ClassEval-Pro: A Cross-Domain Benchmark for Class-Level Code Generation

Anonymous Author(s)

ABSTRACT

LLMs have achieved strong results on both function-level code synthesis and repository-level code modification, yet a capability that falls between these two extremes—*compositional code creation*, i.e., building a complete, internally structured class from a specification—remains underserved. Current evaluations are either confined to isolated functions or rely on manually curated class-level tasks that are expensive to scale and increasingly susceptible to data contamination. We introduce ClassEval-Pro, a benchmark of 300 class-level tasks spanning 11 domains, constructed through an automated three-stage pipeline that combines complexity enhancement, cross-domain class composition, and integration of real-world GitHub code contributed after January 2025. Every task is validated by an LLM Judge Ensemble and must pass test suites with over 90% line coverage. We evaluate five frontier LLMs under five generation strategies. The best model achieves only 45.6% class-level Pass@1, with a 17.7-point gap between the strongest and weakest models, confirming the benchmark's discriminative power. Strategy choice strongly interacts with model capability: structured approaches such as bottom-up improve weaker models by up to 9.4 percentage points, while compositional generation collapses to as low as 1.3%. Error analysis over 500 manually annotated failures reveals that logic errors (56.2%) and dependency errors (38.0%) dominate, identifying cross-method coordination as the core bottleneck.

ACM Reference Format:

Anonymous Author(s). 2026. ClassEval-Pro: A Cross-Domain Benchmark for Class-Level Code Generation. In . ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

LLMs now excel at two distinct modes of code generation. On function-level benchmarks, frontier models surpass 90% Pass@1 on HumanEval [2] and MBPP [1], approaching human-level accuracy on isolated programming tasks. On repository-level benchmarks, coding agents resolve over 75% of real-world GitHub issues on SWE-bench Verified [15]. Yet real software development also demands a capability that falls between these two extremes: building a complete, internally structured class from a specification—coordinating multiple methods, managing shared state, and integrating logic across domains [13, 22]. Current function-level benchmarks (HumanEval, MBPP, BigCodeBench [34]) test isolated logic synthesis, while repository-level benchmarks (SWE-bench, DevEval [18]) test code modification within existing codebases; neither evaluates whether models can architect a coherent multi-method artifact from scratch. ClassEval [7] pioneered class-level evaluation, yet its reliance on manual curation (500 person-hours for 100 tasks across 7 domains) makes it difficult to scale, and its 2023-vintage data is increasingly exposed to training-set contamination [8, 23]. We refer to this underserved evaluation dimension as *compositional code creation*.

To address this gap, we introduce ClassEval-Pro, a benchmark of 300 class-level tasks spanning 11 domains, designed to evaluate compositional code creation. Three design choices distinguish it from prior work. (1) *Cross-domain composition*—tasks require integrating heterogeneous domain logic (e.g., configuration management + financial computation) into a single coherent class, testing compositional reasoning that single-domain tasks cannot capture. (2) *Automated, contamination-resistant construction*—an LLM Judge Ensemble pipeline [11] replaces manual curation, sourcing from GitHub repositories contributed after January 2025 [33] and employing multi-model collaboration for skeleton generation, test alignment, and reference implementation. (3) *Strict validation*—every task is verified against automatically generated test suites that achieve over 90% line coverage, ensuring that both the specification and the reference solution are functionally correct.

We evaluate five frontier LLMs—GPT-5.1, Gemini-2.5-Pro, Qwen3-480B, Qwen3-30B, and Kimi-K2—under five generation strategies (holistic, incremental, compositional, top-down, and bottom-up) and manually annotate 500 failed cases. Class-level Pass@1 ranges from 27.9% to 45.6% (17.7-point gap), far wider than on function-level benchmarks, confirming that compositional tasks discriminate frontier models effectively. Strategy choice strongly interacts with model capability: structured decomposition (bottom-up) improves weaker models by up to 9.4 percentage points, while compositional generation collapses to as low as 1.3%. Error analysis reveals that logic errors (56.2%) and dependency errors (38.0%) dominate failures, identifying cross-method coordination—not surface-level code formation—as the core bottleneck in class-level code generation.

Our major contributions are as follows:

- We propose ClassEval-Pro, a benchmark of 300 class-level tasks spanning 11 domains, targeting *compositional code creation*—architecting coherent, multi-method classes from specifications.
- We design an automated, contamination-resistant construction pipeline using LLM Judge Ensembles and post-2025 GitHub code.
- We provide diagnostic insights—strategy–model interactions and a fine-grained error taxonomy—that go beyond pass/fail rates to inform model and agent design.

2 RELATED WORK

2.1 Large Language Models for Code Generation

Large Language Models (LLMs) have significantly advanced code generation [34], repair [3, 25], translation [16], and reasoning [10].

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.

Conference'17, July 2017, Washington, DC, USA

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

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1: Comparison of code generation benchmarks.

Benchmark	Task	Avg. LOC	Cross Domain	Auto Construct	Real Source	From Scratch
Function-level generation						
HumanEval	164	11.5	×	×	×	✓
MBPP	974	6.8	×	×	×	✓
APPS	5000	21.4	×	×	✓	✓
HumanEval+	164	11.5	×	×	×	✓
LCBench	1055	–	×	✓	✓	✓
Repository-level modification						
SWE-bench	2294	–	✓	✓	✓	×
DevEval	1874	–	✓	×	✓	×
Class-level creation						
ClassEval	100	45.7	✓	×	×	✓
CoderEval	230	30.0	✓	×	✓	✓
ClassEval-Pro	**300**	**113**	✓	✓	✓	✓

This progress is driven by both open-weight models (e.g., DeepSeek-Coder [12], Qwen3-Coder [30]) and proprietary models (e.g., GPT-5 [26], Gemini 3 [5]). Despite these advances, current benchmarks predominantly evaluate atomic function-level tasks or repository-level modifications. Despite object-oriented programming dominating real-world software [22], evaluations for compositional class-level creation remain limited.

2.2 Benchmarks for Code Generation

Early code generation benchmarks target function-level synthesis (e.g., HumanEval [2], MBPP [1]). While subsequent works enhance evaluation rigor [20], API diversity [34], and data freshness [14], they fundamentally fail to capture the structural complexity of real-world software [13].

To address this, recent evaluations diverge into two trajectories: repository-level modification and compositional generation. Modification benchmarks evaluate issue resolution within existing codebases (e.g., SWE-bench [15], DevEval [18]) or utilize repository context [31]. Generation benchmarks, conversely, target structural creation, such as context-dependent units (CoderEval [32]) or entire classes (ClassEval [7]). However, reliance on manual curation restricts existing class-level benchmarks in scalability and domain coverage [7, 23]. While several existing benchmarks incorporate *multi-domain evaluation*, none support the automated construction of from-scratch, real-source tasks at the class level. ClassEval-Pro bridges this gap by uniquely integrating an auto-construct pipeline with multi-domain, real-source grounding for from-scratch generation.

2.3 Dataset Augmentation

Traditional data augmentation relies on semantic-preserving transformations [24, 28] to improve generalization. Recently, LLM-driven synthesis has emerged as a scalable alternative, utilizing seed-based bootstrapping (e.g., Self-Instruct [27]) and iterative complexity escalation (e.g., Evol-Instruct [29]). However, these methods primarily target training data generation rather than benchmark curation. Repurposing them for evaluation introduces strict requirements: generated tasks must guarantee functional correctness, align with executable test suites, and prevent training data contamination. Existing frameworks fail to systematically enforce these constraints

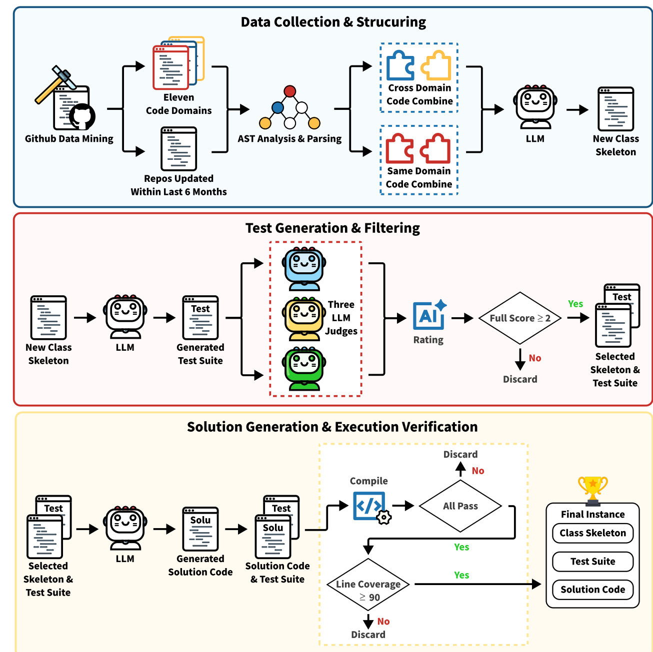

Figure 1: Overview of the benchmark construction pipeline.

for compositional class-level generation. ClassEval-Pro bridges this by implementing a rigorous auto-construct pipeline, which integrates LLM-based skeleton generation, multi-model peer validation, and strict line-coverage filtering to synthesize high-quality, multi-domain evaluation tasks.

3 BENCHMARK CONSTRUCTION

This section introduces ClassEval-Pro, a benchmark of 300 multi-domain class-level code generation tasks spanning 11 domains.

3.1 Pipeline Overview

To mitigate data contamination [14] and the manual overhead of benchmark curation [7], we propose a fully automated construction pipeline (Figure 1). First, we harvest code snippets across 11 domains from GitHub repositories updated within the past six months [9]. Through AST parsing, candidate classes are structurally merged either within similar domains or across distinct domains. This structural fusion enforces compositional reasoning while synthesizing novel configurations to strictly minimize pre-training exposure [6].

Subsequently, an LLM generates a test suite for each skeleton. To ensure rigorous alignment, three LLM-judges [11] evaluate the skeleton-test compatibility (0–10 scale). Only pairs securing a perfect 10 from at least two judges proceed. Finally, a separate LLM synthesizes the solution code. We retain only instances that achieve a 100% pass rate and ≥ 90% line coverage during execution [21], guaranteeing high-quality evaluation samples [23].

233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290

Table 2: Domain Distribution of ClassEval-Pro Tasks

Domain	Same-domain	Cross-domain	Total
Finance & Ecommerce	8	33	41
File Handling	8	30	38
Management Systems	18	19	37
Network & Web	3	33	36
Database Operations	5	27	32
Security & Crypto	3	23	26
Natural Lang. Process.	5	18	23
Game Development	3	18	21
Mathematical Operations	6	12	18
Utils	6	9	15
Data Formatting	2	11	13
Total	**67**	**233**	**300**

3.2 Stage 1: Complexity Enhancement

Before combining classes across domains, we first enhance the complexity of individual classes from the original ClassEval benchmark [7]. For each of the 100 original tasks, an LLM augments the class skeleton by (1) injecting additional methods that introduce non-trivial intra-class dependencies (e.g., helper methods invoked by multiple public methods), (2) deepening control-flow logic within existing methods (e.g., adding edge-case handling, state validation, or multi-step transformations), and (3) expanding the attribute space to require coordinated state management across methods. The enhanced class may exhibit different or extended functionality compared to the original, as augmentation can reorganize methods and introduce new inter-method interactions that compose novel behavior. AST-level constraints ensure structural validity rather than behavioral equivalence, and the LLM Judge Ensemble validation and coverage-based filtering described in Section 3.1 verify that the resulting skeleton–test pair is self-consistent and executable. This stage serves as the foundation for subsequent combination stages, providing richer building blocks whose internal dependencies already demand multi-step reasoning.

3.3 Stage 2: Domain-Based Class Combination

Building upon the original ClassEval taxonomy [7], ClassEval-Pro expands the scope from 7 to 11 domains (e.g., adding Data Structures and System Utilities), yielding 300 diverse class-level tasks (Table 2). To evaluate compositional code creation [13], we implement two structural combination strategies driven by AST-augmented prompts that specify target functionalities and interface constraints [6]:

- **Intra-Domain Composition:** Merging classes within a single domain. For example, fusing `SinglyLinkedList` and `Stack` into a `LinkedStack` requires the model to optimize internal method calls while maintaining structural integrity.
- **Cross-Domain Composition:** Human experts first categorize composable domains based on real-world applications. To simulate complex scenarios, classes are then paired across these domains. For instance, coupling `SQLQueryBuilder` (Database) with `PlayerInventory` (Game) necessitates unified state management and cross-domain functional mapping.

As detailed in Table 3, these compositional strategies yield tasks with significantly higher complexity—quantified by average lines of code (LOC), dependency depth, and method count—exhibiting denser intra-class dependencies than standalone tasks in prior benchmarks [7].

Table 3: Code Metrics Statistics

Dataset	Type	Number	LOC		Dep.		Method	
			Avg	Med	Avg	Med	Avg	Med
Original	Standard	100	45.7	33	1.77	2.0	4.97	5.0
ClassEval-Pro	Cross-Domain	233	117.0	114	3.01	3.0	9.53	9.0
	Same-Domain	67	122.0	100	2.85	3.0	8.60	8.0

3.4 Stage 3: GitHub Integration and Quality Control

While Stages 1–2 enhance and combine classes derived from the original ClassEval corpus, Stage 3 introduces an independent source of real-world code to further diversify the benchmark. We collect classes from actively maintained open-source repositories and feed them into the same domain-based combination pipeline described in Stage 2 (both same-domain and cross-domain pairings), so the final benchmark contains tasks originating from both the enhanced ClassEval lineage and fresh GitHub code. This stage implements a four-phase process—discovery, extraction, validation, and serialization—that converts raw GitHub data into structured, self-contained components for downstream combination.

We query the GitHub Search API [9] for Python repositories created on or after January 1, 2025, ensuring that all collected code postdates the knowledge cutoffs of current LLMs and thereby mitigates the risk of data contamination [15]. To achieve broad and representative coverage, we expanded the original seven domains of ClassEval into an eleven-domain keyword matrix. Keywords are partitioned into groups of three and issued as disjunctive queries across three star-count tiers (100–500, 500–1,000, and >1,000), following a stratified sampling strategy similar to that of The Stack [17]. Duplicate repositories are removed via URL deduplication.

Each discovered repository is shallow-cloned, and all Python source files—excluding test files and migration directories—are parsed with the `ast` module. An AST visitor performs *import purity analysis*: any file referencing a third-party package or using a relative import is discarded entirely. Only classes whose imports resolve exclusively to the Python standard library are retained, guaranteeing that every extracted class can be compiled and tested in isolation—a stricter variant of the self-containedness criterion adopted by CoderEval [32]. We further enforce structural constraints: each candidate class must contain at least 5 methods and span between 40 and 800 lines of code, filtering out both trivial data containers and monolithic files. In total, 206 repositories were crawled, yielding 1,114 extracted classes, of which 383 survived all filtering criteria.

Candidates that survive extraction then undergo three-level verification: (1) *content integrity*—files are checked for non-empty content and the absence of HTML or XML artifacts from malformed API responses; (2) *compilation*—each file is parsed with `ast.parse` and compiled with Python's built-in `compile()`, catching syntax errors and structural violations such as `return` statements outside

291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348

functions; and (3) *reference completeness*—a custom AST visitor performs scope-aware analysis to detect unresolved name references by maintaining a scope stack that tracks imports, function parameters, class definitions, and local assignments, flagging any Load-context name unresolvable in any enclosing scope or the builtin namespace. This static check goes beyond the AST-parsability filter used in StarCoder's data pipeline [19] and ensures each class is genuinely self-contained and executable.

Validated classes are converted into structured JSON via a second AST pass. Each record contains repository metadata (origin, domain label), import statements, the class-level docstring, and per-method entries comprising the full signature, parameter list, docstring, and source code—a format designed to support downstream prompt construction without additional parsing. Each GitHub class inherits a domain label from the query that discovered it; classes are then paired through both *same-domain* and *cross-domain* combinations.

The dataset comprises class names, skeletons, test codes, corresponding test classes, and solution code. To ensure data integrity, human experts performed a final verification to guarantee that all problem descriptions are accurate and unambiguous.

4 EXPERIMENTAL DESIGN

Research Questions. To benchmark LLMs on ClassEval-Pro, we study four complementary research questions covering benchmark difficulty, overall model performance, strategy effects, and failure modes:

- **RQ1:** How effectively does our pipeline increase task difficulty compared to the original ClassEval? This RQ evaluates whether ClassEval-Pro is more challenging in a controlled and meaningful way.
- **RQ2:** How do state-of-the-art LLMs perform on ClassEval-Pro under holistic generation? This RQ provides the main benchmark comparison under a unified setting.
- **RQ3:** How do different generation strategies affect class-level code generation performance across models? This RQ examines the sensitivity of ClassEval-Pro to different class construction strategies.
- **RQ4:** What types of errors dominate LLM failures on class-level code generation? This RQ identifies the main failure patterns exposed by ClassEval-Pro.

4.1 Studied LLMs

We study five LLMs, covering both proprietary and open-weight systems. Specifically, our evaluation includes two commercial frontier models, GPT-5.1 and Gemini-2.5 Pro, together with three open-weight instruction-tuned MoE models, Kimi-K2, Qwen3-480B-A35B, and Qwen3-30B-A3B. These models span a wide range of scales, from 30.5B parameters to trillion-scale MoE architectures, with active parameter sizes ranging from 3.3B to 35B. For the open-weight models, the officially released specifications indicate training corpora of 15.5–36 trillion tokens and context windows ranging from 32k to 256k tokens, while the proprietary models provide comparably strong long-context support but do not publicly disclose parameter size or training-token details. This selection enables a balanced comparison between closed and open-weight models, while also supporting within-family scaling analysis through the two Qwen3 variants.

4.2 Studied Generation Strategies

For class-level code generation [7], we investigate five generation strategies that span the spectrum from single-pass synthesis to structured decomposition and iterative refinement [25].

- Holistic: Given a code class skeleton as input, the LLM is required to generate the complete class implementation in a single pass [7].
- Incremental: Following Du et al. [7], classes are generated by iteratively predicting methods in their declaration order. Each step leverages the previously generated code as context, continuing until class completion.
- Compositional: Import statements, class definitions, and method bodies are generated independently from the skeleton [7]. Each fragment is produced without inter-method context, then assembled to form the final class.
- Bottom-Up: Utilizing a two-stage dependency-guided framework [4], this approach categorizes methods into hierarchical levels based on call graphs and shared states. Methods are generated incrementally from the lowest dependency level to the highest.
- Top-Down: This approach employs the same dependency analysis as Bottom-Up but inverts the implementation sequence, prioritizing high-level methods before their lower-level dependencies.

We provide one example for each generation strategy in the released code repository for clearer illustration.[1]

4.3 Evaluation Metrics

To evaluate generation correctness, we follow established literature by adopting Pass@k as our primary metric.

To ensure a reliable evaluation, the underlying benchmark is rigorously validated during the dataset construction phase. We automatically execute candidate solutions against the test suites in an isolated environment. A generation task is included in the final dataset only if its reference solution satisfies strict execution criteria: (1) passing all method-level and class-level tests; (2) achieving a statement coverage strictly greater than 90%; and (3) completing execution within a 60-second timeout limit to prevent unbounded resource consumption.

To maintain practical computational overhead and response times, we set n to five. To mitigate high sampling variance, we employ the unbiased estimator consistent with prior work [7].

4.4 Implementation Details

We evaluate our benchmark using five models: GPT-5.1, Gemini-2.5-Pro, Qwen3-480B-A35B, Qwen3-30B-A3B and Kimi-K2. All models are accessed via their respective APIs. For all generation tasks, we set temperature=0.2 and max_tokens=16438. To compute Pass@k, we sample $k = 5$ candidate solutions per task. Our automated evaluation pipeline, including code execution and test verification via pytest, runs on a local server equipped with 8× NVIDIA A800

[1]Code repository: https://anonymous.4open.science/r/anonymous-repo-5153/

(80GB) GPUs. We enforce a strict 60-second timeout per task to bound resource consumption. Due to API inference cost constraints, all models are evaluated with a single run. To ensure reproducibility, all constructed data, generated outputs, and evaluation scripts will be open-sourced upon acceptance.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Dataset Difficulty

To address RQ1, we evaluate task difficulty across two dimensions: structural complexity and semantic diversity.

Structural Complexity: Using internal method dependencies as a proxy (Figure 2), ClassEval-Pro exhibits a significant rightward shift. Unlike the original ClassEval, which peaks at 1–2 layers, our benchmark centers at 3 layers with a long tail extending to 7. This deeper hierarchy amplifies difficulty by strictly requiring multi-step compositional reasoning rather than localized function completion.

Semantic Diversity and Rigor: The t-SNE projection of CodeBERT embeddings (Figure 3) displays wide dispersion, validating that our combination strategies yield a highly diverse task space. Crucially, the superimposed gradient confirms that nearly all generated tasks maintain ∼100% test coverage, ensuring that increased structural complexity does not compromise evaluation rigor.

> ✎ **Finding 1:** Our automated pipeline increases benchmark difficulty along both structural and semantic dimensions without sacrificing evaluation rigor. As a result, ClassEval-Pro provides a more challenging and more discriminative testbed for class-level code generation than the original ClassEval.

5.2 RQ2: Overall Performance Analysis

We use *holistic* generation—where each model generates the entire target class in a single pass—as a unified baseline for fair comparison across models.

Comparison among LLMs. Under holistic generation, clear capability differences emerge across models. Qwen3-480B-A35B achieves the strongest Pass@1 Class-level Full Success (45.6%), followed closely by Kimi-K2 (45.1%) and Gemini-2.5-Pro (44.7%). In contrast, GPT-5.1 lags substantially behind at 27.9%. The 17.7-point gap between the best and worst models confirms the benchmark's discriminative power, and the detailed per-metric breakdown (Table 4) shows that these disparities persist across Partial Success and Method-level metrics.

A clear scaling effect is also evident within the Qwen family: the larger Qwen3-480B-A35B outperforms Qwen3-30B-A3B by 5.1 points on Pass@1 Class-level Full Success (45.6% vs. 40.5%), with consistent gains across all metrics. This confirms that larger model capacity remains beneficial for class-level generation requiring long-range dependency planning. Rankings are broadly stable under Pass@3 and Pass@5, though Gemini-2.5-Pro shows the largest gain with increasing k (from 44.7% to 62.7%), suggesting greater solution diversity.

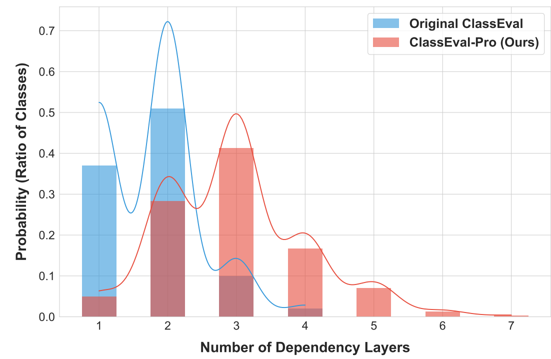

Figure 2: Normalized Distribution of Dependency Layers

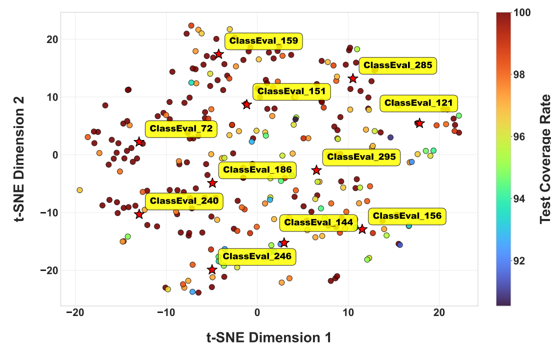

Figure 3: Semantic Landscape of ClassEval-Pro

> ✎ **Finding 2:** Under holistic generation, ClassEval-Pro clearly differentiates frontier LLMs: strong models cluster around 45% class-level full success at Pass@1, while weaker ones lag far behind, and larger models within the same family retain a consistent advantage.

5.3 RQ3: Impact of Generation Strategies

Different generation strategies may favor different combinations of model capability and task complexity. We therefore evaluate five representative strategies—*holistic*, *incremental*, *compositional*, *top-down*, and *bottom-up*—to examine how strategy choice affects class-level code generation performance.

As shown in Table 4, strategy effectiveness is strongly model-dependent, and no single strategy dominates across all settings. Nevertheless, clear trends emerge. *Top-down* and *bottom-up* are the most consistently competitive structured strategies, often improving over *holistic* generation by better organizing inter-method dependencies. For example, on GPT-5.1, *bottom-up* achieves the best class-level full success at Pass@1 (37.3%), outperforming both *holistic* (27.9%) and *top-down* (34.7%). On Gemini-2.5-Pro, *top-down* also surpasses *holistic* (48.4% vs. 44.7%). However, this advantage is not universal: on Kimi-K2, *holistic* (45.1%) and *top-down* (44.5%) remain strongest, while *bottom-up* drops to 31.5%, suggesting that explicit decomposition can also introduce new inconsistencies when global coherence is not maintained. In contrast, *incremental* generation is generally less stable, likely because step-wise synthesis weakens global context across methods, and *compositional* performs worst

Table 4: Results of different generation strategies with various models

Model	Generation Strategy	Class-level						Method-level					
		Full Success			Partial Success			Full Success			Partial Success		
		Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
GPT-5.1	Holistic	27.9%	35.8%	39.0%	47.5%	55.1%	57.3%	61.3%	67.3%	69.0%	70.0%	74.9%	76.1%
	Incremental	20.9%	26.0%	28.0%	33.3%	39.5%	41.7%	43.6%	49.0%	50.8%	51.0%	55.4%	56.8%
	Compositional	17.7%	23.4%	26.3%	30.5%	38.5%	42.3%	40.1%	46.3%	48.8%	48.1%	54.0%	56.3%
	Top-down	34.7%	47.9%	53.0%	53.1%	66.2%	70.0%	66.8%	79.7%	83.1%	73.8%	85.4%	88.1%
	Bottom-up	37.3%	47.2%	51.3%	56.9%	64.7%	68.3%	73.0%	78.7%	80.8%	80.4%	84.4%	85.9%
Gemini-2.5-Pro	Holistic	44.7%	58.1%	62.7%	69.3%	78.6%	80.7%	80.5%	88.1%	90.3%	89.1%	93.7%	94.8%
	Incremental	45.4%	57.4%	61.7%	66.5%	76.5%	78.3%	77.1%	86.0%	87.9%	84.4%	91.1%	92.2%
	Compositional	1.3%	2.6%	3.0%	1.8%	3.3%	3.7%	8.0%	15.4%	19.4%	19.2%	32.2%	39.0%
	Top-down	48.4%	63.0%	68.0%	69.9%	80.9%	84.7%	82.2%	90.4%	92.2%	89.5%	95.0%	96.0%
	Bottom-up	45.5%	59.5%	64.0%	66.7%	79.0%	81.7%	76.7%	88.1%	90.9%	83.4%	93.0%	94.8%
qwen3-480b-a35b	Holistic	45.6%	52.9%	56.3%	74.1%	79.4%	81.0%	84.2%	87.5%	88.7%	92.6%	94.4%	95.0%
	Incremental	37.9%	50.9%	56.0%	54.9%	70.0%	76.0%	63.6%	77.8%	83.0%	69.0%	83.1%	88.3%
	Compositional	13.5%	25.2%	30.3%	31.1%	52.7%	61.0%	48.9%	68.8%	74.3%	65.5%	83.4%	86.9%
	Top-down	45.3%	55.1%	58.7%	73.9%	81.1%	83.3%	83.5%	88.7%	90.2%	91.6%	94.8%	95.5%
	Bottom-up	46.3%	54.1%	56.3%	72.7%	77.0%	78.3%	82.7%	86.2%	87.4%	90.2%	92.2%	93.0%
qwen3-30b-a3b	Holistic	40.5%	51.1%	55.7%	67.3%	77.2%	81.0%	78.2%	85.6%	88.0%	87.9%	92.9%	94.3%
	Incremental	38.3%	47.1%	50.7%	64.3%	73.2%	76.0%	76.2%	83.7%	85.5%	85.7%	91.4%	92.5%
	Compositional	7.5%	17.1%	22.7%	22.8%	42.2%	50.3%	38.8%	60.6%	67.2%	55.6%	76.3%	81.4%
	Top-down	35.3%	49.0%	55.0%	61.9%	74.4%	78.0%	73.8%	84.0%	87.0%	83.4%	91.2%	93.1%
	Bottom-up	38.6%	51.0%	55.3%	67.5%	76.6%	79.0%	77.1%	85.3%	87.4%	86.8%	92.6%	93.8%
Kimi-K2	Holistic	45.1%	53.1%	56.0%	72.1%	78.5%	81.0%	83.0%	87.3%	88.7%	91.0%	93.5%	94.3%
	Incremental	32.7%	45.2%	49.3%	52.3%	65.3%	68.7%	61.9%	73.4%	76.2%	68.6%	78.8%	81.0%
	Compositional	21.7%	31.6%	34.0%	41.5%	54.0%	57.0%	56.1%	67.5%	70.2%	69.6%	79.0%	81.0%
	Top-down	44.5%	52.5%	55.0%	70.8%	78.0%	80.3%	82.9%	87.5%	88.7%	91.1%	93.7%	94.4%
	Bottom-up	31.5%	46.7%	54.0%	49.7%	68.7%	76.0%	59.1%	78.6%	85.7%	65.0%	84.7%	91.6%

overall, with severe degradation on several models such as Gemini-2.5-Pro (1.3%) and qwen3-30b-a3b (7.5%). Overall, strategy choice substantially changes class-level generation outcomes: structured decomposition is often helpful, but excessive fragmentation can severely harm coherence.

✎ **Finding 3:** Large performance differences across strategies show that ClassEval-Pro is highly sensitive to class construction order and decomposition style, effectively exposing challenges in dependency coordination and whole-class coherence.

5.4 RQ4: Error Analysis

To understand the main remaining bottlenecks in class-level code generation, we analyze failed cases across five generation strategies and five frontier LLMs, using them to identify recurring error patterns at both the model and strategy levels.

5.4.1 Methodology. We focus on partially failed cases, namely those with non-zero method-level success but class-level failure, as they provide the clearest diagnostic signal. For each strategy, we select the best-performing model as the representative source and apply stratified random sampling across the five test categories, yielding 100 cases per strategy and 500 manually annotated cases in total.

The analysis proceeds in two stages. We first use automated log extraction to collect tracebacks, exception types, and failed test names for coarse categorization. We then conduct manual thematic annotation on the 500-case sample with three annotators who each have more than three years of Python experience, examining prompts, generated code, and tracebacks to identify root causes, with multiple labels allowed when defects co-occur. The taxonomy is developed iteratively on a pilot subset and refined before the main round. Inter-annotator agreement, measured by Fleiss' κ over top-level categories, reaches substantial agreement, and all remaining disagreements are resolved through consensus review.

5.4.2 Error Taxonomy. We organize consensus annotations into five primary error categories and one benchmark-specific cross-cutting tag. Annotators may assign multiple labels during review, but for aggregate reporting each failed case is mapped to its most direct root cause, with the cross-cutting tag retained separately when needed.

Concretely, **Syntax Errors** capture low-level code invalidity such as unresolved names or malformed symbol references; **Dependency Errors** arise when the generated class fails to preserve required field, method, or library dependencies; **Logic Errors** describe runnable but semantically incorrect implementations; **Integration Errors** reflect class-level inconsistencies where individually plausible methods do not function coherently together; and **Other Errors** cover remaining runnable failures, mainly unsupported or hallucinated behavior beyond the specification.

5.4.3 Error Distribution Analysis. Figure 4 shows that the dominant bottleneck among failed-but-runnable cases lies in semantic correctness rather than basic executability. For readability, slices below 1% are slightly enlarged in the figure, although the reported

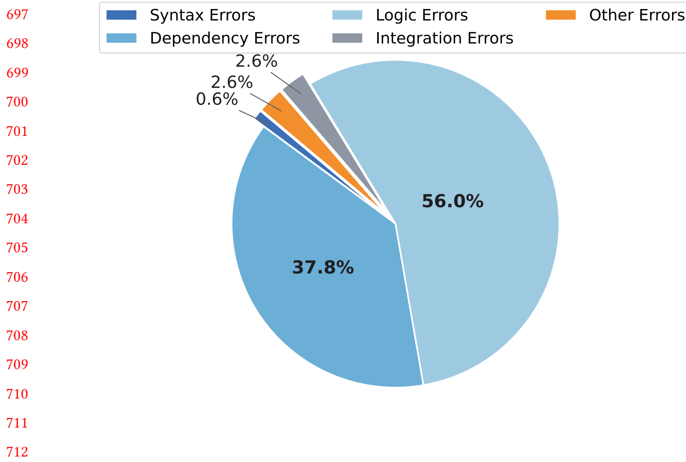

Figure 4: Overall distribution of primary error categories

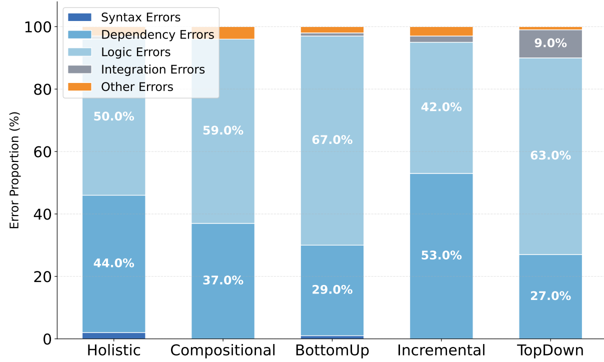

Figure 5: Error distribution across generation strategies

percentages remain unchanged. Across the 500 annotated cases, **Logic Errors** constitute the largest share at **56.2%**, followed by **Dependency Errors** at **38.0%**, whereas **Syntax Errors** account for only **0.6%**. **Integration Errors** and **Other Errors** each remain limited at **2.6%**. Taken together, these results indicate that once modern LLMs produce runnable class code, their main difficulty is no longer surface-level syntax, but preserving correct semantics, inter-method consistency, and dependency structure.

Overall, the error distribution suggests that class-level code generation is primarily constrained by reasoning and coordination failures rather than by low-level code formation. In particular, the clear dominance of logic and dependency errors implies that improving benchmark performance will depend less on further reducing syntax mistakes and more on strengthening models' ability to maintain coherent class semantics, accurate method interactions, and reliable use of required state or libraries.

5.4.4 Error Pattern Analysis by Strategy. Figure 5 shows that strategies fail in systematically different ways. Holistic has a balanced profile: **Logic Errors (50.0%)** and **Dependency Errors (44.0%)** occur at comparable rates, suggesting full-class generation retains global context but struggles with both semantic correctness and dependency consistency. Incremental has the highest **Dependency Errors (53.0%)**, indicating step-wise generation is especially vulnerable to broken signatures, missing fields, and cross-method context fragmentation. BottomUp and TopDown shift failures toward **Logic Errors (67.0%** and **63.0%**, respectively) while keeping dependency errors lower, suggesting hierarchical structuring preserves interfaces but does not resolve semantic reasoning at the class level. TopDown also has the largest **Integration Errors** share (**9.0%**), implying high-level planning can leave inconsistencies in state evolution and cross-method coordination. Compositional shows a mixed profile with **Logic Errors** at **59.0%** and **Dependency Errors** at **37.0%**, indicating independently generated fragments remain hard to reconcile into a coherent class.

Overall, these results reveal a trade-off between dependency preservation and semantic correctness. Strategies with explicit decomposition (BottomUp, TopDown) alleviate dependency failures but shift the burden toward logic and integration problems, whereas Incremental suffers most from context fragmentation. Generation strategy thus changes *how* models fail rather than eliminating failure: unstructured generation exposes dependency drift, while structured generation reveals unresolved semantic composition and class-level coordination challenges.

Dependency Error: Signature Drift

GPT-5.1 · Incremental

Required functionality

base_select(fields=None, condition=None) should use self._context_table internally, without adding an explicit table argument.

Generated code

```
def base_select(self, table, fields=None, condition=None):
    # Incorrectly adds 'table' as explicit parameter
    cols = ", ".join(fields) if fields else "*"
    sql = f"SELECT {cols} FROM {table}"
    if condition:
        sql += f" WHERE {condition}"
    return sql
```

Test failure

```
TypeError: ContextualQueryRegistry.base_select() missing 1
required positional argument: 'table'
Target: base_select | Input: None
```

Root cause / correction hint

The generated method changes the original interface by inventing a new required parameter. This breaks existing callers and shows that the model failed to preserve the class-level method contract. The correct implementation should keep the original signature and read from self._context_table internally.

Figure 6: A representative dependency error case

5.4.5 Concrete Error Examples. Figure 6 shows a representative **Dependency Error** in which the generated class fails to preserve an existing method contract. The required functionality specifies that `base_select(fields=None, condition=None)` should read the table name internally from `self._context_table`, without introducing any additional interface requirement. However, the generated implementation changes the method signature to `base_select(self, table, fields=None, condition=None)`, thereby inventing a new required positional parameter. As a result, the method remains locally executable, but immediately breaks existing callers and triggers a `TypeError` at test time.

This example illustrates a central challenge in class-level code generation: many failures do not arise from syntax or obviously nonsensical code, but from subtle violations of class-internal dependencies and interface consistency. The generated method is

superficially plausible, yet it no longer conforms to the surrounding class design because it fails to preserve how shared state is accessed and how downstream callers invoke the method. More broadly, this case reinforces our earlier findings that runnable generations can still fail when the model does not reliably maintain class-level contracts, dependency structure, and cross-method consistency across the full implementation.

6 THREATS TO VALIDITY

External validity. Our study targets Python exclusively; results may not generalize to statically-typed or systems-level languages. We mitigate temporal bias by sourcing only GitHub data from January 2025 onward, and our pipeline is language-agnostic by design. **Internal validity.** LLM-assisted construction may introduce biases; we mitigate this via multi-stage quality control including compilation checks and test-suite validation with >90% line coverage. Annotation subjectivity is addressed by three independent annotators with inter-annotator agreement (Kappa > 0.7) and consensus-based conflict resolution.

7 CONCLUSION

We presented ClassEval-Pro, a benchmark targeting *compositional code creation*—the underserved evaluation dimension between function-level synthesis and repository-level code modification. Our automated, contamination-resistant pipeline constructs 300 cross-domain class-level tasks spanning 11 domains from post-January-2025 GitHub code, replacing manual curation that limits scalability and temporal validity in prior benchmarks. Experiments with five frontier LLMs and five generation strategies reveal that compositional creation remains far from solved: the best model (Qwen3-480B) achieves only 45.6% class-level Pass@1, with a 17.7-point gap separating the strongest and weakest models. Strategy choice substantially impacts performance—top-down and bottom-up approaches improve weaker models by up to 9.4 points, while compositional generation collapses to as low as 1.3%. Error analysis over 500 manually annotated failed cases shows that logic errors (56.2%) and dependency errors (38.0%) dominate, identifying cross-method coordination as the core bottleneck. We release the benchmark, pipeline, and evaluation scripts to support future research.

DATA AVAILABILITY

Our code and benchmark are available at https://anonymous.4open.science/r/anonymous-repo-5153.

REFERENCES

[1] Jacob Austin, Augustus Odena, Maxwell Nye, et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[2] Mark Chen, Jerry Tworek, and ohers. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[3] Silin Chen, Shaoxin Lin, Xiaodong Gu, et al. 2025. Swe-exp: Experience-driven software issue resolution.

[4] Xinyun Chen, Ryan A. Chi, Xuezhi Wang, and Denny Zhou. 2024. Premise Order Matters in Reasoning with Large Language Models. *arXiv preprint arXiv:2402.08939* (2024).

[5] Google DeepMind. 2026. *Gemini 3 Pro Model Card.* Technical Report.

[6] 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).

[7] Xueying Du, Mingwei Liu, Kaixin Wang, et al. 2023. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation. *arXiv preprint arXiv:2308.01861* (2023).

[8] Yixiong Fang, Tianran Sun, Yuling Shi, Min Wang, and Xiaodong Gu. 2025. LastingBench: Defend Benchmarks Against Knowledge Leakage.

[9] GitHub. 2026. GitHub REST API Documentation. https://docs.github.com/en/rest.

[10] Alex Gu, Baptiste Rozière, Hugh Leather, et al. 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. In *Proc. ICML*, Vol. 235. 16568–16621.

[11] Jiawei Gu, Xuhui Jiang, Zhichao Shi, et al. 2025. A Survey on LLM-as-a-Judge. *arXiv preprint arXiv:2411.15594* (2025).

[12] Daya Guo, Qihao Zhu, Dejian Yang, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence.

[13] Xinyi Hou, Yanjie Zhao, Yue Liu, et al. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *arXiv preprint arXiv:2308.10620* (2024).

[14] Naman Jain, King Han, Alex Gu, et al. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *arXiv preprint arXiv:2403.07974* (2024).

[15] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *Proc. ICLR*.

[16] Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. XCodeEval: An Execution-based Large Scale Multilingual Multitask Benchmark for Code Understanding, Generation, Translation and Retrieval. In *Proc. ACL*. 6766–6805.

[17] Denis Kocetkov, Raymond Li, Loubna Ben Allal, et al. 2022. The Stack: 3 TB of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).

[18] Jia Li, Ge Li, Yunfei Zhao, et al. 2024. DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv preprint arXiv:2405.19856* (2024).

[19] Raymond Li, Loubna Ben Allal, Yangtian Zi, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[20] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *arXiv preprint arXiv:2305.01210* (2023).

[21] 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 *Proc. NeurIPS*, Vol. 36. 21558–21570.

[22] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proc. ICSE*. 111–120.

[23] 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).

[24] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Improving Neural Machine Translation Models with Monolingual Data. *arXiv preprint arXiv:1511.06709* (2016).

[25] Yuling Shi, Songsong Wang, Chengcheng Wan, Min Wang, and Xiaodong Gu. 2024. From Code to Correctness: Closing the Last Mile of Code Generation with Hierarchical Debugging.

[26] Aaditya Singh, Adam Fry, et al. 2025. OpenAI GPT-5 System Card. *arXiv preprint arXiv:2601.03267* (2025).

[27] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. *arXiv preprint arXiv:2212.10560* (2023).

[28] Jason Wei and Kai Zou. 2019. EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. *arXiv preprint arXiv:1901.11196* (2019).

[29] Can Xu, Qingfeng Sun, Kai Zheng, et al. 2025. WizardLM: Empowering large pre-trained language models to follow complex instructions. *arXiv preprint arXiv:2304.12244* (2025).

[30] An Yang, Anfeng Li, et al. 2025. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388* (2025).

[31] Fengji Zhang, Bei Chen, Yue Zhang, et al. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. *arXiv preprint arXiv:2303.12570* (2023).

[32] Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proc. ICSE*. 1–12.

[33] Qiming Zhu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Le Sun, and Shing-Chi Cheung. 2024. DOMAINEVAL: An Auto-Constructed Benchmark for Multi-Domain Code Generation. *arXiv preprint arXiv:2408.13204* (2024).

[34] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, et al. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *Proc. ICLR*.