

OPENCSSGEN: A Large-Scale Corpus of Real-World Python Classes for LLM Research

Musfiqur Rahman
Concordia University
Montréal, Canada

musfiqur.rahman@mail.concordia.ca

SayedHassan Khatoonabadi
Concordia University
Montréal, Canada

sayedhassan.khatoonabadi@concordia.ca

Emad Shihab
Concordia University
Montréal, Canada

emad.shihab@concordia.ca

Abstract

Existing class-level code generation datasets are either synthetic (ClassEval: 100 classes) or insufficient in scale for modern training needs (RealClassEval: 400 classes), hindering robust evaluation and empirical analysis. We present **OpenClassGen**, a large-scale corpus of 324,843 Python classes extracted from 2,970 engineered open-source projects. Each entry pairs a human-written class with its corresponding skeleton, which comprises class and method signatures with associated docstrings, and is enriched with 27 static code metrics covering complexity, coupling, cohesion, and inheritance properties. Unlike prior benchmarks that require repository-level context resolution, OpenClassGen provides self-contained class skeletons that serve as complete generation specifications. We demonstrate the corpus's utility by evaluating three LLMs (GPT-4-mini, Claude-4-Sonnet, Qwen-3-Coder) on a curated, executable subset of 300 classes, enriched with test suites achieving 58% branch coverage. Results show strong semantic similarity (CodeBERTScore-F3: 0.89) but moderate functional correctness (pass rate: 0.33), with substantial variance across models. This variance, along with diverse class characteristics, confirms that OpenClassGen enables meaningful differentiation of LLM capabilities. The dataset supports diverse use cases, including fine-tuning, retrieval-augmented generation, difficulty modelling, and failure mode analysis. The complete dataset and curation scripts are publicly available at <https://zenodo.org/records/18409150>.

CCS Concepts

• **Software and its engineering** → **Software libraries and repositories**; *Object oriented languages*; • **Computing methodologies** → **Natural language processing**.

Keywords

Large Language Models, Code Generation, Software Repository Mining

ACM Reference Format:

Musfiqur Rahman, SayedHassan Khatoonabadi, and Emad Shihab. 2026. OPENCSSGEN: A Large-Scale Corpus of Real-World Python Classes for

LLM Research. In *Proceedings of The 30th International Conference on Evaluation and Assessment in Software Engineering (EASE 2026)*. ACM, New York, NY, USA, 10 pages. <https://doi.org/XXXXXXXX.XXXXXXX>

1 Introduction

Recent advancements in large language models (LLMs) have led to the development of various code generation benchmarks [8, 9, 12, 17, 19]. However, these benchmarks predominantly focus on function-level code generation, evaluating LLMs on isolated, self-contained functions. At the other extreme, repository-level benchmarks such as RepoClassBench [13] and JavaBench [11] situate code generation within full project contexts, requiring complex agentic architectures to navigate codebases and retrieve relevant dependencies. This leaves an important middle ground underexplored: class-level code generation, where the structural complexity of object-oriented design is present, but the task remains tractable without heavyweight retrieval infrastructure.

Recent evidence suggests that this middle ground is particularly valuable. RepoClassBench reports up to 29% pass rate in Python class generation with agentic architectures that have full repository access, while RealClassEval [27] achieves comparable rates (25–34%) without agents or repository-level context using only class skeletons. They also find that adding RAG-based retrieval yields up to a 7 percentage point increase in functional correctness. This suggests that well-designed class skeletons capture sufficient structural information for effective code generation. This offers a favourable trade-off between task complexity and infrastructure requirements. However, existing class-level datasets remain limited. ClassEval [14] provides 100 manually crafted Python classes, but its synthetic nature limits generalizability. For example, LLMs achieve 84–89% correctness on ClassEval but only 25–34% on real-world code [27]. RealClassEval addresses realism by curating 400 classes from open-source projects, but this scale remains insufficient for training, fine-tuning, or large-scale empirical studies.

To address this gap, we present **OpenClassGen**: a large-scale corpus of 324,843 Python classes extracted from 2,970 engineered open-source projects. Each entry consists of a human-written class paired with its corresponding skeleton, comprising the class and method signatures with associated docstrings. We enrich the dataset with 27 static code metrics covering complexity, coupling, cohesion, and inheritance properties. Unlike prior datasets, OpenClassGen is designed not as a fixed benchmark but as a research infrastructure supporting multiple use cases:

- **Fine-tuning and training:** 324,843 (skeleton, implementation) pairs provide a supervised learning signal for class-level code generation.

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.

EASE 2026, Glasgow, Scotland, United Kingdom

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXXXX.XXXXXXX>

- **Retrieval-augmented generation (RAG):** Structural metadata enables similarity-based retrieval of relevant examples during generation.
- **Difficulty modelling:** Static metrics can be correlated with LLM performance to predict challenging class characteristics.
- **Empirical studies:** The scale supports statistically robust analysis of LLM failure modes, documentation effects, and architectural patterns.

This paper makes the following contributions:

- (1) **OpenClassGen**, the first large-scale corpus of real-world Python classes specifically curated for LLM-assisted class-level code generation research.
- (2) A rigorous curation pipeline with engineering-based project filtering, AST-based skeleton extraction, and test class removal.
- (3) An evaluation demonstrating dataset utility across three LLMs (GPT-o4-mini, Claude-4-Sonnet, Qwen-3-Coder) and four quality dimensions (syntactic, semantic, structural, functional).
- (4) Public release of the dataset and replication scripts to support reproducibility and future research (See Data Availability section).

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 details the curation process. Section 4 characterizes the dataset. Section 5 presents our evaluation. Sections 6 and 7 discuss implications and limitations. Section 8 concludes the paper.

2 Related Work

We review existing benchmarks for LLM-based code generation, focusing on the progression from function-level to class-level evaluation. We distinguish between *class-level generation*, which concerns the granularity of the generated artifact (producing a complete class as the unit of output), and *repository-level generation*, which concerns the context required for generation (resolving cross-file dependencies, imports, and project-specific configurations). While related, these dimensions are orthogonal: repository-level benchmarks may generate functions or methods that depend on external context, whereas class-level benchmarks focus on producing structurally complete classes regardless of external dependencies.

2.1 Function-Level Benchmarks

HumanEval [12] is a widely used benchmark consisting of 164 hand-written Python programming problems with function signatures and test cases. It introduced Pass@k as an evaluation metric, measuring how often a model generates a correct solution within k attempts. MBPP [9] extends function-level evaluation with 974 Python programs, each paired with natural language descriptions and test cases. Multi-HumanEval [8] builds on HumanEval by introducing multilingual code generation benchmarks covering over 10 programming languages. CoderEval [35] addresses context dependence by including non-standalone functions that rely on surrounding code, such as variables, functions, or classes within the same file or project. HumanEvo [37] addresses a different dimension of realism: temporal evolution. By rolling back repositories to the commit state before target functions were written, HumanEvo avoids future context leakage and reveals that evolution-ignored evaluation inflates LLM performance by 10–61%.

While these benchmarks have advanced LLM evaluation for code generation, they remain limited to function-level tasks. Functions in isolation do not capture the complexity of real-world software, where code typically depends on class attributes, inheritance hierarchies, and project-specific dependencies.

2.2 Class-Level and Repository-Level Benchmarks

ClassEval [14] introduced the first benchmark specifically designed for class-level code generation, containing 100 manually crafted Python classes with test suites achieving 98.2% branch coverage. The study revealed that LLMs perform significantly worse on class-level tasks compared to function-level benchmarks. For instance, GPT-4 achieved only 37% holistic pass rate. The authors found that holistic generation (generating the full class at once) works better for advanced models, while incremental generation benefits weaker models. Yuen et al. [36] extended this work, demonstrating that chain-of-thought prompting improves functional correctness by up to 25%.

RealClassEval [27] addressed the realism gap by curating 400 classes from GitHub repositories. The study found that LLMs achieve 84–89% correctness on synthetic benchmarks but only 25–34% on real-world code, with no significant generalization gap between seen and unseen repositories. Error analysis revealed that `AttributeError`, `TypeError`, and `AssertionError` dominate real-world failures. While RealClassEval advances ecological validity, its 400-sample scale remains insufficient for training, fine-tuning, or large-scale empirical analysis.

Adjacent to class-level generation, several benchmarks target repository-level code generation, where challenges involve resolving cross-file dependencies and navigating project context. JavaBench [11] addresses the language imbalance in existing benchmarks, noting that 95.8% of code generation benchmarks involve Python. It comprises 4 Java projects originally designed as undergraduate course assignments, with 389 methods across 106 classes targeting object-oriented features such as encapsulation, inheritance, and polymorphism. RepoClassBench [13] extends evaluation to repository contexts across multiple languages (Java, Python, and C#), where each class has cross-file dependencies and corresponding test cases. However, this benchmark requires sophisticated agentic architectures to retrieve relevant context and a dependency-resolution infrastructure, both of which add complexity and cost to evaluation.

2.3 Positioning OpenClassGen

Table 1 compares OpenClassGen against existing datasets that target class-level or repository-level code generation. We include repository-level benchmarks (JavaBench, RepoClassBench) because they also address object-oriented code structures, though they differ in granularity and context requirements.

OpenClassGen fills a critical gap by providing two orders of magnitude more samples than prior class-level datasets while maintaining real-world origin. Unlike synthetic benchmarks (ClassEval), our classes are extracted from production repositories with varying documentation quality. Unlike existing real-world class-level

Table 1: Comparison of class-level and repository-level code generation datasets.

Dataset	Classes	Projects	Real-World
ClassEval [14]	100	—	✗
RealClassEval [27]	400	12	✓
JavaBench [11]	106	4	✗
RepoClassBench [13]	130	26	✓
OpenClassGen	324,843	2,970	✓

datasets (RealClassEval), our scale supports fine-tuning, retrieval-augmented generation, and statistically robust empirical studies. Unlike repository-level benchmarks, OpenClassGen isolates the class-level synthesis problem: as noted in Section 1, RealClassEval achieves pass rates comparable to RepoClassBench’s agentic approach using only class skeletons, suggesting that well-designed skeletons capture sufficient context for effective generation. Our corpus enables researchers to study this core challenge without the confounding factors of repository navigation. Additionally, we provide 27 static code metrics per class, enabling research on difficulty modelling and structural analysis not possible with existing datasets.

3 Dataset Curation

This section describes the process followed to curate OpenClassGen. Our corpus targets Python due to its widespread adoption in both industry and open-source communities [3], LLMs’ strongest code generation performance on this language [12], and alignment with existing benchmarks, over 95% of which target Python [11]. Figure 1 provides an overview of the curation pipeline.

3.1 Project Selection

We build our dataset on existing filtered projects shared in [27]. These projects were selected based on “engineered project” [23] criteria established by Xiao et al. [33] to identify well-maintained, production-quality projects. Three categories of filters were applied:

- (1) **License filtering:** Repositories without licenses or with non-standard licenses were excluded, including those with Creative Commons licenses and the SIL Open Font License 1.1, which are not typically used for software projects.
- (2) **Activity filtering:** Repositories without any releases, with fewer than two contributors, or marked as archived were removed.
- (3) **Quality filtering:** The distribution of repository properties (pull requests, issues, and lines of code) was analyzed, and repositories in the first quartile for each metric were excluded. Additionally, repositories outside the 97% confidence interval for code-to-comment ratio were excluded, as engineered software projects are typically documented with code comments.

This process yielded 2,970 repositories spanning diverse domains, including web frameworks, machine learning libraries, and data processing utilities.

3.2 Repository Analysis

After identifying the projects, we cloned each repository, preserving the complete structure and dependencies of the codebases. This step ensured that all relevant files were available for analysis and maintained the integrity of the original repositories while enabling large-scale processing.

To characterize class complexity, we computed 27 static code metrics per class using Understand™ by SciTools [6]. These metrics span four categories:

- **Size metrics:** Lines of code, comment lines, blank lines, number of methods.
- **Complexity metrics:** Cyclomatic complexity, essential complexity, maximum nesting depth.
- **Coupling metrics:** Fan-in, fan-out, number of dependencies.
- **Inheritance metrics:** Depth of inheritance tree, number of children, coupling between objects.

This analysis identified 434,518 Python classes across the 2,970 repositories.

3.3 Class Extraction

We parse all .py files using Python’s ast module [1] to extract class skeletons, comprising class signatures, method signatures, and docstrings (when present), with method bodies replaced by pass statements.

Listings 1 and 2 illustrate the extraction: the skeleton preserves structural information and documentation while removing implementations.

Listing 1: Human-written class from nuagenetworks/monolith [2].

```

1 class TaskManager(object):
2     """Multi threading manager"""
3     def __init__(self):
4         """Initializes a TaskManager"""
5         self.threads = list()
6
7     def wait_until_exit(self):
8         """Wait until all threads are finished."""
9         [t.join() for t in self.threads]
10        self.threads = list()
11
12    def start_task(self, method, *args, **kwargs):
13        """Start a task in a separate thread
14        Args:
15            method: the method to start
16            args: Accept args/kwargs arguments
17        """
18        thread = threading.Thread(
19            target=method, args=args, kwargs=kwargs)
20        thread.is_daemon = False
21        thread.start()
22        self.threads.append(thread)

```

Listing 2: Extracted class skeleton.

```

1 class TaskManager(object):
2     """Multi threading manager"""
3     def __init__(self):
4         """Initializes a TaskManager"""
5         pass
6
7     def wait_until_exit(self):
8         """Wait until all threads are finished."""
9         pass
10
11    def start_task(self, method, *args, **kwargs):

```

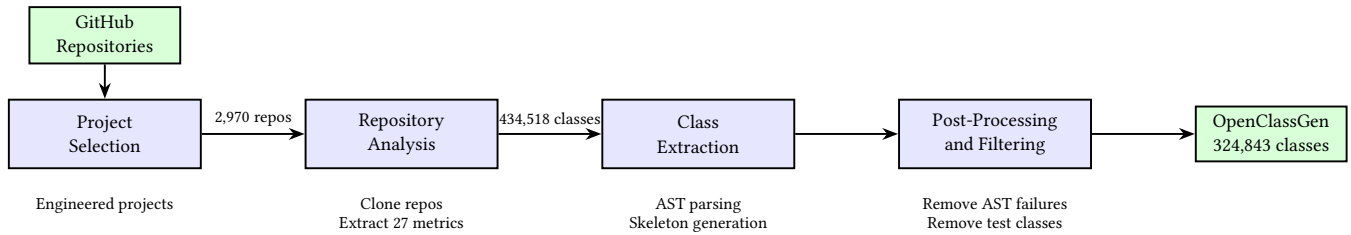


Figure 1: Overview of the OpenClassGen curation pipeline. Starting from 2,970 engineered GitHub repositories, we extract classes via AST parsing, generate skeletons, and apply filtering to produce 324,843 production Python classes.

```

12 """Start a task in a separate thread
13 Args:
14     method: the method to start
15     args: Accept args/kwargs arguments
16 """
17 pass
  
```

3.4 Post-Processing and Filtering

After extracting class skeletons, we performed post-processing to refine the dataset:

- (1) **AST failure removal:** Classes that failed AST parsing were removed, reducing the dataset by 3,557 classes. Qualitative analysis revealed these typically involved Python 2 syntax (e.g., `print "..."` without parentheses), incompatible with Python 3’s `ast` module.
- (2) **Test class removal:** Test classes were identified and removed using a multi-pattern heuristic on class names and file paths: class names starting or ending with “Test” or “Case”, or located in paths containing “test(s)” or “unit”. Test classes were excluded because they represent a fundamentally different generation task—they verify behavior rather than implement functionality, follow rigid framework-specific patterns (e.g., `unittest`, `pytest`), and depend on the classes they test. This step removed 106,118 classes. This ensures OpenClassGen focuses on production code generation, aligning with the scope of existing class-level benchmarks [14, 27].

The final dataset comprises 324,843 production Python classes.

3.5 Dataset Structure

For each class, we record the fields described in Table 2. The complete list of 27 static code metrics is available in the replication package.

Table 3 presents summary statistics for the curated dataset. Classes vary substantially in size and documentation coverage, reflecting the diversity of real-world software. The median class contains 24 lines of code with 1 method, while larger classes exceed 200 lines with complex method structures.

4 Dataset Characterization

To assess the quality and diversity of OpenClassGen, we analyze the dataset along three dimensions: documentation coverage, structural composition, and domain diversity.

Table 2: Data dictionary for OpenClassGen.

Field	Description
<code>id</code>	Unique index for each class.
<code>repository_name</code>	Name of the GitHub repository.
<code>file_path</code>	Path to the file containing the class.
<code>class_name</code>	Name of the class.
<code>human_written_class</code>	Complete class implementation with docstrings.
<code>class_skeleton</code>	Extracted skeleton with method bodies replaced by <code>pass</code> .
<code>comment_to_code_ratio</code>	Ratio of comment lines to code lines.
<code>total_program_units</code>	Number of classes and methods in the skeleton.
<code>total_doc_str</code>	Number of program units with docstrings.

Table 3: Summary statistics of OpenClassGen (324,843 classes).

Property	Avg.	Std.	25%	Med.	75%
Method Count	3.52	7.36	0.00	1.00	4.00
Total Lines	80.20	253.92	8.00	24.00	72.00
Code Lines	50.19	160.74	6.00	15.00	44.00
Comment Lines	20.86	102.85	0.00	3.00	15.00
Blank Lines	10.93	33.50	1.00	3.00	10.00
Comment-to-Code Ratio	0.53	1.71	0.00	0.22	0.63

4.1 Documentation Coverage

Documentation quality directly impacts LLM performance on code generation tasks, as docstrings provide natural language guidance for implementation. We categorize classes by documentation level:

- **Fully documented:** All program units (class and methods) have docstrings.
- **Partially documented:** Some but not all program units have docstrings.
- **Undocumented:** No program units have docstrings.

Figure 2 shows the distribution across these categories. Approximately 20% of classes are fully documented, 44% are partially documented, and 36% have no docstrings. This distribution reflects real-world software practices, where documentation coverage varies

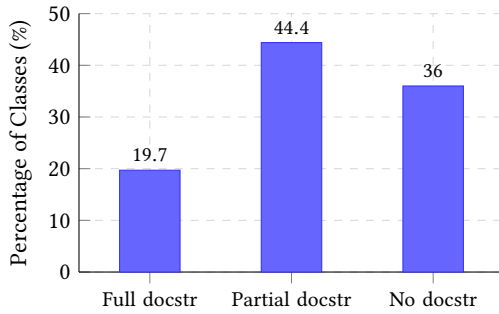


Figure 2: Distribution of classes by documentation coverage in OpenClassGen.

substantially across projects and developers [16]. Unlike synthetic benchmarks such as ClassEval, which provide complete and well-structured docstrings for all classes, OpenClassGen captures the documentation variability that LLMs encounter in production environments.

The dataset includes fields `total_program_units` and `total_doc_str` for each class, enabling researchers to filter by documentation level. For instance, selecting classes where `total_program_units` equals `total_doc_str` yields 63,971 fully documented classes, which in and of itself is significantly larger than existing class-level benchmarks.

4.2 Structural Composition

Classes in OpenClassGen exhibit substantial structural diversity. Table 3 shows that the median class contains 1 method and 24 lines of code, while the 75th percentile reaches 4 methods and 72 lines. This range spans simple data containers and utility classes to complex stateful managers with extensive method interactions.

We observe the following structural patterns:

- **Data classes:** Classes with few or no methods, primarily defining attributes. These constitute approximately 28% of the dataset (classes with zero methods).
- **Utility classes:** Small classes with 1–3 focused methods providing specific functionality. These represent approximately 44% of the dataset.
- **Manager classes:** Larger classes with 4+ methods managing state and coordinating complex operations. These comprise approximately 28% of the dataset.

The 27 static metrics enable fine-grained filtering by structural properties. For example, researchers studying inheritance can filter by depth of the inheritance tree; those investigating complexity can stratify by cyclomatic complexity.

4.3 Domain Diversity

OpenClassGen spans 2,970 repositories covering diverse application domains. Based on qualitative analysis of the repositories, the dataset includes cloud and infrastructure tooling (14%), scientific computing (8%), data processing and ETL (6%), machine learning and AI (6%), DevOps and CI/CD (5%), API clients and SDKs (5%), networking libraries (4%), web frameworks (2%), and testing tools (2%), with the remainder spanning domain-specific utilities across

Table 4: Dataset characteristics comparison.

Characteristic	ClassEval	RealClassEval	OpenClassGen
Classes	100	400	324,843
Documentation	Complete	Variable	Variable
Structural metrics	✗	✗	27 metrics
Domain diversity	Limited	Moderate	High

robotics, gaming, cryptography, and other specialized fields. This diversity ensures that models trained or evaluated on OpenClassGen encounter varied coding patterns, naming conventions, and architectural styles representative of the broader Python ecosystem.

4.4 Comparison with Existing Datasets

Table 4 compares OpenClassGen’s characteristics with existing class-level datasets. We restrict this comparison to ClassEval and RealClassEval as they are the only datasets that share our granularity (class-level) and language (Python), enabling direct comparison of documentation coverage, structural metrics, and domain diversity. As shown in the table, OpenClassGen provides not only greater scale but also greater variability in documentation and structure, enabling more robust evaluation of LLM generalization.

5 Evaluation

To demonstrate the utility of OpenClassGen for class-level code generation research, we evaluate LLM-generated classes using extracted skeletons as structured prompts. This evaluation is explicitly illustrative: its purpose is to confirm that OpenClassGen supports meaningful multi-dimensional LLM assessment, not to provide definitive model rankings or characterize the state of the art. The three models selected are representative of current general-purpose and code-specialized LLMs, but are not claimed to be the strongest available baselines. We assess generation quality by comparing LLM outputs against their human-written counterparts across four complementary dimensions: syntactic quality (lexical overlap), semantic quality (meaning preservation), structural quality (AST-level alignment), and functional correctness (executable behaviour).

5.1 Approach

Sampling. We randomly sample 300 classes from the dataset using stratified sampling to ensure balanced representation across three documentation levels: fully documented (all program units have docstrings), partially documented (some program units have docstrings), and undocumented (no program units have docstrings). This stratification enables evaluation of LLM performance under varying levels of natural language guidance. A sample of 300 classes (100 per documentation tier) exceeds the minimum required for a 90% confidence level with a 5% margin of error ($n = 271$), providing sufficient statistical resolution for this illustrative evaluation; a more comprehensive evaluation leveraging the full dataset’s scale is an intended direction for future work.

Generation. For each sampled class, we prompt three representative LLMs to generate the complete class implementation given its skeleton:

- **GPT-04-mini** [5]: A reasoning-focused model from OpenAI.

- **Claude-4-Sonnet** [4]: A general-purpose model from Anthropic.
- **Qwen-3-Coder** [7]: A code-specialized open-weight model.

We use the following prompt:

You are an expert Python programmer who can correctly implement complete Python classes based on the provided class skeletons. Implement the following class:
[CLASS SKELETON]

This prompt design follows established practices in code generation evaluation [12, 14]. The “expert” persona encourages the model to produce professional-quality code rather than simplified or pedagogical implementations; prior work has shown that expert identity prompting improves LLM response quality across diverse tasks [34]. We intentionally avoid chain-of-thought instructions or few-shot examples to isolate the model’s inherent class-level generation capability, ensuring results are comparable across models and reproducible by other researchers.

5.2 Metrics

We employ five metrics spanning four quality dimensions. We provide formal definitions below, following prior work in code generation evaluation [14, 27].

5.2.1 Syntactic Quality. We measure lexical overlap using two complementary n-gram-based metrics. Following prior work [18, 29], we use $n \leq 3$ as higher-order n-grams provide diminishing returns for code representation.

BLEU. The BLEU (Bilingual Evaluation Understudy) score [25] measures precision-oriented n-gram overlap between generated code C and reference code R :

$$\text{BLEU} = BP \cdot \exp \left(\sum_{n=1}^N w_n \log p_n \right) \quad (1)$$

where p_n is the precision of n-grams up to length N , w_n is the weight assigned to each n-gram (typically uniform), and BP is a brevity penalty that adjusts for short generations:

$$BP = \begin{cases} 1 & \text{if } |C| > |R| \\ e^{(1-|R|/|C|)} & \text{if } |C| \leq |R| \end{cases} \quad (2)$$

where $|C|$ and $|R|$ denote the lengths of the candidate and reference, respectively.

ROUGE-L. The ROUGE (Recall-Oriented Understudy for Gisting Evaluation) score [21] measures recall-oriented overlap. We use the ROUGE-L variant, which computes the longest common subsequence (LCS) between generated and reference code:

$$\text{ROUGE-L} = \frac{\text{LCS}(C, R)}{|R|} \quad (3)$$

where $\text{LCS}(C, R)$ is the length of the longest common subsequence. Unlike n-gram metrics, ROUGE-L captures sequence-level structure without requiring contiguous matches.

Together, BLEU emphasizes precision (how much of the generated code is correct) while ROUGE-L emphasizes recall (how much of the reference code is covered).

5.2.2 Semantic Quality.

CodeBERTScore. Lexical metrics fail to capture semantic equivalence when implementations differ syntactically but preserve meaning. CodeBERTScore [38] addresses this by computing similarity in the embedding space of CodeBERT [15], a transformer model pre-trained on code and natural language.

Given token embeddings c_i for generated code and r_j for reference code, CodeBERTScore computes precision P , recall R , and F-score:

$$P = \frac{1}{|C|} \sum_{c_i \in C} \max_{r_j \in R} c_i^\top r_j \quad (4)$$

$$R = \frac{1}{|R|} \sum_{r_j \in R} \max_{c_i \in C} r_j^\top c_i \quad (5)$$

$$F_\beta = (1 + \beta^2) \cdot \frac{P \cdot R}{\beta^2 \cdot P + R} \quad (6)$$

We report F_3 (i.e., $\beta = 3$), which weights recall three times more than precision. This choice reflects that in code generation, capturing the full semantic content of the reference is more critical than avoiding extraneous tokens.

5.2.3 Structural Quality.

Tree Similarity Edit Distance (TSED). Lexical and embedding-based metrics operate on token sequences and may miss structural differences in code organization. TSED [31, 32] measures structural similarity by comparing abstract syntax trees (ASTs).

Given ASTs T_1 and T_2 for generated and reference code, the tree edit distance $\Delta(T_1, T_2)$ is the minimum cost of edit operations (insertion, deletion, renaming) to transform T_1 into T_2 :

$$\Delta(T_1, T_2) = \min_{ops} \sum_{i=1}^n w(ops_i) \quad (7)$$

where ops is a sequence of edit operations and $w(ops_i)$ is the cost of each operation. We compute this using the APTED algorithm [26], which optimizes over all possible path strategies.

To ensure interpretability, we normalize by the size of the larger AST:

$$\text{TSED} = 1 - \frac{\Delta(T_1, T_2)}{\max(|T_1|, |T_2|)} \quad (8)$$

where $|T|$ denotes the number of nodes in tree T . TSED ranges from 0 (completely different structures) to 1 (identical ASTs).

5.2.4 Functional Correctness.

Pass Rate. The above metrics assess similarity to reference implementations but do not verify executable correctness. We compute the pass rate on automatically generated unit tests:

$$\text{Pass Rate} = \frac{\text{Number of passing tests}}{\text{Total number of tests}} \quad (9)$$

Tests are generated using Pyguint [22], a search-based test generation [24] framework for Python that applies genetic algorithms to maximize code coverage [10].

We use automated test generation rather than repository test suites for two reasons. First, executing repository tests at scale is impractical: it requires resolving project-specific dependencies [20], handling environment mismatches, and managing unmaintained

projects. These challenges do not scale to 324K classes across 2,970 repositories. Second, Pynguin generates tests against the *generated* code in isolation, providing a controlled assessment of functional correctness without conflating generation quality with dependency resolution failures. To validate the quality of these test suites, we measured their branch coverage against the human-written ground truth for our 300 evaluation classes, achieving an average of 58%. This is only marginally lower than Pynguin’s reported performance on Python code (63–68% branch coverage using search-based algorithms [22]), confirming that our test suites provide a robust baseline for assessing functional correctness.

5.3 Results

Table 5 presents results for each LLM across the four quality dimensions.

Semantic and Structural Similarity. Generated classes demonstrate strong semantic similarity (average CodeBERTScore-F3: 0.89) and structural alignment (average TSED: 0.83), indicating that LLMs effectively capture the meaning and architecture of human-written classes. The low standard deviation across models (0.07 for both metrics) suggests consistent high-level understanding regardless of model architecture.

Syntactic Variation. Syntactic overlap is moderate (average ROUGE-L: 0.65, average BLEU: 0.48), reflecting natural variations in implementation style while preserving semantic intent. LLMs prioritize meaning over exact token reproduction, which is a necessary strength, as multiple valid implementations exist for most class specifications.

Functional Correctness. Pass rates average 0.33 across the three models, with Claude-4-Sonnet and Qwen-3-Coder achieving 0.35–0.36 and GPT-o4-mini at 0.29. The variance in pass rates demonstrates that OpenClassGen provides meaningful differentiation between models—an essential characteristic for a useful evaluation corpus.

Implications. These results confirm that class skeletons provide sufficient context for LLM generation. The combination of strong semantic/structural scores with moderate syntactic overlap indicates that LLMs understand class design but vary in surface-level implementation choices. The functional correctness gap (0.33 pass rate vs. 0.89 semantic similarity) highlights the distinction between understanding intent and producing executable code, indicating it as a key challenge for future research.

These results confirm that OpenClassGen enables meaningful differentiation of LLM capabilities across multiple quality dimensions.

6 Discussion

We discuss how OpenClassGen addresses critical gaps in the current landscape of code generation research, outline concrete use cases, and identify the unique value proposition of our contribution.

6.1 Addressing the Scale-Realism Trade-off

Existing class-level datasets force researchers to choose between scale and realism. Synthetic benchmarks like ClassEval [14] offer

controlled evaluation with complete documentation and test suites, but their 100 hand-crafted samples lack the diversity and noise of real-world software. Real-world datasets like RealClassEval [27] capture authentic coding patterns, but their 400-sample scale limits statistical power and precludes use for training or fine-tuning.

OpenClassGen resolves this trade-off by providing 324,843 real-world classes, which is three orders of magnitude larger than existing class-level datasets while preserving the documentation variability, structural diversity, and domain heterogeneity characteristic of production software. This scale enables:

- **Statistically robust evaluation:** Large sample sizes support meaningful confidence intervals and effect size estimation across class characteristics.
- **Fine-tuning and training:** Sufficient data volume for supervised fine-tuning of code generation models on class-level tasks.
- **Stratified analysis:** Researchers can subset by documentation level, complexity, or domain while retaining adequate sample sizes per stratum.

6.2 Complementing Evolution-Aware Evaluation

Recent work has highlighted the importance of temporal considerations in code generation evaluation. HumanEvo [37] demonstrated that evolution-ignored evaluation inflates LLM performance by 10–61% due to future context leakage and missing deleted code. Their solution rolls back repositories to the commit state when the target functions were written.

OpenClassGen complements this evolution-aware perspective differently. Rather than providing repository context that requires temporal alignment, we extract *self-contained class skeletons* that specify the generation task without external dependencies. This design choice offers two advantages:

- (1) **Context isolation:** Each class can be evaluated independently without repository rollback infrastructure, enabling scalable evaluation across hundreds of thousands of samples.
- (2) **Controlled prompting:** The skeleton serves as a complete specification, such as what the class should do (docstrings) and how it should be structured (method signatures), without leaking implementation details from surrounding code.

While HumanEvo addresses the question “Can LLMs generate correct code given realistic repository context?”, OpenClassGen addresses “Can LLMs generate complete class implementations given only structural specifications?” These are complementary research questions, and progress on both is necessary for robust LLM-assisted software development.

6.3 Enabling Structural Analysis

Unlike existing datasets, OpenClassGen provides 27 static code metrics per class, enabling research directions not possible with prior benchmarks:

Difficulty Prediction. Our evaluation revealed substantial variance in pass rates, indicating that certain classes are inherently more challenging. The structural metrics enable correlation studies: Which properties predict generation difficulty? Prior work suggests

Table 5: Generation quality across four dimensions (300 samples). We report average scores per model.

Model	Syntactic		Semantic	Structural	Functional
	ROUGE-L	BLEU	CodeBERTScore-F3	TSED	Pass Rate
GPT-o4-mini	0.60	0.46	0.88	0.82	0.29
Claude-4-Sonnet	0.68	0.50	0.90	0.84	0.36
Qwen-3-Coder	0.67	0.49	0.90	0.84	0.35
Average	0.65	0.48	0.89	0.83	0.33

cyclomatic complexity and dependency depth impact LLM performance [14, 27], but systematic validation requires large-scale data with per-class metrics, and this is precisely what OpenClassGen provides.

Retrieval-Augmented Generation. For RAG-based code generation, retrieval quality depends on meaningful similarity measures. Lexical similarity (e.g., BM25 [30]) may retrieve syntactically similar but structurally different classes. OpenClassGen’s metrics enable retrieval strategies based on structural properties: retrieve classes with similar complexity profiles, inheritance patterns, or coupling characteristics as few-shot examples.

Curriculum Learning. Training or fine-tuning LLMs on progressively difficult examples can improve learning efficiency. The structural metrics provide a principled basis for ordering training samples by difficulty, enabling curriculum learning strategies for class-level code generation.

6.4 Limitations of Existing Benchmarks

Table 6 summarizes limitations of existing benchmarks and how OpenClassGen addresses them. We include class-level datasets (ClassEval, RealClassEval) and repository-level benchmarks that target object-oriented structures (JavaBench, RepoClassBench). Function-level benchmarks such as HumanEvo address orthogonal challenges (temporal evolution) and are discussed in Section 6 but excluded here as their limitations fall outside the scope of class-level generation.

6.5 Intended Use Cases

The following use cases represent intended and anticipated directions enabled by OpenClassGen’s scale and metadata. While the current evaluation demonstrates dataset utility for zero-shot generation assessment, experimental validation of fine-tuning, RAG, and other downstream applications is left to future work. Based on our analysis, we identify the following concrete use cases for OpenClassGen:

Use Case 1: Fine-Tuning for Class-Level Generation. The 324,843 (skeleton, implementation) pairs provide a supervised training signal for future fine-tuning studies. Researchers can fine-tune base models to improve class-level generation, then evaluate on held-out subsets or external benchmarks like ClassEval.

Use Case 2: RAG Corpus Construction. The structural metadata enables the construction of retrieval corpora where similar classes (by complexity, documentation, or domain) serve as few-shot examples.

This is a particularly promising direction for future work, as structural similarity-based retrieval may outperform lexical approaches for class-level generation.

Use Case 3: Failure Mode Analysis. The scale supports statistically robust analysis of LLM failure patterns. Researchers can correlate structural properties with error types (e.g., do high-coupling classes produce more `AttributeErrors`?) to identify systematic LLM weaknesses.

Use Case 4: Documentation Impact Studies. The documentation stratification (20% fully documented, 44% partial, 36% undocumented) enables controlled studies of how documentation quality affects generation correctness. Prior work has explored this question through ablation studies that randomly remove docstrings from fully documented classes [27]; OpenClassGen instead provides naturally occurring documentation variation, enabling more realistic evaluation. This is a question with direct practical implications for developer workflows.

Use Case 5: Benchmark Subset Curation. Researchers can use the structural metrics to curate focused benchmarks: e.g., “high-complexity classes with full documentation” or “simple utility classes without docstrings.” This enables targeted evaluation of specific LLM capabilities.

7 Limitations

We discuss the limitations of OpenClassGen and suggest possible mitigation strategies.

Data Contamination Risk. LLMs are trained on public GitHub repositories, and our source projects may overlap with training data, potentially inflating performance estimates. Importantly, the source repositories were collected prior to the training cutoffs of all three evaluated models (GPT-o4-mini, Claude-4-Sonnet, and Qwen-3-Coder), meaning that data leakage cannot be ruled out for any of them. However, existing work [27] found no significant performance difference between seen and unseen real-world classes, suggesting memorization plays a minimal role in class-level generation where structural understanding matters more than verbatim recall. We therefore treat the reported results as potentially optimistic upper bounds, and recommend that future evaluations cross-reference repository names and commit timestamps against model-specific training cutoff dates to identify and exclude potentially memorized samples.

Functional Validation Scope. Our evaluation uses Pynguin for automated test generation, achieving 58% branch coverage on our

Table 6: Limitations of existing class-level benchmarks and how OpenClassGen addresses them.

Benchmark	Limitation	How OpenClassGen Addresses It
ClassEval	Synthetic data with complete documentation do not reflect real-world variability; 100 samples are insufficient for training or robust evaluation.	324,843 classes from production repositories with natural documentation variability (19.7% fully documented, 44.4% partial, 36.0% undocumented).
RealClassEval	400 samples limit statistical power; no structural metrics for difficulty analysis.	Three orders of magnitude larger scale enables robust statistical analysis; 27 static metrics per class support difficulty modelling and stratified evaluation.
JavaBench	Project-level granularity conflates class generation with dependency resolution; Java-only.	Class-level granularity isolates the generation task from dependency management; self-contained skeletons serve as complete specifications.
RepoClassBench	Repository context requirements limit scalability; small sample size.	No repository context required; skeletons are self-contained, enabling evaluation of 324,843 classes without dependency resolution infrastructure.

evaluation set, which is close to Pynguin’s reported performance (63–68%) on Python code [22]. The remaining 42% of unexercised branches means that pass rates may underestimate true functional correctness, particularly for stateful or dependency-heavy classes. Additionally, relying on a single automated test generation tool introduces oracle risk: test inadequacy may be conflated with model failure, causing correct implementations to appear incorrect when tests fail to exercise the relevant code paths. Taken together, the reported pass rate of 0.33 should be interpreted as a conservative lower bound rather than a precise measure of functional correctness. Researchers requiring higher-fidelity correctness evaluation can use OpenClassGen’s structural metrics to identify classes with simpler dependency structures, or supplement Pynguin with manual test curation for focused studies.

Python-Only Scope. OpenClassGen contains only Python classes, limiting generalizability to other object-oriented languages such as Java, C#, or TypeScript. This focus was intentional: Python dominates existing code generation research [11], enabling direct comparison with prior benchmarks. That said, the curation pipeline is language-agnostic at the project selection and filtering stages; extending to other languages requires only language-specific AST parsing, which we leave to future work.

Despite these limitations, OpenClassGen’s scale, structural metadata, and real-world fidelity provide a robust foundation for advancing class-level code generation research, transforming known constraints into structured opportunities for future investigation.

8 Conclusion

We introduced OpenClassGen, a large-scale corpus of 324,843 real-world Python classes extracted from 2,970 engineered open-source projects. Each entry pairs a human-written class implementation with its corresponding skeleton—comprising class and method signatures with associated docstrings—and is complemented with 27 static code metrics.

Our evaluation across three contemporary LLMs (GPT-o4-mini, Claude-4-Sonnet, Qwen-3-Coder) demonstrates that OpenClassGen

enables meaningful assessment of class-level code generation. Generated classes exhibit strong semantic similarity (CodeBERTScore-F3: 0.89) and structural alignment (TSED: 0.83) compared to their human-written counterparts, yet functional correctness remains challenging (pass rate: 0.33), aligning with prior findings on real-world class-level generation [27]. This gap between similarity metrics and executable correctness confirms the dataset’s utility for differentiating model capabilities and studying factors that influence generation quality.

OpenClassGen addresses a critical gap in the code generation landscape: existing class-level datasets are either synthetic and small (ClassEval: 100 classes) or real-world but limited in scale (RealClassEval: 400 classes). By providing three orders of magnitude more samples while preserving real-world documentation variability, structural diversity, and domain heterogeneity, OpenClassGen enables research directions previously infeasible, including fine-tuning for class-level generation, retrieval-augmented generation with structural metadata, and statistically robust failure mode analysis. We hope this corpus serves as a foundation for advancing LLM-assisted code generation toward the structural complexity of real-world software development.

Data Availability

Our replication package, which includes the analysis scripts, can be found here: <https://zenodo.org/records/18409150>. The dataset is also available on Hugging Face [28] and can be loaded using the datasets Python library as follows:

Listing 3: Loading OpenClassGen from Hugging Face.

```

1 # pip install datasets
2
3 from datasets import load_dataset
4
5 dataset = load_dataset("mrahman2025/OpenClassGen")

```

Acknowledgment

We used AI-powered tools (Claude and Grammarly) to assist with manuscript revisions. All AI-assisted content was reviewed and verified for accuracy by the first author.

References

- [1] 2013. ast – Abstract Syntax Trees – docs.python.org. <https://docs.python.org/3/library/ast.html>. [Accessed 28-02-2025].
- [2] 2015. GitHub - nuagenetworks/monolithe: generic and extendable code generator from specifications. – github.com. <https://github.com/nuagenetworks/monolithe>. [Accessed 07-11-2025].
- [3] 2024. GitHub Language Stats – madnight.github.io. https://madnight.github.io/github/#/pull_requests/2024/1. [Accessed 12-02-2026].
- [4] 2024. Introducing Claude 4 – anthropic.com. <https://www.anthropic.com/news/claude-4>. [Accessed 07-11-2025].
- [5] 2024. OpenAI Platform – platform.openai.com. <https://platform.openai.com/docs/models/o4-mini>. [Accessed 07-11-2025].
- [6] 2024. Understand: The Software Developer’s Multi-Tool – scitools.com. <https://scitools.com/>. [Version 7.0, Build 1217, Accessed 28-02-2025].
- [7] 2025. Qwen/Qwen3-Coder-30B-A3B-Instruct · Hugging Face – huggingface.co. <https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct>. [Accessed 07-11-2025].
- [8] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868* (2022).
- [9] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [10] José Campos, Yan Ge, Nasser Albutian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.
- [11] Jialun Cao, Zhiyong Chen, Jiarong Wu, Shing-Chi Cheung, and Chang Xu. 2024. Javabench: A benchmark of object-oriented code generation for evaluating large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 870–882.
- [12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [13] Ajinkya Deshpande, Anmol Agarwal, Shashank Shet, Arun Iyer, Aditya Kanade, Ramakrishna Bairi, and Suresh Parthasarathy. 2024. Class-Level Code Generation from Natural Language Using Iterative, Tool-Enhanced Reasoning over Repository. *arXiv preprint arXiv:2405.01573* (2024).
- [14] 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 *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiao Cheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
- [16] Hao He. 2019. Understanding source code comments at large-scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1217–1219.
- [17] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).
- [18] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.
- [19] Srinivasan Iyer, Ioannis Konstantas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).
- [20] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. 2022. Not all dependencies are equal: An empirical study on production dependencies in npm. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
- [21] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.
- [22] Stephan Lukaszczuk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.
- [23] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating github for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.
- [24] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology* 104 (2018), 236–256.
- [25] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
- [26] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Information Systems* 56 (2016), 157–173.
- [27] Musfiqur Rahman, SayedHassan Khatoonabadi, and Emad Shihab. 2025. Beyond Synthetic Benchmarks: Evaluating LLM Performance on Real-World Class-Level Code Generation. *arXiv preprint arXiv:2510.26130* (2025).
- [28] Musfiqur Rahman, SayedHassan Khatoonabadi, and Emad Shihab. 2025. Open-ClassGen (Revision of 572f3). doi:10.57967/hf/6321
- [29] Musfiqur Rahman, Dharani Palani, and Peter C Rigby. 2019. Natural software revisited. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 37–48.
- [30] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and trends® in information retrieval* 3, 4 (2009), 333–389.
- [31] Yewei Song, Saad Ezzini, Xunzhu Tang, Cedric Lothritz, Jacques Klein, Tegawendé Bissyandé, Andrey Boytsov, Ulrick Ble, and Anne Goujon. 2024. Enhancing Text-to-SQL translation for financial system design. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 252–262.
- [32] Yewei Song, Cedric Lothritz, Daniel Tang, Tegawendé F Bissyandé, and Jacques Klein. 2024. Revisiting code similarity evaluation with abstract syntax tree edit distance. *arXiv preprint arXiv:2404.08817* (2024).
- [33] Tao Xiao, Youmei Fan, Fabio Calefato, Christoph Treude, Raula Gaikovina Kula, Hideaki Hata, and Sebastian Balthes. 2025. Self-admitted GenAI usage in open-source software. *arXiv preprint arXiv:2507.10422* (2025).
- [34] Benfeng Xu, An Yang, Junyang Lin, Quan Wang, Chang Zhou, Yongdong Zhang, and Zhendong Mao. 2023. Expertprompting: Instructing large language models to be distinguished experts. *arXiv preprint arXiv:2305.14688* (2023).
- [35] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
- [36] Adam Yuen, John Pangas, Md Mainul Hasan Polash, and Ahmad Abdellatif. 2025. Prompting Matters: Assessing the Effect of Prompting Techniques on LLM-Generated Class Code. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 803–808.
- [37] Dewu Zheng, Yanlin Wang, Ensheng Shi, Ruikai Zhang, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. 2024. Humanevo: An evolution-aware benchmark for more realistic evaluation of repository-level code generation. *arXiv preprint arXiv:2406.06918* (2024).
- [38] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. (2023). <https://arxiv.org/abs/2302.05527>