An Automated Methodology for Generating Labeled Datasets of Semantic Errors in Code

Mahmoud Kassem
mahmoud.kassem@nyu.edu
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates

Francisco Ribeiro
francisco.ribeiro@nyu.edu
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates

Sarah Nadi
sarah.nadi@nyu.edu
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates

Abstract

Code generated by language models frequently contains subtle semantic errors that are difficult to detect with conventional static analysis and machine learning classifiers. Studying these errors requires specialized datasets that are laborious to create manually. To address this, we present an automated methodology for generating a labeled dataset of semantic errors based on an established taxonomy. Our approach uses GPT-4.1 to systematically introduce single, isolated semantic bugs into correct code from the HumanEval and BigCodeBench benchmarks. We also present a resulting artifact: a labeled dataset of Python code with semantic errors, designed to facilitate classification research.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; **Mutation testing**; *Software defect analysis*; • **Computing methodologies** → *Natural language processing*.

Keywords

Automated Software Engineering, Large Language Models, Semantic Errors, Fault Injection, Synthetic Datasets, Mutation Analysis, Prompt Engineering

ACM Reference Format:
Mahmoud Kassem, Francisco Ribeiro, and Sarah Nadi. 2026. An Automated Methodology for Generating Labeled Datasets of Semantic Errors in Code. In *3rd International Workshop on Large Language Models For Code (LLM4Code '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3786181.3788730

1 Introduction

Datasets of correct and incorrect code pairs [28, 30, 33, 65] are essential for developing and evaluating many software engineering tools. This data drives the creation of automated program repair (APR) tools [40], fault localization (FL) techniques [61], and code intelligence systems [5].

A common method for creating these datasets is by mining real developer fixes from software repositories [29, 31]. Undoubtedly, this method has created notable datasets that helped advance the state of the art in the above research areas. However, mining historic data from version control systems has some inherent problems. First, the resulting data is often noisy [21, 27, 63] due to tangled changes [22]. A bug-fix commit frequently contains more than only the bug fix, as it is often bundled with additional unrelated changes, like refactoring or logging. As an example, consider the bug-fix pair shown in Listing 1 taken from the *ManySStuBs4J* dataset [31, 32], where the bug fix (changing a condition to check for null) is combined with an unrelated change in functionality (converting a date value to a string). The presence of such unrelated modifications means that models trained on this data may learn to associate irrelevant patterns with specific bug types.

Listing 1: Noisy fix: identified by fixCommitSHA1 `aa90e04`

```
-  if(submittedNode == null || submittedNode.get("values") != null
       ) {
+  if(submittedNode == null || submittedNode.get("values") == null
       ) {
...
-  variables.put(field.getId(), dateValue);
+  variables.put(field.getId(), dateValue.toString("yyyy-M-d"));
```

Second, defect artifacts are hard to reproduce and keep executable [66]. Moreover, many of these pipelines are language- and toolchain-specific, which makes them hard to extend [27, 59]. As such, adding new examples, bug types, or languages is often impractical. Ultimately, this forces researchers to work with data that is often limited or static.

Recently, large language models (LLMs) have been used as powerful tools for code generation [10]. By leveraging the capability of many of these models to follow natural language instructions, researchers have recently leveraged LLMs to inject targeted errors in source code to improve mutation testing [53]. Inspired by this idea, we argue that such targeted mutations offer a more flexible approach to creating cleanly labeled single-fault executable datasets.

In this work, we propose an automated methodology to generate a labeled dataset of semantic errors by prompting an LLM to introduce specific bugs into correct code. Such an approach has the following advantages:

(1) It eliminates the noise of tangled changes: By design, the only difference between the correct and incorrect code is the single and intended bug.
(2) It is flexible: Our methodology is lightweight and automated. It can be re-run at any time to generate new examples, target different bug types, or support new languages without a complex mining and code analysis infrastructure.

Our primary contributions are:

(1) **An Automated Methodology for Dataset Generation:** We introduce a novel foundational technique that uses an LLM (in our case: GPT-4.1) to systematically inject single, isolated semantic errors into correct reference solutions.

(2) **A Labeled Dataset for Semantic Errors:** The primary output of our methodology is a new Python dataset of **1,217** labeled samples **spanning six semantic error categories.**[1] Each sample consists of a correct code solution paired with an incorrect, LLM-generated variant, labeled with a specific semantic error type.

While our current taxonomy focuses on six core error types, this work serves as a proof-of-concept for the viability of LLM-driven bug injection, establishing a baseline for future expansion into more complex defect categories.

Listing 2: Simplified structure of the LLM mutation prompt.

```
Introduce a single, natural-looking logical error into the given
      Python function.

**Task Description:** {task_desc}

**Correct Code:** {correct_code}

**Error Type to Introduce:** {error_type}
**Definition:** {error_type_description}
**Example:** {error_type_example}

**SPECIFIC GUIDANCE FOR THIS ERROR TYPE:**
<Dos and don'ts for this error type>

**CRITICAL REQUIREMENTS (Summarized):**
<General rules for subtlety, code integrity, and label fidelity>

[... The full prompt continues with more detailed examples of good
      /bad errors, specific rules to prevent applying the wrong
      change for a given error type, and a final validation
      checklist to guide the model's self-correction process ...]

**Output Format:**
Return your response as a JSON object with the keys: "mutated_code
      " and "explanation".
```

Overall, we argue that leveraging the code-generation capabilities of LLMs is an effective way to overcome the lack of available datasets. Our methodology allows us to rapidly generate large amounts of labeled data, overcoming the scalability barrier of traditional mining.

2 Methodology

This section details our methodology for the automated generation of a labeled dataset of semantic errors. Our approach is centered on a mutational strategy where we use an LLM to inject specific errors into correct code. We describe the source of our data, the prompt engineering techniques used to control the LLM, and the validation process to ensure the quality of the generated dataset.

2.1 Data Source

The foundation of our dataset are the two the well-established datasets *HumanEval* benchmark [10] and the hard subset of the *BigCodeBench* [68] benchmark. *HumanEval* provides a collection of 164 function-level programming problems, each accompanied by a natural language task description, a canonical correct solution in Python, and a test suite. Likewise, the hard subset of BigCodeBench provides 296 challenging programming tasks, each supplying complex natural-language instructions and a rigorous suite of test cases designed to evaluate a model's ability to utilize diverse function calls and perform compositional reasoning.

Both of these benchmarks serve as ideal sources, as they provide the necessary components — the task to incorporate in our prompt, the ground-truth code, and the means for verification — to generate and validate our dataset.

[1] Our dataset and supplemental materials are available at: https://github.com/sanadlab/auto-sem-gen

Table 1: Semantic error types considered by our dataset.

Error Type	Example of Incorrect Code Transformation
`incorrect_condition`	`if score > 50: -> if score >= 50:`
`off_by_one`	`for i in range(1, n): -> for i in range(n):`
`incorrect_variable_name`	`x, y = 5, 10; return x -> ...; return y`
`constant_value_error`	`TIMEOUT = 30 -> TIMEOUT = 60`
`incorrect_arthematic_operator`	`result = x + y -> result = x - y`
`incorrect_function_arguments`	`plot(x_data, y_data)-> plot(y_data, x_data)`

2.2 Mutational Approach and Prompt Engineering

Our data generation process adapts concepts from LLM-driven mutation testing, such as the LLMorpheus [53] tool, which uses models to inject faults for evaluating test suites. While LLMorpheus prompts for general buggy replacements at placeholder locations, our approach is specifically tailored for dataset generation by prompting GPT-4.1 to rewrite the function, deliberately introducing a single semantic error from a predefined taxonomy, which is a subset from the classification of LLM-generated code errors by Wang et al. [58]. Their taxonomy classifies errors along two dimensions: 13 semantic characteristics (describing the error's root cause) and 14 syntactic characteristics (describing the error's location). For this work, we selected a subset of six common semantic error types from their 13 semantic categories, which are well-suited for our mutational approach and listed in Table 1. The prompt template, summarized in Listing 2, explicitly instructs the model to inject a specific fault type (e.g., an off-by-one error or an incorrect logical operator) while preserving the syntactic correctness and the overall structure of the original code. This controlled injection of a single fault per solution is a key feature of our methodology, ensuring the resulting dataset provides unambiguous examples of each error type.

2.3 Validation

To ensure the integrity of the generated dataset, we employ a semi-automated validation process, which is illustrated in the final stages of the pipeline in Figure 1. This process consists of two key stages: automated testing and manual verification.

First, in the automated stage, each LLM-generated code variant is executed against the corresponding test suite (from *HumanEval* or

BigCodeBench). This step is crucial for filtering out equivalent mutants, variants that, despite syntactic changes, still produce correct outputs and pass all tests. Only variants that fail at least one test case are retained for the final dataset. We emphasize that the manual verification described in this section was conducted solely to validate the effectiveness of our methodology and establish a baseline of confidence. The generation technique itself is fully automated and requires no human intervention to produce new datasets.

Second, to verify the quality of the injected errors, we conducted a manual analysis on random samples from both datasets. This manual check verified two critical criteria: (1) that the injected bug precisely matched the requested semantic error type, and (2) that no other confounding, unintended errors were introduced. We adopted a consistent statistical standard for our sampling, aiming for a 90% confidence level with a 5% margin of error. For the *HumanEval* dataset, our manual analysis of 200 random samples out of 745 total entries yielded an accuracy of 90%. For the *BigCodeBench Hard* dataset, from its total population of 472 generated errors, this standard required a sample size of 173. Manual analysis of these 173 samples revealed an accuracy of 75.7%. This lower accuracy on *BigCodeBench Hard*, compared to *HumanEval*, is attributable to its higher task complexity. We observed that as the solution logic becomes more complex and less straightforward, it is more challenging for the LLM to inject a single, isolated semantic error without also introducing other confounding bugs.

This two-stage validation process gives us confidence in the overall label quality and provides a clear accuracy metric for each dataset.

3 The Generated Dataset

The primary artifact of our methodology is a labeled dataset of semantic errors. Listings 3 and 4 show a representative sample from *HumanEval* and the corresponding mutant generated using our methodology, while Listings 5 and 6 show a sample from *BigCodeBench* with its corresponding generated mutant.

Listing 3: Correct solution for HumanEval/8 (sum_product).

```python
def sum_product(numbers: List[int])
                -> Tuple[int, int]:
    sum_value = 0
    prod_value = 1
    for n in numbers:
        sum_value += n
        prod_value *= n
    return sum_value, prod_value
```

Listing 4: Incorrect solution for HumanEval/8, labeled as "Incorrect Arithmetic Operation".

```python
def sum_product(numbers: List[int])
                -> Tuple[int, int]:
    sum_value = 0
    prod_value = 1
    for n in numbers:
        sum_value += n
        prod_value += n # Bug: should be prod_value *= n
    return sum_value, prod_value
```

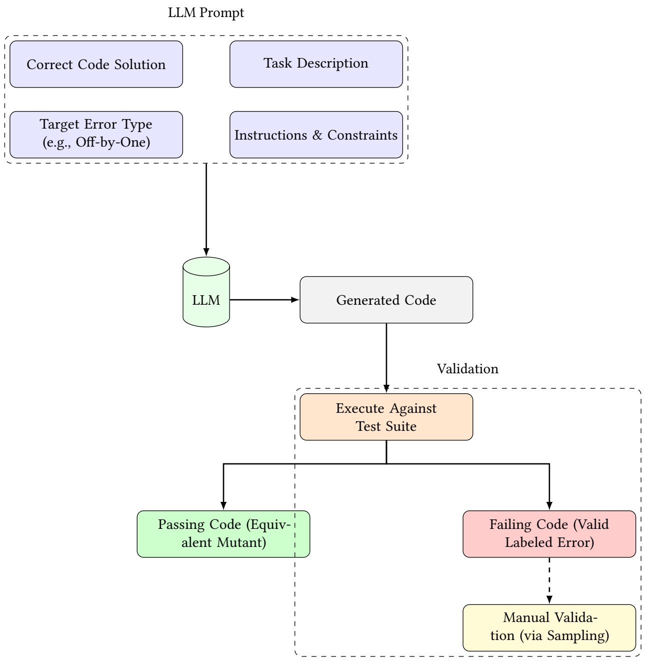
LLM Prompt
Correct Code Solution
Task Description
Target Error Type (e.g., Off-by-One)
Instructions & Constraints
LLM
Generated Code
Validation
Execute Against Test Suite
Passing Code (Equivalent Mutant)
Failing Code (Valid Labeled Error)
Manual Validation (via Sampling)

Figure 1: Pipeline for generating and validating a dataset sample. A formatted prompt is sent to the LLM. The output is tested to filter equivalent mutants (passing). The remaining valid errors (failing) are collected, and their label accuracy is then verified through manual sampling.

Listing 5: Correct solution for BigCodeBench/952 (task_func).

```python
def task_func(task_list, n_tasks,
              employees, seed=None):
    assignment_data = []
    for _ in range(n_tasks):
        task_name = random.choice(
            task_list).replace(" ", "_")
        employee = random.choice(employees)
        assignment_data.append(
            [task_name, employee, due_date])
    return pd.DataFrame(assignment_data,
        columns=["Task Name", "Assigned To",
                 "Due Date"])
```

Listing 6: Incorrect solution for BigCodeBench/952, labeled as "Constant Value Error".

```python
def task_func(task_list, n_tasks,
              employees, seed=None):
    assignment_data = []
    for _ in range(n_tasks):
        task_name = random.choice(
            task_list).replace(" ", "-") # Bug: should be "_"
        employee = random.choice(employees)
        assignment_data.append(
            [task_name, employee, due_date])
    return pd.DataFrame(assignment_data,
        columns=["Task Name", "Assigned To",
                 "Due Date"])
```

This section describes the structure and characteristics of this dataset. For accessibility and ease of use, the dataset is stored in JSON Lines format.

3.1 Dataset Schema

Each line in the dataset corresponds to an entry and is a self-contained JSON object representing a single programming task and its corresponding faulty code variant. Each JSON object contains the following key-value pairs:

- `task_description`: A string containing the complete natural language description of the programming task, as provided by the original *HumanEval* or *BigCodeBench* benchmarks.
- `correct_solution`: A string containing the canonical, correct Python code for the task from the reference benchmarks, *HumanEval* or *BigCodeBench*.
- `incorrect_solution`: A string containing the LLM-generated Python code, which includes a single, isolated semantic error.
- `error_type`: A string label specifying the class of the semantic error introduced in the `incorrect_solution`. Table 1 lists the possible error types.

We elaborate on the research opportunities enabled by this schema in Section 4.2.

3.2 Dataset Statistics

Our final dataset contains 1,217 samples, composed of 745 generated from the *HumanEval* benchmark and 472 from the *BigCodeBench Hard* benchmark. Table 2 provides a detailed distribution of these samples by their injected error type.

Table 2: Distribution of Error Types in the Dataset.

Error Type	Percentage	Count
Constant Value Error	15.9%	193
Incorrect Arithmetic Operation	18.4%	224
Incorrect Condition	11.5%	140
Incorrect Function Arguments	16.8%	205
Incorrect Variable Name	18.5%	225
Off-by-One Error	18.9%	230
Total	**100.0%**	**1,217**

The distribution of error types is relatively balanced, making the dataset suitable for training and evaluating multi-class classification models. We elaborate on this in Section 4.1.

4 Applications and Future Work

Our work in generating this dataset of fine-grained semantic errors is not an end in itself, but rather a foundation for further research and development. In this section, we first explore the immediate potential applications of the dataset as a new benchmark for code intelligence tools. We then outline several promising directions for future work, focusing on how our bug generation methodology can be extended and refined.

4.1 Potential Applications

As we noted in Section 3, the structure of the dataset artifact enables broader research into the nature of programming errors. By including the natural language task description, our dataset facilitates research into the root causes of semantic bugs. For example, researchers may explore:

- **Fine-Grained Error Classification:** The dataset's structure naturally lends itself to a classification problem, making it a fitting resource for training and evaluating supervised models (from simple classifiers to more complex neural architectures) to automatically classify the specific type of semantic error in a code snippet. This inherent fit suggests direct applications in IDEs and automated tutoring systems, providing developers and learners with more specific feedback beyond a simple "wrong answer".
- **Error Correlations:** Whether certain requirement patterns (e.g., boundary conditions) are more likely to result in specific bugs (e.g., off-by-one errors).
- **Program Repair:** Using the paired correct/incorrect solutions and error labels to develop targeted Automated Program Repair (APR) approaches.
- **Failure Analysis:** Studying the generated errors to better understand the scenarios more prone to failure in LLMs, informing the development of more reliable code models.

4.2 Future Work

Our work also establishes a foundation for several promising research directions where the methodology itself can be extended and refined. Future work could involve:

- **Expanding Scope and Generalizability:** Applying our mutational approach to other programming languages (e.g., Java, C++) and different source code benchmarks. This would test the generalizability of our technique and produce a more diverse collection of datasets. For instance, recent work like LLMorpheus [53] has also explored mutational strategies for languages such as JavaScript/TypeScript, albeit with a different goal (mutation testing), which highlights the broad utility of such methods.
- **Enriching the Error Taxonomy:** Expanding the taxonomy to include more complex semantic error types. This includes critical but hard-to-generate bugs such as concurrency issues (e.g., data races, deadlocks) [38] and incorrect API usage (e.g., violating call order, misusing parameters) [64], which remain significant challenges in software engineering.
- **Varying the Mutator LLM:** Experimenting with different foundational models (e.g., open-source models like Llama[55] or Mixtral[26]) to perform the bug injection. This is important for assessing how a model's underlying architecture and training data influence the diversity and realism of the generated bugs, as well as for exploring the feasibility of using more accessible, non-proprietary models for this task.
- **Improving Validation:** To address the lower accuracy observed in complex tasks (like *BigCodeBench*), future work could use iterative prompting or LLM-as-a-judge [16] to filter out unclear examples before manual review. Some code specific LLM-as-a-judge methods exist such as CodeJudge that is specifically designed for evaluating semantic code [54] or ICE-Score [67] which mainly overcomes the problem

of lacking test-suites, making the methodology applicable for code datasets with no provided test suites.

5 Threats to Validity

We identify several potential threats to the validity of our methodology and the resulting dataset.

Internal Validity: The primary threat to internal validity is the fidelity of the semantic error labels. Although our methodology is designed to generate specific error types, the stochastic nature of LLMs could lead to mislabeled examples (e.g., the model introducing a different bug than instructed). We mitigate this by using highly constrained prompts and performing a manual validation on a random sample, which confirmed 90% and 75.7% labeling accuracies, as described in Section 2.3. However, a small degree of label noise is an inherent limitation of this automated approach. Additionally, since all mutations were generated using a single model (GPT-4.1), there is a potential risk of model bias, where the dataset may over-represent specific error patterns favored by this model's training distribution or tend to produce repetitive variations of the same error type. Future work should explore ensuring distribution diversity by, for example, employing a wider range of LLMs for mutation.

External Validity: Our findings may have limited generalizability since our dataset is based on a single programming language (Python) and two data sources (*HumanEval* and *BigCodeBench Hard*). The distribution and nature of semantic errors might differ in other languages or in larger and more complex real-world software. Nevertheless, by using a well-established benchmark, our work provides a reproducible baseline that enables the community to extend this methodology to other contexts.

Construct Validity: The validity of our conclusions depends on our error taxonomy. While based on prior work [57], this taxonomy is a representative subset and may not be exhaustive. We acknowledge that it currently excludes complex categories such as concurrency issues, state-management defects, or security vulnerabilities. We position this work as a foundational proof-of-concept for validating the generative methodology; however, the lower validation accuracy observed on the *BigCodeBench* benchmark (75.7%) compared to *HumanEval* (90%) highlights the trade-off between task complexity and the difficulty of injecting isolated, precision errors without unintended side effects.

6 Related Work

Our work introduces a methodology for the automated generation of labeled datasets and is best understood in the context of two areas: (1) existing approaches for collecting bug data, and (2) the emerging study of errors in LLM-generated code.

6.1 Approaches to Bug Data Collection

The predominant method for creating bug datasets is to mine the version control history of open-source projects. Large-scale datasets like ManySStuBs4J [31] and TSSB-3M [47] have successfully applied this to collect millions of single-statement bug fixes. While these resources share our focus on fine-grained edits, they do not guarantee that a buggy version only represents a single behavioral fault. This ambiguity, which our introductory example in Listing 1 shows, introduces noise and hinders automated analysis. In contrast, our methodology takes a different route: it programmatically injects exactly one isolated semantic error into a known-correct solution and validates it via tests. This controlled, low-noise approach provides the precision needed to build tools that target the core of the error, rather than being misled by unrelated or confounding aspects that are tangled with a bug.

Other research recognizes the noise in mined commits and focuses on improving dataset precision. Techniques like Flexeme [43] and LLM-based detectors [42] untangle commits that mix multiple changes. BugMiner [49] isolates bug-inducing changes. These approaches attempt to clean up the noise inherent in mined data, while our generation method sidesteps this problem by producing single-fault instances by design.

Other datasets also aim for high confidence in the validity of their bugs. While some achieve this using test execution to validate bugs, others rely on different artifacts. For example, ReDef [41] uses reverted commits and PreciseBugCollector [20] uses bug-tracker information. While this improves label reliability, their focus differs from ours. They typically operate at a coarser granularity (classifying code as simply "defective" or "clean") and do not enforce our strict requirement for a single, isolated semantic error per sample.

Some approaches have learned mutation operators from real bug-fixes to create more realistic faults [56]. Recent work such as LLMorpheus [53] has also explored using LLMs for mutation testing in JavaScript and TypeScript. While our approach shares the underlying concept of LLM-driven mutation, it differs fundamentally in scope and application. LLMorpheus generates mutants to evaluate the quality of existing test suites, whereas our methodology targets the generation of a labeled dataset for training models. Additionally, our work currently focuses on Python, while LLMorpheus focuses on JS/TS. Similarly, mined datasets like ManySStuBs4J [31] focus on Java. However, unlike tool-specific static analysis, our prompt-driven methodology is inherently language-agnostic. This allows our approach to be adapted to other languages with relatively low effort. Our methodology can be seen as a specialized form of mutation generation, but our goal is not to evaluate an existing test suite. Instead, we use a constrained process to create a clean, labeled dataset. As studies on distribution shift have shown [19], such controlled synthetic datasets are valuable and complement datasets mined from real-world bugs.

6.2 Datasets for Analyzing LLM-Generated Code

In order to study the reliability of LLM-generated code, some studies analyze LLM outputs to create taxonomies of common errors [11, 14, 35, 51, 57]. These taxonomies show that LLM errors have patterns, a finding that is the basis for our work. However, these studies only categorize errors manually. They do not offer an automated way to generate a labeled dataset. Our methodology fills this gap by automating the creation of a structured dataset from an error taxonomy.

The buggy-HumanEval dataset [13] creates buggy code prefixes by flipping operators in correct solutions. Their goal is to test how LLMs complete code when the provided prefix already contains a bug. In contrast, our methodology generates a complete but incorrect function, not just a prefix, and provides an explicit label for the error type. Our goal is to create a labeled dataset, not to test code completion. Other benchmarks also use bug injection to evaluate LLM debugging. For instance, DebugBench [52] uses GPT-4 to implant diverse bugs, while MdEval [37] uses custom tools for multilingual faults. Their goal is to test general debugging. Our methodology is different: we use highly constrained prompts to inject a single, specific error type. This creates a dataset for classifying error types, not for evaluating debugging abilities.

7 Conclusion

This work makes two key contributions to the study of semantic errors in code. First, we introduced an automated and flexible methodology that leverages an LLM to systematically inject single, isolated semantic errors into correct code, guided by a predefined taxonomy. Second, using this methodology, we produced and validated a new, publicly available dataset of 1,217 labeled Python code snippets. This artifact, derived from the *HumanEval* and *BigCodeBench* benchmarks, spans six common semantic error categories and provides a clean, task-aligned resource for the research community.

The primary practical implication of our work is its ability to overcome the key limitations of traditional bug-data mining. By generating data by design, our approach avoids the "tangled change" problem, ensuring that each sample contains only the intended, labeled fault. This provides the community with a reliable benchmark for developing and evaluating a new generation of code intelligence tools. As discussed in Section 4.1, this dataset can directly facilitate research into fine-grained error classification, targeted program repair, and the analysis of LLM failure modes, enabling tools that offer more specific, actionable feedback than a simple pass/fail verdict.

Ultimately, this work establishes a new foundation for creating high-quality, labeled data for code analysis. Future efforts can extend this methodology to other programming languages, richer error taxonomies, and different code benchmarks, further accelerating research into automated debugging and code reliability.

References

[1] Altaf Allah Abbassi, Leuson Da Silva, Amin Nikanjam, and Foutse Khomh. 2025. Unveiling Inefficiencies in LLM-Generated Code: Toward a Comprehensive Taxonomy. arXiv:2503.06327 [cs.SE] doi:10.48550/arXiv.2503.06327

[2] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. 2001. On the Surprising Behavior of Distance Metrics in High Dimensional Space. In *Database Theory — ICDT 2001*. Springer-Verlag Berlin Heidelberg, 420–434.

[3] E. Akimova, I. Klyuchnikov, N. Sorokin, et al. 2021. A Survey on Software Defect Prediction Using Deep Learning. *Mathematics* 9, 24 (2021), 3281.

[4] Elena N. Akimova, Alexander Yu. Bersenev, Artem A. Deikov, Konstantin S. Kobylkin, Anton V. Konygin, and Ilya P. Mezentsev. 2021. PyTraceBugs: A Large Python Code Dataset for Supervised Machine Learning in Software Defect Prediction. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Taipei, Taiwan, 113–123. doi:10.1109/APSEC53868.2021.00022

[5] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (July 2018), 37 pages. doi:10.1145/3212695

[6] M.F.I. Amin, M.M.H. Nibir, M.M. Rahman, and S. Sharmin. 2023. Multi-label Code Error Classification Using CodeT5 and ML-KNN. In *2023 5th International Conference on Sustainable Technologies for Industry 5.0 (STI)*. IEEE, Dhaka, Bangladesh, 1–6. doi:10.1109/STI59943.2023.10255392

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL] doi:10.48550/arXiv.2108.07732

[8] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 8 (2013), 1798–1828. doi:10.1109/TPAMI.2013.50

[9] Satish Chandra and Maxim Tabachnyk. 2024. AI in Software Engineering at Google: Progress and the Path Ahead. Google Research Blog. https://research.google/blog/ai-in-software-engineering-at-google-progress-and-the-path-ahead/

[10] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] doi:10.48550/arXiv.2107.03374

[11] QiHong Chen, Jiachen Yu, Jiawei Li, Jiecheng Deng, Justin Tian Jin Chen, and Iftekhar Ahmed. 2024. A Deep Dive Into Large Language Model Code Generation Mistakes: What and Why? arXiv:2411.01414 [cs.SE]

[12] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. arXiv:2304.05128 [cs.CL] doi:10.48550/arXiv.2304.05128

[13] Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. 2023. Large language models of code fail at completing code with potential bugs. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1794, 27 pages.

[14] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, Yan Liu, Enyu Zhou, Ming Zhang, Yuhao Zhou, Yueming Wu, Rui Zheng, Ming Wen, Rongxiang Weng, Jingang Wang, Xunliang Cai, Tao Gui, Xipeng Qiu, Qi Zhang, and Xuanjing Huang. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. arXiv:2407.06153 [cs.SE] https://arxiv.org/abs/2407.06153

[15] Zhangyin Feng, Daya Guo, Duyu Tang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[16] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. 2025. A Survey on LLM-as-a-Judge. arXiv:2411.15594 [cs.CL] https://arxiv.org/abs/2411.15594

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225. doi:10.18653/v1/2022.acl-long.508

[18] D. Guo, S. Ren, S. Lu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[19] Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. 2022. On Distribution Shift in Learning-based Bug Detectors. arXiv:2204.10049 [cs.LG] https://arxiv.org/abs/2204.10049

[20] Ye He, Zimin Chen, and Claire Le Goues. 2023. PreciseBugCollector: Extensible, Executable and Precise Bug-Fix Collection: Solution for Challenge 8: Automating Precise Data Collection for Code Snippets with Bugs, Fixes, Locations, and Types. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1899–1910. doi:10.1109/ASE56229.2023.00163

[21] Steffen Herbold, Alexander Trautsch, and Benjamin Ledel. 2020. Large-Scale Manual Validation of Bugfixing Changes. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 611–614. doi:10.1145/3379597.3387504

[22] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 121–130.

[23] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04) (OOPSLA '04)*. ACM, 22–26. doi:10.1145/1028664.1028668

[24] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. arXiv:2303.18184 [cs.SE] doi:10.48550/arXiv.2303.18184

[25] Yuheng Huang, Lei Ma, Keizaburo Nishikino, and Takumi Akazaki. 2025. Risk Assessment Framework for Code LLMs via Leveraging Internal States. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE '25)*. ACM, Trondheim, Norway. doi:10.1145/3696630.3728566

[26] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] https://arxiv.org/abs/2401.04088

[27] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 686–698. doi:10.1109/ICSE43902.2021.00069

[28] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[29] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, San Jose, CA, USA, 437–440. doi:10.1145/2610384.2628045

[30] Vinay Kabadi, Dezhen Kong, Siyu Xie, Lingfeng Bao, Gede Artha Azriadi Prana, Tien-Duy B. Le, Xuan-Bach D. Le, and David Lo. 2023. The Future Can't Help Fix The Past: Assessing Program Repair In The Wild. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 50–61. doi:10.1109/ICSME58846.2023.00017

[31] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. ACM, Seoul, Republic of Korea, 358–362. doi:10.1145/3379597.3387491

[32] Rafael Michael Karampatsis and Charles Sutton. 2020. *ManySStuBs4J Dataset*. doi:10.5281/zenodo.3653444

[33] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256. doi:10.1109/TSE.2015.2454513

[34] Meir M. Lehman. 1979. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1, 3 (jul 1979), 213–221. doi:10.1016/0164-1212(79)90022-0

[35] Xiaoli Lian, Shuaisong Wang, Jieping Ma, Fang Liu, Xin Tan, Li Zhang, Lin Shi, and Cuiyun Gao. 2024. Uncovering Weaknesses in Neural Code Generation. arXiv:2407.09793 [cs.SE] https://arxiv.org/abs/2407.09793

[36] 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. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=vTfV3l2vLg

[37] Shukai Liu, Linzheng Chai, Jian Yang, Jiajun Shi, He Zhu, Liran Wang, Ke Jin, Wei Zhang, Hualei Zhu, Shuyue Guo, Tao Sun, Jiaheng Liu, Yunlong Duan, Yu Hao, Liqun Yang, Guanglin Niu, Ge Zhang, and Zhoujun Li. 2025. MdEval: Massively Multilingual Code Debugging. arXiv:2411.02310 [cs.CL] https://arxiv.org/abs/2411.02310

[38] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 329–339. doi:10.1145/1346281.1346323

[39] Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. 2023. The LMs: Understanding Code Syntax and Semantics for Code Analysis. arXiv:2305.12138 [cs.LG]

[40] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report hal-01956501. HAL/archives-ouvertes.fr.

[41] Doha Nam, Taehyoun Kim, Duksan Ryu, and Jongmoon Baik. 2025. Probing Pre-trained Language Models on Code Changes: Insights from ReDef, a High-Confidence Just-in-Time Defect Prediction Dataset. arXiv:2509.09192 [cs.SE] https://arxiv.org/abs/2509.09192

[42] Md Nahidul Islam Opu, Shaowei Wang, and Shaiful Chowdhury. 2025. LLM-Based Detection of Tangled Code Changes for Higher-Quality Method-Level Bug Datasets. arXiv:2505.08263 [cs.SE] https://arxiv.org/abs/2505.08263

[43] Profir-Petru Pârundefinedachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2020. Flexeme: untangling commits using lexical flows. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/3368089.3409693

[44] Shaoming Qiu, Bicong E, Jingjie He, and Liangyu Liu. 2024. Survey of Software Defect Prediction Features. *Neural Computing and Applications* 37, 4 (dec 2024), 2113–2144. doi:10.1007/s00521-024-10937-1

[45] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 2836–2847. doi:10.18653/v1/2020.findings-emnlp.255

[46] Henry Gordon Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. doi:10.2307/1990888

[47] Cedric Richter and Heike Wehrheim. 2022. TSSB-3M: mining single statement bugs at massive scale. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 418–422. doi:10.1145/3524842.3528505

[48] Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. 2024. Using AI-Based Coding Assistants in Practice: State of Affairs, Perceptions, and Ways Forward. arXiv:2406.07765 [cs.SE]

[49] Xuezhi Song, Yijian Wu, Junming Cao, Bihuan Chen, Yun Lin, Zhengjie Lu, Dingji Wang, and Xin Peng. 2023. BugMiner: Automating Precise Bug Dataset Construction by Code Evolution History Mining. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1919–1929. doi:10.1109/ASE56229.2023.00201

[50] 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] doi:10.48550/arXiv.2403.08937

[51] Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. 2025. Bugs in large language models generated code: an empirical study. *Empirical Softw. Engg.* 30, 3 (Feb. 2025), 48 pages. doi:10.1007/s10664-025-10614-4

[52] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Hui Haotian, Liu Weichuan, Zhiyuan Liu, and Maosong Sun. 2024. DebugBench: Evaluating Debugging Capability of Large Language Models. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 4173–4198. doi:10.18653/v1/2024.findings-acl.247

[53] Frank Tip, Jonathan Bell, and Max Schäfer. 2025. LLMorpheus: Mutation Testing Using Large Language Models. *IEEE Transactions on Software Engineering* 51, 6 (2025), 1645–1665. doi:10.1109/TSE.2025.3562025

[54] 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

[55] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[56] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning How to Mutate Source Code from Bug-Fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 301–312. doi:10.1109/ICSME.2019.00046

[57] Z. Wang, H. Zhang, L. Zhang, et al. 2024. RFCScan: An Autonomous Agent for Detecting Functional Bugs in Network Protocol Implementations. *arXiv preprint arXiv:2405.08447* (2024).

[58] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2025. Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE '25)*. IEEE/ACM.

[59] Hiroya Watanabe, Masanari Kondo, Eunjong Choi, and Osamu Mizuno. 2024. Benefits and Pitfalls of Token-Level SZZ: An Empirical Study on OSS Projects. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 776–786. doi:10.1109/SANER60148.2024.00084

[60] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, et al. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/Fse '20)*. ACM, 1556–1560. doi:10.1145/3368089.3417943

[61] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. doi:10.1109/TSE.2016.2521368

[62] Angus Yang, Zehan Li, and Jie Li. 2024. Advancing GenAI Assisted Programming–A Comparative Study on Prompt Efficiency and Code Quality Between GPT-4 and GLM-4. arXiv:2402.12782 [cs.SE] doi:10.48550/arXiv.2402.12782

[63] Deheng Yang, Yan Lei, Xiaoguang Mao, David Lo, Huan Xie, and Meng Yan. 2021. Is the Ground Truth Really Accurate? Dataset Purification for Automated Program Repair. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 96–107. doi:10.1109/SANER50967.2021.00018

[64] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. 318–343. doi:10.1007/978-3-642-03013-0_15

[65] Hao-Nan Zhu, Robert M. Furth, Michael Pradel, and Cindy Rubio-González. 2025. From Bugs to Benchmarks: A Comprehensive Survey of Software Defect Datasets. *CoRR* abs/2504.17977 (2025). https://arxiv.org/abs/2504.17977

[66] Hao-Nan Zhu and Cindy Rubio-González. 2023. On the Reproducibility of Software Defect Datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2324–2335. doi:10.1109/ICSE48619.2023.00195

[67] Terry Yue Zhuo. 2024. ICE-Score: Instructing Large Language Models to Evaluate Code. arXiv:2304.14317 [cs.AI] https://arxiv.org/abs/2304.14317

[68] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, et al. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *International Conference on Learning Representations (ICLR)*.