-based and numerical augmentation methods for code summarization have demonstrated improvements in BLEU-4, METEOR [110], and ROUGE-L [111] scores, with gains ranging from 1.37% to 2.24% [45]. Similarly, multilingual and back-translation approaches have not only enhanced style consistency but also improved BLEU scores by up to 6.9% for code translation and 7.5% for code summarization, showcasing their utility in multilingual and multi-domain tasks [97].

In code translation, augmentation techniques have shown remarkable results. Leveraging similar corpora and generating multiple reference translations increased Compute Accuracy (CA@1) by 7.5% across various language pairs, highlighting the role of functional equivalence in improving translation quality [92]. Rule-based and retrieval-based methods have further demonstrated their effectiveness, achieving up to a 35% increase in translation accuracy for tasks such as Java-to-C# translation [93,94]. While rule-based transformations occasionally risk overfitting, retrieval-based approaches consistently outperformed traditional NLP-based methods, underlining the importance of tailoring augmentation techniques to the unique requirements of GSETs.

Future directions: Data augmentation methods may evolve by leveraging advanced generative models to create diverse, high-quality synthetic data. These techniques should aim to better simulate real-world scenarios, address low-resource programming languages, and explore cross-language augmentation to enhance model generalization and robustness in multilingual and multi-domain tasks.

6.3 Dataset construction methods

Methodology: Dataset construction plays a key role in ensuring reliable evaluations in GSETs. Some methods aim to introduce strategies to improve data quality at the time of constructing datasets. [41] proposed standardized guidelines, comparing dataset division by method versus project. Their findings showed that project-based division prevents information leakage between training and test sets, leading to more reliable results. They also recommended removing automatically generated code to avoid inflating model performance. With a processed dataset of 2.1 million Java method-comment pairs, this work offered a unified benchmark, enhancing comparability and consistency across studies.

Effectiveness: These standards significantly improved research outcomes by ensuring consistency and preventing information leakage. [41] allowed for more accurate comparisons between models and reduced variability caused by differing preprocessing techniques. Removing automatically generated code ensured that performance improvements reflected the actual improvement of the model. This approach improves research reproducibility and ensures more reliable evaluations, advancing the field with a stable benchmark dataset for future studies.

Future directions: Dataset construction methods should prioritize standardization and scalability, emphasizing reproducible guidelines for segmentation, annotation, and preprocessing. Future work should integrate robust mechanisms to prevent data leakage, ensure consistent quality, and incorporate diverse, real-world programming scenarios to improve dataset relevance and applicability.

6.4 New training methods

Methodology: To address data scarcity and diversity, [98] introduced multilingual joint training for GSETs. By leveraging shared code patterns and identifiers across different languages, the approach allows models to benefit from resource-rich languages, improving performance in low-resource languages like Ruby and JavaScript. This data augmentation enhances tasks such as code summarization, retrieval, and function naming by enabling models to learn useful patterns from other languages.

Effectiveness: The multilingual joint training method significantly boosts performance in low-resource languages by transferring knowledge from better-resourced ones. This improves model generalization and enhances the automation of software engineering tasks in multilingual environments, making it a valuable solution for increasing the robustness of GSETs models across various languages.

Future directions: New training methods should further explore multilingual and multitask learning frameworks to enhance cross-language and cross-domain generalization. By leveraging transfer learning and shared representations across tasks, future developments should aim to address data scarcity issues and improve model performance in low-resource environments.

6.5 Expanding test cases

Methodology: To address insufficient testing and vague descriptions in existing datasets, [85] proposed the EvalPlus framework. EvalPlus combines LLM-based and mutation-based methods for automatic test input generation. It first uses ChatGPT [112] to generate high-quality seed inputs, followed by type-aware mutations to create additional test cases that expand dataset coverage. EvalPlus also verifies the functional correctness of generated code using differential testing and optimizes evaluation efficiency with a test suite reduction algorithm. This method significantly enhances the test coverage and rigor of existing datasets, enabling more precise evaluations of code generated by LLMs.

Effectiveness: Implementing EvalPlus revealed that the performance of many LLM code generation models had been overestimated due to insufficient testing. When evaluated on the expanded dataset, HUMANEVAL+ [113], models showed a significant drop in pass rates (pass@k), indicating that prior evaluations missed many errors. Additionally, EvalPlus reordered model performance rankings, with some open-source models, like WizardCoder-CodeLlama [114], outperforming ChatGPT on HUMANEVAL+, contrary to previous results. This underscores the need for more rigorous testing to accurately assess LLMs and avoid overestimating their code generation capabilities.

Future directions: Expanding test cases requires the development of automated, context-aware testing techniques that target edge cases and ensure thorough evaluation. Future work should focus on generating diverse, high-coverage test suites that assess functionality, robustness, and semantic correctness, enabling more accurate and reliable evaluation of code generation models.

6.6 Answer to RQ3:

To address data quality issues in GSETs, several solutions have been proposed. Most methods are data cleaning and filtering approaches, this type of technology can design corresponding templates based on the characteristics of the data. Data augmentation techniques focus on expanding datasets by generating new, diverse examples, which helps mitigate issues like data scarcity and imbalance. Additionally, standardized dataset construction methods have been suggested to ensure more reliable and consistent model evaluations by preventing information leakage and enhancing comparability across studies.

These solutions collectively improve the quality of datasets, leading to better model performance, increased generalization, and more accurate evaluation results in GSETs.

Takeaways. The key insight is that while current solutions offer promising improvements, there remains a significant gap between the number of identified data quality issues and the availability of comprehensive, validated solutions. This highlights an important direction for future research: developing integrated frameworks that combine multiple solution approaches to systematically address the complex, multifaceted nature of data quality challenges in GSETs.

7 Recommendations

To build high-quality datasets that can significantly enhance the performance of DNN models in GSETs, researchers must adopt a comprehensive and meticulous dataset construction strategy. Based on the findings of this study, which analyzed the processes (RQ1), data quality issues (RQ2), and proposed solutions (RQ3) in existing research, we recommend several best practices across three critical areas: data selection, dataset annotation, and data cleaning and validation. These areas are pivotal in ensuring that datasets are robust, representative, and capable of facilitating the development of high-performance models.

Data selection: ensuring representativeness and diversity. One of the primary challenges identified in RQ2 is the lack of representativeness and diversity in datasets used for GSETs, which can severely hinder model generalization.

Datasets often suffer from limited programming language coverage, an over-representation of specific types of code snippets, or an imbalance in the distribution of code patterns. To mitigate these issues, we recommend that researchers prioritize the selection of diverse code samples from a wide array of programming languages, domains, and projects, including both open-source repositories and industry-level projects. By incorporating code from different languages and paradigms (e.g., object-oriented, functional), researchers can ensure that the resulting dataset is rich in structural and syntactic variation, helping models generalize better across unseen tasks.

Moreover, as indicated by the analysis in RQ1, it is essential to consider the scalability of the dataset. Larger datasets are often necessary to train DNN models effectively, but the size must be balanced with the quality and diversity of the data. Researchers should also account for timeliness, ensuring that the dataset contains recent code samples that reflect modern software engineering practices and trends. This helps DNN models remain relevant and effective in contemporary code-related tasks.

Dataset annotation: improving accuracy and consistency. Accurate and consistent annotation is crucial for training DNN models in GSETs, yet data annotation is often fraught with challenges such as noise, human error, and inconsistencies, as explored in RQ2.

To address these issues, researchers should consider employing semi-automated annotation techniques. which can significantly reduce the time and effort required for manual labeling while ensuring a certain level of precision. Automated techniques, such as leveraging static analysis tools or abstract syntax tree (AST) parsers, can be used to extract metadata and generate labels for code-related datasets. However, these methods are not infallible, and manual verification is still necessary to ensure correctness. In cases where manual annotation is required, crowdsourcing and expert-driven approaches are both valid options, but they should be supplemented

by rigorous guidelines and quality control mechanisms to ensure consistency across annotators.

Additionally, it's important to align annotation granularity with the specific needs of the task. For instance, code summarization may require function-level annotations, whereas program repair might require line-level or even token-level annotations. Researchers should also consider the annotation format, ensuring that it is both human-readable and machine-processable, to facilitate easy integration into future studies and model training pipelines. As highlighted in RQ3, providing clear documentation and justifications for the annotation process helps future researchers understand the dataset's limitations and facilitates reproducibility.

Data cleaning and validation: enhancing data quality. To further enhance the quality and reliability of datasets for GSETs, researchers must remain vigilant about the pervasive data quality issues that can undermine model performance.

As emphasized in RQ2, problems like data noise, imbalances, and insufficient representativeness continue to be significant barriers to the effectiveness of DNN models. Thus, it is critical for researchers to not only adopt robust dataset construction methodologies but also to actively seek out and address these data quality issues throughout the dataset lifecycle. This involves conducting thorough preprocessing checks, maintaining transparency in data collection methods, and employing advanced techniques such as data cleaning, balancing, and noise reduction.

More importantly, researchers should strive to innovate and propose new solutions to prevalent data quality challenges. As highlighted by the findings in RQ3, while some techniques like data augmentation and noise filtering have been effective, there is ample room for improvement and the development of more sophisticated methods. Future work should explore novel strategies that go beyond existing approaches, such as leveraging hybrid techniques that combine machine learning with expert-driven insights to refine datasets. By prioritizing data quality and continuously improving solutions to common issues, the research community can help ensure that DNN models for GSETs are trained on robust and high-quality datasets, ultimately leading to more reliable and impactful outcomes.

8 Threats to validity

Internal validity concerns relate to the accuracy of our research process. The main threat stems from potential subjectivity in manual screening and quality assessment. To mitigate this, we established predefined quality criteria, involved multiple reviewers in the assessment process, and maintained detailed documentation of our decision-making. Our data extraction process posed another internal validity challenge. To address this, we implemented a comprehensive data extraction form, conducted multiple review rounds, and employed systematic coding to ensure consistency and thoroughness in extracting relevant information.

External validity addresses the generalizability of our findings. Although we analyzed 70 papers, this sample may not fully represent the entire landscape of dataset construction for GSETs. To mitigate this limitation, we employed a comprehensive search strategy across multiple databases and supplemented our initial search with forward and backward snowballing to expand our coverage. The rapid evolution of deep learning approaches creates temporal constraints on our study. To address this challenge, we included papers published through the end of our search period and used forward snowballing to identify more recent relevant publications that cite our core papers.

Construct validity relates to whether our study accurately measures what it intends to measure. Despite our rigorous search methodology, relevant studies may have been missed due to terminology variations or database limitations. To mitigate this threat, we developed our search string using the PICO framework, incorporated multiple synonyms and related terms, and searched across three major databases (ACM Digital Library, IEEE Xplore, and Scopus). Our initial screening based on abstracts and titles might have excluded relevant papers. To counter this risk, we conducted thorough forward and backward snowballing, enabling us to identify papers that may have been missed in our initial database search due to terminology differences or indexing limitations.

9 Conclusion

In conclusion, this review has provided a comprehensive analysis of the dataset construction processes, common data quality issues, and proposed solutions in the context of DNN-driven GSETs. Our review highlights that the methodologies for constructing datasets vary significantly, with differences in data sources, annotation methods, collection techniques, and dataset granularity. These variations, while offering flexibility, also contribute to key data quality issues such as noise, sparsity, imbalance, and lack of representativeness, all of which adversely affect the performance and generalization capabilities of DNN models in tasks like code summarization, repair, generation, and translation. Although several strategies, including data augmentation, noise reduction, and resampling approaches, have been proposed to mitigate these issues, challenges remain in constructing high-quality, large-scale datasets that are both representative and diverse. Our findings underscore the critical need for more robust and standardized approaches to dataset curation, with particular attention to quality assurance throughout the data collection and annotation phases. By addressing these challenges, future research can significantly improve the reliability and effectiveness of DNN models in GSETs, ultimately advancing the field of deep learning in software engineering and automating code-related tasks more efficiently.

Acknowledgements

We would like to thank reviewers for their constructive comments. This project was partially funded by the National Natural Science Foundation of China (Grant Nos. 62372225 and 62272220).

Competing interests

Baowen XU is an Editorial Board member of the journal and a co-author of this article. To minimize bias, he was excluded from all editorial decision-making related to the acceptance of this article for publication. The remaining authors declare no conflict of interest.

References

[1] Huang Y, Chen Y, Chen X, Chen J, Peng R, Tang Z, Huang J, Xu F, Zheng Z. Generative software engineering. 2024, arXiv preprint arXiv: 2403.02583

[2] Palacio D N, Velasco A, Rodriguez-Cardenas D, Moran K, Poshyvanyk D. Evaluating and explaining large language models for code using syntactic structures. 2023, arXiv preprint arXiv: 2308.03873

[3] LeClair A, Jiang S, McMillan C. A neural model for generating natural language summaries of program subroutines. In: Proceedings of 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 2019, 795−806

[4] Alon U, Brody S, Levy O, Yahav E. code2seq: generating sequences from structured representations of code. In: Proceedings of the 7th International Conference on Learning Representations. 2019

[5] Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. In: Proceedings of the 6th International Conference on Learning Representations. 2018

[6] Ben-Nun T, Jakobovits A S, Hoefler T. Neural code comprehension: a learnable representation of code semantics. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems. 2018, 3589–3601

[7] Gupta R, Pal S, Kanade A, Shevade S. DeepFix: fixing common C language errors by deep learning. In: Proceedings of the 31st AAAI Conference on Artificial Intelligence. 2017, 1345–1351

[8] Bader J, Scott A, Pradel M, Chandra S. Getafix: learning to fix bugs automatically. Proceedings of the ACM on Programming Languages, 2019, 3(OOPSLA): 159

[9] Liu K, Koyuncu A, Kim D, Bissyandé T F. TBar: revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2019, 31−42

[10] Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M. CodeBERT: a pre-trained model for programming and natural languages. In: Proceedings of the Association for Computational Linguistics: EMNLP 2020. 2020, 1536–1547

[11] Brown T B, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, Agarwal S, Herbert-Voss A, Krueger G, Henighan T, Child R, Ramesh A, Ziegler D M, Wu J, Winter C, Hesse C, Chen M, Sigler E, Litwin M, Gray S, Chess B, Clark J, Berner C, McCandlish S, Radford A, Sutskever I, Amodei D. Language models are few-shot learners. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. 2020, 159

[12] Wang Y, Wang W, Joty S, Hoi S C H. CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of 2021 Conference on Empirical Methods in Natural Language Processing. 2021, 8696–8708

[13] Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng S K, Clement C, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M. GraphCodeBERT: pre-training code representations with data flow. In: Proceedings of the 9th International Conference on Learning Representations. 2021

[14] Dabre R, Chu C, Kunchukuttan A. A survey of multilingual neural machine translation. ACM Computing Surveys (CSUR), 2020, 53(5): 99

[15] Le T H M, Chen H, Babar M A. Deep learning for source code modeling and generation: models, applications, and challenges. ACM Computing Surveys (CSUR), 2020, 53(3): 62

[16] Yang Y, Xia X, Lo D, Grundy J. A survey on deep learning for software engineering. ACM Computing Surveys (CSUR), 2022, 54(10s): 206

[17] Jain A, Patel H, Nagalapatti L, Gupta N, Mehta S, Guttula S, Mujumdar S, Afzal S, Sharma Mittal R, Munigala V. Overview and importance of data quality for machine learning tasks. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2020, 3561−3562

[18] Whang S E, Lee J G. Data collection and quality challenges for deep learning. Proceedings of the VLDB Endowment, 2020, 13(12): 3429−3432

[19] Elman J L. Finding structure in time. Cognitive Science, 1990, 14(2): 179−211

[20] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 1997, 9(8): 1735−1780

[21] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A N, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. 2017, 6000−6010

[22] Ahmad W, Chakraborty S, Ray B, Chang K W. Unified pre-training for program understanding and generation. In: Proceedings of 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2021, 2655−2668

[23] Zhang C, Wang J, Zhou Q, Xu T, Tang K, Gui H, Liu F. A survey of automatic source code summarization. Symmetry, 2022, 14(3): 471

[24] Zhu Y, Pan M. Automatic code summarization: a systematic literature review. 2019, arXiv preprint arXiv: 1909.04352

[25] Wang Z, Cuenca G, Zhou S, Xu F F, Neubig G. MCoNaLa: a benchmark for code generation from multiple natural languages. In: Proceedings of the Association for Computational Linguistics: EACL 2023. 2023, 265–273

[26] Zhang F, Zhang Z, Keung J W, Tang X, Yang Z, Yu X, Hu W. Data preparation for deep learning based code smell detection: a systematic literature review. Journal of Systems and Software, 2024, 216:

112131

[27] Liu H, Jin J, Xu Z, Zou Y, Bu Y, Zhang L. Deep learning based code smell detection. IEEE Transactions on Software Engineering, 2021, 47(9): 1811−1837

[28] Croft R, Xie Y, Babar M A. Data preparation for software vulnerability prediction: a systematic literature review. IEEE Transactions on Software Engineering, 2023, 49(3): 1044−1063

[29] Watson C, Cooper N, Palacio D N, Moran K, Poshyvanyk D. A systematic literature review on the use of deep learning in software engineering research. ACM Transactions on Software Engineering and Methodology (TOSEM), 2022, 31(2): 32

[30] Ghaisas S, Singhal A. Dealing with data for RE: mitigating challenges while using NLP and generative AI. In: Ferrari A, Ginde G, eds. Handbook on Natural Language Processing for Requirements Engineering. Cham: Springer, 2025, 457−486

[31] Wang S, Huang L, Gao A, Ge J, Zhang T, Feng H, Satyarth I, Li M, Zhang H, Ng V. Machine/deep learning for software engineering: a systematic literature review. IEEE Transactions on Software Engineering, 2023, 49(3): 1188−1231

[32] Kitchenham B. Procedures for performing systematic reviews. Keele: Keele University, 2004, 1−26

[33] Zhang H, Babar M A, Tell P. Identifying relevant studies in software engineering. Information and Software Technology, 2011, 53(6): 625−637

[34] Schardt C, Adams M B, Owens T, Keitz S, Fontelo P. Utilization of the PICO framework to improve searching PubMed for clinical questions. BMC Medical Informatics and Decision Making, 2007, 7: 16

[35] GitHub. Corporation for digital scholarship. See Github.com/digitalscholar website, 2023

[36] Liu B, Wang T, Zhang X, Fan Q, Yin G, Deng J. A neural-network based code summarization approach by using source code and its call dependencies. In: Proceedings of the 11th Asia-Pacific Symposium on Internetware. 2019, 12

[37] Liu S, Chen Y, Xie X, Siow J K, Liu Y. Retrieval-augmented generation for code summarization via hybrid GNN. In: Proceedings of the 9th International Conference on Learning Representations. 2021

[38] Choi Y, Kim S, Lee J H. Source code summarization using attention-based keyword memory networks. In: Proceedings of 2020 IEEE International Conference on Big Data and Smart Computing (BigComp). 2020, 564−570

[39] Yin P, Deng B, Chen E, Vasilescu B, Neubig G. Learning to mine parallel natural language/source code corpora from stack overflow. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. 2018, 388−389

[40] Hasan M, Muttaqueen T, Ishtiaq A A, Mehrab K S, Haque M M A, Hasan T, Ahmad W, Iqbal A, Shahriyar R. CoDesc: a large code-description parallel dataset. In: Proceedings of the Association for Computational Linguistics: ACL-IJCNLP 2021. 2021, 210−218

[41] LeClair A, McMillan C. Recommendations for datasets for source code summarization. In: Proceedings of 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2019, 3931−3937

[42] Mastropaolo A, Ciniselli M, Pascarella L, Tufano R, Aghajani E, Bavota G. Towards summarizing code snippets using pre-trained transformers. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension. 2024, 1−12

[43] Shi L, Mu F, Chen X, Wang S, Wang J, Yang Y, Li G, Xia X, Wang Q. 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. 2022, 107−119

[44] Ding X, Peng R, Chen X, Huang Y, Bian J, Zheng Z. Do code summarization models process too much information? Function signature may be all that is needed. ACM Transactions on Software Engineering and Methodology, 2024, 33(6): 160

[45] Song Z, Shang X, Li M, Chen R, Li H, Guo S. Do not have enough data? An easy data augmentation for code summarization. In: Proceedings of the 13th IEEE International Symposium on Parallel Architectures, Algorithms and Programming (PAAP). 2022, 1−6

[46] Zhu T, Li Z, Pan M, Shi C, Zhang T, Pei Y, Li X. Revisiting information retrieval and deep learning approaches for code summarization. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). 2023, 328−329

[47] Madeiral F, Urli S, Maia M, Monperrus M. BEARS: an extensible java bug benchmark for automatic program repair studies. In: Proceedings of the 26th IEEE international conference on software analysis, evolution and reengineering (SANER). 2019, 468−478

[48] Jiang Y, Liu H, Luo X, Zhu Z, Chi X, Niu N, Zhang Y, Hu Y, Bian P, Zhang L. BugBuilder: an automated approach to building bug repository. IEEE Transactions on Software Engineering, 2023, 49(4): 1443−1463

[49] Fakhoury S, Chakraborty S, Musuvathi M, Lahiri S K. Nl2fix: generating functionally correct code edits from bug descriptions. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering: Companion Proceedings. 2024, 410−411

[50] Tang H, Nadi S. On using stack overflow comment-edit pairs to recommend code maintenance changes. Empirical Software Engineering, 2021, 26(4): 68

[51] Antal G, Vándor N, Kolláth I, Mosolygó B, Hegedűs P, Ferenc R. PyBugHive: a comprehensive database of manually validated, reproducible python bugs. IEEE Access, 2024, 12: 123739−123756

[52] Lin D, Koppel J, Chen A, Solar-Lezama A. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In: Proceedings of 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. 2017, 55−56

[53] Campos E C, de Almeida Maia M. Common bug-fix patterns: a large-scale observational study. In: Proceedings of 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 2017, 404−413

[54] Ye H, Martinez M, Durieux T, Monperrus M. A comprehensive study of automatic program repair on the QuixBugs benchmark. Journal of Systems and Software, 2021, 171: 110825

[55] Yang D, Lei Y, Mao X, Lo D, Xie H, Yan M. Is the ground truth really accurate? Dataset purification for automated program repair. In: Proceedings of 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 2021, 96−107

[56] Zhong W, Li C, Zhang Y, Ge Z, Wang J, Ge J, Luo B. An automated and flexible multilingual bug-fix dataset construction system. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2023, 1881−1886

[57] Yadav A S, Wilson J N. BOSS: a dataset to train ML-based systems to repair programs with out-of-bounds write flaws. In: Proceedings of the 5th ACM/IEEE International Workshop on Automated Program Repair. 2024, 26−33

[58] Yang Y, He T, Feng Y, Liu S, Xu B. Mining python fix patterns via analyzing fine-grained source code changes. Empirical Software Engineering, 2022, 27(2): 48

[59] Pramod D, De Silva T, Thabrew U, Shariffdeen R, Wickramanayake S. BugsPHP: a dataset for automated program repair in PHP. In: Proceedings of the 21st International Conference on Mining Software Repositories. 2024, 128−132

[60] Wu Y, Li Z, Zhang J M, Liu Y. ConDefects: a complementary dataset to address the data leakage concern for LLM-based fault localization and program repair. In: Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering. 2024, 642−646

[61] Jin M, Shahriar S, Tufano M, Shi X, Lu S, Sundaresan N, Svyatkovskiy A. Inferfix: end-to-end program repair with LLMs. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2023, 1646−1656

[62] Chen Z, Kommrusch S, Tufano M, Pouchet L N, Poshyvanyk D, Monperrus M. SequenceR: sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering, 2021, 47(9): 1943−1959

[63] Richter C, Wehrheim H. TSSB-3M: mining single statement bugs at massive scale. In: Proceedings of the 19th International Conference on Mining Software Repositories. 2022, 418−422

[64] Bai J, Zhou L, Blanco A, Liu S, Wei F, Zhou M, Li Z. Jointly learning to repair code and generate commit message. In: Proceedings of 2021 Conference on Empirical Methods in Natural Language Processing. 2021, 9784−9795

[65] Xia C S, Wei Y, Zhang L. Automated program repair in the era of large pre-trained language models. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE). 2023, 1482−1494

[66] Kacmajor M, Kelleher J D. Automatic acquisition of annotated training corpora for test-code generation. Information, 2019, 10(2): 66

[67] Zhong L, Wang Z. Can LLM replace stack overflow? A study on robustness and reliability of large language model code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence. 2024, 21841−21849

[68] Yu H, Shen B, Ran D, Zhang J, Zhang Q, Ma Y, Liang G, Li Y, Wang Q, Xie T. CoderEval: a benchmark of pragmatic code generation with generative pretrained models. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 428−439

[69] Lai Y, Li C, Wang Y, Zhang T, Zhong R, Zettlemoyer L, Yih W T, Fried D, Wang S, Yu T. DS-1000: a natural and reliable benchmark for data science code generation. In: Proceedings of the 40th International Conference on Machine Learning. 2023, 756

[70] Du X, Liu M, Wang K, Wang H, Liu J, Chen Y, Feng J, Sha C, Peng X, Lou Y. Evaluating large language models in class-level code generation. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 982−994

[71] Tiwari S P, Prasad S, Thushara M. Machine learning for translating pseudocode to python: a comprehensive review. In: Proceedings of the 7th International Conference on Intelligent Computing and Control Systems (ICICCS). 2023, 274−280

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

[73] Li J, Sangalay A, Cheng C, Tian Y, Yang J. Fine tuning large language model for secure code generation. In: Proceedings of 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering. 2024, 86−90

[74] Peng Q, Chai Y, Li X. HumanEval-XL: a multilingual code generation benchmark for cross-lingual natural language generalization. In: Proceedings of 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation. 2024, 8383−8394

[75] Feng Y, Vanam S, Cherukupally M, Zheng W, Qiu M, Chen H. Investigating code generation performance of ChatGPT with crowdsourcing social data. In: Proceedings of the 47th IEEE Annual Computers, Software, and Applications Conference (COMPSAC). 2023, 876−885

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

[77] Tony C, Mutas M, Ferreyra N E D, Scandariato R. LLMSecEval: a dataset of natural language prompts for security evaluations. In: Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories (MSR). 2023, 588−592

[78] Akinobu Y, Kajiura T, Obara M, Kuramitsu K. NMT-based code generation for coding assistance with natural language. Journal of Information Processing, 2022, 30: 443−450

[79] Li H S, Mesgar M, Martins A, Gurevych I. Python code generation by asking clarification questions. In: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics. 2023, 14287−14306

[80] Shen S, Zhu X, Dong Y, Guo Q, Zhen Y, Li G. 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. 2022, 1533−1543

[81] Yin P, Deng B, Chen E, Vasilescu B, Neubig G. Learning to mine aligned code and natural language pairs from stack overflow. In: Proceedings of the 15th International Conference on Mining Software Repositories. 2018, 476−486

[82] Athiwaratkun B, Gouda S K, Wang Z, Li X, Tian Y, Tan M, Ahmad W U, Wang S, Sun Q, Shang M, Gonugondla S K, Ding H,

Kumar V, Fulton N, Farahani A, Jain S, Giaquinto R, Qian H, Ramanathan M K, Nallapati R. Multi-lingual evaluation of code generation models. In: Proceedings of the 11th International Conference on Learning Representations. 2023

[83] Zhu J, Shen M. Research on deep learning based code generation from natural language description. In: Proceedings of the 5th IEEE International Conference on Cloud Computing and Big Data Analytics (ICCCBDA). 2020, 188−193

[84] Cosma A, Iordache I B, Rosso P. RoCode: a dataset for measuring code intelligence from problem definitions in Romanian. In: Proceedings of 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation. 2024, 14173−14185

[85] Liu J, Xia C S, Wang Y, Zhang L. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In: Proceedings of the 37th International Conference on Neural Information Processing Systems. 2023, 943

[86] Jain N, Zhang T, Chiang W L, Gonzalez J E, Sen K, Stoica I. LLM-assisted code cleaning for training accurate code generators. In: Proceedings of the 12th International Conference on Learning Representations. 2024

[87] Yan W, Tian Y, Li Y, Chen Q, Wang W. CodeTransOcean: a comprehensive multilingual benchmark for code translation. In: Proceedings of the Association for Computational Linguistics: EMNLP 2023. 2023, 5067−5089

[88] Lei B, Ding C, Chen L, Lin P H, Liao C. Creating a dataset for high-performance computing code translation using LLMs: a bridge between OpenMP Fortran and C++. In: Proceedings of 2023 IEEE High Performance Extreme Computing Conference (HPEC). 2023, 1−7

[89] Jiao M, Yu T, Li X, Qiu G, Gu X, Shen B. On the evaluation of neural code translation: taxonomy and benchmark. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2023, 1529−1541

[90] Roziere B, Lachaux M A, Chanussot L, Lample G. Unsupervised translation of programming languages. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. 2020, 1730

[91] Rithy I J, Shakil H H, Mondal N, Sultana F, Shah F M. XTest: a parallel multilingual corpus with test cases for code translation and its evaluation. In: Proceedings of the 25th International Conference on Computer and Information Technology (ICCIT). 2022, 623−628

[92] Xie Y, Naik A, Fried D, Rose C. Data augmentation for code translation with comparable corpora and multiple references. In: Proceedings of the Association for Computational Linguistics: EMNLP 2023. 2023, 13725−13739

[93] Chen B, Golebiowski J, Abedjan Z. Data augmentation for supervised code translation learning. In: Proceedings of the 1st IEEE/ACM International Conference on Mining Software Repositories (MSR). 2024, 444−456

[94] Chen B, Golebiowski J, Abedjan Z. Towards data augmentation for supervised code translation. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering: Companion Proceedings. 2024, 352−353

[95] Pan R, Ibrahimzada A R, Krishna R, Sankar D, Wassi L P, Merler M, Sobolev B, Pavuluri R, Sinha S, Jabbarvand R. Lost in translation: a study of bugs introduced by large language models while translating code. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024, 82

[96] Nguyen D, Nam L, Dau A, Nguyen A, Nghiem K, Guo J, Bui N. The vault: a comprehensive multilingual dataset for advancing code understanding and generation. In: Proceedings of the Association for Computational Linguistics: EMNLP 2023. 2023, 4763−4788

[97] Chen P, Lampouras G. Exploring data augmentation for code generation tasks. In: Proceedings of the Association for Computational Linguistics: EACL 2023. 2023, 1542−1550

[98] Ahmed T, Devanbu P. Multilingual training for software engineering. In: Proceedings of the 44th International Conference on Software Engineering. 2022, 1443−1455

[99] Liu C, Lu S, Chen W, Jiang D, Svyatkovskiy A, Fu S, Sundaresan N, Duan N. Code execution with pre-trained language models. In: Proceedings of the Association for Computational Linguistics: ACL 2023. 2023, 4984−4999

[100] Saavedra N, Silva A, Monperrus M. Gitbug-actions: building reproducible bug-fix benchmarks with GitHub actions. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering: Companion Proceedings. 2024, 1−5

[101] Silva A, Saavedra N, Monperrus M. GitBug-Java: a reproducible benchmark of recent java bugs. In: Proceedings of the 1st IEEE/ACM International Conference on Mining Software Repositories (MSR). 2024, 118−122

[102] Wu Y, Jiang N, Pham H V, Lutellier T, Davis J, Tan L, Babkin P, Shah S. How effective are neural networks for fixing security vulnerabilities. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023, 1282−1294

[103] Karampatsis R M, Sutton C. How often do single-statement bugs occur?: the ManySStuBs4J dataset. In: Proceedings of the 17th International Conference on Mining Software Repositories. 2020, 573−577

[104] Liu K, Han Y, Liu Y, Zhang J M, Chen Z, Sarro F, Huang G, Ma Y. TrickyBugs: a dataset of corner-case bugs in plausible programs. In: Proceedings of the 21st International Conference on Mining Software Repositories. 2024, 113−117

[105] Herbold S, Trautsch A, Ledel B. Large-scale manual validation of bugfixing changes. In: Proceedings of the 17th International Conference on Mining Software Repositories. 2020, 611−614

[106] Allamanis M, Barr E T, Devanbu P, Sutton C. A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR), 2018, 51(4): 81

[107] Nguyen H A, Nguyen T T, Wilson G Jr, Nguyen A T, Kim M, Nguyen T N. A graph-based approach to API usage adaptation. ACM SIGPLAN Notices, 2010, 45(10): 302−321

[108] Husain H, Wu H H, Gazit T, Allamanis M, Brockschmidt M. CodeSearchNet challenge: evaluating the state of semantic code search. 2019, arXiv preprint arXiv: 1909.09436

[109] Papineni K, Roukos S, Ward T, Zhu W J. Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. 2002, 311−318

[110] Banerjee S, Lavie A. METEOR: an automatic metric for MT

evaluation with improved correlation with human judgments. In: Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization. 2005, 65–72

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

[112] Radford A, Narasimhan K, Salimans T, Sutskever H. Improving language understanding by generative pre-training. 2018

[113] Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto H P, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A, Puri R, Krueger G, Petrov M, Khlaaf H, Sastry G, Mishkin P, Chan B, Gray S, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Such F P, Cummings D, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss W H, Nichol A, Paino A, Tezak N, Tang J, Babuschkin I, Balaji S, Jain S, Saunders W, Hesse C, Carr A N, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight M, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W. Evaluating large language models trained on code. 2021, arXiv preprint arXiv: 2107.03374

[114] Rozière B, Gehring J, Gloeckle F, Sootla S, Gat I, Tan X E, Adi Y, Liu J, Sauvestre R, Remez T, Rapin J, Kozhevnikov A, Evtimov I, Bitton J, Bhatt M, Ferrer C C, Grattafiori A, Xiong W, Défossez A, Copet J, Azhar F, Touvron H, Martin L, Usunier N, Scialom T, Synnaeve G. Code Llama: open foundation models for code. 2023, arXiv preprint arXiv: 2308.12950

Shihao WENG received the BE degree in computer science in 2022 from the School of Artificial Intelligence and Computer Science, Jiangnan University, China. He is currently working toward the PhD degree in software engineering with the Software Institute, Nanjing University, China. His research interests lie in quality assurance of complex software systems, particularly in data quality assurance for intelligent software.

Yang FENG received the BS and PhD degrees from the Nanjing University, China. and the University of California, USA, respectively. He is currently an assistant research professor in the Department of Computer Science and Technology at Nanjing University, China. His research interests lie in the quality assurance of complex software systems and the implementation of reliable software infrastructure.

Yining YIN received the BE degree in Mathematics and Applied Mathematics from the Department of Mathematics, Wuhan University, China. She is currently working toward the PhD degree in software engineering with the Software Institute, Nanjing University, China. Her research interests lie in software testing, particularly in intelligent software testing and data quality assurance for intelligent software.

Zhenlun ZHANG received the BE degree in software engineering in 2024 from the School of Mechanical, Electrical & Information Engineering, Shandong University, China. He is currently working toward the ME degree in Nanjing University, China. His research focuses on quality assurance of complex software systems, especially data quality in intelligent software.

Baowen XU received the BS, MS, and PhD degrees in computer science from Wuhan University, China, Huazhong University of Science and Technology, China, and Beihang University, China respectively. He is a professor in the Department of Computer Science and Technology at Nanjing University, China. His main research interests are programming languages, software testing, software maintenance, and software metrics. He is a member of the IEEE and the IEEE Computer Society.