Bridging Generation and Training: A Systematic Review of Quality Issues in LLMs for Code

KAIFENG HE*, Sun Yat-sen University, China
XIAOJUN ZHANG*, Sun Yat-sen University, China
PEILIANG CAI*, Sun Yat-sen University, China
MINGWEI LIU†, Sun Yat-sen University, China
YANLIN WANG, Sun Yat-sen University, China
CHONG WANG, Nanyang Technological University, Singapore
KAIFENG HUANG, Tongji University, China
BIHUAN CHEN, Fudan University, China
XIN PENG, Fudan University, China
ZIBIN ZHENG, Sun Yat-sen University, China

Large language models (LLMs) frequently generate defective outputs in code generation tasks, ranging from logical bugs to security vulnerabilities. While these generation failures are often treated as model-level limitations, empirical evidence increasingly traces their root causes to imperfections within the training corpora. Yet, the specific mechanisms linking training data quality issues to generated code quality issues remain largely unmapped. This paper presents a systematic literature review of 114 primary studies to investigate how training data quality issues propagate into code generation. We establish a unified taxonomy that categorizes generated code quality issues across nine dimensions and training data quality issues into code and non-code attributes. Based on this taxonomy, we formalize a causal framework detailing 18 typical propagation mapping mechanisms. Furthermore, we synthesize state-of-the-art detection and mitigation techniques across the data, model, and generation lifecycles. The reviewed literature reveals a clear methodological shift: quality assurance is transitioning from reactive, heuristic-based post-generation filtering toward proactive, data-centric governance and closed-loop repair. Finally, we identify open challenges and outline research directions for developing reliable LLMs for code through integrated data curation and continuous evaluation. Our repository is available at https://github.com/SYSUSELab/From-Data-to-Code.

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: Code Generation, Large Language Model, Code Quality, Data Quality, Systematic Literature Review, Quality Detection, Mitigation

*These authors contributed equally to this work.
†Corresponding author.

Authors' Contact Information: Kaifeng He, hekf5@mail2.sysu.edu.cn, Sun Yat-sen University, Zhuhai, China; Xiaojun Zhang, zhangxj229@mail2.sysu.edu.cn, Sun Yat-sen University, Zhuhai, China; Peiliang Cai, caipliang3@mail2.sysu.edu.cn, Sun Yat-sen University, Zhuhai, China; Mingwei Liu, liumw26@mail.sysu.edu.cn, Sun Yat-sen University, Zhuhai, China; Yanlin Wang, wangylin36@mail.sysu.edu.cn, Sun Yat-sen University, Zhuhai, China; Chong Wang, chong.wang@ntu.edu.sg, Nanyang Technological University, Singapore, Singapore; Kaifeng Huang, kaifengh@tongji.edu.cn, Tongji University, Shanghai, China; Bihuan Chen, bhchen@fudan.edu.cn, Fudan University, Shanghai, China; Xin Peng, pengxin@fudan.edu.cn, Fudan University, Shanghai, China; Zibin Zheng, zhzibin@mail.sysu.edu.cn, Sun Yat-sen University, Zhuhai, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

ACM Reference Format:

Kaifeng He, Xiaojun Zhang, Peiliang Cai, Mingwei Liu, Yanlin Wang, Chong Wang, Kaifeng Huang, Bihuan Chen, Xin Peng, and Zibin Zheng. 2026. Bridging Generation and Training: A Systematic Review of Quality Issues in LLMs for Code. 1, 1 (April 2026), 50 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In recent years, large language models (LLMs) have demonstrated unprecedented capabilities across diverse domains [12]. Built on the Transformer architecture and trained on massive corpora of open-source code, both general-purpose LLMs and code-specialized variants have emerged as transformative tools in modern development workflows when applied to software engineering tasks [14]. These models now support or fully automate core software tasks, including code completion [47], program repair [27, 62], test case generation [144, 145], and code documentation [31]. This trend presents significant opportunities to improve software development efficiency and quality, shortening the cycle of routine development tasks and lowering the threshold for novice developers, which is gradually reshaping the traditional paradigm of software engineering [112].

```
1:  from flask import Flask, request
2:  import sqlite3, pandas as pd
3:
4:  app = Flask(__name__)
5:
6:  @app.route('/register', methods=['POST'])
7:  def register_user():
8:      # Raw input (no validation)
9:      username, email, pwd = request.form['username'], request.form['email'], request.form['password']

10:     # 🚨SECURITY: SQL Injection (unparameterized query)
11:     query = f"INSERT INTO users VALUES ('{username}', '{email}', '{pwd}')"

12:     # ❌CORRECTNESS: Deprecated pandas.append() (pandas 2.0+)
13:     df = pd.read_csv('user_db.csv').append(pd.DataFrame({'user':[username]}))
14:     df.to_csv('user_db.csv', index=False)

15:     # 🛡ROBUSTNESS: No exception handling
16:     conn = sqlite3.connect('user.db')
17:     conn.execute(query)  # Unsafe execution
18:     conn.commit()
19:     conn.close()

20:     return "Success", 200
21:
22: if __name__ == '__main__':
23:     app.run(debug=True)
```

Fig. 1. LLM-generated code with multiple quality defects.

However, despite their remarkable utility, LLMs frequently produce code outputs with critical quality issues that hinder practical adoption [88]. Empirical studies confirm that unfiltered LLM-generated code suffers from widespread syntax and logical errors, numerous security vulnerabilities, and significant maintainability flaws [13]. For example, Figure 1 illustrates a typical LLM-generated user registration endpoint, which exhibits multiple overlapping quality defects: an unparameterized SQL query (Line 11) that exposes critical SQL injection vulnerabilities, use of the deprecated `pandas.append()` method (Line 13) that breaks compatibility with modern pandas versions, and complete lack of exception handling (Lines 16-19) that leaves the application prone to runtime crashes. These defects not only diminish model utility but also introduce tangible risks: incorrect core business code has led to substantial financial losses for

enterprises, while security vulnerabilities in network-facing code have triggered data breaches affecting thousands of users [87]. Systematically dissecting these quality issues is therefore both theoretically meaningful and practically urgent.

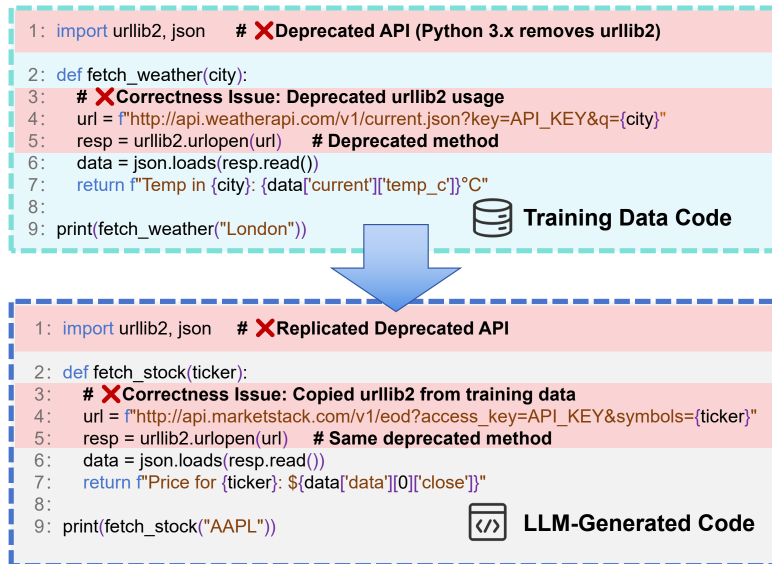

Fig. 2. Training data quality issues propagated to generated code.

Training data serves as the foundation of LLM learning. Recent studies confirm that its quality directly dictates the reliability of LLM-generated code [16, 133]. Low-quality datasets primarily drive generation defects: duplicated samples cause models to overfit, weakening generalization [58]; untested snippets propagate poor programming practices [114]; and imbalanced data limits adaptability to specialized domains [153]. As shown in Figure 2, a flawed snippet using the deprecated `urllib2` library in the training data can be directly replicated by LLMs in new contexts, even though modern Python versions have removed this API entirely. Critically, the specific mechanisms translating training data quality issues into generated code quality issues remain insufficiently explored. The inability to trace generation failures back to their dataset origins prevents targeted remediation, as developers cannot effectively identify which data flaws require resolution.

Beyond the missing link between dataset and generation quality, existing research on detection and mitigation methods is fragmented. Techniques for identifying quality issues are scattered across data curation, model training, and post-generation refinement, with no unified framework to guide their application [44]. Similarly, diverse mitigation strategies, such as data cleaning and prompt engineering, are frequently evaluated in isolation. Consequently, it remains unclear how these methods complement one another or scale across different scenarios [100]. This fragmentation undermines the practical impact of research, as industrial users lack clear guidance on building end-to-end quality assurance pipelines for LLMs in code generation.

Despite progress in individual subtopics, existing research suffers from three critical gaps that hinder a holistic understanding of quality governance for LLM code generation: ❶ Fragmented taxonomies: Most studies focus on isolated quality dimensions, such as evaluating security vulnerabilities in isolation [95]. Consequently, recent literature

highlights the persistent lack of a unified framework to systematically categorize both generated code quality issues [51] and training data quality issues [132]; ❷ Unclear causal mappings and backtracking mechanisms: While large-scale model reports acknowledge the opacity between data curation and output quality [63], few works clarify how specific dataset flaws directly or indirectly translate into code defects, nor do they establish effective ways to trace generation issues back to their data origins; ❸ Disjoint detection and mitigation methods: As emphasized by recent systematic reviews [42], techniques for identifying and resolving quality issues remain scattered across different phases, lacking an integrated framework to guide their application across the full lifecycle of LLMs for code generation.

To address these gaps, this paper presents a systematic literature review of 114 primary studies on LLM training data and code generation. Through an in-depth analysis of these studies, our findings reveal that generated code quality issues (e.g., correctness flaws and security vulnerabilities) are rarely isolated reasoning deficits, but rather deeply rooted in training data quality issues. These training data quality issues span two dimensions: code attributes (defects in individual code samples) and non-code attributes (textual noise and macro-level dataset anomalies). Consequently, they propagate into code outputs through two distinct mechanisms: direct mappings (memorizing explicit code defects) and indirect mappings (distorting the learning distribution). Furthermore, because current mitigation efforts rely heavily on reactive post-generation filtering, there is an urgent need to transition toward proactive, data-centric governance.

In summary, the main contributions of this work are as follows:

- We propose a **dual-dimensional taxonomy** that categorizes generated code quality issues across nine quality dimensions and organizes training data quality issues into two major groups.
- We develop a **causal framework** that characterizes propagation mechanisms linking training data quality issues to generated code quality issues, enabling systematic tracing of generation failures to potential dataset origins.
- We synthesize **existing detection and mitigation techniques** across the lifecycle of LLMs for code generation, covering data curation, model training, and post-generation validation.

The remainder of this paper is organized as follows: Section 2 provides background on LLMs for code generation, generated code quality, and dataset quality. Section 3 details our systematic review methodology. Section 4 presents our core findings addressing the three research gaps, with a focus on mapping relationships, backtracking mechanisms, and integrated detection/mitigation frameworks. Section 5 discusses key insights, challenges, and implications for data-centric quality assurance. Section 6 analyzes threats to validity. Section 7 concludes with a summary and future work.

2 Background

This section provides the conceptual background for the subsequent review. We first introduce LLMs in code generation to clarify the technical context in which the reviewed studies are situated. We then describe the data-driven lifecycle of LLM-based code generation, because this lifecycle provides the basis for understanding how training data quality issues may propagate into generated code. Next, we define generated code quality issues and training data quality issues from a dual perspective to establish a consistent terminology for the later synthesis. Finally, we review representative surveys to position this study in relation to existing review efforts and to clarify the gap addressed in this work.

2.1 LLMs in Code Generation

LLMs have recently demonstrated remarkable capability in generating, understanding, and reasoning about source code [147]. Based on the Transformer architecture [121], both general-purpose LLMs such as GPT-4 [92], Gemini [116], Qwen [9] and code-specialized variants such as Codex [14], CodeT5 [128], DeepSeekCoder [37], CodeLlama [106], and StarCoder [63] learn statistical and semantic patterns of programming languages from a massive corpus of source code collected from public repositories.

Compared to traditional NLP models, LLMs applied to code generation must satisfy significantly stricter constraints regarding syntax, semantics, and executability [4]. Code is inherently more brittle than natural text, as minor syntactic deviations can result in compilation errors, while semantically inconsistent logic can cause catastrophic functional failures in real-world software systems. Although LLMs have enabled a broad range of code-related applications, including automatic bug fixing, code translation, and test synthesis [147], they frequently produce outputs that are structurally plausible but functionally incorrect. This phenomenon is often referred to as "hallucinated code" [148]. This discrepancy between surface-level fluency and actual software reliability requires a deeper, systematic examination of code generation quality.

2.2 Data-Driven Lifecycle of LLM-based Code Generation

Understanding the root causes of degraded generation quality [71] requires examining the data-driven lifecycle of LLMs in code generation tasks. As illustrated in Figure 3, this lifecycle can be visualized as an end-to-end continuous process moving from Raw Data through Data Processing, Model Training, and Code Generation.

The performance of LLMs in code generation is fundamentally constrained by the data they consume throughout their pipeline [59]. Specifically, the Model Training stage typically consists of two primary phases [94]. The first is large-scale pre-training on raw, heterogeneous code corpora and documentation. While this phase embeds broad programming knowledge, it also indiscriminately absorbs unstructured noise, deprecated APIs, and insecure coding practices prevalent in open-source repositories [95]. The second phase involves instruction-tuning or reinforcement learning with human feedback (RLHF) on curated datasets to align the model with specific human intents [36].

Consequently, the Code Generation phase (inference) is intrinsically tied to this data lineage. LLMs used for code generation operate as advanced statistical pattern matchers; therefore, the quality of their generated artifacts is a direct reflection of their training distribution. From a software engineering perspective, unmanaged flaws introduced during the Data Processing stage accumulate as a form of data-driven technical debt [109]. As depicted by the "Mappings" module on the left side of Figure 3, if the initial data pipeline suffers from training data quality issues (such as imbalance, vulnerabilities, or redundancy), these training data quality issues propagate through the training process and are inevitably externalized as structural and functional defects (such as incorrectness, unreadability, or insecurity) in the generated code [126]. To counter this, a multi-layered quality governance framework is required. As illustrated on the right side of Figure 3, this involves targeted mitigation strategies applied at the Data-level, the Model-level, and the Generation-level.

2.3 Quality Issues: A Dual-Perspective Definition

Traditional software quality frameworks (e.g., ISO/IEC 25010 [46]) evaluate systems across dimensions like correctness, reliability, and maintainability, assuming defects arise from human error, architectural decay, or environmental misconfigurations. LLM-generated code complicates this landscape. While it must meet these conventional standards, it also

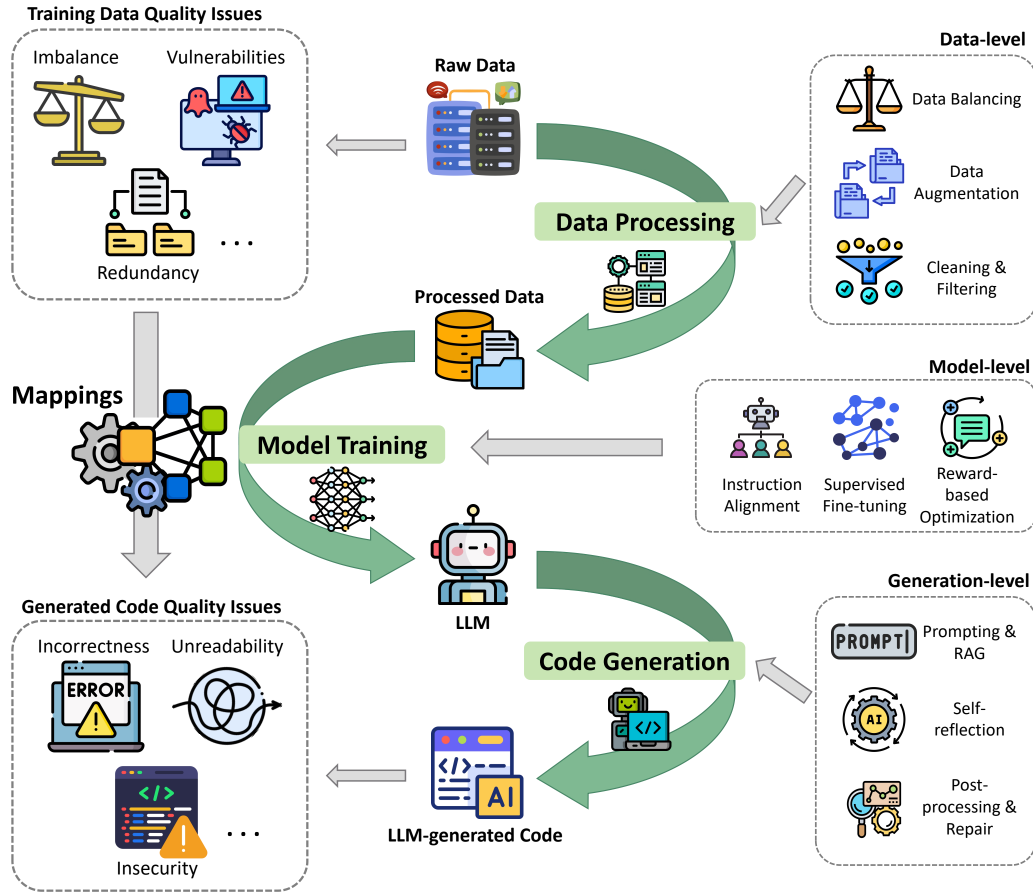

Fig. 3. Conceptual framework of quality issues and mitigation in the LLM lifecycle.

exhibits generation-specific anomalies that transcend traditional defect taxonomies. Given the fragmented terminology currently used to describe these emergent issues, we first formalize the two core concepts investigated in this review to establish a consistent vocabulary.

First, we define a **Generated Code Quality Issue** as any anomaly in LLM-generated code that degrades its functional correctness, non-functional quality, risk-related security and compliance, or generation-specific attributes, strictly confining our scope to the output artifacts rather than intrinsic model metrics. These issues range from traditional syntax and logic errors to unique anomalies like hallucinated APIs and redundant outputs. Unlike human-written bugs, they stem directly from the interplay between the model's generative mechanisms and its underlying training data distributions [3, 40, 118].

Second, we define a **Training Data Quality Issue** as any defect or distributional bias within the pre-training and fine-tuning corpora specifically curated for code generation tasks. These issues encompass various facets of data imperfection. They include explicit flaws embedded in specific training samples, such as code snippets harboring incorrect, deprecated, or unmaintainable logic, alongside noisy textual data. Furthermore, they involve broader corpus-wide anomalies like

language imbalance, data redundancy, inadequate diversity, benchmark contamination, and low-value noise [59, 85]. These data-centric problems ultimately distort the programming semantics learned by the LLM.

The mechanism by which training data quality issues transform into generated code quality issues represents a critical gap in current software engineering research [29]. Recognizing and formalizing these dual perspectives is the essential first step toward shifting from reactive post-generation repair to proactive, data-centric quality governance.

2.4 Related Surveys

Recent literature has extensively explored LLMs from either the data management perspective or the code generation quality perspective.

From the data perspective, existing work tends to isolate specific phases of the model lifecycle. For pre-training, researchers have systematized data processing pipelines, detailing mechanisms like data deduplication, quality filtering, and domain mixing [154]. For post-training, efforts focus on data-efficient paradigms—such as data selection, distillation, and synthetic data generation—to mitigate high annotation costs and diminishing returns from data scaling [79].

Conversely, evaluation-centric studies assess the artifacts produced by LLMs. Security-oriented surveys identify severe vulnerabilities introduced during code generation (e.g., hardcoded credentials and out-of-bounds accesses) and review automated testing countermeasures [104]. Extending beyond security, empirical studies like Kharma et al. [54] conduct multi-dimensional evaluations across diverse programming languages, revealing pronounced architectural and memory-safety degradation in low-level languages like C/C++. Furthermore, recent work highlights non-functional discrepancies, demonstrating that while LLM-generated code achieves baseline readability, it exhibits significant stylistic inconsistencies in API preferences and formatting when compared to human developers [127].

Despite their respective depths, these two tracks remain disconnected. Data curation surveys rarely trace how specific corpus flaws manifest as downstream software defects, while evaluation studies primarily benchmark generation failures without investigating the underlying data provenance. To our knowledge, this is the first systematic literature review to explicitly bridge this gap, formalizing the causal mechanisms that map training data quality issues to generated code quality issues.

3 Review Methodology

We follow established guidelines for conducting software engineering systematic literature reviews [56]. The process of paper collection and filtering, depicted in Figure 4, encompasses seven structured stages: defining research questions, designing search strings, conducting database searches, performing backward and forward snowballing, consolidating and de-duplicating records, applying inclusion and exclusion criteria, and assessing quality.

3.1 Research Questions

This survey systematically investigates generated code quality issues in LLM-generated code and their potential origins in training datasets. To structure this research problem, we formulate the following five research questions (RQs).

- **RQ1: What generated code quality issues exist in LLM-generated code, and how can they be systematically categorized?** This question identifies and organizes the spectrum of quality issues in generated code, establishing a taxonomy of generated code quality issues.

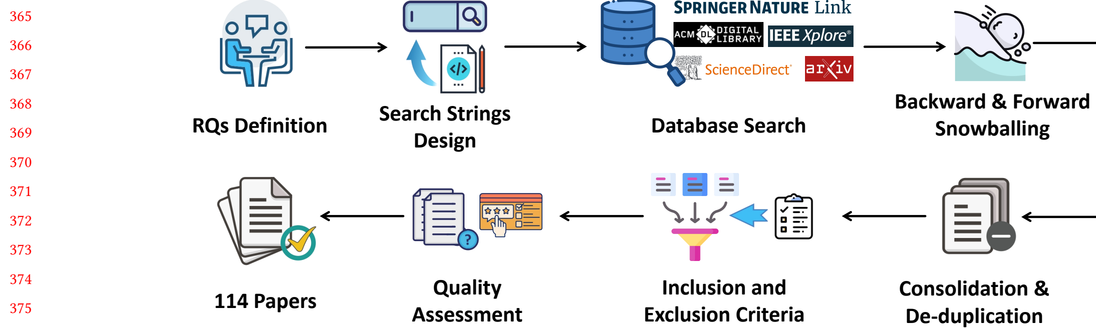

Fig. 4. Overview of the process of paper collection and filtering.

- **RQ2: What training data quality issues exist in datasets used for LLM training, and how can they be categorized?** This question examines training data quality issues in training datasets and organizes them into a structured taxonomy.
- **RQ3: How do training data quality issues influence generated code quality issues?** This question explores the underlying connections between training data quality issues and generated code quality issues, aiming to uncover and formalize the mapping mechanisms through which data issues propagate.
- **RQ4: What techniques exist for detecting generated code quality issues and training data quality issues?** This question reviews existing methods for identifying quality issues in both generated code and training data.
- **RQ5: What mitigation strategies have been proposed to address these quality issues?** This question synthesizes existing mitigation approaches into a lifecycle-oriented framework for improving the quality of LLM-based code generation systems.

Overall, these RQs are organized to progressively move from problem characterization to mechanism analysis and finally to issue handling. RQ1 and RQ2 first characterize the quality issues in generated code and training datasets, respectively. Building on this foundation, RQ3 analyzes how training data quality issues propagate to generated code quality issues, providing insights into the mechanisms underlying LLM-based code generation. Finally, RQ4 and RQ5 focus on issue handling by reviewing existing detection techniques and mitigation strategies across the LLM lifecycle.

3.2 Search Strategy

We employed a multi-stage search strategy combining systematic keyword-based querying with iterative snowballing [134]. We designed Boolean search expressions comprising three core conceptual groups:

(1) **LLMs for Code:** (`“large language model” OR “LLM” OR “Code LLM” OR “code generation” OR “code completion” OR “program repair”`)
(2) **Generated Code Quality Issues:** `AND (“code quality” OR “defect” OR “bug” OR “vulnerability” OR “hallucination” OR “correctness” OR “security” OR “compliance” OR “robustness” OR “maintainability” OR “understandability” OR “efficiency” OR “repetition”`

(3) **Training Data Quality Issues:** `AND (“training data” OR “dataset” OR “corpus” OR “data quality” OR “data noise” OR “data contamination” OR “data leakage” OR “imbalance” OR “redundancy” OR “duplication” OR “diversity”)`

The initial database search was executed across five major digital libraries representing software engineering and artificial intelligence venues:

- **IEEE Xplore:** Captures works published in venues such as *IEEE Transactions on Software Engineering (TSE)*, *ICSE*, and *ASE*.
- **ACM Digital Library:** Serves as a repository for software engineering and programming language research, including venues such as *FSE*, *TOSEM*, *PLDI*, and *OOPSLA*.
- **SpringerLink:** Offers access to journals and conference proceedings on empirical methods and data-driven software engineering.
- **ScienceDirect:** Hosts domain-specific journals, including *Information and Software Technology (IST)* and *Journal of Systems and Software (JSS)*.
- **arXiv:** Acts as the primary repository for rapid dissemination of LLM research, capturing early empirical explorations prior to formal publication.

To minimize the omission of relevant works, we subsequently performed backward and forward snowballing on the initially retrieved papers [134].

3.3 Study Selection

We established formal inclusion and exclusion criteria aligned with our research questions. Primary studies satisfying all the following **Inclusion Criteria** were retained:

- **Relevance to LLMs and Code:** Investigates LLMs or auxiliary neural approaches applied explicitly to source code generation, completion, repair, or synthesis tasks.
- **Focus on Quality Dimensions:** Analyzes specific quality attributes of generated code or training corpora.
- **Empirical Grounding:** Provides verifiable empirical evaluation, reproducible artifacts, or concrete quantitative/qualitative findings.
- **Publication Standards:** Published in peer-reviewed venues or available as methodologically sound preprints.

Conversely, studies meeting any of the following **Exclusion Criteria** were discarded:

- **Domain Mismatch:** Focuses exclusively on natural language tasks without code-centric analysis, or addresses traditional (non-neural) code generation.
- **Methodological Opacity:** Lacks adequate empirical grounding, methodological transparency, or presents conceptual opinions without verifiable data.
- **Language and Accessibility:** Written in languages other than English or lacks full-text accessibility.
- **Duplication:** Represents an earlier or subsumed version of a retained extended study.

Snowballing and Consolidation: Prior to the formal screening, we performed backward and forward snowballing on the initially retrieved papers to mitigate the omission of relevant works. Subsequently, all records gathered from the database searches and the snowballing process were consolidated, and a rigorous de-duplication step was executed to eliminate identical entries across different sources.

469 470 471 472 473

Screening Process: Based on the consolidated and de-duplicated literature pool, three authors independently screened the titles and abstracts. Papers passing this phase underwent a full-text review against the predefined criteria. Discrepancies were documented and resolved through iterative consensus meetings to ensure full agreement on the final inclusion set.

474 475

3.4 Quality Assessment

476 477 478 479 480 481 482

We conducted a structured quality assessment of all candidate studies passing the full-text screening. Each paper was independently evaluated by three reviewers against eight predefined criteria (summarized in Table 1): (1) Relevance to Code Generation Quality, (2) Relevance to Dataset Quality, (3) Methodological Rigor, (4) Depth of Causal Analysis, (5) Detection and Mitigation Techniques, (6) Data and Experimental Transparency, (7) Clarity and Reporting Completeness, and (8) Publication Quality.

483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517

Table 1. Quality Assessment Criteria for Included Studies

Criterion	Description	Score Options
Relevance to Code Generation Quality	Whether the study investigates LLM-based code generation, completion, repair, or understanding tasks, and explicitly addresses quality aspects such as correctness, robustness, or maintainability.	Yes / No
Relevance to Dataset Quality	Whether the study discusses or analyzes the quality of training or fine-tuning datasets, including issues such as noise, duplication, contamination, imbalance, or outdated data.	Yes / No
Methodological Rigor	Whether the study clearly describes its objectives, hypotheses, research design, controlled variables, and experimental procedures, demonstrating methodological validity.	Yes / Partially / No
Depth of Causal Analysis	Whether the study goes beyond descriptive observations to analyze, model, or empirically verify causal relationships between data quality and generation quality.	Yes / No
Detection and Mitigation Techniques	Whether the study proposes concrete techniques, frameworks, or tools for detecting or mitigating code or dataset quality issues.	Yes / No
Data and Experimental Transparency	Whether the study releases or sufficiently documents experimental code, datasets, parameter configurations, or evaluation metrics to enable reproducibility.	Yes / No
Clarity and Reporting Completeness	Whether the paper presents a clear structure, comprehensive experimental reporting, and explicit discussion of limitations.	Yes / Partially / No
Publication Quality	Whether the study was published in a peer-reviewed venue (e.g., TSE, TOSEM, ICSE, FSE). If it is an arXiv preprint, whether its methodological quality meets comparable standards.	Yes / Partially / No

518 519 520

The evaluation utilized a mixed scoring system depending on the specific criterion, with options comprising either binary (*Yes*/*No*) or ternary (*Yes*/*Partially*/*No*) responses. Responses were scored as follows: *Yes* = 2, *Partially* = 1, and *No*

521 = 0 (maximum attainable score: 16). Studies scoring below 10 were excluded from the final synthesis. Discrepancies
522 among reviewers were resolved through consensus discussions.
523
524
525 ### 3.5 Data Extraction and Synthesis

526 We executed a structured data extraction protocol to capture information relevant to our research questions. Three
527 authors independently extracted the data, resolving inconsistencies through consensus.
528
529 *Data Extraction.* For each included primary study, we extracted both bibliometric metadata and analytical findings,
530
531 mapping directly to our RQs:

532
- **Bibliographic & Contextual Data:** Title, authors, publication year, and venue.
533
- **Code Quality Observations (RQ1):** Specific quality dimensions evaluated (e.g., correctness, security, main-
534
535 tainability) and the observed generated code quality issues.
536
- **Dataset Quality Aspects (RQ2):** Identified training data quality issues (e.g., contamination, noise, imbalance,
537 duplication).
538
- **Causal Mappings (RQ3):** Empirical or conceptual links established between dataset characteristics and
539
540 downstream generation failures.
541
- **Detection & Mitigation Interventions (RQ4 & RQ5):** Proposed diagnostic tools, metrics, model-level
542 alignments, or data curation strategies.
543
544 *Data Synthesis and Coding.* To synthesize the extracted data, we employed an iterative qualitative coding approach.
545 The procedure consisted of three stages:
546
547
(1) **Open Coding:** Initial review of the papers to identify and label specific phenomena.
548
(2) **Axial Coding:** Grouping the initial labels into higher-level conceptual categories representing broader dimen-
549 sions.
550
551
(3) **Selective Coding:** Aligning these consolidated categories with our predefined research questions (RQ1–RQ5)
552 to establish the overarching taxonomies and causal mappings.
553
554 Complementing the qualitative synthesis, we quantitatively aggregated the frequencies of specific issue types and
555 mitigation strategies.
556
557 ### 3.6 Demographic Overview of Primary Studies

558
559 The final synthesis encompasses 114 primary studies. This section provides a demographic overview of the included
560 literature.
561
562 *Temporal Distribution.* As illustrated in Figure 5, research on generated code quality issues has experienced rapid
563 and accelerating growth. The cumulative distribution spans from the third quarter of 2021 to the first quarter of 2026,
564 reaching a total of 114 primary studies. Notably, the volume of publications surged significantly starting in late 2023, a
565 trajectory that closely aligns with the widespread release and adoption of foundational LLMs.
566
567 *Publication Venues.* As shown in Figure 7, the selected studies are distributed across a wide range of publication
568 venues. The largest proportion is from arXiv, suggesting that this area is developing rapidly and that many results are
569
570 disseminated first as preprints. In addition, the studies also appear in recognized SE venues (e.g., ICSE, FSE, ASE, TSE,
571 TOSEM) and AI/NLP venues (e.g., ACL, NeurIPS, ICLR), indicating the interdisciplinary nature of this research topic.
572

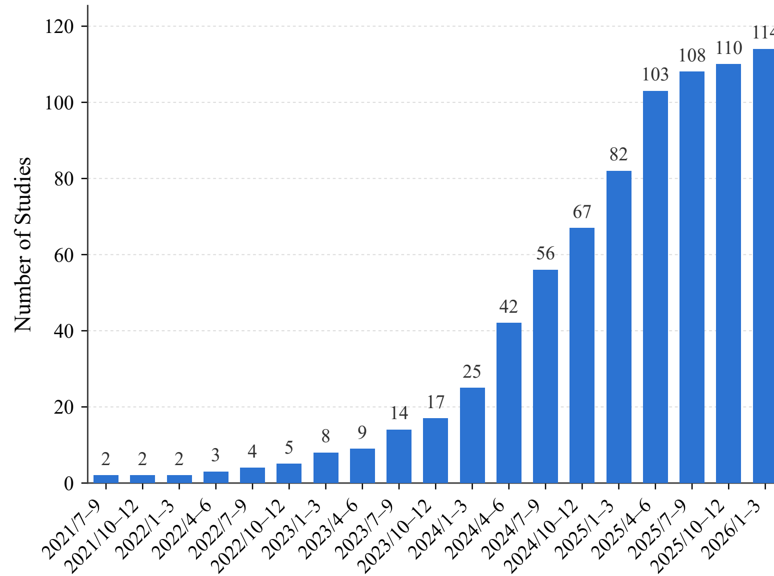

Fig. 5. Cumulative number of included studies by publication period.

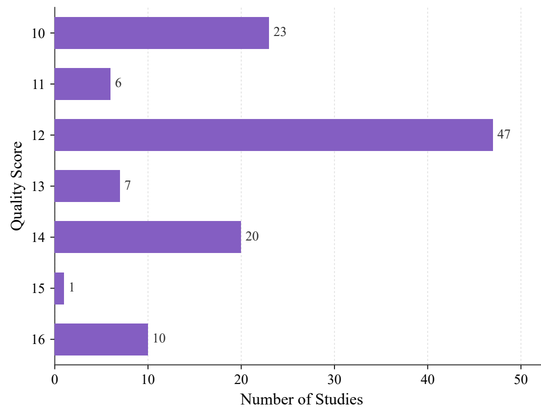

Fig. 6. Distribution of Included Studies by Quality Score.

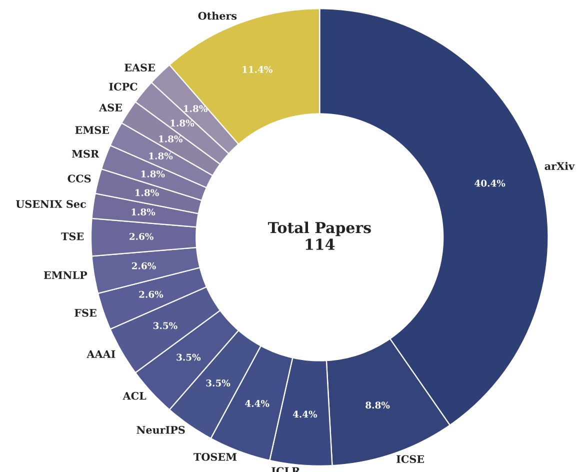

Fig. 7. Venue Distribution of Included Studies.

Quality Score Distribution. The distribution of their quality assessment scores is presented in Figure 6. The scores range from 10 to 16, with the most frequent score being 12 (47 studies), followed by 10 (23 studies) and 14 (20 studies).

4 Findings

This section presents our findings organized around five research questions (RQs). Each RQ addresses a distinct aspect of quality issues in LLM code generation and their underlying data. The structure follows: (1) identifying generated

code quality issues, (2) identifying training data quality issues, (3) mapping between them, (4) detecting quality issues, and (5) mitigating quality issues.

4.1 RQ1: What generated code quality issues exist in LLM-generated code, and how can they be systematically categorized?

We begin by examining the spectrum of quality deficiencies exhibited in LLM-generated code. Building on the formal definition of generated code quality issues established in Section 2.3 (i.e., structural, functional, or non-functional anomalies in LLM-generated code that degrade its core attributes), we synthesize findings from primary studies to address the fragmentation of existing classifications. Specifically, we establish a unified, multi-dimensional taxonomy that integrates functional, non-functional, risk-related, and generation-specific quality concerns, with detailed classifications and supporting references summarized in Figure 8.

4.1.1 Dimensions of generated code quality issues. After synthesizing prior taxonomies and empirical reports, we identify nine major and recurring dimensions that constitute our final classification of quality issues in LLM-generated code:

Correctness. Emphasizes failure to generate the intended semantics or valid execution results, with common symptoms including syntax errors, logical errors, and improper use of APIs or dependencies. These issues can manifest in different ways:

- **Syntax Errors**: Occur when the generated code cannot compile or parse, often due to missing semicolons, mismatched brackets, or incomplete statements. These errors prevent the code from executing properly [88].
- **Logical Errors**: Arise when the generated code produces unexpected results during runtime, even though the code compiles correctly. This could lead to runtime errors or incorrect outputs. For instance, a program might fail to handle boundary conditions correctly or throw exceptions during specific operations, despite successful compilation [50].
- **Improper Use of APIs or Dependencies**: Happens when the generated code references non-existent functions, deprecated APIs, or uses incorrect versions of libraries. This could be due to model hallucinations or outdated training data, leading to invalid or unsafe code. For example, a function from the deprecated `urllib` module in Python or a mismatch in library versions might cause the code to fail in the current environment [125].Therefore, correctness problems are not only limited to syntax or logic errors in the code itself but are also closely tied to how the model understands and references the libraries and tools in the current software ecosystem. As training data and API versions evolve, LLMs face continually changing correctness challenges, especially in long-running projects or applications spanning multiple technology stacks [115].

Security. Patterns that expose code to attacks, data leakage, or unsafe execution. Security issues can be further categorized into external-triggered vulnerabilities and inherent vulnerabilities, based on the conditions that trigger them and their root causes.

- **External-Triggered Vulnerabilities**: These rely on external interactions (such as user input or network requests) to be activated, with the core issue being insufficient protection against untrusted external factors. For example, generated code may have SQL injection vulnerabilities when user input is directly concatenated into database queries without filtering; or it may transmit sensitive data (such as user IDs) via HTTP without transport layer security, risking eavesdropping [10].

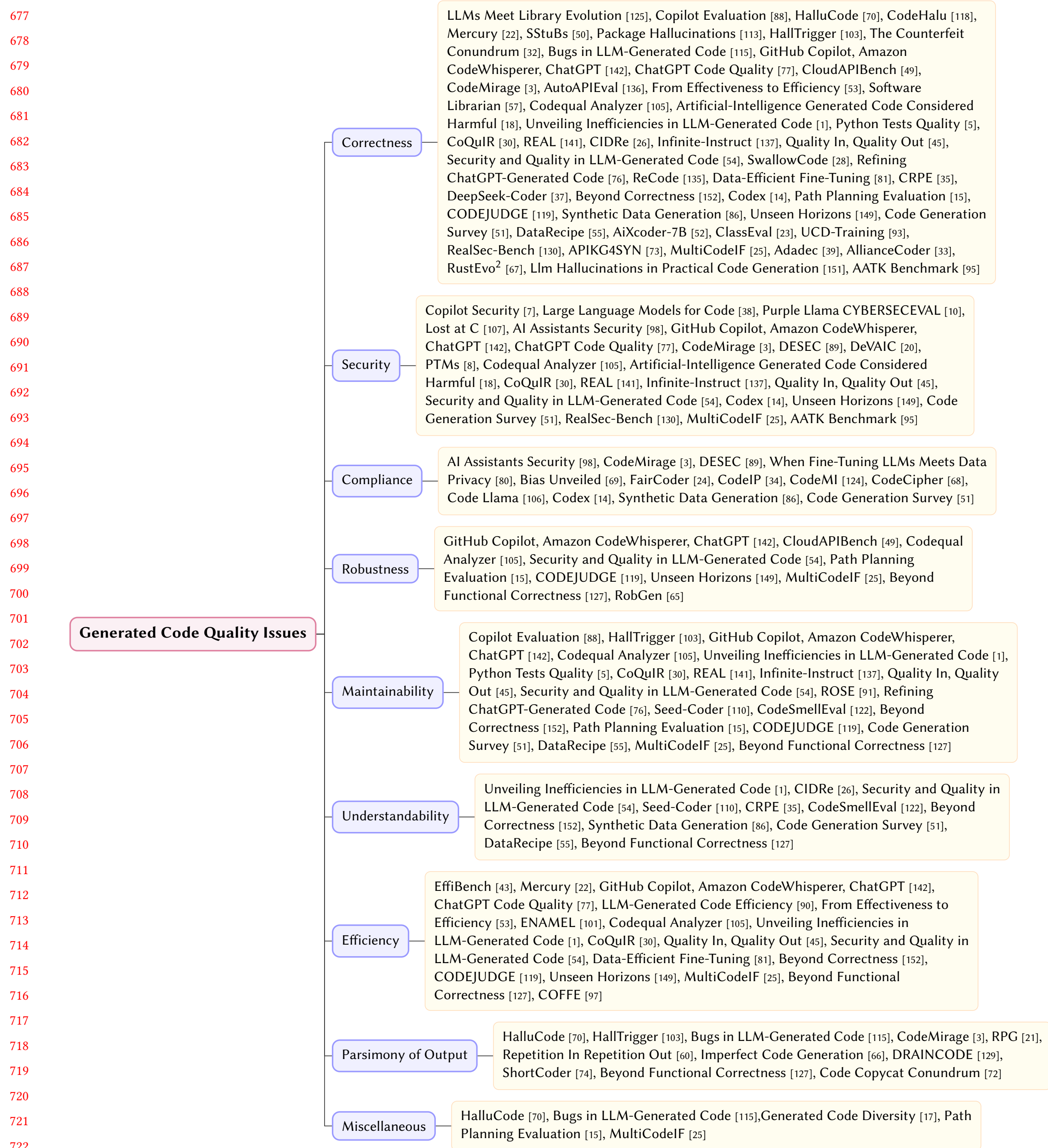

Fig. 8. Taxonomy of generated code quality issues with corresponding literature references.

- **Inherent Vulnerabilities**: These are intrinsic design or implementation flaws that pose risks without requiring external triggers. A typical example is improper encryption usage, such as hardcoding encryption keys in source code, which leads to sensitive data leakage if the code is exposed [18]. Additionally, poor memory management and unsafe API calls are simple instances of inherent vulnerabilities, both of which introduce hidden security risks by design.

Compliance. Violations of legal, ethical, or regulatory constraints. Compliance issues typically involve generating harmful, biased, or illegal content, which may lead to legal lawsuits, ethical disputes, or privacy breaches. These issues can be further categorized into specific types:

- **Malicious Toxic Code**: A typical compliance issue, distinct from security concerns, because it is specifically designed to harm others' rights and disrupt social order. Unlike security issues, which arise from vulnerabilities that are exploited externally, malicious toxic code is intended to intentionally cause damage. For example, generated code may intentionally embed backdoors, ransomware, or other malicious software, which not only disrupt the normal functioning of the system but also violate user privacy and property rights, thereby breaching relevant laws and social norms [14].
- **Privacy Issues**: These involve the accidental leakage of personal or sensitive information through generated code. This can occur when the code processes data in ways that do not comply with privacy protection regulations, such as GDPR. For example, the generated code might unintentionally expose users' personal details or fail to anonymize data properly, thus violating privacy laws [80].
- **Copyright Issues**: Occur when generated code unintentionally copies code snippets protected by copyright, violating intellectual property laws. This may happen when the model references or generates parts of code that are under copyright protection without proper attribution or permission, leading to potential legal disputes over intellectual property [34].

These compliance issues are not just technical challenges but also involve wide-ranging legal and social responsibilities, requiring strict regulation during the code generation process.

Robustness. The inability to handle abnormal inputs, boundary conditions, or runtime disruptions. These issues often manifest as a lack of necessary fault tolerance and validation design in the code. For example, generated array operation code may fail to check whether the accessed index is within the array's bounds, causing an out-of-bounds error and crashing the program when an index value beyond the range is provided. Similarly, when receiving numeric data such as order amounts or ages from user input, the code may not verify whether the input is a valid number, resulting in a type conversion exception if the user accidentally enters characters or special symbols, which then interrupts the business flow [119].

In our definition, there is a core distinction between robustness and security. Robustness focuses on the system's stability in the face of abnormal inputs, environmental fluctuations, or non-malicious disruptions, with the main goal of ensuring the program does not crash and can tolerate errors. In contrast, security is more concerned with defending against external malicious attacks (such as injection attacks or data theft), with the core objective of protecting system data and core functions from unauthorized use or damage [38]. The two concepts focus on different dimensions: "system fault tolerance" for robustness and "defense against external risks" for security.

Maintainability. Difficulty in modifying or extending code without introducing risks. This issue typically arises when code lacks clear structure or is overly complex, making it harder to understand, modify, or extend. Common

deficiencies include large, monolithic code units that handle too many responsibilities, deeply nested logic that complicates readability, and excessive coupling between components that reduces flexibility [91]. Additionally, code may exhibit data clumping, where multiple unrelated values are grouped together, making it difficult to adapt the system to new requirements [76].

Weak modularization and a lack of reuse also hinder maintainability, as small changes may require significant alterations across the codebase [110]. Notably, maintainability also encompasses code reusability, which ensures that components or functions can be reused across different parts of the system without modification. Reusable components not only reduce redundancy but also make it easier to maintain and extend the system over time, as updates to one module can be reflected across all instances where it is used.

Understandability. The ability for humans to easily comprehend the logic, intent, and structure of code. Issues often arise from poor or inconsistent naming conventions, where variable and function names fail to clearly reflect their purpose, making the code harder to follow and prone to misinterpretation [86]. Additionally, missing or misleading comments further obscure the intent of the code, leaving developers to decipher the rationale behind certain decisions or implementations. The use of "magic numbers", hardcoded constants without explanation, also contributes to confusion, as these values lack context and make the code less intuitive [152].

It is important to distinguish between understandability and maintainability, as while clear and understandable code certainly aids in maintenance, maintainability itself focuses more on the code's structure, modularity, and reusability. Understandability, on the other hand, centers on making the code logical and transparent, ensuring that anyone can quickly grasp its functionality and intent without requiring deep expertise or context.

Efficiency. The unnecessary consumption of computational resources in terms of time or space. These issues typically arise from inefficient computational logic design or improper resource allocation [43].

- **Execution Time Efficiency**: Generated code may use suboptimal algorithms, such as employing bubble sort with a time complexity of $O(n^2)$ to process large datasets in image processing or machine learning model training, instead of using the more efficient quicksort with a time complexity of $O(n \log n)$. This leads to a significant increase in runtime as the data size grows, severely impacting performance [90].
- **Memory Usage Efficiency**: There may be memory wastage due to inefficient handling of memory. For instance, in big data processing or real-time systems, the code might repeatedly create temporary objects, such as repeatedly copying large data blocks in image processing, rather than using caching or buffering. This results in frequent memory allocations and deallocations, increasing garbage collection overhead and reducing system responsiveness [101].

Parsimony of Output. Inefficiencies introduced during generation that degrade the structural quality of the produced artifact. These issues often manifest as overly long, redundant, or repetitive code blocks, unnecessary comments, or verbose scaffolding, significantly increasing the post-processing workload [21]. Even if the output is not truncated, the generated code may still contain large amounts of redundancy and repetition, such as repeatedly generated if-else statements or multiple redundant loop structures, which make the code bloated and unusable, severely reducing its usability. Unlike typical efficiency issues related to time and space complexity, these generation-level inefficiencies arise from the model producing unnecessary content, wasting computational resources, and making the code harder to read, modify, and maintain.

Miscellaneous. Quality issues that don't fit into the previous eight dimensions, primarily related to the alignment between generated code and external constraints. A key example is poor instruction-following, where LLMs generate code that meets basic syntax and functionality but deviates from specific user instructions [15]. Unlike correctness issues, which are typically related to logic or syntax, instruction-following problems often stem from the model's misunderstanding of task details or its "hallucination" of information not in the instructions. These issues highlight the difficulty of aligning LLM outputs with precise user expectations, which can affect the practical usability of the code.

4.1.2 Taxonomies and Classifications. Existing research frequently proposes fragmented classification systems that focus on isolated quality dimensions. For instance, some classifications concentrate strictly on correctness by categorizing syntax errors and logical flaws [115], while others focus exclusively on vulnerability detection, cataloging injection attacks and cryptographic misuse [20]. A few studies also address code style issues, such as inconsistent indentation patterns [122]. However, these approaches fail to cover the full spectrum of generated code quality issues. They often suffer from overlapping problem definitions, such as conflating robustness with security [8], and critically lack a unified framework to integrate diverse quality concerns.

In contrast, the comprehensive, nine-dimensional classification system proposed in this paper systematically resolves these gaps. Our framework integrates functional attributes (correctness), non-functional attributes (efficiency, maintainability, understandability, robustness), risk-related attributes (security, compliance), and generation-specific attributes (parsimony of output). Crucially, it emphasizes clear demarcation between dimensions to avoid unnecessary coupling. For example, we explicitly distinguish between security and robustness: security focuses on defending against malicious attacks and inherent vulnerabilities, whereas robustness concerns the handling of non-malicious anomalies. This critical distinction is rarely clarified in existing classifications, where the two are routinely conflated.

Importantly, we intentionally exclude certain vaguely defined dimensions prevalent in previous literature, such as generation "hallucination" [70], "code smells" [5], and "style issues" [152]. Because these concepts lack definitional accuracy, typicality, and clarity, they cannot serve as independent quality dimensions. Instead, we systematically map the valid anomalies underlying these concepts into our established nine dimensions. For instance, manifestations of "generation hallucinations", such as incorrect API or dependency usage, are explicitly categorized under the correctness dimension. Similarly, structural "code smells" like massive code units, deep nesting, and excessive coupling are mapped to maintainability, while inconsistent naming and missing comments are classified under understandability. Style issues are subsequently integrated into either understandability or maintainability, depending on whether they hinder human comprehension or complicate future modifications.

Representative cases across primary studies further validate the stability of our framework. Beyond mapping injection flaws to security, copyright violations to compliance, and redundant comments to the parsimony of output dimension, our framework consistently absorbs previously scattered problems. For instance, "data aggregation code smells" seamlessly integrate into maintainability, while "hallucinatory dependency calls" are strictly treated as correctness issues. This cross-study consistency verifies the comprehensiveness of our taxonomy in encompassing all key quality issues of LLM-generated code. Ultimately, it demonstrates the framework's applicability for systematic analysis, effectively resolving the fragmentation and ambiguity prevalent in existing classification systems.

> ***Summary and Insights.***

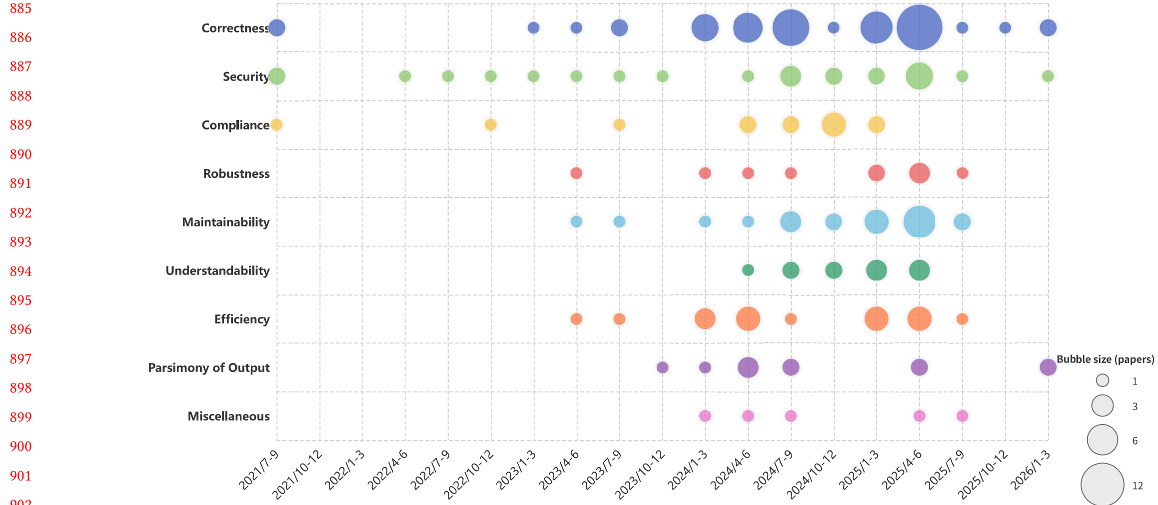

Fig. 9. Temporal distribution of studies across nine quality dimensions (bubble size denotes the number of studies in the corresponding time period).

Research on generated code quality has expanded rapidly since 2023, with explosive growth in 2024–2025, reflecting the accelerating development and industrial adoption of LLMs for code generation (Figure 9). Early studies primarily focused on functional correctness, which remains the most frequently investigated dimension (52 studies) and continues to dominate the literature. However, the research scope has gradually broadened toward a more comprehensive view of code quality.

Security (24 studies) and maintainability (22 studies) have emerged as major concerns, reflecting growing awareness of the risks and long-term maintenance costs associated with deploying LLM-generated code in real-world systems. Meanwhile, efficiency (19 studies) and compliance (13 studies) have attracted increasing attention since 2024 due to practical deployment constraints. In contrast, understandability (11 studies), parsimony of output (11 studies), and robustness (10 studies) remain comparatively underexplored, largely because they are more difficult to evaluate and less frequently prioritized in existing benchmarks. The miscellaneous dimension (5 studies), dominated by instruction-following issues, highlights emerging challenges specific to LLM-based code generation.

Overall, the literature reveals a clear evolution in research priorities: from an initial focus on functional correctness toward a multi-dimensional understanding of generated code quality. This shift suggests that the evaluation of LLM-generated code is gradually aligning with broader software quality principles, where functional correctness represents only one aspect of overall software quality. The observed diversity of issues also confirms that generated code quality issues are systematic and structurally classifiable, motivating the unified nine-dimension taxonomy proposed in this study. Future research should further expand evaluation beyond correctness-oriented benchmarks to better capture the full spectrum of quality requirements in practical software engineering.

4.2 RQ2: What training data quality issues exist in datasets used for LLM training, and how can they be categorized?

Next, we examine quality issues at the dataset level, which fundamentally shape model behavior and downstream code generation. As established in Section 2.3, training data quality issues are characterized as intrinsic flaws within pre-training and fine-tuning corpora. To provide a precise analysis, our investigation structurally divides these issues into two core dimensions: *code attributes* (defects localized within individual code samples) and *non-code attributes* (which encompass both textual noise and macro-level properties such as distribution, redundancy, and diversity). Specific classifications and supporting references are detailed in Figure 10.

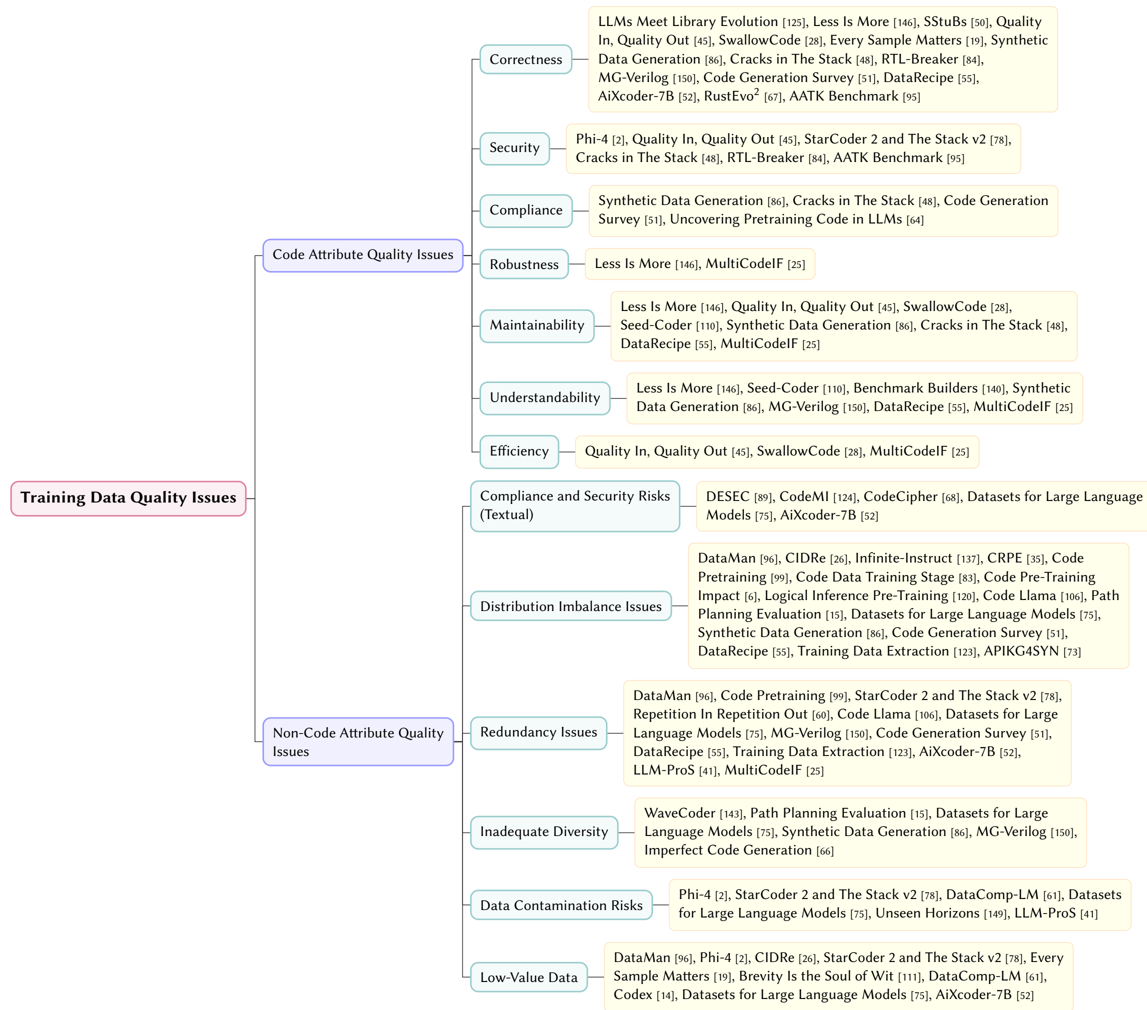

Fig. 10. Taxonomy of training data quality issues with corresponding literature references.

4.2.1 Code Attribute Quality Issues. This category investigates inherent defects within individual code samples in the training dataset. These sample-level flaws propagate through the learning process and closely mirror the generated code quality dimensions defined in RQ1. Notably, we exclude generation-specific dimensions, such as Parsimony of Output, because verbosity or redundancy are structural artifacts of the model's generation process rather than intrinsic properties of the training data. The specific sub-dimensions are detailed as follows:

Correctness. Correctness is the most fundamental quality issue in dataset code, manifested as syntax errors, logical flaws, or the failure to achieve the intended functionality. A typical issue is API misuse, where outdated API calls, incorrect parameter usage, or similar error patterns frequently present in the dataset are learned by the model as valid programming practices. This directly leads to generation failures or abnormal functionality when the generated code is used in real environments [89]. Syntax errors and logical inconsistencies similarly interfere with the model's understanding of correct programming patterns, reducing the reliability of the generated code [28].

Security. Code samples in the dataset often contain inherent security vulnerabilities or high-risk programming patterns, which serve as negative security examples for the model. For instance, SQL queries that do not filter user inputs expose the model to injection attack risks, while samples containing hardcoded keys or access tokens may guide the model to expose sensitive information in generated code. These flaws directly lower the security threshold of the generated code, making it vulnerable to external attacks or potential data breachess [45].

Compliance. Compliance issues in the dataset refer to code samples that violate legal, ethical, or industry regulatory requirements, rather than simple programming style problems. For example, unauthorized copying of copyrighted code snippets may lead the model to generate infringing content [124], while samples with discriminatory logic could guide the model to produce code that involves gender, racial, or age discriminations [92]. These defects not only expose the generated code to legal risks but may also spark serious ethical controversie.

Robustness. Code samples in the dataset often lack necessary fault tolerance, making them unable to handle abnormal inputs, boundary conditions, or runtime fluctuations. A large number of samples that do not include boundary checks or exception handling will lead the model to overlook the importance of error management and learn fragile programming patterns [146]. This results in generated code that is prone to crashes and instability when faced with non-standard inputs or environmental changes, making it unable to maintain stable operation.

Maintainability. Code samples in the dataset often exhibit disorganized structures and poor extensibility, making them difficult to modify or reuse. Samples with excessive coupling between modules and a lack of necessary documentation and comments will teach the model bad coding practices, leading to the generation of code that is overly complex and logically tangled. This type of code will require significant modification effort in future iterations, severely impacting development efficiency [55].

Understandability. Code samples in the dataset that exhibit unclear logic or obscure expressions prevent the model from learning clear programming patterns. Samples with poor naming conventions or hardcoded constants without explanation will interfere with the model's understanding of code semantics, making it difficult to grasp the principle of "semantic consistency in expression". This directly leads to generated code with poor readability, increasing the difficulty for developers to understand and collaborate [110].

Efficiency. Code samples in the dataset that waste resources or contain suboptimal algorithm designs lead the model to learn inefficient programming logic. Samples using algorithms with high time complexity for large datasets will cause the model to overlook performance optimization during code generation. Similarly, samples with poor memory management will lead to the generation of resource-intensive code, causing the final output to run slowly or consume excessive resources in real deployment [45].

4.2.2 Non-Code Attribute Quality Issues. This dimension covers all quality issues unrelated to the quality of individual code data points, focusing on defects in non-code textual data itself and macro-level attribute flaws of datasets. The specific subcategories are as follows:

Compliance and Security Risks (Textual). Unlike compliance issues in code attributes, this type of risk focuses on compliance and security hazards inherent in textual data itself, rather than risks at the code execution level.

- **Illegal and Harmful Text**: Includes content involving violence, pornography, distorted values, or speech inciting hatred or discrimination against specific groups (gender, race, age). Such text may guide the model to generate biased or harmful code and related content [75].
- **Copyright-Infringing Text**: Copyright-protected textual data used without authorization, such as pirated technical documents, excerpts from paid tutorials, or proprietary specifications. This may lead to copyright disputes in the comments and documents associated with the generated code [124].
- **Privacy-Leaking Text**: Text containing personal sensitive information (ID numbers, phone numbers, emails, or bank account information). Such data increases the risk of privacy leakage during code generation and may violate data protection regulations [52].

Distribution Imbalance Issues. Distribution imbalance causes the model to develop learning biases, making it difficult to adapt to various programming needs in a balanced manner.

- **Multilingual Support Imbalance**: The coverage of programming and natural languages is highly uneven. Mainstream languages (Java, Python) dominate, while niche programming and non-English languages are underrepresented, resulting in poor performance in low-resource scenarios [137].
- **Domain and Resource Attention Imbalance**: Textual information about popular technical domains and common APIs is abundant, while niche domains and uncommon APIs are underrepresented. Task and scenario types are also skewed, with excessive data on general algorithm implementations but scarce data on system architecture design or specialized environment adaptation [89].
- **Data Type Proportion Imbalance**: The ratio of code data to non-code data is unreasonably configured. Excessive code data limits the model's natural language understanding, while excessive textual data weakens programming syntax learning, both harming generation performance [6].
- **Data Difficulty Distribution Imbalance**: The dataset's complexity is polarized, dominated by either simple syntax examples or high-complexity algorithms, and lacking a reasonable gradient from basic to advanced. This imbalance hinders the model's ability to perform consistently across difficulty levels, particularly in medium-difficulty business code generation [35].

Redundancy Issues. The core manifestation of redundancy is duplicate or nearly duplicate data samples, which cause the model to overfit repetitive patterns and lose generalization ability. A growing concern is *training data degradation*: many datasets now include synthetic data generated by models, which often exhibit content repetition and

low diversity [60]. Prolonged reliance on such data leads to rigid, uncreative knowledge representations, resulting in code that lacks innovation and flexibility [55].

Inadequate Diversity. Datasets fail to sufficiently cover diverse industry scenarios, edge cases, and specialized domains. For example, business logic in finance, healthcare, or industrial control is underrepresented, as are programming needs in special operating environments (such as limited computing power) or edge-case exception handling. Consequently, models struggle to handle non-general or novel programming tasks effectively [143].

Data Contamination Risks. Inadequate filtering during data collection may include benchmark test data (such as HumanEval, MBPP) in training sets. The model may then memorize solutions rather than reason through tasks, inflating benchmark scores and misleading assessments of its true generation and reasoning capabilities [41].

Low-Value Data. Such data contributes little or even negatively impacts learning, introducing noise and inefficiency.

- **Meaningless Text**: Short texts containing only a URL, single words, or templated content (such as repetitive copyright notices, slogans) [61].
- **Format Noise**: Residual webpage elements like navigation bars or ads, or interface text accidentally extracted from GUI elements (such as "Login/Register" buttons) [75].
- **Low-Information-Density Text**: Spam, meaningless character sequences, invalid or malformed documents, or unparsed encoded fragments (such as Base64, hex). Texts where visible valid content is less than 20% of the file also fall into this category [78].
- **Erroneous Text**: Texts with logical fallacies, factual inaccuracies, or serious grammatical errors that mislead model understanding [2].
- **Incomplete Data**: Poorly documented API references lacking parameters, explanations, or structured comments. For instance, datasets like Stack, Vault, CodeSearchNet, and MCoNaLa often lack structured annotation design, limiting the model's ability to generate high-quality documentation or comments [26].

4.2.3 Taxonomies and Classifications. Our taxonomy of training data quality issues is designed to be **comprehensive** and **systematic**, covering both the micro-level quality of individual code/text samples and the macro-level properties of datasets. Compared to existing classification frameworks for training data quality issues, our approach offers several advantages.

First, it explicitly differentiates between *code attribute* and *non-code attribute* issues, which is crucial for establishing the mapping relationship with generated code quality issues (addressed in RQ3). By isolating code-specific defects, we enable a more precise analysis of how single-point code quality flaws propagate to generated outputs. Second, the non-code attribute category is granularly segmented into sub-dimensions like distribution, redundancy, and low-value data, which are often overlooked or generalized in other frameworks. This granularity allows for targeted identification and mitigation of textual and macro-level anomalies that subtly degrade model performance. Third, our taxonomy is tailored to the context of LLM code generation, incorporating dimensions such as multilingual support imbalance and code complexity distribution. These factors are uniquely impactful in shaping a model's ability to generate high-quality code across diverse scenarios. In summary, this taxonomy provides a holistic lens to diagnose training data quality issues, bridging the gap between data-level defects and their manifestations in LLM-generated code.

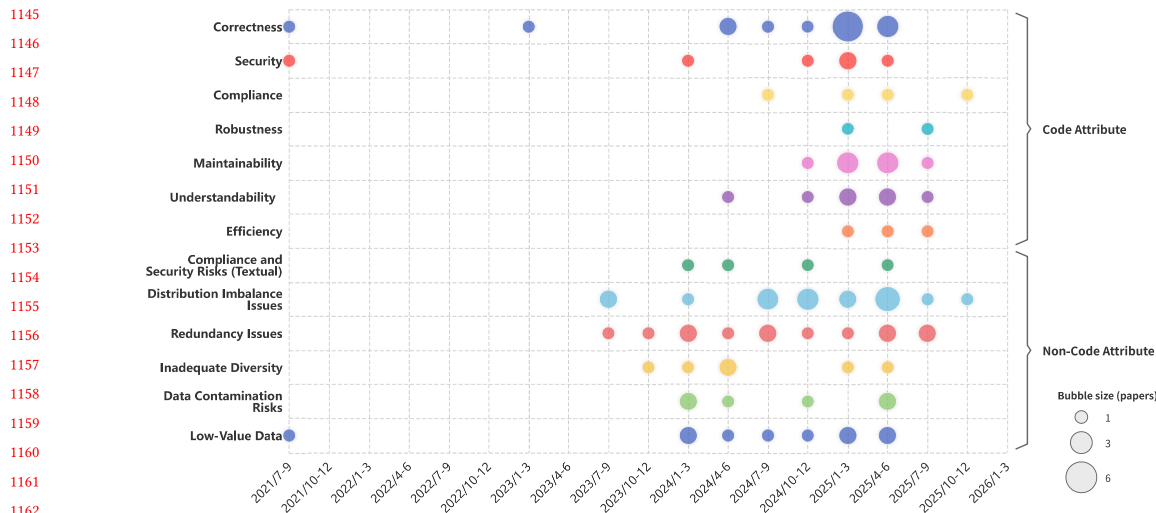

Fig. 11. Temporal distribution of studies across code and non-code attribute quality dimensions (bubble size denotes the number of studies in the corresponding time period).

Summary and Insights.

This section synthesizes findings from the analyzed studies. As illustrated in Figure 11, research on training data quality issues exhibits a clear temporal growth trend, consistent with the observations reported in Section 4.1.

Among code attribute issues, correctness (15 studies) remains the most dominant and persistent concern, reflecting its fundamental role in code-centric datasets. Maintainability (8 studies), understandability (7 studies), and security (6 studies) follow, while code compliance (4 studies), efficiency (3 studies), and robustness (2 studies) remain relatively underexplored, largely due to limited standardized benchmarks and high evaluation complexity.

For non-code attribute issues, distribution imbalance (17 studies), redundancy (13 studies), and low-value data (10 studies) emerge as the most prevalent concerns, showing increasing research attention in recent years. Inadequate diversity (6 studies) and data contamination risks (6 studies) are also gaining traction, while textual compliance and security risks (4 studies) remain comparatively underrepresented.

A key insight is that non-code attribute issues, although general in LLMs, have disproportionately amplified effects in code generation scenarios. Distribution imbalance introduces systematic learning bias and reduces generalization to underrepresented programming languages and tasks. Redundancy reinforces repetitive patterns and increases the risk of overfitting. Low-value data injects noise that degrades learning efficiency and output reliability. Meanwhile, inadequate diversity and data contamination further restrict the robustness and applicability of generated code in real-world settings.

Overall, our analysis reveals that training data quality issues are not independent artifacts but form a coupled and compounding system across both code and non-code dimensions. Mitigating one category often requires addressing others jointly, underscoring the need for holistic dataset curation strategies in code-oriented LLM development.

4.3 How do training data quality issues influence generated code quality issues?

This section establishes the linkage between training data quality issues and the observable generated code quality issues. To systematically trace the origins of generated code quality issues, we bridge the dimensions established in RQ1 and RQ2. Specifically, training data quality issues (RQ2) manifest into generated code quality issues (RQ1) through two primary propagation mechanisms: direct and indirect mappings.

Generally, direct mappings are closely associated with *Code Attribute Quality Issues* in the training data, where explicit code-level defects are memorized, generalized, or reproduced by LLMs. Conversely, indirect mappings typically stem from *Non-Code Attribute Quality Issues*, where dataset-level distributional anomalies or textual noise distort the learning distribution and indirectly induce generated code quality issues.

To provide a comprehensive overview of these causal and correlative relationships, Figure 12 illustrates the flow from training data quality issues (left) through their specific propagation mechanisms (middle) to the resulting generated code quality issues (right). Through iterative synthesis of the extracted literature and continuous collaborative discussions, we consolidated these relationships into 18 typical propagation mapping mechanisms (10 direct mappings and 8 indirect mappings), corresponding to the distinct flow paths in the middle layer of the Sankey diagram. The following subsections elaborate on each of these 18 mechanisms in detail.

4.3.1 Direct Mappings.

Direct mappings describe causal relationships where code attribute defects embedded in the training corpus are learned and directly reproduced by LLMs.

Deprecated or Obsolete APIs → Compatibility and Execution Failures. One recurring mapping involves outdated or deprecated APIs within large-scale datasets such as The Stack or CodeSearchNet [125]. These corpora often include legacy repositories using API versions no longer compatible with current dependencies. When models trained on such data generate code invoking obsolete interfaces, the resulting programs suffer from runtime or compilation errors. For instance, LLMs frequently generate Python code importing `tensorflow.compat.v1` or Java code referencing outdated `javax` packages [136]. This pattern exemplifies *memorization-driven replication*, where temporal drift in data leads to obsolete knowledge being reproduced in outputs.

Syntax and Structural Defects → Compilation Errors and Fragile Logic. Another direct mapping arises from syntactically or structurally invalid code snippets in the training corpus [28]. Public datasets often aggregate unverified code fragments from forums, incomplete answers, or partial code blocks, leading to missing brackets, undefined variables, or inconsistent indentation. These defects manifest in generation as syntax errors or semantically incomplete implementations. The mechanism is *pattern propagation*, where the model learns frequent yet invalid structural templates. Models trained on raw GitHub code, for instance, frequently generate Python functions lacking indentation alignment or missing return statements. Empirical analysis shows a strong correlation between syntax error frequency in datasets and failure rates during execution-based evaluation [45].

Low-level Bug Patterns (SStuBs) → Replication of Common Logic Bugs. Small, shallow bugs (SStuBs) represent another class of training data quality issues that directly propagate to generated code [50]. These micro-bugs are prevalent in open-source corpora and are easily memorized by autoregressive models. Empirical evidence shows that LLMs trained on unfiltered GitHub data tend to reproduce frequent bug patterns (using <= instead of <, or neglecting to close I/O streams) [50]. The underlying mechanism combines memorization and pattern reinforcement: repeated

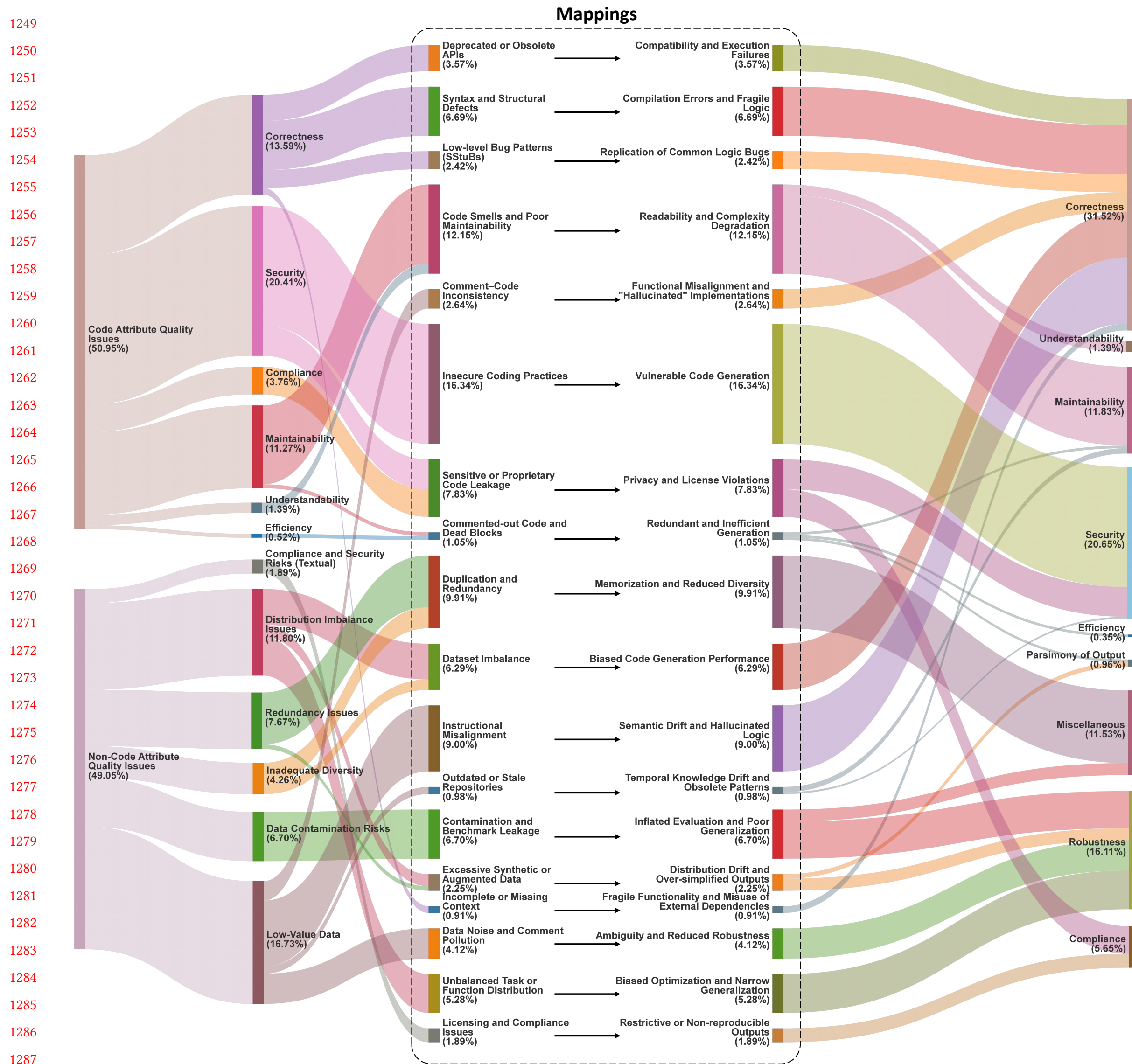

Fig. 12. A Sankey diagram illustrating the mappings from training data quality issues (left) to generated code quality issues (right). The intermediate layer details the specific propagation mechanisms.

exposure to buggy samples increases their likelihood in the model's learned distribution. Static and dynamic analyses of generated outputs often reveal bug frequency distributions similar to those found in the training data [55].

Code Smells and Poor Maintainability → Readability and Complexity Degradation. Datasets containing large proportions of code with poor structural design lead to models producing equally unmaintainable code [122]. Such defects are typical in repositories never subjected to peer review or refactoring. Generated outputs exhibit excessive verbosity, repetition, or low cohesion, reflecting the stylistic distribution of the source corpus. This mapping is mediated

by *pattern propagation and representation bias*: LLMs internalize both the logic and stylistic tendencies of their training samples. A corpus overrepresented with imperative, monolithic Java files biases the model toward verbose procedural structures, diminishing code readability and maintainability [76].

Comment–Code Inconsistency → Functional Misalignment and "Hallucinated" Implementations. In some datasets, comments and accompanying code blocks are semantically misaligned—the comment describes a sorting algorithm while the code implements searching. This mismatch introduces noisy supervision that confuses instruction-following models, especially during fine-tuning. Consequently, LLMs tend to generate functions fulfilling the comment text syntactically but not semantically. The mapping mechanism is *semantic misalignment*: the model associates incorrect natural language–code pairs, leading to hallucinated or intent-divergent outputs. Evidence shows that instruction-tuned models trained on misaligned pairs (Docstring–Code datasets) produce higher semantic divergence scores under code-to-text consistency evaluation [55].

Insecure Coding Practices → Vulnerable Code Generation. When training data includes insecure idioms such as hard-coded credentials, unsafe deserialization, or outdated cryptographic functions, the model replicates these practices, resulting in code that is functionally correct but inherently insecure [48]. Models trained on public Python repositories often generate code containing `eval()` or unvalidated user input, reflecting unsafe patterns common in the data. This direct mapping operates through *memorization of vulnerable code snippets*, propagating inherent vulnerabilities from the dataset directly into the generated artifacts.

Sensitive or Proprietary Code Leakage → Privacy and License Violations. A further direct mapping pertains to training data containing proprietary or personal information, including API keys, user identifiers, or code under restrictive licenses [124]. When exposed to such data, models can memorize unique identifiers and later reproduce them verbatim during generation, resulting in privacy leaks or license infringements. The mechanism is classic *data leakage through memorization*. Empirical studies demonstrate that LLMs can output exact code fragments or credentials found in training repositories when prompted with similar contexts [68].

Commented-out Code and Dead Blocks → Redundant and Inefficient Generation. Many code corpora include commented-out sections, temporary debugging blocks, or incomplete prototypes [21]. These artifacts bias models toward generating redundant or non-functional code. The mechanism combines pattern propagation and entropy amplification: the model captures the structure of "half-finished" code as a valid pattern, inflating generation length and redundancy. Output examples include functions containing commented debugging code or unused variables.

Outdated or Stale Repositories → Temporal Knowledge Drift and Obsolete Patterns. A direct mapping linked to code age involves repositories containing entirely obsolete project structures or unsupported dependencies [75]. Even when syntactically valid, such codebase samples encode practices or language idioms that have been deprecated across the software ecosystem. Consequently, generated code tends to use outdated library versions or inefficient paradigms (e.g., generating Python 2 style print statements, or employing insecure hashing methods deprecated in modern standards). The propagation mechanism is *temporal drift memorization*, where stale code attributes are reinforced due to a lack of temporal weighting.

Licensing and Compliance Issues → Restrictive or Non-reproducible Outputs. While not affecting functional correctness directly, licensing constraints embedded within code attributes introduce practical compliance degradation [124]. Models unknowingly replicate copyrighted code fragments or entire files from restrictive sources. This

mapping represents an *ethical-legal leakage* problem at the code attribute level. Empirical evidence shows verbatim replication of GPL-covered code snippets by autoregressive models [34].

4.3.2 Indirect Mappings. Indirect mappings refer to cases where non-code attribute quality issues, such as dataset imbalance, textual noise, or data contamination, distort the model's learned distribution, representation bias, or alignment, thereby inducing generated code quality issues. These effects emerge via mechanisms like entropy collapse and semantic drift, which often manifest in aggregate behavioral changes rather than explicit syntactic errors.

Duplication and Redundancy → Memorization and Reduced Diversity. Large-scale code corpora are often dominated by duplicated or near-duplicated samples, constructed by indiscriminate crawling of GitHub or Stack Overflow [75]. Repeated code blocks, copy-pasted templates, and auto-generated boilerplate significantly reduce corpus entropy, causing models to overfit frequent patterns. The result is an overrepresentation of "safe" or memorized responses in generated code, with reduced creativity and exploration capability. Empirical studies show models trained on heavily duplicated subsets of The Stack tend to regenerate entire library functions verbatim or produce highly similar implementations across different prompts [78]. The mechanism here is *entropy collapse*, where repeated samples narrow the token probability distribution.

Dataset Imbalance → Biased Code Generation Performance. Another indirect mapping arises from severe data imbalance across programming languages, domains, and difficulty levels. Public corpora such as CodeParrot or The Stack contain a disproportionate amount of Python and Java code, while low-resource languages (Rust, Kotlin) are underrepresented [106]. Consequently, models exhibit marked performance disparities across languages and domains. This mapping operates through *representation bias*, where the model's internal representation space is dominated by high-frequency samples, skewing token prediction probabilities. Evidence shows multilingual LLMs achieve up to 40% higher pass@k scores in Python than in Go or C# [137]. Similarly, data imbalance between high-level API tasks and low-level algorithmic tasks results in uneven reasoning ability.

Instructional Misalignment → Semantic Drift and Hallucinated Logic. Beyond individual sample defects, entire datasets may suffer from *instructional misalignment* between natural language and code segments, particularly in instruction-tuning datasets. The Docstring–Code dataset and many instruction synthesis corpora contain mismatched function descriptions or misaligned summaries [55]. When such noise is learned, models internalize spurious correlations between intent and implementation, leading to hallucinations or logically incoherent outputs. A common failure mode is that the model generates code adhering syntactically to the prompt but failing to fulfill its semantics—for instance, implementing addition when subtraction is described. This phenomenon exemplifies *semantic misalignment propagation*.

Contamination and Benchmark Leakage → Inflated Evaluation and Poor Generalization. Dataset contamination causes models to overfit and artificially inflate evaluation scores [41]. Contaminated benchmarks (HumanEval, MBPP, CodeContests) have been found embedded verbatim in the pretraining corpora of several open-source models [149]. When such leakage occurs, models can reproduce solutions from memory rather than through generalization, leading to deceptively high performance. This mapping operates through the mechanism of *data leakage and memorization.*

Excessive Synthetic or Augmented Data → Distribution Drift and Over-simplified Outputs. Synthetic datasets generated via model self-sampling or rule-based expansion can amplify low-quality patterns and introduce distributional artifacts [86]. Overuse of synthetic code examples with repetitive or simplified structures leads models to generate

uniform, shallow code lacking abstraction and robustness. The underlying mechanism is *representation drift and entropy distortion*: synthetic data reinforces dominant patterns and suppresses the diversity of natural code. Empirical evidence shows excessive self-generated fine-tuning data increases token-level repetition and reduces functional correctness [60].

Incomplete or Missing Context → Fragile Functionality and Misuse of External Dependencies. Many code datasets extract isolated functions or snippets without their surrounding context (imports, dependencies, or usage scenarios) [26]. This leads to context fragmentation: during training, the model fails to capture necessary inter-file or module-level dependencies. Consequently, generated code may omit critical initialization steps, misuse imported libraries, or generate undefined references. The propagation mechanism is *contextual under-specification*, where incomplete training examples bias the model toward fragmentary understanding. Models trained on function-level datasets often generate code referencing undeclared variables or failing to import dependencies like `numpy`.

Data Noise and Comment Pollution → Ambiguity and Reduced Robustness. A substantial fraction of mined repositories contains low-information or noisy comments (auto-generated docstrings, advertisements, irrelevant annotations) [75]. Although seemingly benign, this "comment pollution" affects models' text–code co-training alignment, making natural language prompts less informative and increasing ambiguity in conditioning. The result is reduced robustness under prompt variation, as the model overfits to irrelevant lexical patterns. The mechanism is *signal-to-noise degradation*: excessive non-informative tokens dilute meaningful learning signals. Empirical findings indicate filtering out noisy comments can improve code generation pass@k by up to 8% [61].

Unbalanced Task or Function Distribution → Biased Optimization and Narrow Generalization. In large corpora, simple tasks (input/output manipulation, arithmetic) are vastly overrepresented compared to complex algorithmic or multi-file implementations [35]. This imbalance drives models to optimize toward shallow completion objectives, impairing their ability to generalize to compositional or reasoning-intensive tasks. The mechanism here is *optimization bias*: gradient updates dominated by easy samples lead to "local minima" in model performance. Evidence shows models fine-tuned on homogeneous or low-complexity data exhibit lower functional correctness on real-world software engineering benchmarks [15].

> ***Summary and Insights.***
>
> **While direct mappings represent explicit "garbage in, garbage out" replication, indirect mappings driven by non-code distributional flaws are equally prevalent but far more insidious.** Direct mappings, such as memorizing deprecated APIs or buggy snippets, are highly visible and relatively easy to detect. In contrast, generation failures are not solely caused by models learning explicitly flawed code. Overwhelming volumes of "safe" but unbalanced data, such as severe duplication, language imbalance, or skewed task complexity, induce deep representation and optimization biases. This lack of dataset entropy pulls gradient updates into suboptimal local minima, severely restricting the model's capability to generalize to complex software engineering tasks, demonstrating that severe distributional imbalance can be an even more profound driver of "garbage out" than explicitly dirty data.
>
> **Consequently, a critical trend in quality assurance is the paradigm shift from microscopic sample-level cleansing to macroscopic corpus-level statistical governance.** Mitigating systemic generated code quality issues now requires moving beyond simple vulnerability scanning or syntax linting. Ensuring semantic

alignment, balancing information entropy, and rigorously stratifying task complexity have become the primary frontiers for building robust and reliable code generation models.

4.4 RQ4: What techniques exist for detecting generated code quality issues and training data quality issues?

This section synthesizes detection techniques for quality issues at both the code generation and dataset levels of LLMs. Detection approaches have evolved from static and rule-based analysis toward dynamic, model-driven, and hybrid evaluation frameworks. These methods are classified into two categories: (1) **generated code quality issue detection**, which identifies functional, structural, and semantic defects in LLM-generated code; and (2) **training data quality issue detection**, which targets noise, duplication, imbalance, and contamination within training corpora. These approaches form the diagnostic foundation of LLM quality governance, providing empirical signals for subsequent mitigation. The following subsections synthesize representative detection methods across both domains.

4.4.1 Generated Code Quality Issue Detection. Generated code quality issue detection identifies functional, structural, and semantic defects in LLM-generated code, including syntax errors, runtime failures, inefficiency, hallucination, and security vulnerabilities. As illustrated in Figure 13, we classify these methods into three families: **dynamic analysis**, **static analysis**, and **model-based detection**. These families are often combined in hybrid workflows to improve coverage and accuracy.

Dynamic Analysis. Dynamic analysis assesses code correctness and performance through execution or runtime observation.

- **Test-based Execution Analysis.** This widely used paradigm validates outputs through compilation, unit tests, and functional benchmarks (e.g., *HumanEval*, *LeetCode*, *CoderEval*) [28, 88, 90, 115]. Automated test harnesses evaluate pass@k accuracy or runtime efficiency to measure correctness, maintainability, and reliability under varying inputs [28, 37, 55]. Frameworks such as *ENAMEL* and *EffiBench* execute code against curated test suites to jointly quantify functional accuracy and runtime cost [43, 101]. Execution-based frameworks like *CodeHalu* [118] and *Mercury* [22] detect hallucinations or inefficiency by comparing runtime behavior to expected semantics. Some studies integrate fuzzing or sandbox execution to detect rare faults and unstable behaviors [9, 18, 107, 108].
- **Runtime Monitoring.** Runtime monitoring continuously captures execution signals such as time, memory consumption, and failure traces. Studies monitor temporal performance variance across multiple generations to identify inefficiency or instability [53, 90]. Metrics like runtime percentile weighting (e.g., the Beyond metric in *Mercury*) couple correctness and efficiency [22]. These methods enable fine-grained analysis of resource utilization, providing feedback for runtime-aware model refinement.

Static Analysis. Static analysis inspects code without execution, leveraging syntactic and semantic rules to detect errors, vulnerabilities, or code smells.

- **Rule-based Detection.** Rule-based static analysis is a primary method for code quality assessment. Tool-based detection employs established analyzers (*SonarQube*, *CodeQL*, *Semgrep*) to identify CWE-related vulnerabilities, code smells, and maintainability defects in LLM-generated code. Studies like *DeVAIC* [20] and *CyberSecEval* [10] extend these analyzers with domain-specific patterns to identify flaws such as SQL injection or buffer overflow. Custom rule-based systems design syntax or pattern constraints tailored to model artifacts. Studies detecting

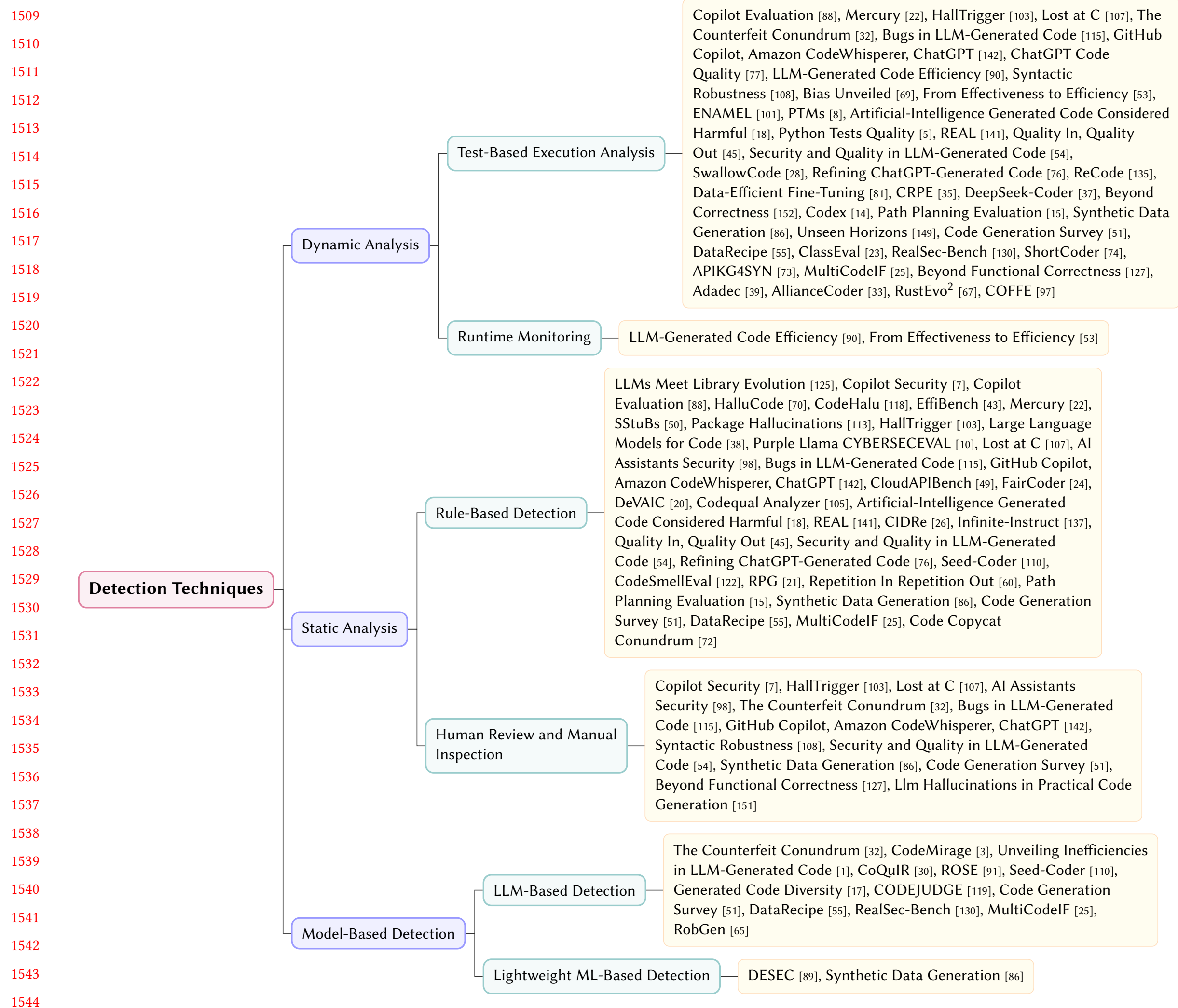

Fig. 13. Taxonomy of generated code quality issue detection techniques with corresponding literature references.

deprecated API calls [125] or duplicated structures [21, 60] define matching rules based on fully qualified names, type hierarchies, or AST similarity. Extended approaches such as *CIDRe* or *Infinite-Instruct* integrate multi-aspect criteria to score code by relevance, completeness, and style adherence [26, 137].

- **Human Review and Manual Inspection.** Human inspection remains necessary for validating high-impact findings or labeling benchmark datasets. Manual review is frequently coupled with automated scans to validate code smells, verify hallucinations, or refine static analysis outputs [115]. Examples include manual validation in vulnerability studies [7] and hallucination detection benchmarks like *CodeMirage* [3]. Human-in-the-loop inspection serves as ground truth and a corrective mechanism for ambiguous automated results.

Model-based Detection. Model-based detection utilizes either LLMs or lightweight machine learning classifiers to identify quality issues via semantic representations.

- **LLM-based Detection.** LLMs act as evaluators of generated code quality in three forms: (1) *direct LLM evaluation*, where an LLM assesses outputs without prompt engineering (e.g., using GPT-4o to classify inefficiency types [1]); (2) *prompt-engineered evaluation*, which augments LLMs with structured instructions or exemplars for code critique, hallucination detection, or fairness auditing [32, 51]; and (3) *fine-tuned evaluation*, where LLMs or code-specific transformers (e.g., CodeBERT, CodeT5) are fine-tuned for tasks like architectural smell detection or comment quality scoring [3, 91]. Frameworks like *HalluCode* [70] and *CodeSmellEval* [122] illustrate this direction.
- **Lightweight ML-based Detection.** Lightweight classifiers or feature-based models pre-screen large volumes of generated code. These models operate on static features (e.g., cyclomatic complexity, AST depth) to achieve scalable filtering. For example, the *codequal_analyzer* [105] and ML-based scoring modules in *DataMan* [96] demonstrate that compact models achieve high consistency with expert judgment. They are frequently combined with LLM evaluators to balance interpretability and computational cost.

Dynamic execution remains the primary measure of correctness, while static and model-based techniques provide structural and semantic coverage. The synergy among these paradigms establishes holistic quality assessment pipelines.

4.4.2 Training Data Quality Issue Detection. Training data quality issue detection focuses on identifying noise, duplication, contamination, imbalance, and other data defects that undermine the training and fine-tuning of LLMs for code generation. Unlike code-level detection, which evaluates generated outputs, training data quality issue detection targets the integrity, provenance, and representativeness of the underlying data sources. As illustrated in Figure 14, the reviewed literature reveals three dominant categories of training data quality issue detection methods: **dynamic analysis**, **static analysis**, and **model-based detection**.

Dynamic Analysis. Dynamic analysis involves executing code samples or monitoring model performance metrics to identify erroneous or low-quality data.

- **Execution-based Validation.** This approach evaluates whether code snippets can compile and execute successfully, leveraging automated testing frameworks. Studies such as *CodeSmellEval* [122] implement large-scale compilation and test pipelines to detect syntax errors, missing dependencies, and incomplete functions. Execution-driven filtering frameworks use unit tests and runtime logs to measure executability rates across datasets (*CodeNet*, *CodeParrot*, *BigCode*) [22, 30, 43, 118, 136]. These analyses report that 20–40% of raw collected code is non-executable or semantically inconsistent. Execution feedback also reveals higher-order issues such as inconsistent input/output formats or incorrect test oracle mappings [118]. For instance, execution traces may expose duplicated samples disguised under different identifiers or data contaminated with pre-solved tasks from evaluation benchmarks [82]. Integrating this feedback into filtering pipelines ensures model training focuses on executable code.
- **Monitoring of Performance or Metric Drift.** Monitoring model behavior provides an indirect signal of data degradation. Researchers track the evolution of loss, accuracy, or diversity metrics during pretraining to detect distribution shifts or contamination [60]. A sudden improvement on benchmark-like tasks during training can signal data leakage, whereas rising validation loss indicates noise accumulation or outdated content [60]. Studies

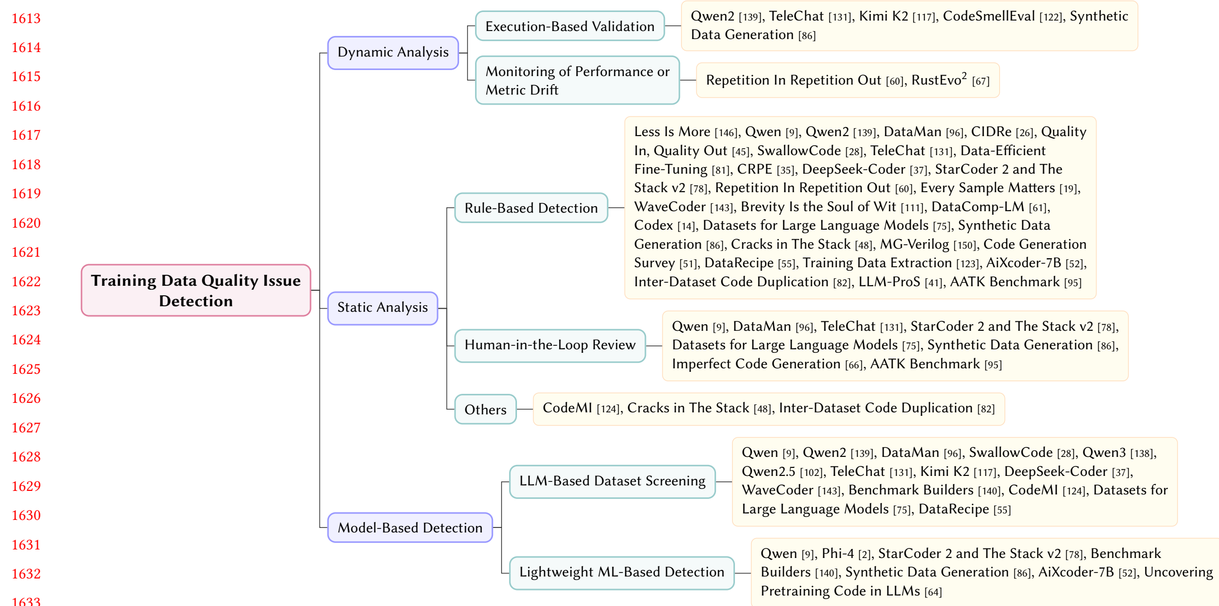

Fig. 14. Taxonomy of training data quality issue detection techniques with corresponding literature references.

like *Cracks in The Stack* and *Inter-Dataset Code Duplication* [48, 82] employ continuous metric monitoring to flag suspect subsets, enabling intervention before training amplifies data biases.

Static Analysis. Static analysis performs structural and semantic checks on datasets without executing code, making it suitable for large-scale repositories.

- **Rule-based Detection.** Rule-based detection employs syntactic, lexical, or metadata rules to identify noisy or redundant samples. *Tool-based static checking* adapts established analyzers (e.g., *PyLint*, *Bandit*, *Semgrep*) to dataset curation workflows [10, 28] to detect unsafe imports, incomplete function definitions, or insecure patterns. *Custom or domain-specific rule definition* enforces dataset-specific heuristics, such as verifying function headers, dependency declarations, or project-level file integrity [86]. Frameworks like *CodeAudit* and *DataLint* formalize rules that quantify syntactic defects, unreachable statements, and deprecated APIs in training corpora.
- **Human-in-the-loop Review.** Human inspection resolves ambiguous cases where automated rules fail. Studies such as *TeleChat* [119] combine crowd-sourced or expert review to verify code functionality, identify label errors, or assess documentation consistency. Reviewers also validate automated flags for high-risk categories (security vulnerabilities or license violations) to ensure compliance [75].
- **Others.** This category includes other detection methods such as data provenance tracking, model feature observation, and privacy-oriented auditing. *Provenance-based detection* analyzes the origin and traceability of code samples. By linking file hashes, repository metadata, or commit histories, these methods identify duplicated or misattributed content [48]. The *World of Code (WoC)* infrastructure enables SHA-1-based traceability to locate the first appearance of code blobs, revealing dataset contamination against benchmark repositories [48]. Provenance tracing also identifies stale code from abandoned projects. *Feature observation* detects anomalies by observing model performance under frozen-layer configurations. A growing performance gap as layers are

frozen suggests memorization or data duplication [82]. *Layer-wise freezing analysis* infers redundancy levels by examining performance degradation patterns under frozen configurations, providing empirical signals of representation bias [82]. *Code inference attacks* detect privacy and compliance violations within training corpora. These attacks infer whether specific sensitive or copyrighted code snippets exist in the training set. For instance, *CodeMI* [124] uses membership inference to check if a model has memorized proprietary code, serving as a tool for detecting data contamination and copyright infringement.

Model-based Detection. Model-based detection employs machine learning or LLMs to identify contaminated data via learned representations.

- **LLM-based Dataset Screening.** These methods prompt an LLM to assess whether a sample is coherent, complete, or redundant, often guided by few-shot exemplars [9, 28, 37]. *Qwen3* and *Kimi K2* utilize LLMs to score samples based on readability, correctness, and style adherence. Other studies integrate LLM judgment into automated pipelines to iteratively refine or exclude low-quality samples, demonstrating potential for semantic validation.
- **Lightweight ML-based Detection.** Lightweight machine learning models (e.g., random forests, small neural classifiers) filter data at scale [2, 52, 78]. These classifiers operate on features such as token entropy, AST metrics, or length-normalized duplication ratios to detect outliers or noise. Systems like *DataMan* [96] integrate such classifiers into pretraining pipelines for large-scale filtering with minimal computational overhead.

Dynamic approaches validate code executability, static rules ensure structural compliance, and model-based detection introduces scalability.

> ***Summary and Insights.***
>
> **Quality detection in LLM-based code generation has fundamentally transitioned from isolated, deterministic rule-checking to hybrid, semantic-aware diagnostic pipelines.** While traditional static analysis and dynamic execution remain the bedrock for verifying structural compliance and functional correctness, the field is increasingly integrating model-driven evaluators to overcome the limitations of rigid heuristics. The emergence of "LLM-as-a-judge" paradigms and lightweight machine learning classifiers enables scalable semantic reasoning, bridging the critical gap between syntactic constraints and nuanced intent alignment. This synergistic approach allows researchers to evaluate complex, non-functional dimensions that single-paradigm methods frequently fail to capture.
>
> **Furthermore, the diagnostic landscape exhibits a critical shift toward data–model co-assessment, establishing bidirectional traceability between training corpora and generated artifacts.** Rather than treating evaluation exclusively as a post-generation filter, contemporary methodologies proactively target dataset integrity through provenance tracking, execution-based pre-validation, and metric drift monitoring. This dual-focus assessment architecture ensures that structural anomalies, benchmark contamination, and representation biases are identified before they are memorized and amplified during the training phase. Ultimately, these integrated assessment ecosystems move beyond reactive error detection, providing the foundational empirical signals necessary for continuous and proactive quality mitigation across the entire model lifecycle.

4.5 RQ5: What mitigation strategies have been proposed to address these quality issues?

This section synthesizes mitigation strategies aimed at improving both the quality of LLM-generated code and the integrity of training data. The literature reveals two complementary streams of efforts: (1) **generated code quality issue mitigation**, which focuses on post-generation refinement, repair, and alignment techniques; and (2) **training data quality issue mitigation**, which addresses data curation, de-duplication, decontamination, rebalancing, and filtering during the data lifecycle. These strategies form a multi-layered governance framework spanning pre-training, fine-tuning, and inference-time interventions.

4.5.1 Generated Code Quality Issue Mitigation. Generated code quality issue mitigation refers to techniques that actively improve the quality, reliability, and maintainability of code generated by LLMs. In contrast to detection methods, mitigation strategies intervene at different stages of the generation and deployment pipeline—at the **data level**, the **model level**, or the **generation level** (Figure 15). The reviewed studies reveal that such mitigation efforts have evolved from manual rule-based corrections to fully automated self-improvement mechanisms driven by reinforcement learning and reflective generation.

Data-level Mitigation. Data-level interventions that aim to enhance code quality by improving training or fine-tuning data are discussed under the dataset governance framework (see Section 4.5.2). These include data cleaning, filtering, and balancing procedures that indirectly reduce generated code quality issues by providing higher-quality supervision signals.

Model-level Mitigation. Model-level mitigation targets the learning behavior and internal optimization of LLMs to produce higher-quality, less erroneous code. Three major approaches dominate the literature: fine-tuning and instruction alignment, reward-based optimization, and regularization-based stabilization.

- **Fine-tuning and Instruction Alignment.** Fine-tuning allows models to adapt to high-quality, task-specific data, thereby reducing hallucinations, style deviations, and incorrect function use. Studies such as *Package Hallucinations* [113] demonstrate that supervised fine-tuning (SFT) on curated coding corpora significantly mitigates hallucination phenomena, effectively curbing the generation of non-existent or mismatched code dependencies. Instruction alignment further refines this process through reinforcement learning with human feedback (RLHF) or preference optimization (DPO), enabling models to follow natural language specifications with higher fidelity [35, 135, 141]. Some works incorporate domain-specific alignment datasets for bug-fixing, security patching, or refactoring to better capture software engineering constraints and real-world coding norms.
- **Reward-based Optimization.** Reward-driven optimization incorporates quality signals including compilation success, test accuracy, and code readability into reinforcement learning objectives. For example, frameworks like *REAL* [141] and *ReCode* [135] define reward functions based on execution correctness and static quality metrics. This paradigm aligns model behavior toward functional and stylistic excellence by penalizing anti-patterns or resource-heavy code. Several studies have also employed hybrid reward designs, combining pass@k metrics with AST-based style consistency to yield both correctness and maintainability improvements [141].
- **Regularization-based Stabilization.** Regularization-based techniques stabilize model predictions to prevent erratic generation or mode collapse. Typical implementations include entropy regularization, dropout adjustment, and consistency loss constraints between successive decoding steps [60]. Empirical studies [60] show that

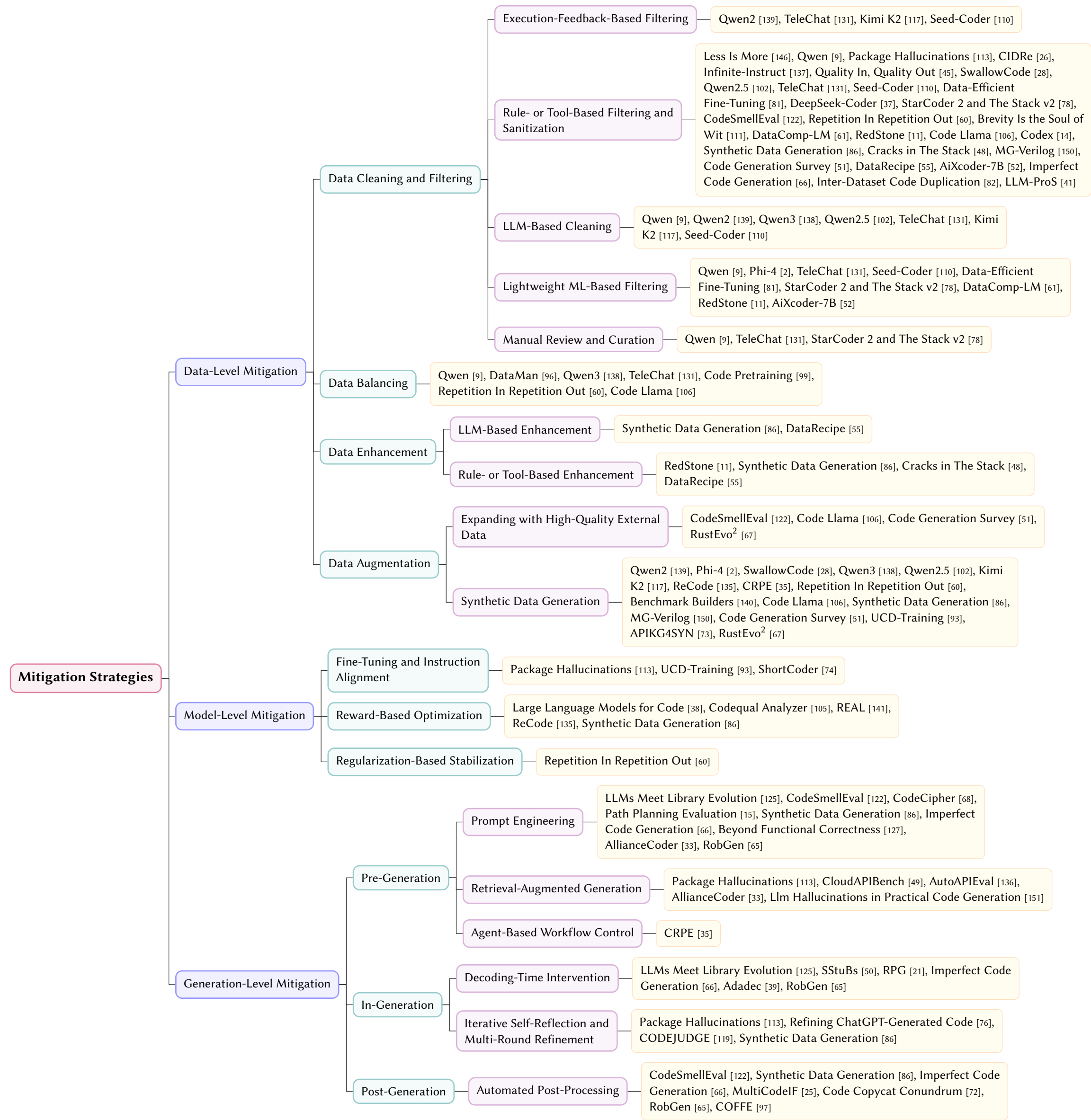

Fig. 15. Taxonomy of generated code quality issue mitigation strategies with corresponding literature references.

these regularization strategies enhance output robustness and reduce randomness-induced quality degradation, particularly in low-temperature beam search or constrained decoding environments.

Generation-level Mitigation. Generation-level mitigation refers to interventions during the code generation process itself, encompassing pre-generation preparation, in-generation control, and post-generation refinement. This category

represents the most flexible and operationally diverse segment of quality governance, enabling real-time adaptation and feedback-guided correction [21].

- **Pre-generation.** *Prompt engineering* introduces structured prompts, explicit constraints, and chain-of-thought exemplars to guide the model toward more accurate and readable outputs. Studies such as *RobGen* [65] and *CodeCipher* [68] show that well-designed prompts significantly reduce hallucination and errors. *Retrieval-Augmented Generation (RAG)* integrates external codebases or documentation retrieval prior to generation, ensuring contextual grounding and factual consistency [113]. Examples like *Package Hallucinations* [113] and *CloudAPIBench* [49] illustrate improvements in correctness and dependency management through retrieval-enhanced decoding. Additionally, *agent-based workflow control* frameworks organize multi-step generation workflows where specialized agents or sub-models handle each phase. Systems like *CRPE* [35] have demonstrated robust performance across diverse benchmarks.
- **In-generation.** *Decoding-time intervention* imposes constraints or rescoring rules on beam search paths, leveraging heuristics like syntax conformity, entropy-based thresholds, or execution feedback. For example, adaptive decoding and entropy-triggered reranking adjust token selection strategies based on uncertainty, thereby balancing diversity and correctness [39]. *Iterative self-reflection and multi-round refinement* enable the model to self-evaluate and correct partial outputs during generation. This paradigm, exemplified by *CODEJUDGE* [119], involves running intermediate tests or analyses, then refining the generated code iteratively until it passes evaluation criteria. Such methods substantially improve functional accuracy and logical coherence, approaching human-like iterative debugging.
- **Post-generation.** *Automated post-processing* techniques apply static or dynamic analyzers (e.g., *SonarQube*, *PyLint*) to automatically repair or reformat generated code [122]. Rule-based fixers or AST-level repair modules can resolve syntax errors, redundant expressions, or style inconsistencies [66, 89, 122]. In addition, tool-based verification frameworks execute generated programs in sandboxes or test harnesses to automatically prune incorrect or unsafe samples [122]. Recent studies also combine post-generation filtering with fine-grained ranking to select the most plausible completions among candidate outputs [119]. The synergy between automated repair and human verification (e.g., manual patch validation or code review integration) further strengthens reliability [89].

4.5.2 Training Data Quality Issue Mitigation. Training data quality issue mitigation encompasses a broad range of strategies aimed at improving, repairing, or rebalancing datasets to ensure the reliability, representativeness, and ethical integrity of LLM training and fine-tuning data. These methods span from cleaning and filtering to data enhancement and augmentation. The reviewed studies reveal four dominant categories of mitigation strategies: **data cleaning and filtering**, **data balancing**, **data enhancement**, and **data augmentation** (Figure 16). Collectively, they represent an evolving ecosystem of dataset governance that integrates automation, semantic reasoning, and human oversight.

Data Cleaning and Filtering. Data cleaning and filtering aim to remove noise, duplication, and harmful or invalid code samples prior to model training. These are among the most widely adopted dataset governance techniques in both academic and industrial LLM pipelines.

- **Execution-feedback-based Filtering.** This approach removes invalid or low-quality samples based on compilation or runtime results. Studies such as *TeleChat* and *Seed-Coder* [110, 131] execute code snippets to eliminate non-compiling or functionally incorrect examples. Execution logs and test feedback provide a direct

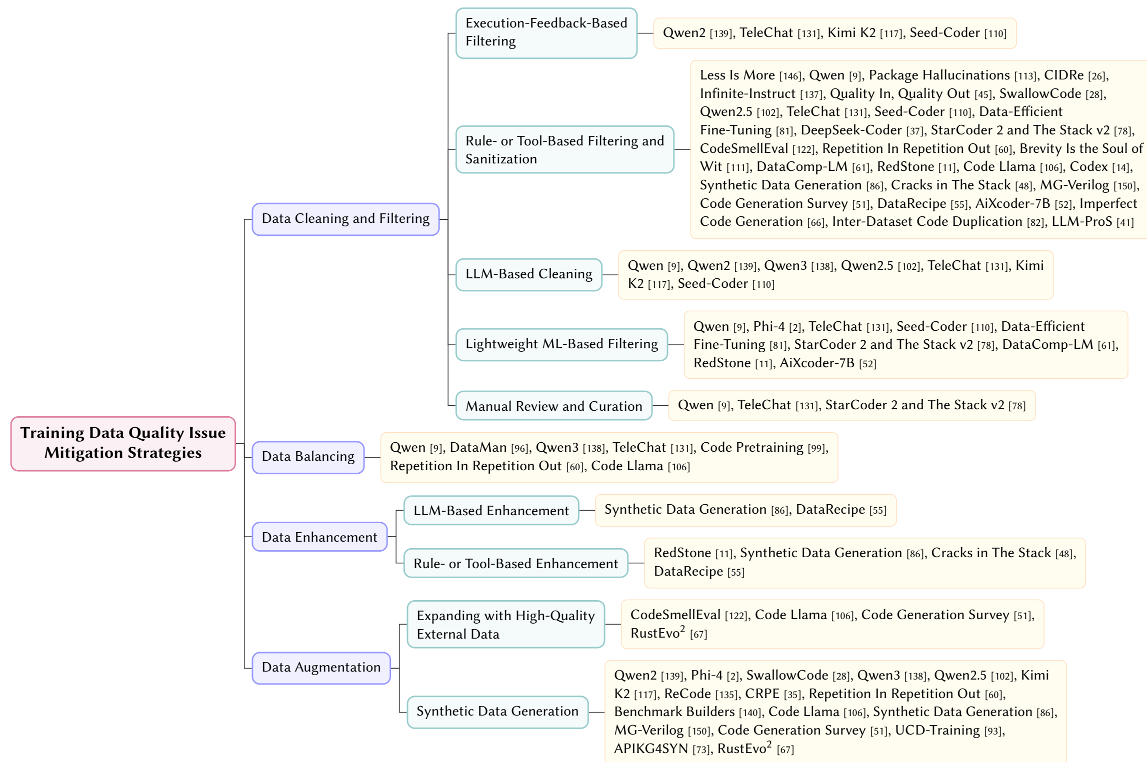

Fig. 16. Taxonomy of training data quality issue mitigation strategies with corresponding literature references.

signal of semantic validity, ensuring that only runnable and logically sound code remains in the dataset [117]. Some frameworks further use automated test harnesses or fuzzing to filter edge cases and unstable code [110].

- **Rule- or Tool-based Filtering and Sanitization.** Static analysis tools such as *Semgrep*, *SonarQube*, and *Bandit* are frequently used to detect vulnerabilities, unsafe imports, or license violations [52]. Rule-based sanitization frameworks like *DataLint* and *CodeAudit* define heuristics for deduplication, code smell removal, and comment normalization. These systems systematically eliminate low-quality patterns such as incomplete functions, empty try-catch blocks, and deprecated APIs, improving both syntactic and structural integrity.
- **LLM-based Cleaning.** Emerging studies leverage LLMs to clean datasets at the semantic level, reformatting, repairing, or rewriting code samples to fix indentation, correct minor logic errors, or standardize naming conventions [9, 138]. *Qwen3* and *Kimi K2* [61] employ prompt-based code correction and documentation completion, improving readability and maintainability. Unlike rule-based filters, these approaches capture contextual nuances and language semantics, effectively bridging the gap between structural and semantic cleansing.
- **Lightweight ML-based Filtering.** Lightweight classifiers are employed for scalable noise detection based on features like token entropy, line length, and AST complexity [11, 61, 110]. Systems such as *MetaFilter* use small models to pre-screen massive datasets before expensive LLM-based filtering, enabling efficient tiered cleaning pipelines.

- **Manual Review and Curation.** Despite increasing automation, human validation remains indispensable for high-stakes or ambiguous cases, such as security-related or licensed code. Manual review has been incorporated in datasets like *The Stack v2*, *CodeSearchNet-Filtered*, and *HumanEvalFix* [78], where experts confirm compliance, repair dataset labels, and remove contaminated benchmarks.

Data Balancing. Data balancing mitigates biases and imbalances across programming languages, difficulty levels, or problem domains, ensuring model generalization and equitable performance. Studies such as *Qwen3* [138] adjust sampling weights to harmonize language representation, such as Python, C++, and Java, while *CodeLlama* [106] employs difficulty-based stratification to equalize sample complexity. Other works rebalance function categories (e.g., data structures vs. algorithms) to prevent overrepresentation biases. Balancing at the corpus level has also been shown to improve fairness and robustness by reducing memorization of dominant distributions [9, 60, 96, 99, 106, 131, 138].

Data Enhancement. Data enhancement focuses on improving the semantic richness, structure, or clarity of existing code samples rather than discarding them.

- **LLM-based Enhancement.** LLMs can refine or rewrite low-quality code to conform to consistent style and design principles. Studies such as *DataRecipe* [55] use prompt-based refinement to correct syntax inconsistencies, improve variable naming, and insert docstrings. This paradigm not only cleans but also enriches data with human-like explanations, thereby enhancing both readability and training value.
- **Rule- or Tool-based Enhancement.** Tool-based enhancement frameworks apply static analyzers and formatters like *Black* and *ClangFormat* to standardize formatting and enforce coding style rules [11, 48, 55, 86]. Some studies also integrate refactoring tools that automatically modularize code or replace deprecated APIs [48]. These methods ensure structural consistency and promote model exposure to modern software engineering practices.

Data Augmentation. Data augmentation expands datasets with additional high-quality samples to increase coverage and diversity.

- **Synthetic Data Generation.** Synthetic code data can be generated using both rule-/tool-based and LLM-based approaches. Rule-based synthesis methods including template-based code generation and mutation operators augment datasets by systematically varying input parameters, function names, or control structures [106]. Examples include *Qwen2.5* [102] and *Code Llama* [106], which generate semantically equivalent variants to improve model robustness. LLM-based synthesis has become the dominant paradigm, where models such as textitStarCoder and *CodeLlama* are prompted to produce new code conditioned on problem descriptions or I/O examples [78, 106]. This strategy enables scalable creation of diverse yet high-quality samples, often filtered through executability and semantic constraints [102, 135].
- **Expanding with High-quality External Data.** Augmentation can also occur through the inclusion of curated external datasets from reputable open-source repositories [51, 106, 122]. For example, *Code Llama* [106] and *Stack V2* [78] integrate filtered GitHub data. Cross-domain augmentation, which combines algorithmic tasks, documentation, and conversational code explanations, further enhances representational richness.

> ***Summary and Insights.***

Quality assurance for LLM-based code generation is transitioning from isolated, stage-specific interventions toward continuous, data-model feedback loops. Current literature demonstrates that treating data curation, model alignment, and inference constraints as disjoint processes is suboptimal. Traditional heuristic filtering is increasingly augmented by execution-aware verification and LLM-driven semantic cleaning. By integrating runtime signals and introspective mechanisms, such as iterative self-reflection and retrieval-augmented generation, mitigation pipelines can dynamically refine both training distributions and generated artifacts. This evolution underscores that reliable code generation requires the deep interdependence of dataset integrity and continuous model optimization, rather than relying solely on post-hoc decoding fixes.

Although automated data augmentation and self-reflective repair mechanisms increasingly enable models to operate as autonomous agents, human oversight remains a strict boundary condition for high-stakes software engineering tasks. Recent studies highlight a clear trajectory toward automated pipelines capable of synthesizing high-quality training variants and correcting structural faults on the fly. However, purely algorithmic quality assurance struggles with semantic ambiguity, legal constraints, and complex security contexts. Consequently, human-in-the-loop validation is consistently required for critical evaluations, including security vulnerability triage, license compliance verification, and bias mitigation. This persistent necessity indicates that future governance frameworks must structurally balance scalable, automated refinement with targeted expert verification.

5 Discussion

This section discusses the broader implications of the findings beyond the descriptive synthesis in Section 4. We first distill several cross-cutting insights that connect generated code quality issues with training data quality issues. We then summarize the main open challenges that continue to limit reliable quality governance in current research. Based on these observations, we outline a roadmap for data-centric quality governance and further position this perspective within the broader LLM engineering ecosystem.

5.1 Cross-Cutting Insights on Code and Data Quality

Misattribution of Generated Code Quality Issues. The software engineering community frequently attributes generated code quality issues to limitations in model reasoning. However, our synthesis demonstrates that these generated code quality issues largely manifest as downstream symptoms of training data quality issues. Because LLMs accurately reflect the distribution of their training corpora, structural vulnerabilities and obsolete coding patterns in generated outputs are direct consequences of inadequately curated training data.

The Insufficiency of Correctness-Centric Evaluation. Evaluating generated code exclusively through functional correctness misrepresents its viability in real-world deployment. Attributes critical to software sustainability, such as maintainability, security, and understandability, remain systematically marginalized in prevailing training paradigms and evaluation benchmarks. Consequently, models frequently generate code that passes execution tests yet introduces architectural smells or known vulnerabilities. This discrepancy underscores a fundamental misalignment between statistical optimization targets and established software engineering principles.

The Persistent Trade-off in Data Distribution. Real-world code corpora are disproportionately skewed toward mainstream languages and boilerplate algorithmic tasks. Allocating extensive training bandwidth to these dominant domains maximizes performance on standard benchmarks, but inevitably compromises model capability in specialized, low-resource languages or complex repository-level scenarios. Balancing this distribution without sacrificing core

capabilities requires sophisticated data curation strategies, indicating that merely scaling dataset volume cannot resolve inherent representational biases.

5.2 Open Challenges in Current Research

Pervasive Benchmark Contamination. The widespread issue of evaluation leakage undermines the integrity of quality assurance efforts. Traditional static datasets are increasingly, albeit inadvertently, included in pre-training corpora, artificially inflating performance metrics and rendering comparative analyses unreliable. When models memorize ground-truth solutions rather than internalizing programming logic, establishing their true generalization capability becomes highly problematic.

Data Opacity and Temporal Drift. Evaluation contamination is exacerbated by a lack of transparency in training data provenance. State-of-the-art models are frequently trained on proprietary or partially disclosed datasets, precluding independent verification. Even within open-source data, the rapid evolution of software ecosystems introduces temporal drift via deprecated APIs and patched vulnerabilities. When a model generates outdated or insecure code, researchers lack robust data-lineage frameworks to trace the output back to specific obsolete repositories, rendering targeted remediation exceedingly difficult.

Absence of Causal Attribution. Causal inference between training data quality and generated code quality remains underexplored. While empirical studies consistently highlight strong correlations, the community lacks formal causal attribution methods to pinpoint how specific training data quality issues induce corresponding generated code quality issues. Without causal grounding, mitigation techniques rely on heuristic trial-and-error rather than principled interventions.

5.3 A Roadmap for Data-Centric Quality Governance

To transition from reactive debugging to proactive governance, future research must operationalize data-centric principles across the model lifecycle. The traditional separation among data curation, model training, and post-generation repair is demonstrably unsustainable. We outline three actionable research avenues:

Data Provenance and Targeted Unlearning. Future architectures must integrate explicit data provenance tracking to facilitate closed-loop debugging. By developing efficient unlearning mechanisms and influence functions tailored for source code, researchers can attribute specific generated code quality issues directly to training data quality issues. This capability enables continuous, targeted dataset sanitization without the computational burden of full model retraining.

Dynamic, Contamination-Resistant Benchmarks. The reliance on static execution benchmarks must pivot toward dynamic evaluation frameworks. Research should focus on constructing continuously updating benchmarks that extract recent, unmemorized issues directly from active repositories. These frameworks must holistically integrate static analysis tools to assess cyclomatic complexity and security, ensuring that generated code is both functionally and structurally sound.

Aligning Models with Software Engineering Standards. Model alignment techniques must evolve to incorporate objective software engineering constraints. Rather than relying solely on human preference or basic test execution, reward functions should synthesize signals from diverse sources, including static analyzers, resource efficiency profilers, and dependency resolution tools. Embedding precise engineering metrics directly into the optimization process compels models to internalize high-level architectural standards, ultimately yielding code that is intrinsically reliable and maintainable.

5.4 Positioning Data-Centric Governance in the Evolving LLM Engineering Ecosystem

Recent LLM deployment trends increasingly emphasize *harness engineering*, where models are embedded into external frameworks such as retrieval pipelines, tool-augmented workflows, and multi-agent systems. This trend changes how code-generation quality is managed in practice, but it does not weaken the relevance of a data-centric perspective.

Harness-level engineering mainly mitigates failures at deployment time. For example, retrieval can supplement missing or outdated knowledge in a specific context, and external validators can intercept insecure or non-executable outputs before use. However, such interventions do not remove the underlying causes discussed in this review. If training corpora contain insecure coding idioms, obsolete APIs, or distorted distributions, these issues remain part of the model's learned behavior and may still surface in scenarios not explicitly covered by the harness. In this sense, harness engineering should be viewed as a layer of compensation and control, rather than a substitute for improving the quality of the training data itself.

A related trend is that some capabilities previously handled by external agents, such as tool use, iterative debugging, and multi-step planning, are increasingly being internalized into the model through pretraining or post-training alignment. This development further strengthens, rather than reduces, the importance of data-centric governance. Once such behaviors are absorbed into model parameters, their reliability becomes more directly tied to the quality of the underlying training examples. As the boundary between model internals and external orchestration becomes less clear, improving training data quality remains essential for building reliable code-generation systems, even when strong engineering frameworks are available.

6 Threats to Validity

Internal Validity. Despite a rigorous multi-database search and forward–backward snowballing, the coverage of relevant studies may be incomplete due to indexing delays or preprints evolving after initial publication. Keyword-based retrieval is inherently sensitive to terminology; the diverse phrasing across software engineering and artificial intelligence communities means some relevant works might evade our search expressions. During data extraction, categorizing complex or hybrid methods involves subjective interpretation. To mitigate this bias, three independent reviewers cross-validated the coding process, resolving discrepancies through consensus meetings.

Construct Validity. The concepts of code and dataset quality are multifaceted and lack universally standardized definitions. While our taxonomy synthesizes existing software quality frameworks and recent literature, this abstraction may oversimplify nuanced dimensions like semantic alignment or safety constraints. Furthermore, evaluating primary studies using our 16-point quantitative matrix (Section 3) relies on the transparency of the original publications; sparse methodological reporting in certain papers may introduce scoring inconsistencies. As the field rapidly evolves, emerging quality attributes may transcend the constructs formalized in this review.

External Validity. Generalizing our findings to the broader code generation landscape presents specific boundaries. The 114 analyzed studies predominantly focus on English-language corpora, mainstream programming languages, and open-source datasets. Therefore, our synthesis may not fully capture the quality dynamics of multilingual settings, low-resource languages, or proprietary enterprise codebases. Additionally, the observed mappings between training data quality issues and generated code quality issues are tied to current LLM architectures and training regimes; future models with distinct inductive biases or data curation pipelines may exhibit different behaviors.

Conclusion Validity. The heterogeneity of experimental designs across primary studies precludes a formal meta-analysis; thus, our derived mappings and categorical trends are qualitative syntheses rather than precise measurements

of effect size. Furthermore, publication bias likely skews the distribution of evaluated techniques: studies reporting successful mitigations or novel frameworks are published more frequently than neutral or negative results. This asymmetry may lead to an overestimation of the maturity of current quality-assurance methods. Consequently, our findings characterize the current research landscape and highlight empirical correlations, rather than establishing definitive causal claims.

7 Conclusion

This systematic review of 114 primary studies establishes a unified taxonomy categorizing generated code quality issues and their corresponding data-level origins. By synthesizing the causal mappings between these two dimensions, this study highlights a fundamental mechanism: generation failures—such as structural vulnerabilities, obsolete APIs, and logic defects—are rarely isolated model reasoning deficits. Instead, they are direct manifestations of upstream data-level technical debt, propagating through dataset noise, benchmark contamination, and representational bias.

Currently, quality assurance in LLM code generation remains methodologically fragmented, heavily relying on reactive, post-generation filtering. The overarching implication of this review is the necessity for a paradigm shift toward proactive, data-centric governance. To achieve sustainable reliability in LLM-generated software, the community should transition from disjointed, stage-specific mitigations to end-to-end quality pipelines. This evolution entails establishing bidirectional traceability between generation artifacts and training corpora, deploying dynamic evaluation frameworks, and integrating objective software engineering constraints directly into model optimization. By bridging the gap between statistical data curation and rigorous software standards, this review lays the empirical foundation for building trustworthy, maintainable LLM-based development tools.

References

[1] Altaf Allah Abbassi, Leuson Da Silva, Amin Nikanjam, and Foutse Khomh. 2025. A Taxonomy of Inefficiencies in LLM-Generated Python Code. arXiv:2503.06327 [cs.SE] https://arxiv.org/abs/2503.06327

[2] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. 2024. Phi-4 Technical Report. arXiv:2412.08905 [cs.CL] https://arxiv.org/abs/2412.08905

[3] Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. 2025. CodeMirage: Hallucinations in Code Generated by Large Language Models. arXiv:2408.08333 [cs.SE] https://arxiv.org/abs/2408.08333

[4] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. arXiv:1709.06182 [cs.SE] https://arxiv.org/abs/1709.06182

[5] Victor Alves, Carla Bezerra, Ivan Machado, Larissa Rocha, Tássio Virgínio, and Publio Silva. 2025. Quality Assessment of Python Tests Generated by Large Language Models. arXiv:2506.14297 [cs.SE] https://arxiv.org/abs/2506.14297

[6] Viraat Aryabumi, Yixuan Su, Raymond Ma, Adrien Morisot, Ivan Zhang, Acyr Locatelli, Marzieh Fadaee, Ahmet Üstün, and Sara Hooker. 2024. To Code, or Not To Code? Exploring Impact of Code in Pre-training. arXiv:2408.10914 [cs.CL] https://arxiv.org/abs/2408.10914

[7] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2024. Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code? arXiv:2204.04741 [cs.SE] https://arxiv.org/abs/2204.04741

[8] Md. Abdul Awal, Mrigank Rochan, and Chanchal K. Roy. 2025. Large Language Models as Robust Data Generators in Software Analytics: Are We There Yet? arXiv:2411.10565 [cs.SE] https://arxiv.org/abs/2411.10565

[9] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. arXiv:2309.16609 [cs.CL] https://arxiv.org/abs/2309.16609

[10] Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, Sasha Frolov, Ravi Prakash Giri, Dhaval Kapil, Yiannis Kozyrakis, David LeBlanc, James Milazzo, Aleksandar

Straumann, Gabriel Synnaeve, Varun Vontimitta, Spencer Whitman, and Joshua Saxe. 2023. Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models. arXiv:2312.04724 [cs.CR] https://arxiv.org/abs/2312.04724

[11] Yaoyao Chang, Lei Cui, Li Dong, Shaohan Huang, Yangyu Huang, Yupan Huang, Scarlett Li, Tengchao Lv, Shuming Ma, Qinzheng Sun, Wenhui Wang, Furu Wei, Ying Xin, Mao Yang, Qiufeng Yin, and Xingxing Zhang. 2024. RedStone: Curating General, Code, Math, and QA Data for Large Language Models. arXiv:2412.03398 [cs.CL] https://arxiv.org/abs/2412.03398

[12] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology* 15, 3 (2024), 1–45.

[13] Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, Wei Ye, and Shikun Zhang. 2025. A Survey on Evaluating Large Language Models in Code Generation Tasks. arXiv:2408.16498 [cs.SE] https://arxiv.org/abs/2408.16498

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[15] Wanyi Chen, Meng-Wen Su, and Mary L. Cummings. 2025. Assessing LLM code generation quality through path planning tasks. arXiv:2504.21276 [cs.SE] https://arxiv.org/abs/2504.21276

[16] Zhengyu Chen, Siqi Wang, Teng Xiao, Yudong Wang, Shiqi Chen, Xunliang Cai, Junxian He, and Jingang Wang. 2025. Revisiting Scaling Laws for Language Models: The Role of Data Quality and Training Strategies. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 23881–23899. https://doi.org/10.18653/v1/2025.acl-long.1163

[17] Heejae Chon, Seonghyeon Lee, Jinyoung Yeo, and Dongha Lee. 2024. Is Functional Correctness Enough to Evaluate Code Language Models? Exploring Diversity of Generated Codes. arXiv:2408.14504 [cs.SE] https://arxiv.org/abs/2408.14504

[18] Chun Jie Chong, Zhihao Yao, and Iulian Neamtiu. 2024. Artificial-Intelligence Generated Code Considered Harmful: A Road Map for Secure and High-Quality Code Generation. arXiv:2409.19182 [cs.CR] https://arxiv.org/abs/2409.19182

[19] Codefuse, Ling Team, :, Wenting Cai, Yuchen Cao, Chaoyu Chen, Chen Chen, Siba Chen, Qing Cui, Peng Di, Junpeng Fang, Zi Gong, Ting Guo, Zhengyu He, Yang Huang, Cong Li, Jianguo Li, Zheng Li, Shijie Lian, BingChang Liu, Songshan Luo, Shuo Mao, Min Shen, Jian Wu, Jiaolong Yang, Wenjie Yang, Tong Ye, Hang Yu, Wei Zhang, Zhenduo Zhang, Hailin Zhao, Xunjin Zheng, and Jun Zhou. 2025. Every Sample Matters: Leveraging Mixture-of-Experts and High-Quality Data for Efficient and Accurate Code LLM. arXiv:2503.17793 [cs.LG] https://arxiv.org/abs/2503.17793

[20] Domenico Cotroneo, Roberta De Luca, and Pietro Liguori. 2024. DeVAIC: A Tool for Security Assessment of AI-generated Code. arXiv:2404.07548 [cs.SE] https://arxiv.org/abs/2404.07548

[21] Yihong Dong, Yuchen Liu, Xue Jiang, Zhi Jin, and Ge Li. 2025. Rethinking Repetition Problems of LLMs in Code Generation. arXiv:2505.10402 [cs.CL] https://arxiv.org/abs/2505.10402

[22] Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A Code Efficiency Benchmark for Code Large Language Models. arXiv:2402.07844 [cs.SE] https://arxiv.org/abs/2402.07844

[23] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 982–994. https://doi.org/10.1145/3597503.3639219

[24] Yongkang Du, Jen tse Huang, Jieyu Zhao, and Lu Lin. 2025. FairCoder: Evaluating Social Bias of LLMs in Code Generation. arXiv:2501.05396 [cs.CL] https://arxiv.org/abs/2501.05396

[25] Guoliang Duan, Mingwei Liu, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. 2025. A Hierarchical and Evolvable Benchmark for Fine-Grained Code Instruction Following with Multi-Turn Feedback. arXiv:2507.00699 [cs.SE] https://arxiv.org/abs/2507.00699

[26] Maria Dziuba and Valentin Malykh. 2025. CIDRe: A Reference-Free Multi-Aspect Criterion for Code Comment Quality Measurement. arXiv:2505.19757 [cs.SE] https://arxiv.org/abs/2505.19757

[27] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[28] Kazuki Fujii, Yukito Tajima, Sakae Mizuki, Hinari Shimada, Taihei Shiotani, Koshiro Saito, Masanari Ohi, Masaki Kawamura, Taishi Nakamura, Takumi Okamoto, Shigeki Ishida, Kakeru Hattori, Youmi Ma, Hiroya Takamura, Rio Yokota, and Naoaki Okazaki. 2025. Rewriting Pre-Training Data Boosts LLM Performance in Math and Code. arXiv:2505.02881 [cs.LG] https://arxiv.org/abs/2505.02881

[29] Cuiyun Gao, Xing Hu, Shan Gao, Xin Xia, and Zhi Jin. 2024. The Current Challenges of Software Engineering in the Era of Large Language Models. arXiv:2412.14554 [cs.SE] https://arxiv.org/abs/2412.14554

[30] Jiahui Geng, Fengyu Cai, Shaobo Cui, Qing Li, Liangwei Chen, Chenyang Lyu, Haonan Li, Derui Zhu, Walter Pretschner, Heinz Koeppl, and Fakhri Karray. 2025. CoQuIR: A Comprehensive Benchmark for Code Quality-Aware Information Retrieval. arXiv:2506.11066 [cs.SE] https://arxiv.org/abs/2506.11066

[31] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[32] Alex Gu, Wen-Ding Li, Naman Jain, Theo X. Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. 2024. The Counterfeit Conundrum: Can Code Language Models Grasp the Nuances of Their Incorrect Generations? arXiv:2402.19475 [cs.SE] https://arxiv.org/abs/2402.19475

[33] Wenchao Gu, Juntao Chen, Yanlin Wang, Tianyue Jiang, Xingzhe Li, Mingwei Liu, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. What to Retrieve for Effective Retrieval-Augmented Code Generation? An Empirical Study and Beyond. arXiv:2503.20589 [cs.SE] https://arxiv.org/abs/2503.20589

[34] Batu Guan, Yao Wan, Zhangqian Bi, Zheng Wang, Hongyu Zhang, Pan Zhou, and Lichao Sun. 2024. CodeIP: A Grammar-Guided Multi-Bit Watermark for Large Language Models of Code. arXiv:2404.15639 [cs.CL] https://arxiv.org/abs/2404.15639

[35] Ningxin Gui, Qianghuai Jia, Feijun Jiang, Yuling Jiao, dechun wang, and Jerry Zhijian Yang. 2025. CRPE: Expanding The Reasoning Capability of Large Language Model for Code Generation. arXiv:2505.10594 [cs.SE] https://arxiv.org/abs/2505.10594

[36] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks Are All You Need. arXiv:2306.11644 [cs.CL] https://arxiv.org/abs/2306.11644

[37] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE] https://arxiv.org/abs/2401.14196

[38] Jingxuan He and Martin Vechev. 2023. Large Language Models for Code: Security Hardening and Adversarial Testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1865–1879. https://doi.org/10.1145/3576915.3623175

[39] Kaifeng He, Mingwei Liu, Chong Wang, Zike Li, Yanlin Wang, Xin Peng, and Zibin Zheng. 2026. Towards Better Code Generation: Adaptive Decoding with Uncertainty Guidance. arXiv:2506.08980 [cs.SE] https://arxiv.org/abs/2506.08980

[40] Tianxing He, Jingzhao Zhang, Zhiming Zhou, and James Glass. 2021. Exposure Bias versus Self-Recovery: Are Distortions Really Incremental for Autoregressive Text Generation? arXiv:1905.10617 [cs.LG] https://arxiv.org/abs/1905.10617

[41] Md Sifat Hossain, Anika Tabassum, Md. Fahim Arefin, and Tarannum Shaila Zaman. 2025. LLM-ProS: Analyzing Large Language Models' Performance in Competitive Problem Solving. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 80–87. https://doi.org/10.1109/llm4code66737.2025.00015

[42] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[43] Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. 2025. EffiBench: Benchmarking the Efficiency of Automatically Generated Code. arXiv:2402.02037 [cs.SE] https://arxiv.org/abs/2402.02037

[44] Nam Huynh and Beiyu Lin. 2025. Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications. arXiv:2503.01245 [cs.SE] https://arxiv.org/abs/2503.01245

[45] Cristina Improta, Rosalia Tufano, Pietro Liguori, Domenico Cotroneo, and Gabriele Bavota. 2025. Quality In, Quality Out: Investigating Training Data's Role in AI Code Generation. arXiv:2503.11402 [cs.SE] https://arxiv.org/abs/2503.11402

[46] International Organization for Standardization. 2023. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. ISO/IEC 25010:2023.

[47] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language models for code completion: A practical evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[48] Mahmoud Jahanshahi and Audris Mockus. 2025. Cracks in The Stack: Hidden Vulnerabilities and Licensing Risks in LLM Pre-Training Datasets. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 104–111. https://doi.org/10.1109/llm4code66737.2025.00018

[49] Nihal Jain, Robert Kwiatkowski, Baishakhi Ray, Murali Krishna Ramanathan, and Varun Kumar. 2024. On Mitigating Code LLM Hallucinations with API Documentation. arXiv:2407.09726 [cs.CL] https://arxiv.org/abs/2407.09726

[50] Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. 2023. Large Language Models and Simple, Stupid Bugs. arXiv:2303.11455 [cs.SE] https://arxiv.org/abs/2303.11455

[51] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology* 35, 2 (Jan. 2026), 1–72. https://doi.org/10.1145/3747588

[52] Siyuan Jiang, Jia Li, He Zong, Huanyu Liu, Hao Zhu, Shukai Hu, Erlu Li, Jiazheng Ding, Yu Han, Wei Ning, Gen Wang, Yihong Dong, Kechi Zhang, and Ge Li. 2025. aiXcoder-7B: A Lightweight and Effective Large Language Model for Code Processing. arXiv:2410.13187 [cs.CL] https://arxiv.org/abs/2410.13187

[53] Weipeng Jiang, Xuanqi Gao, Juan Zhai, Shiqing Ma, Xiaoyu Zhang, Ziyan Lei, and Chao Shen. 2025. From Effectiveness to Efficiency: Uncovering Linguistic Bias in Large Language Model-based Code Generation. arXiv:2406.00602 [cs.SE] https://arxiv.org/abs/2406.00602

[54] Mohammed Kharma, Soohyeon Choi, Mohammed AlKhanafseh, and David Mohaisen. 2025. Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis. arXiv:2502.01853 [cs.CR] https://arxiv.org/abs/2502.01853

[55] Kisub Kim, Jounghoon Kim, Byeongjo Park, Dongsun Kim, Chun Yong Chong, Yuan Wang, Tiezhu Sun, Daniel Tang, Jacques Klein, and Tegawende F. Bissyande. 2024. DataRecipe — How to Cook the Data for CodeLLM?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1206–1218. https://doi.org/10.1145/3691620.3695593

[56] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Keele University and Durham University Joint Report, EBSE 2007-001.

[57] Jasmine Latendresse, SayedHassan Khatoonabadi, Ahmad Abdellatif, and Emad Shihab. 2024. Is ChatGPT a Good Software Librarian? An Exploratory Study on the Use of ChatGPT for Software Library Recommendations. arXiv:2408.05128 [cs.SE] https://arxiv.org/abs/2408.05128

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

[59] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2022. Deduplicating Training Data Makes Language Models Better. arXiv:2107.06499 [cs.CL] https://arxiv.org/abs/2107.06499

[60] Huayang Li, Tian Lan, Zihao Fu, Deng Cai, Lemao Liu, Nigel Collier, Taro Watanabe, and Yixuan Su. 2023. Repetition In Repetition Out: Towards Understanding Neural Text Degeneration from the Data Perspective. arXiv:2310.10226 [cs.CL] https://arxiv.org/abs/2310.10226

[61] Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, Saurabh Garg, Rui Xin, Niklas Muennighoff, Reinhard Heckel, Jean Mercat, Mayee Chen, Suchin Gururangan, Mitchell Wortsman, Alon Albalak, Yonatan Bitton, Marianna Nezhurina, Amro Abbas, Cheng-Yu Hsieh, Dhruba Ghosh, Josh Gardner, Maciej Kilian, Hanlin Zhang, Rulin Shao, Sarah Pratt, Sunny Sanyal, Gabriel Ilharco, Giannis Daras, Kalyani Marathe, Aaron Gokaslan, Jieyu Zhang, Khyathi Chandu, Thao Nguyen, Igor Vasiljevic, Sham Kakade, Shuran Song, Sujay Sanghavi, Fartash Faghri, Sewoong Oh, Luke Zettlemoyer, Kyle Lo, Alaaeldin El-Nouby, Hadi Pouransari, Alexander Toshev, Stephanie Wang, Dirk Groeneveld, Luca Soldaini, Pang Wei Koh, Jenia Jitsev, Thomas Kollar, Alexandros G. Dimakis, Yair Carmon, Achal Dave, Ludwig Schmidt, and Vaishaal Shankar. 2025. DataComp-LM: In search of the next generation of training sets for language models. arXiv:2406.11794 [cs.LG] https://arxiv.org/abs/2406.11794

[62] Jia Li, Zeyang Zhuang, Zhuangbin Chen, Yuxin Su, Wei Meng, and Michael R. Lyu. 2026. ComBench: A Repo-level Real-world Benchmark for Compilation Error Repair. arXiv:2603.27333 [cs.SE] https://arxiv.org/abs/2603.27333

[63] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL] https://arxiv.org/abs/2305.06161

[64] Yuanheng Li, Zhuoyang Chen, Xiaoyun Liu, Yuhao Wang, Mingwei Liu, Yang Shi, Kaifeng Huang, and Shengjie Zhao. 2025. Uncovering Pretraining Code in LLMs: A Syntax-Aware Attribution Approach. arXiv:2511.07033 [cs.CR] https://arxiv.org/abs/2511.07033

[65] Zike Li, Mingwei Liu, Anji Li, Kaifeng He, Yanlin Wang, Xin Peng, and Zibin Zheng. 2025. A Preliminary Study on the Robustness of Code Generation by Large Language Models. arXiv:2503.20197 [cs.SE] https://arxiv.org/abs/2503.20197

[66] Xiaoli Lian, Shuaisong Wang, Jieping Ma, Xin Tan, Fang Liu, Lin Shi, Cuiyun Gao, and Li Zhang. 2024. Imperfect Code Generation: Uncovering Weaknesses in Automatic Code Generation by Large Language Models. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 422–423. https://doi.org/10.1145/3639478.3643081

[67] Linxi Liang, Jing Gong, Mingwei Liu, Chong Wang, Guangsheng Ou, Yanlin Wang, Xin Peng, and Zibin Zheng. 2025. RustEvo2̂: An Evolving Benchmark for API Evolution in LLM-based Rust Code Generation. arXiv:2503.16922 [cs.SE] https://arxiv.org/abs/2503.16922

[68] Yalan Lin, Chengcheng Wan, Yixiong Fang, and Xiaodong Gu. 2024. CodeCipher: Learning to Obfuscate Source Code Against LLMs. arXiv:2410.05797 [cs.CL] https://arxiv.org/abs/2410.05797

[69] Lin Ling, Fazle Rabbi, Song Wang, and Jinqiu Yang. 2025. Bias Unveiled: Investigating Social Bias in LLM-Generated Code. arXiv:2411.10351 [cs.SE] https://arxiv.org/abs/2411.10351

[70] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. arXiv:2404.00971 [cs.SE] https://arxiv.org/abs/2404.00971

[71] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210 [cs.SE] https://arxiv.org/abs/2305.01210

[72] Mingwei Liu, Juntao Li, Ying Wang, Xueying Du, Zuoyu Ou, Qiuyuan Chen, Bingxu An, Zhao Wei, Yong Xu, Fangming Zou, Xin Peng, and Yiling Lou. 2025. Code Copycat Conundrum: Demystifying Repetition in LLM-based Code Generation. arXiv:2504.12608 [cs.SE] https://arxiv.org/abs/2504.12608

[73] Mingwei Liu, Zheng Pei, Yanlin Wang, Zihao Wang, Zikang Li, Enci Lin, Xin Peng, and Zibin Zheng. 2025. Framework-Aware Code Generation with API Knowledge Graph-Constructed Data: A Study on HarmonyOS. arXiv:2512.00380 [cs.SE] https://arxiv.org/abs/2512.00380

[74] Sicong Liu, Yanxian Huang, Mingwei Liu, Jiachi Chen, Ensheng Shi, Yuchi Ma, Hongyu Zhang, Yin Zhang, and Yanlin Wang. 2026. ShortCoder: Knowledge-Augmented Syntax Optimization for Token-Efficient Code Generation. arXiv:2601.09703 [cs.SE] https://arxiv.org/abs/2601.09703

[75] Yang Liu, Jiahuan Cao, Chongyu Liu, Kai Ding, and Lianwen Jin. 2024. Datasets for Large Language Models: A Comprehensive Survey. arXiv:2402.18041 [cs.CL] https://arxiv.org/abs/2402.18041

[76] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. 2023. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. arXiv:2307.12596 [cs.SE] https://arxiv.org/abs/2307.12596

[77] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2024. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. arXiv:2308.04838 [cs.SE] https://arxiv.org/abs/2308.04838

[78] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE] https://arxiv.org/abs/2402.19173

[79] Junyu Luo, Bohan Wu, Xiao Luo, Zhiping Xiao, Yiqiao Jin, Rong-Cheng Tu, Nan Yin, Yifan Wang, Jingyang Yuan, Wei Ju, and Ming Zhang. 2025. A Survey on Efficient Large Language Model Training: From Data-centric Perspectives. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 30904–30920. https://doi.org/10.18653/v1/2025.acl-long.1493

[80] Wenqiang Luo, Jacky Wai Keung, Boyang Yang, He Ye, Claire Le Goues, Tegawende F. Bissyande, Haoye Tian, and Bach Le. 2024. When Fine-Tuning LLMs Meets Data Privacy: An Empirical Study of Federated Learning in LLM-Based Program Repair. arXiv:2412.01072 [cs.SE] https://arxiv.org/abs/2412.01072

[81] Weijie Lv, Xuan Xia, and Sheng-Jun Huang. 2025. Data-efficient LLM Fine-tuning for Code Generation. arXiv:2504.12687 [cs.CL] https://arxiv.org/abs/2504.12687

[82] José Antonio Hernández López, Boqi Chen, Mootez Saad, Tushar Sharma, and Dániel Varró. 2025. On Inter-Dataset Code Duplication and Data Leakage in Large Language Models. *IEEE Transactions on Software Engineering* 51, 1 (2025), 192–205. https://doi.org/10.1109/TSE.2024.3504286

[83] Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At Which Training Stage Does Code Data Help LLMs Reasoning? arXiv:2309.16298 [cs.CL] https://arxiv.org/abs/2309.16298

[84] Lakshmi Likhitha Mankali, Jitendra Bhandari, Manaar Alam, Ramesh Karri, Michail Maniatakos, Ozgur Sinanoglu, and Johann Knechtel. 2024. RTL-Breaker: Assessing the Security of LLMs against Backdoor Attacks on HDL Code Generation. arXiv:2411.17569 [cs.CR] https://arxiv.org/abs/2411.17569

[85] Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. 2024. On Leakage of Code Generation Evaluation Datasets. arXiv:2407.07565 [cs.CL] https://arxiv.org/abs/2407.07565

[86] Mihai Nadăş, Laura Dioşan, and Andreea Tomescu. 2025. Synthetic Data Generation Using Large Language Models: Advances in Text and Code. *IEEE Access* 13 (2025), 134615–134633. https://doi.org/10.1109/ACCESS.2025.3589503

[87] Claudia Negri-Ribalta, Rémi Geraud-Stewart, Anastasia Sergeeva, and Gabriele Lenzini. 2024. A systematic literature review on the impact of AI models on the security of code generation. *Frontiers in Big Data* 7 (2024), 1386720.

[88] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3524842.3528470

[89] Yuqing Nie, Chong Wang, Kailong Wang, Guoai Xu, Guosheng Xu, and Haoyu Wang. 2025. Decoding Secret Memorization in Code LLMs Through Token-Level Characterization. arXiv:2410.08858 [cs.CR] https://arxiv.org/abs/2410.08858

[90] Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. 2024. On Evaluating the Efficiency of Source Code Generated by LLMs. arXiv:2404.06041 [cs.SE] https://arxiv.org/abs/2404.06041

[91] Samal Nursapa, Anastassiya Samuilova, Alessio Bucaioni, and Phuong T. Nguyen. 2025. ROSE: Transformer-Based Refactoring Recommendation for Architectural Smells. arXiv:2507.12561 [cs.SE] https://arxiv.org/abs/2507.12561

[92] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse

Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[93] Guangsheng Ou, Qiming Zhang, Sirong Chen, Anji Li, Dong Xu, Tiancheng Luo, Dekun Dai, Cuiyun Gao, Long Wang, Jun Zhou, Mingwei Liu, and Zibin Zheng. 2026. Unseen-Codebases-Domain Data Synthesis and Training Based on Code Graphs. arXiv:2602.20799 [cs.SE] https://arxiv.org/abs/2602.20799

[94] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL] https://arxiv.org/abs/2203.02155

[95] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (Jan. 2025), 96–105. https://doi.org/10.1145/3610721

[96] Ru Peng, Kexin Yang, Yawen Zeng, Junyang Lin, Dayiheng Liu, and Junbo Zhao. 2025. DataMan: Data Manager for Pre-training Large Language Models. arXiv:2502.19363 [cs.CL] https://arxiv.org/abs/2502.19363

[97] Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. 2025. COFFE: A Code Efficiency Benchmark for Code Generation. arXiv:2502.02827 [cs.SE] https://arxiv.org/abs/2502.02827

[98] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2785–2799. https://doi.org/10.1145/3576915.3623157

[99] Jackson Petty, Sjoerd van Steenkiste, and Tal Linzen. 2025. How Does Code Pretraining Affect Language Model Task Performance? arXiv:2409.04556 [cs.CL] https://arxiv.org/abs/2409.04556

[100] Ihor Pysmennyi, Roman Kyslyi, and Kyrylo Kleshch. 2025. AI-driven tools in modern software quality assurance: an assessment of benefits, challenges, and future directions. *Technology audit and production reserves* 3, 2(83) (May 2025), 44–54. https://doi.org/10.15587/2706-5448.2025.330595

[101] Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. 2025. How Efficient is LLM-Generated Code? A Rigorous & High-Standard Benchmark. arXiv:2406.06647 [cs.SE] https://arxiv.org/abs/2406.06647

[102] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[103] Mirza Masfiqur Rahman and Ashish Kundu. 2024. Code Hallucination. arXiv:2407.04831 [cs.AI] https://arxiv.org/abs/2407.04831

[104] Md Bajlur Rashid, Mohammad Shafayet Jamil Hossain, Mohammad Ishtiaque Khan, Sharaban Tahora, Aiasha Siddika, Mahmudul Islam Prakash, Sharmin Yeasmin, and Hossain Shahriar. 2026. A Survey on Large Language Models in Software Security: Opportunities and Threats. *Computers* 15, 4 (2026). https://doi.org/10.3390/computers15040226

[105] Maxime Robeyns and Laurence Aitchison. 2025. Improving LLM-Generated Code Quality with GRPO. arXiv:2506.02211 [cs.AI] https://arxiv.org/abs/2506.02211

[106] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[107] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. arXiv:2208.09727 [cs.CR] https://arxiv.org/abs/2208.09727

[108] Laboni Sarker, Mara Downing, Achintya Desai, and Tevfik Bultan. 2024. Syntactic Robustness for LLM-based Code Generation. arXiv:2404.01535 [cs.SE] https://arxiv.org/abs/2404.01535

[109] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in Machine learning systems. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 2503–2511.

[110] ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, Yifan Huang, Guanghan Ning, Xierui Song, Jiaze Chen, Siyao Liu, Kai Shen, Liang Xiang, and Yonghui Wu. 2025. Seed-Coder: Let the Code Model Curate Data for Itself. arXiv:2506.03524 [cs.CL] https://arxiv.org/abs/2506.03524

[111] Aaditya K. Singh, Yu Yang, Kushal Tirumala, Mostafa Elhoushi, and Ari S. Morcos. 2024. Brevity is the soul of wit: Pruning long files for code generation. arXiv:2407.00434 [cs.CL] https://arxiv.org/abs/2407.00434

[112] Fangchen Song, Ashish Agarwal, and Wen Wen. 2025. The Impact of Generative AI on Collaborative Open-Source Software Development: Evidence from GitHub Copilot. arXiv:2410.02091 [cs.SE] https://arxiv.org/abs/2410.02091

[113] Joseph Spracklen, Raveen Wijewickrama, A H M Nazmus Sakib, Anindya Maiti, Bimal Viswanath, and Murtuza Jadliwala. 2025. We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs. arXiv:2406.10279 [cs.SE] https://arxiv.org/abs/2406.10279

[114] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1609–1620. https://doi.org/10.1145/3510003.3510160

[115] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. 2024. Bugs in Large Language Models Generated Code: An Empirical Study. arXiv:2403.08937 [cs.SE] https://arxiv.org/abs/2403.08937

[116] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[117] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. 2025. Kimi K2: Open Agentic Intelligence. arXiv:2507.20534 [cs.LG] https://arxiv.org/abs/2507.20534

[118] Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. 2025. CodeHalu: Investigating Code Hallucinations in LLMs via Execution-based Verification. arXiv:2405.00253 [cs.CL] https://arxiv.org/abs/2405.00253

[119] Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating Code Generation with Large Language Models. arXiv:2410.02184 [cs.LG] https://arxiv.org/abs/2410.02184

[120] Fumiya Uchiyama, Takeshi Kojima, Andrew Gambardella, Qi Cao, Yusuke Iwasawa, and Yutaka Matsuo. 2025. Which Programming Language and What Features at Pre-training Stage Affect Downstream Logical Inference Performance? arXiv:2410.06735 [cs.CL] https://arxiv.org/abs/2410.06735

[121] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[122] Alejandro Velasco, Daniel Rodriguez-Cardenas, Luftar Rahman Alif, David N. Palacio, and Denys Poshyvanyk. 2025. How Propense Are Large Language Models at Producing Code Smells? A Benchmarking Study. arXiv:2412.18989 [cs.SE] https://arxiv.org/abs/2412.18989

[123] Yuki Wakai, Toshiki Shibahara, Rina Okada, Takayuki Miura, and Hiroki Kinoshita. 2025. Understanding Privacy Risks of Large Language Models in Japanese Based on Training Data Extraction Attacks. In *Proceedings of the Workshop on Privacy in Large Language Models (LLM) and Natural Language*

Processing (NLP) 2025 (LM-SHIELD '25). Association for Computing Machinery, New York, NY, USA, 25–31. https://doi.org/10.1145/3709018.3736331

[124] Yao Wan, Guanghua Wan, Shijie Zhang, Hongyu Zhang, Pan Zhou, Hai Jin, and Lichao Sun. 2024. Does Your Neural Code Completion Model Use My Code? A Membership Inference Approach. arXiv:2404.14296 [cs.SE] https://arxiv.org/abs/2404.14296

[125] Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. 2025. LLMs Meet Library Evolution: Evaluating Deprecated API Usage in LLM-based Code Completion. arXiv:2406.09834 [cs.SE] https://arxiv.org/abs/2406.09834

[126] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2022. ReCode: Robustness Evaluation of Code Generation Models. arXiv:2212.10264 [cs.LG] https://arxiv.org/abs/2212.10264

[127] Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Mingzhi Mao, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models. arXiv:2407.00456 [cs.SE] https://arxiv.org/abs/2407.00456

[128] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 [cs.CL] https://arxiv.org/abs/2109.00859

[129] Yanlin Wang, Jiadong Wu, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Chong Wang, Ensheng Shi, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2026. DRAINCODE: Stealthy Energy Consumption Attacks on Retrieval-Augmented Code Generation via Context Poisoning. arXiv:2601.20615 [cs.SE] https://arxiv.org/abs/2601.20615

[130] Yanlin Wang, Ziyao Zhang, Chong Wang, Xinyi Xu, Mingwei Liu, Yong Wang, Jiachi Chen, and Zibin Zheng. 2026. RealSec-bench: A Benchmark for Evaluating Secure Code Generation in Real-World Repositories. arXiv:2601.22706 [cs.CR] https://arxiv.org/abs/2601.22706

[131] Zihan Wang, Xinzhang Liu, Yitong Yao, Chao Wang, Yu Zhao, Zhihao Yang, Wenmin Deng, Kaipeng Jia, Jiaxin Peng, Yuyao Huang, Sishi Xiong, Zhuo Jiang, Kaidong Yu, Xiaohui Hu, Fubei Yao, Ruiyu Fang, Zhuoru Jiang, Ruiting Song, Qiyi Xie, Rui Xue, Xuewei He, Yanlei Xue, Zhu Yuan, Zhaoxi Zhang, Zilu Huang, Shiquan Wang, Xin Wang, Hanming Wu, Mingyuan Wang, Xufeng Zhan, Yuhan Sun, Zhaohu Xing, Yuhao Jiang, Bingkai Yang, Shuangyong Song, Yongxiang Li, Zhongjiang He, and Xuelong Li. 2025. Technical Report of TeleChat2, TeleChat2.5 and T1. arXiv:2507.18013 [cs.CL] https://arxiv.org/abs/2507.18013

[132] Zige Wang, Wanjun Zhong, Yufei Wang, Qi Zhu, Fei Mi, Baojun Wang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Data management for large language models: A survey. *arXiv preprint arXiv:2312.01700* 1, 2.3 (2023).

[133] Alexander Wettig, Aatmik Gupta, Saumya Malik, and Danqi Chen. 2024. QuRating: Selecting High-Quality Data for Training Language Models. arXiv:2402.09739 [cs.CL] https://arxiv.org/abs/2402.09739

[134] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14)*. Association for Computing Machinery, New York, NY, USA, Article 38, 10 pages. https://doi.org/10.1145/2601248.2601268

[135] Haoze Wu, Yunzhi Yao, Wenhao Yu, and Ningyu Zhang. 2025. ReCode: Updating Code API Knowledge with Reinforcement Learning. arXiv:2506.20495 [cs.CL] https://arxiv.org/abs/2506.20495

[136] Yixi Wu, Pengfei He, Zehao Wang, Shaowei Wang, Yuan Tian, and Tse-Hsun Chen. 2024. A Comprehensive Framework for Evaluating API-oriented Code Generation in Large Language Models. arXiv:2409.15228 [cs.SE] https://arxiv.org/abs/2409.15228

[137] Wenjing Xing, Wenke Lu, Yeheng Duan, Bing Zhao, Zhenghui kang, Yaolong Wang, Kai Gao, and Lei Qiao. 2025. Infinite-Instruct: Synthesizing Scaling Code instruction Data with Bidirectional Synthesis and Static Verification. arXiv:2505.23177 [cs.CL] https://arxiv.org/abs/2505.23177

[138] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[139] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. 2024. Qwen2 Technical Report. arXiv:2407.10671 [cs.CL] https://arxiv.org/abs/2407.10671

[140] Kang Yang, Xinjun Mao, Shangwen Wang, Yanlin Wang, Tanghaoran Zhang, Bo Lin, Yihao Qin, Zhang Zhang, Yao Lu, and Kamal Al-Sabahi. 2025. Large Language Models are Qualified Benchmark Builders: Rebuilding Pre-Training Datasets for Advancing Code Intelligence Tasks. arXiv:2504.19444 [cs.SE] https://arxiv.org/abs/2504.19444

[141] Feng Yao, Zilong Wang, Liyuan Liu, Junxia Cui, Li Zhong, Xiaohan Fu, Haohui Mai, Vish Krishnan, Jianfeng Gao, and Jingbo Shang. 2025. Training Language Models to Generate Quality Code with Program Analysis Feedback. arXiv:2505.22704 [cs.CL] https://arxiv.org/abs/2505.22704

[142] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. arXiv:2304.10778 [cs.SE] https://arxiv.org/abs/2304.10778

[143] Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. WaveCoder: Widespread And Versatile Enhancement For Code Large Language Models By Instruction Tuning. arXiv:2312.14187 [cs.CL] https://arxiv.org/abs/2312.14187

[144] Zhiqiang Yuan, Yiling Bai, et al. 2024. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*.

[145] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.

[146] Junwei Zhang, Xing Hu, Shan Gao, Xin Xia, David Lo, and Shanping Li. 2025. Less is More: On the Importance of Data Quality for Unit Test Generation. arXiv:2502.14212 [cs.SE] https://arxiv.org/abs/2502.14212

[147] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2024. A Survey on Large Language Models for Software Engineering. arXiv:2312.15223 [cs.SE] https://arxiv.org/abs/2312.15223

[148] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Chen Xu, Yulong Chen, Longyue Wang, Anh Tuan Luu, Wei Bi, Freda Shi, and Shuming Shi. 2025. Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models. arXiv:2309.01219 [cs.CL] https://arxiv.org/abs/2309.01219

[149] Yuanliang Zhang, Yifan Xie, Shanshan Li, Ke Liu, Chong Wang, Zhouyang Jia, Xiangbing Huang, Jie Song, Chaopeng Luo, Zhizheng Zheng, Rulin Xu, Yitong Liu, Si Zheng, and Xiangke Liao. 2025. Unseen Horizons: Unveiling the Real Capability of LLM Code Generation Beyond the Familiar. arXiv:2412.08109 [cs.SE] https://arxiv.org/abs/2412.08109

[150] Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. 2024. MG-Verilog: Multi-grained Dataset Towards Enhanced LLM-assisted Verilog Generation. arXiv:2407.01910 [cs.LG] https://arxiv.org/abs/2407.01910

[151] Ziyao Zhang, Yanlin Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. 2025. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. arXiv:2409.20550 [cs.SE] https://arxiv.org/abs/2409.20550

[152] Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2024. Beyond Correctness: Benchmarking Multi-dimensional Code Generation for Large Language Models. arXiv:2407.11470 [cs.SE] https://arxiv.org/abs/2407.11470

[153] Wanwan Zheng and Mingzhe Jin. 2020. The effects of class imbalance and training data size on classifier learning: an empirical study. *SN Computer Science* 1, 2 (2020), 71.

[154] Xuanhe Zhou, Junxuan He, Wei Zhou, Haodong Chen, Zirui Tang, Haoyu Zhao, Xin Tong, Guoliang Li, Youmin Chen, Jun Zhou, Zhaojun Sun, Binyuan Hui, Shuo Wang, Conghui He, Zhiyuan Liu, Jingren Zhou, and Fan Wu. 2025. A Survey of LLM × DATA. arXiv:2505.18458 [cs.DB] https://arxiv.org/abs/2505.18458