PostCondBench: Benchmarking Correctness and Completeness in Formal Postcondition Inference

Gehao Zhang and Juan Zhai
Manning College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA, USA
{gehaozhang,juanzhai}@umass.edu

Abstract

Formal postconditions precisely characterize program behavior and support debugging, testing, and verification, but writing them requires substantial expertise and effort. This has motivated recent work on automatically generating postconditions from code and natural-language artifacts using large language models (LLMs). However, evaluation remains a key bottleneck. Existing benchmarks primarily emphasize correctness under limited evaluation settings, often relying on surface-form matching or manual assessment on small or synthetic datasets.

We introduce PostCondBench, a multilingual benchmark for evaluating method-level postcondition generation from real-world software. PostCondBench comprises 420 Python and Java tasks drawn from 121 open-source projects, each paired with a high-quality ground-truth postcondition set constructed with expert involvement. To enable automatic evaluation, PostCondBench provides a runnable execution environment and operationalizes completeness via *defect discrimination*: a postcondition set is more complete if it is violated by more defective implementations while remaining satisfied on correct executions. Using PostCondBench, we formulate three generation settings and evaluate five SOTA LLMs. Our results reveal a substantial gap between correctness and completeness, and show that repository-level dependencies and method complexity exacerbate this gap.

1 Introduction

A formal specification expresses the properties that a program is expected to satisfy (Lamsweerde, 2000). By precisely characterizing program behavior, specifications play a central role in debugging, testing, and verification. Despite their importance, formal specifications are scarce in real-world software, as writing them is time-consuming, error-prone, and requires substantial expertise (Lamsweerde, 2000; Snook, 2001; Henkel et al., 2008). As a result, a long line of work has studied the automatic generation of specifications from source code and natural-language (NL) artifacts, aiming to reduce the cost and expertise required for writing specifications (Goffi et al., 2016; Blasi et al., 2018; Zhai et al., 2020; Zhang et al., 2020; Xie et al., 2022; Pandita et al., 2012).

Large language models (LLMs) have significantly improved generative capabilities for software engineering tasks, including code generation and testing (Chen et al., 2021; Islam et al., 2024; Xue et al., 2024). Several studies have shown that LLMs can also generate method-level postconditions directly from code or NL descriptions (Xie et al., 2025; Endres et al., 2024; Ma et al., 2025). However, *how to reliably evaluate generated postconditions* remains an open challenge.

Most existing automatic evaluations focus on *correctness*. Under sufficient test inputs, a postcondition is deemed correct if it is never violated when executed against a correct implementation (Zhai et al., 2020; Endres et al., 2024). Correctness alone is insufficient as a correct postcondition set can still overlook important behavioral aspects of the method, i.e., they can be incomplete.

Evaluating *completeness* is fundamentally difficult because there is no single canonical specification. The same behavioral property can often be expressed by many syntactically different but semantically equivalent postconditions (Endres et al., 2024; Xie et al., 2025). This diversity makes matching against a fixed ground truth, e.g., via string or AST similarity, brittle and prone to penalizing semantically correct specifications. Consequently, prior work has largely relied on manual assessment for evaluating completeness (Xie et al., 2025; Zhai et al., 2020; Ma et al., 2025) (examples in Figure 11). Existing attempts at automatic completeness evaluation remain limited. For instance, Endres et al. (2024) propose a completeness metric, but evaluate it only on HumanEval+ (Liu

Findings of the Association for Computational Linguistics: ACL 2026, pages 35478–35507

et al., 2023), a small dataset of standalone tiny Python programs. To date, there is no benchmark that simultaneously provides (i) multilingual, real-project-derived tasks, (ii) high-quality ground-truth postconditions, and (iii) automatic support for correctness and completeness evaluation.

We present PostCondBench, a multilingual benchmark for evaluating the correctness and completeness of method-level postconditions, a widely used form of formal specification that constrains program states after method execution (Lamsweerde, 2000). PostCondBench comprises 420 postcondition generation tasks in Python and Java, drawn from 121 diverse and popular open-source projects. Each task is paired with a comprehensive, high-quality postcondition set constructed via a semi-automated pipeline with domain-expert involvement. Moreover, PostCondBench provides a runnable execution environment with abundant context and tests, together with code-coverage analysis to ensure sufficiently exercised executions.

Crucially, PostCondBench operationalizes *completeness* via *defect discrimination*: a postcondition set is more complete if it distinguishes more defective implementations from the reference set while remaining satisfied on correct executions. This formulation enables automatic computation without relying on brittle equivalence checking against a single canonical specification.

Using PostCondBench, we formulate three postcondition generation tasks and evaluate five state-of-the-art (SOTA) LLMs. Our results show that completeness remains challenging even for strong models (e.g., at most 17% in Python and 43% in Java for GPT-5), and that there is a substantial gap between being *correct* and being *complete*. We further analyze common causes of incorrect postconditions and the behavioral aspects most frequently overlooked by incomplete ones, highlighting concrete directions for future research.

Our main contributions are:

- PostCondBench, a multilingual benchmark of 420 Python/Java method-level postcondition generation tasks from 121 real-world repositories, with high-quality ground-truth postconditions. Unlike prior benchmarks based on standalone programs, PostCondBench captures repository-level context, including inter-method dependencies and non-trivial method complexity. We quantitatively characterize these factors via dependency analysis and method size (MLOC).
- An automatic evaluation platform that supports both correctness and completeness, with completeness measured via defect-discriminative power.
- Extensive evaluation of five SOTA LLMs on three generation tasks reveals a substantial gap between correctness and completeness, and shows that repository-level dependencies and method complexity further widen this gap.

2 Background and Related Work

Prior work on specification generation includes static analysis, dynamic analysis, and data-driven approaches. Static techniques infer specifications from program structure, often providing soundness guarantees but struggling with scalability and precision (Chen et al., 2016; Shoham et al., 2007; Flanagan and Leino, 2001). Dynamic approaches learn likely specifications from execution traces, but their effectiveness depends heavily on test coverage (Ernst et al., 2001; Nimmer and Ernst, 2002). Repository-mining methods exploit recurring patterns across large codebases, yet are sensitive to repository quality and consistency (Ramanathan et al., 2007; Nguyen et al., 2014).

Recently, LLMs have been applied to generate formal specifications from NL and/or code (Kreber and Hahn, 2021; Cosler et al., 2023; Endres et al., 2024; Xie et al., 2025). While much of this work focuses on temporal properties such as linear temporal logic (LTL), recent studies demonstrate the promise of LLMs for generating method-level postconditions. However, evaluations are conducted on small, standalone programs, limiting their ability to reflect real-world software complexity.

Evaluating generated specifications remains difficult because there is no single canonical "correct and complete" specification: multiple formulations may be semantically valid yet differ substantially in behavioral coverage (Endres et al., 2024; Xie et al., 2025). As a result, prior work frequently relies on manual evaluation or restricts attention to tasks with well-defined ground truths (Zhai et al., 2020; Kreber and Hahn, 2021).

Automatic evaluation of *completeness* has received limited attention. Endres et al. (2024) explore completeness checking for postconditions, but is limited to a small, Python-only dataset of standalone programs. Existing benchmarks rarely

capture repository-level code, cross-language diversity, or the gap between test-based correctness and true behavioral completeness.

In contrast, PostCondBench provides a multilingual, repository-level benchmark with automatically evaluable ground-truth postconditions. By combining test-based validation with mutant-based completeness checking, it enables systematic measurement of both correctness and completeness in realistic software settings.

3 Methodology

3.1 Benchmark Overview

Our benchmark targets formal postcondition inference, evaluating whether approaches can generate precise and comprehensive postconditions from NL descriptions and/or code implementations. We construct a multilingual, repository-level dataset with automated evaluation capabilities. Formally, each PostCondBench instance is a tuple that captures a method, its documentation, implementation, and evaluation artifacts: $x = \langle \text{sig}, \text{nl}, \text{impl}, \mathcal{T}, \mathcal{M}, \mathcal{P} \rangle$ where sig is the method signature; nl, the NL description; impl, the code implementation; $\mathcal{T}$, a set of test cases for correctness evaluation; $\mathcal{M}$, a set of implementation mutants for completeness evaluation; and $\mathcal{P}$, a set of ground-truth postconditions. Figure 1 shows a representative instance example from `joowani/binarytree` (Joowani, 2022).

Our postconditions are written in widely adopted formats, Java Modeling Language (JML (Leavens et al., 2008)) for Java and icontract (Parquery, 2025) for Python. Unlike Endres et al. (2024), which uses native assertion statements as postconditions, and Xie et al. (2025), which uses a custom-defined format, the postcondition languages/frameworks used here are mature and more expressive.

Our postconditions capture intra- and inter-class dependencies, and preserve pre-state information. For example, the postcondition `Arrays.deepEquals(\old(this.items()), \result.items())` uses `\old(this.items())` to preserve the value of `this.items()` prior to method execution. Further, JML provides full support for logical connectives (implication, equivalence, and inequivalence; written as $\Rightarrow$, $\Leftrightarrow$, and $\not\Leftrightarrow$) While existing research highlights that these logical connectives improve writing and understanding (Brown, 2024; Baron et al., 2024), Endres et al. (2024) and Xie et al. (2025) do not provide sufficient support for them.

This dataset is designed for flexibility and broad applicability, with a primary focus on specification generation. For generation from NL, we use $x^{\text{genFromNL}} = (\text{sig}, \text{nl}, \mathcal{T}, \mathcal{M}, \mathcal{P})$; for inference from implementations, $x^{\text{genFromCode}} = (\text{sig}, \text{impl}, \mathcal{T}, \mathcal{M}, \mathcal{P})$. Beyond specification tasks, the dataset also enables applications such as code generation using $x^{\text{code}} = (\text{sig}, \text{nl}, \text{impl}, \mathcal{T}, \mathcal{P})$, which supports workflows that require grounding code synthesis in developer intent.

3.2 Dataset Construction Pipeline

We construct PostCondBench through a multi-step pipeline, illustrated in Figure 2.

3.2.1 Target Method Selection

We select candidate target methods from open-source repositories, requiring runnable environments and methods with adequate documentation, non-trivial complexity, and high test coverage.

Repository Selection. We mine popular and diverse GitHub repositories: for Python/Java, we retrieve repos that (1) use a top-200 frequent tag, (2) are $\leq$ 20 MB, and (3) have a permissive license, then keep the top 5,000 most-starred per language. These sets cover all top-200 tags, ensuring broad topical diversity.

Automated Environment Setup. To enable testing and coverage, each repository must run locally with tests passing and provide a coverage report (e.g., JaCoCo (jacoco, 2025), pytest-cov (pytest-dev, 2025)). To scale the otherwise labor-intensive setup (e.g., Pan et al. (2025) reports over 200 hours spent on setup), we adopt an LLM-driven loop: starting from a clean machine with dependency managers (e.g., Poetry (python-poetry, 2026), Maven (Miller et al., 2010)), our module checks requirements and, upon failure, queries an LLM (`gpt-o4-mini`) to propose edits to core config files (e.g., `pyproject.toml`, `pom.xml`), applies them, and iterates for a bounded number of rounds. Out of 10,000 repositories, 941 are successfully set up (9.4%), slightly better than SWE-agent (Hu et al., 2025; Yang et al., 2024).

Method Parsing and Filtering. We parse methods from the AST (Latif et al., 2023), align coverage reports to obtain per-method line coverage, and retain methods that satisfy: ❶ *Comment Quality:* a comment with >15 English words (and we manually remove obvious mismatches); ❷ *Code*

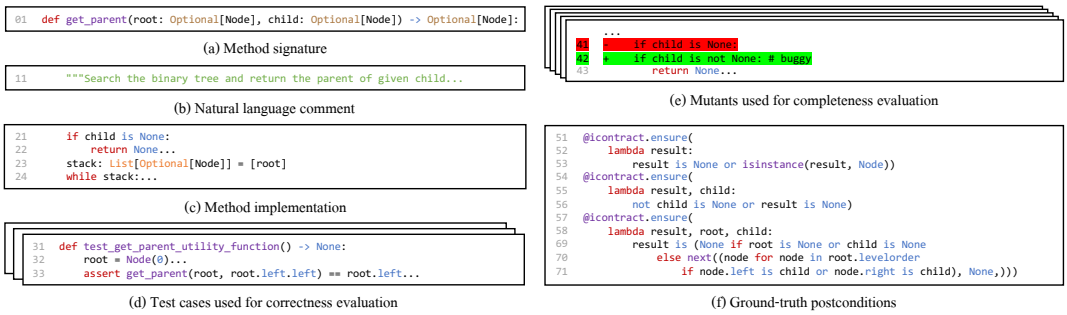


Figure 1: An example instance from POSTCONDBENCH, illustrating the components of a benchmark instance. The instance corresponds to the target method `get_parent`.

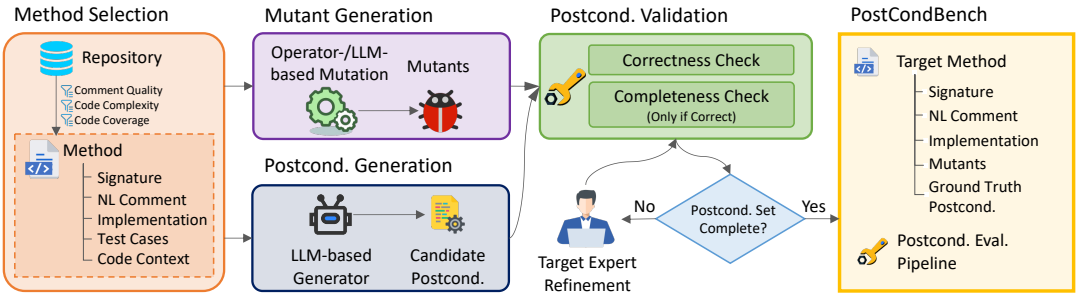


Figure 2: Overall workflow of POSTCONDBENCH.

Complexity: at least 15 lines of code or cyclomatic complexity (McCabe, 1976) ≥ 3; ❸ *Code Coverage:* $\geq 90\%$ line coverage. We exclude methods that throw exceptions under normal execution. This yields **1,030** Python and **1,476** Java methods.

3.2.2 Mutant Generation

We generate mutants for each selected method following mutation-testing principles, which introduce small, controlled perturbations that approximate realistic bugs. This supports completeness evaluation: a complete postcondition set should be violated when the method runs on a defective implementation. We use operator-based mutation and LLM-based mutation, and keep only defective mutants that fail the original test suite.

Operator-based Mutation. We apply a single, localized syntactic transformation to the original method body (e.g., replacing a return value with `None`, modifying conditional operators, or removing void method calls). For Python, we reuse mutation operators from Mutmut (Hovmöller, 2016). For Java, we reproduce the default mutation operators from PIT (Coles et al., 2016).

LLM-based Mutation. Operator-based mutation is efficient but limited in diversity and coupling strength (Wang et al., 2024). To complement it, we use an LLM-based mutation strategy. We parse the AST to locate lines that define conditions in `if`, `while`, and `do` statements, loop headers in `for` statements, and method calls. We replace each target line with a placeholder and prompt `gpt-4o-mini` to generate an alternative line that introduces a defect.

We discard mutants that still pass all tests and remove those that throw exceptions under normal execution, to ensure postconditions are exercised. We retain only methods with at least five valid mutants. This process yields 999 Python methods and 1,431 Java methods, with an average of 24.8 mutants per method.

3.2.3 Postcondition Generation

We use an LLM to draft postcondition sets for each method, reducing manual effort in ground truth construction. The model is given (i) the target method, (ii) repository code, and (iii) a list of mutants, and is asked to propose postconditions that hold for the original code but fail for defective mutants.

We use `gpt-5-mini` with reasoning enabled. The prompt includes the method and, for each mutant, a line-by-line diff. We provide repository context by uploading a zip archive and using OpenAI's code interpreter tool (OpenAI, 2023) to access relevant files.

3.2.4 Postcondition Validation

We validate each postcondition set along two axes: correctness on the original implementation and completeness against buggy mutants. A set is correct if the original method satisfies it across the provided tests; it is complete if it also distinguishes every generated mutant.

Correctness. A postcondition set is correct if a correct implementation satisfies it for all (legal) inputs. We treat the developer test suite as the input distribution and require high test quality (line coverage $\geq 90\%$). For a target method m and a postcondition set p, we instrument m with p and run the test suite, obtaining $\mathsf{eval}(m, p) \in \{-1, 0, 1\}$: $\mathsf{eval}(m, p) == 1$ iff all tests pass and all postconditions hold; $\mathsf{eval}(m, p) == 0$ iff any postcondition is violated; and $\mathsf{eval}(m, p) == -1$ iff the run fails due to other exceptions. We define correctness as

$$\mathsf{corr}(m, p) == True \iff \mathsf{eval}(m, p) == 1$$

Our evaluation of correctness relies on test suites for the execution distribution. To reduce noise from weak or sparse test suites, we filter tasks by test coverage and retain only methods with high line coverage. Although high code coverage does not necessarily imply high test quality (e.g., assertions may be weak and corner cases may remain undertested), which can lead to optimistic estimates in dynamic evaluation, we expect this criterion to preferentially select tasks with relatively stronger test suites in practice. The final benchmark achieves an average line coverage of 99%.

Completeness. Correctness alone can be vacuous; a postcondition set may pass tests yet fail to capture essential behavior. We assess completeness via mutation testing. For each method m, we generate a mutant set $\{b_1, \ldots, b_q\}$ (Section 3.2.2). For a correct postcondition set p, we say p kills mutant b_h if $\mathsf{eval}(b_h, p) == 0$. A postcondition set is bug-complete if it is correct and kills all mutants:

$$\mathsf{comp}(m, p) == True \iff$$
$$\mathsf{eval}(m, p) == 1 \wedge \left[\bigwedge_{h=1}^{q} \mathsf{eval}(b_h, p) == 0\right]$$

This mutation-based completeness is an operational proxy for behavioral coverage rather than a definitive measure of semantic completeness. Our mutation approach supports sufficient mutant quality and diversity, leading to a comprehensive and stable evaluation on completeness (see Appendix F).

3.2.5 Expert Refinement and Finalization

If the LLM-generated postcondition set is not bug-complete, two domain experts refine it by inspecting the method, repository context, and mutants, and re-running validation until bug-completeness is reached; disagreements are resolved by consensus. Each expert has over five years of programming experience and more than one year of experience with formal specifications. If the generated set is already bug-complete, we adopt it as the ground truth. We exclude methods (or specific mutants) that require unsupported specification constructs or remain unkillable under the available tests. All removals are reported in the Appendix H. Most of these exclusions arise from representational limits of the underlying specification frameworks.

Benchmark Finalization. From 999 Python and 1,431 Java methods with sufficient mutants, we select a diverse subset via farthest-first traversal (Rosenkrantz et al., 1977) within each language, using cosine distance over CodeBERT (Feng et al., 2020) embeddings of method headers. We then interleave automated generation with expert refinement and filtering, yielding PostCondBench with 210 methods per language, each with a bug-complete postcondition set.

As shown in Table 1, methods in PostCondBench exhibit non-trivial size and complexity, with rich test coverage and multiple ground-truth postconditions per method, reflecting realistic repository-level scenarios.

Table 1: Statistics of PostCondBench for Python and Java. All metrics are reported as method-level averages, except for the number of methods and repositories.

Statistic	Python	Java
Methods	210	210
Repositories	61	60
Comment Word Length	44.6	46.9
Test Cases	27.4	40.0
Line Coverage	98.8%	99.3%
Lines of Code	30.4	30.6
Cyclomatic Complexity	5.0	5.2
# Mutant	26.6	21.9
# Postcondition	3.6	5.5

4 Experimental Setup

For each target method and each LLM, we evaluate postcondition generation under three input

settings. Under each setting, we independently sample 5 candidate postcondition sets. Each candidate is evaluated for test-based correctness and bug-completeness. Unless otherwise stated, we report single-sample (@1) results in the main paper and defer multi-sample (@k) results to the appendix.

Input Settings. We evaluate postcondition generation under three input settings: ❶ Code-only (C2P), which provides only the method implementation; ❷ NL-only (N2P), which provides only NL comments; and ❸ Code+NL (F2P), which provides both. These settings allow us to study how different input modalities affect correctness, completeness, and their discrepancy.

Models. We evaluate 5 SOTA LLMs, covering proprietary and open-weight families: GPT-5 (OpenAI, 2025), Claude-4.5 (Sonnet) (Anthropic, 2025), LLaMA-4 (Maverick) (AI, 2025), Qwen3-32B (Yang et al., 2025), and Gemma-3-27B (Mesnard et al., 2024). Details are in Appendix D.1.

Evaluation Metrics. We report Corr@k and Comp@k, which assess that when we generate k postcondition sets for a method independently, the expectation that at least one generation result is correct/complete; full details are in Appendix D.3.

Evaluation Protocol. For each model, input setting, and method, we evaluate generated postconditions using the metrics above. We report model-level aggregates (e.g., Corr@1, Comp@1) in tables and method-level gap distributions in violin plots.

5 Results

We now present experimental results on PostCondBench, covering overall performance, the relationship between correctness and completeness, the effect of different input modalities, and fine-grained analyses of model failures.

5.1 How Large Is the Gap Between Correctness and Completeness?

We evaluate LLM-generated postcondition sets under both single-sample and multi-sample settings. Table 2 reports Corr@k and Comp@k, aggregated across languages and input settings. Across all k, Claude-4.5 achieves the highest correctness, while GPT-5 achieves the strongest completeness.

To quantify this discrepancy, we measure the absolute gap $\Delta@1 = \text{Corr@1} - \text{Comp@1}$ and the conditional completeness rate $\rho@1 = \text{Comp@1}/\text{Corr@1}$. Across models, $\Delta@1$ is large (0.063–0.423) and $\rho@1$ is below 0.5 for most models, indicating fewer than half of test-correct outputs are complete. Notably, higher Corr@1 can coincide with a larger gap (e.g., Claude-4.5).

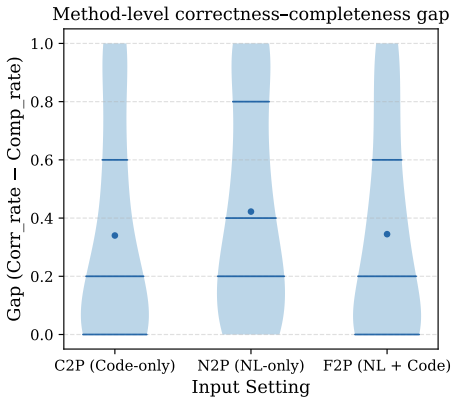


Figure 3: Distribution of method-level correctness–completeness gaps under different input settings. For each method m, the gap is computed as $\Delta(m) = \text{corr\_rate}(m) - \text{comp\_rate}(m)$, where corr_rate and comp_rate denote the fraction of generated postcondition sets that are test-correct and complete, respectively. Each violin aggregates gap values across methods for which at least one test-correct postcondition is generated; methods with Corr@k = 0 are excluded. Horizontal lines indicate the 25th, 50th, and 75th percentiles, while dots denote the mean.

Increasing the sample budget (i.e., increasing k in Corr/Comp@k metrics) improves both correctness and completeness, but does not remove the gap between them. For Corr@k/Comp@k, GPT-5 improves from 0.483/0.255 at @1 to 0.802/0.446 at @5, while Claude-4.5 improves from 0.629/0.207 at @1 to 0.822/0.292 at @5. Additional sampling increases the chance of obtaining a usable specification. However, completeness still lags substantially behind correctness.

Figure 3 analyzes method-level gaps, computed as $\Delta(m) = \text{corr\_rate}(m) - \text{comp\_rate}(m)$ over multiple generations. We focus on methods with Corr@$k > 0$ (excluding Corr@$k = 0$) to isolate incompleteness *after* correctness is achieved. The distribution is broad and right-skewed: the median gap is consistently positive ($\sim$0.2) and the upper quartile reaches roughly 0.6 to 0.8, showing the gap is widespread rather than driven by a few outliers.

Input modality affects the *severity* of the gap but not its existence. All settings exhibit sizable gaps; even F2P has median $\sim$0.2 with a non-trivial tail above 0.5. The gap is most pronounced under N2P,

Table 2: Postcondition generation performance of LLMs. We report Corr@k, Comp@k, Δ@k, and Comp/Corr for $k \in \{1, 3, 5\}$, where Corr@k indicates the expectation that at least one of the k generated postcondition sets is test-correct and Comp@k indicates for at least one is complete. Results are aggregated across input settings and programming languages.

Model	@1				@3				@5			
	Corr	Comp	Δ	Comp/ Corr	Corr	Comp	Δ	Comp/ Corr	Corr	Comp	Δ	Comp/ Corr
GPT-5	0.483	**0.255**	0.227	**0.529**	0.714	**0.386**	0.329	**0.540**	0.802	**0.446**	0.356	**0.556**
Claude-4.5	**0.629**	0.207	**0.423**	0.329	**0.777**	0.268	**0.509**	0.345	**0.822**	0.292	**0.530**	0.355
LLaMA-4	0.214	0.094	0.120	0.441	0.336	0.128	0.209	0.379	0.395	0.144	0.251	0.365
Qwen3-32B	0.118	0.055	0.063	0.466	0.189	0.080	0.109	0.423	0.229	0.093	0.136	0.406
Gemma-3-27B	0.110	0.044	0.066	0.401	0.171	0.058	0.113	0.339	0.204	0.064	0.140	0.315

with the largest median and widest spread (many methods above 0.6, extremes near 1.0). C2P and F2P are more concentrated; F2P slightly reduces the upper tail compared to C2P, but the two remain close overall, suggesting code provides most of the constraining signal for completeness.

> Correctness does not reliably imply completeness: large, systematic correctness–completeness gaps persist across methods, input settings, and sample budgets. NL-derived postconditions exhibit greater variance and heavier upper tails

5.2 What Are the Major Failure Modes of Incorrect Postconditions?

To understand why LLMs produce invalid specifications, we analyze failure modes of incorrect postconditions. We sample 50 incorrect postcondition sets from GPT-5 and Claude-4.5; for each set, we select one postcondition that triggers the failure and manually label its primary cause.

Specification Language / API Misuse (54%). Most failures stem from incorrect contract syntax or framework APIs (54%), leading to compilation/-type errors or runtime contract violations.

Note that many of these misuses are not superficial syntax mistakes. More than half of Java mistakes in this category are misusing post-state variables in `\old(·)`. In JML, the `\old(·)` is used for snapshotting the value of the inputted expression at the pre-state. Expression that input to `\old(·)` should only rely on variables that existed in the pre-state. However, as an example in Figure 4, the generated postcondition uses `entry` in `\old(·)`, while `entry` is a temporary variable that is created when calling the `allMatch(·)` API in the post-state. Rather than merely a syntax error, this highlights an insufficient understanding of the boundary between pre-state and post-state.

To test whether this failure mode is actionable, we add a brief grammar-guidance block to the F2P prompt, summarized from the official icontract and JML documentation, including an explanation of the specification language / API in use, with brief examples. We test GPT-5 and Claude-4.5. Table 11 shows that both correctness and completeness improve noticeably (e.g., 0.629 → 0.814 for Claude-4.5 Corr@1, and 0.207 → 0.283 for Comp@1). However, the gap between correctness and completeness remains large. Our claim still holds: even after these improvements, completeness remains challenging even for state-of-the-art models, and a large correctness–completeness gap is retained.

Semantic Overreach (34%). This error arises when LLMs enforce behaviors unsupported by the implementation or documentation. For example, Figure 9 (Appendix) shows cases where LLMs formalize non-existent semantics.

Other Low-frequency Issues (12%). The remaining cases include basic syntactic mistakes such as mismatched brackets (6%) and null dereferences (6%), where postconditions access `None/null` values without proper guards.

> Incorrect postconditions are primarily caused by specification-language misuse (54%), indicating that improved model awareness of contract syntax and APIs could eliminate a large fraction of failures.

5.3 Which Behaviors Are Missed by Correct but Incomplete Postconditions?

We study postcondition sets that are test-correct on the reference implementation but fail to kill at least one mutant, revealing missing behavioral con-

```
01 //@ ensures !model.containsKey("g") || model.get("g").entrySet()
       .stream().allMatch(entry -> entry.getValue() ==
       \old(model.get("g").get(entry.getKey())));
```

(a) Incorrect postcondition

```
11 //@ ensures !model.containsKey("g") || model.get("g").entrySet()
       .stream().allMatch(entry -> entry.getValue() ==
       \old(Map.copyOf(model)).get("g").get(entry.getKey()));
```

(b) Correct postcondition

Figure 4: An example of incorrect postconditions caused by LLMs' misuse of specification languages or APIs, with corrected versions shown on the right. Highlighted regions indicate key corrections.

straints. We focus on GPT-5 and Claude-4.5, and manually inspect 50 randomly sampled (postcondition set, unkilled mutant) pairs.

Under-specified Return Value Behavior (78%). Most incompleteness arises from weak constraints on return values: postconditions often validate only special cases, types, or coarse ranges, leaving core computation insufficiently specified. Consequently, output-perturbing or logic-bypassing mutants go undetected (Figure 8 in Appendix). Across the sampled cases, missed return-value behaviors span multiple return types: scalar or primitive-like values (34%), built-in collections of scalars (26%), repository-defined types (14%), and third-party library types (4%). Notably, over one third of missed cases involve simple scalar returns, indicating that incompleteness often stems from weak semantic constraints rather than from output complexity.

```
def __init__(self, *, points):
    self._points = points

01  def __init__(self, *, points,
02 -    super().__init__(points=points)
03 +    super().__init__(points=None)...
11  @icontract.snapshot(lambda points: points, name="points")...
12  @icontract.ensure(lambda OLD, self: self._points is OLD.points)
21  @icontract.ensure(lambda self, colors: self.colors is colors)
22  @icontract.ensure(lambda self, n_x,
                                  No validation on self._points
```

Figure 5: Example of a correct but incomplete postcondition set. Top: Target method with mutant changes ("+"/"–"). Middle: Complete postconditions. Bottom: Test-correct but incomplete postconditions.

Unconstrained Object State (16%). Incompleteness also stems from missing constraints on object state. As shown in Figure 5, while other fields are validated, the initialization of `self._points` remains unconstrained, allowing a state-altering mutant to pass undetected.

Lack of Defensive Guards (6%). Some postconditions lack basic robustness, raising runtime exceptions (e.g., dereferencing None) under mutant-induced states rather than failing as specifications. Adding simple defensive guards before attribute access would correctly reject these mutants.

Table 3: Postcondition generation performance on non-standalone (**Dep.**) and standalone (**Solo.**) methods. **Dep.** requires repository-specific dependencies, while **Solo.** is runnable with only built-ins. Bold indicates the higher score within each comparison.

Language	Group	Corr@1	Comp@1
Python	Dep.	0.181	0.035
	Solo.	**0.239**	**0.090**
Java	Dep.	0.410	0.188
	Solo.	**0.488**	**0.316**

Most incompleteness stems from weak constraints on return values, often involving simple scalar outputs. This suggests that systematic strengthening of return-value specifications, rather than handling complex structures, could substantially improve completeness.

5.4 How Do Method Characteristics Affect Postcondition Quality?

We analyze two method-level factors that affect postcondition generation: dependency complexity (standalone vs. non-standalone) and method length.

Dependency Complexity. Following prior work (Yu et al., 2024), we label a method as *standalone* if it runs with only built-in types and standard libraries; otherwise it is *non-standalone*. As shown in Table 3, when averaging across models and input settings, standalone methods consistently achieve higher scores. The stratified results (Table 10 in Appendix) show this trend is robust: across 60 paired comparisons (2 metrics $\times$ 2 languages $\times$ 5 models $\times$ 3 settings), standalone wins in 51 cases (85%). The gap is larger for completeness: average Comp@1 rises from 0.035 to 0.090 in Python (2.6$\times$) and from 0.188 to 0.316 in Java (1.7$\times$), suggesting that external dependencies make it harder to produce sufficiently constraining postconditions.

Method Length. We bucket methods by lines of code (LoC): $[0, 20)$, $[20, 40)$, and $[40, \infty)$. As shown in Figure 6, performance generally de-

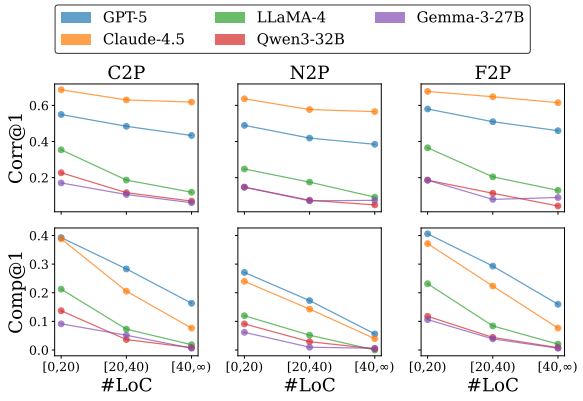


Figure 6: Corr@1 (top) and Comp@1 (bottom) across target methods grouped by lines of code (LoC).

creases with LoC, and the drop is much steeper for completeness. For example, under C2P with Claude-4.5, Comp@1 falls from 0.388 to 0.206 and 0.076 from short to medium and long methods. Aggregated over models, languages, and input settings, the overall average score decreases from 0.296 to 0.205 and then to 0.149 as LoC increases, confirming that longer methods pose a substantially harder generation problem.

> Method characteristics strongly affect postcondition generation: standalone methods are easier than dependency-heavy ones, and longer methods degrade performance monotonically, with completeness being especially sensitive.

5.5 How Stable Is Completeness Under Mutant-Set Changes?

We provide ablations to analyze how mutation design affects completeness (details in Appendix F).

Operator-level ablation. Each language includes 11 operator-based mutation types. We remove each operator individually and compute Comp@1 (averaged across models). In Python, dropping any single operator changes Comp@1 by at most 0.004; in Java, most operators shift Comp@1 by $\leq$0.001 (see Table 9).

Mutation-strength ablation. We further vary mutation "strength" by (i) randomly removing 5 of the 11 operators and (ii) randomly removing 50% of LLM-based mutants (50 trials each). Across all models, the mean Comp@1 changes by less than 4% relative to the full-mutant baseline, with standard deviation $\leq$0.004 in all settings (see Table 8).

Mutant budget ablation. For each method, we randomly sample subsets containing 10%, 20%, ..., 100% of the full mutant set and compute Comp@1. We observe a clear saturation pattern once $\geq$80% of mutants are included. For GPT-5, mean Comp@1 decreases only slightly from 0.2616 (80%) to 0.2585 (90%) and 0.2551 (100%), with absolute differences of $\approx$0.003. Claude-4.5 shows the same pattern (0.2120 $\rightarrow$ 0.2091 $\rightarrow$ 0.2068) (see Table 8).

> Comp@1 meaningfully reflects mutation design, remaining stable under substantial perturbations of operator sets and mutation budget. This indicates that the benchmark is neither dominated by a specific mutation family nor overly sensitive to particular operator choices.

6 Conclusion

We presented PostCondBench, a multilingual benchmark for evaluating method-level postconditions in real-world software. PostCondBench provides high-quality ground-truth postconditions and a runnable platform that evaluates correctness with tests and completeness via defect discrimination. Evaluating five SOTA LLMs across three generation tasks reveals a persistent gap between correctness and completeness. By enabling reliable completeness evaluation, PostCondBench aims to support future work on generating more behaviorally complete postconditions.

Limitations

We exclude methods (or specific mutants) that require unsupported specification constructs or remain unkillable under the available tests. It is possible that this exclusion introduces bias in the PostCondBench evaluation of the affected behaviors, such as iterator or concurrency behaviors. However, filtering affects a minority of the sampled methods. Most of these exclusions arise from representational limits of the underlying specification frameworks (Appendix H). The exceptional behaviors are also not in PostCondBench's scope since we focus on normal postconditions (in contrast to exceptional postconditions). Besides, we restrict on methods with high code coverage due to the execution-based nature of our metrics.

Ethical Considerations

Data sources and licensing. Our benchmark is constructed from publicly available open-source repositories. We only include projects with permissive licenses and preserve attribution to the original authors and repositories. Any release of the benchmark and derived artifacts should comply with the licenses of the included projects.

Privacy and sensitive information. The benchmark is derived from public code and related artifacts (e.g., tests and comments). The dataset underwent end-to-end inspection during curation and validation. Across these inspections, we did not observe privacy-related or sensitive information in the data intended for release.

Human involvement. Some parts of dataset creation and validation involve domain-expert participation (e.g., refining or verifying postconditions and resolving ambiguities). The experts are invited; no public recruitment. Domain experts provide assistance on a voluntary basis. While administrative contact information may be required to coordinate participation, we did not collect such information as part of the dataset itself, did not use it beyond study administration, and it is not included in any released artifacts.

Acknowledgments

This paper uses AI writing assistance purely with the language of the paper, which covers models used for paraphrasing or polishing the author's original content, rather than for suggesting new content.

Data Availability

To follow the Open Science Policy and support reproducibility, we have released code about our implementations and evaluations. All source code and data used in our work can be found at `https://github.com/zhang-ge-hao/postcond-bench`.

References

Meta AI. 2025. Gpt-4o mini: advancing cost-efficient intelligence. Meta.

Anthropic. 2025. Introducing claude sonnet 4.5. Anthropic.

Aviad Baron, Ilai Granot, Ron Yosef, and Dror Feitelson. 2024. Understanding logical expressions with negations: Its complicated. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 303–312.

Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 242–253, Amsterdam, Netherlands. ACM.

Walter E. Brown. 2024. Implication for c++. WG21 Paper P2971R2 P2971R2, ISO/IEC JTC1/SC22/WG21.

Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. 2016. Supporting oracle construction via static analysis. *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–189.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 34 others. 2021. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 449–452.

Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*, pages 383–396. Springer.

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering*, 1(FSE):1889–1912.

Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an annotation assistant for esc/java. In *FME*.

Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 213–224. ACM.

Johannes Henkel, Christoph Reichenbach, and Amer Diwan. 2008. Developing and debugging algebraic specifications for java classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(3):1–37.

Anders Hovmöller. 2016. Mutmut: a python mutation testing system. https://kodare.net/2016/12/01/mutmut-a-python-mutation-testing-system.html. Blog post on En kodare.

Ruida Hu, Chao Peng, Junjielong Xu, Cuiyun Gao, and 1 others. 2025. Repo2run: Automated building executable environment for code repository at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. In *Annual Meeting of the Association of Computational Linguistics 2024*, pages 4912–4944. Association for Computational Linguistics (ACL).

jacoco. 2025. JaCoCo: Java code coverage library. https://github.com/jacoco/jacoco. GitHub repository.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

jhy. 2025. jsoup: the java html parser, built for html editing, cleaning, scraping, and xss safety. https://github.com/jhy/jsoup/. GitHub repository, accessed 2026-01-06.

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 *12th International Conference on Learning Representations, ICLR 2024*.

Joowani. 2022. binarytree: A python library for studying binary trees. https://github.com/joowani/binarytree. GitHub repository, accessed 2026-01-06.

René Just, Darioush Jalali, and Michael D Ernst. 2014a. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014b. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 654–665.

Keon. 2025. algorithms: Minimal examples of data structures and algorithms in python and other languages. https://github.com/keon/algorithms. GitHub repository, accessed 2026-01-06.

Jens U Kreber and Christopher Hahn. 2021. Generating symbolic reasoning problems with transformer gans. *arXiv preprint arXiv:2110.10054*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.

Axel van Lamsweerde. 2000. Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159.

Afshan Latif, Farooque Azam, Muhammad Waseem Anwar, and Amina Zafar. 2023. Comparison of leading language parsers–antlr, javacc, sablecc, tree-sitter, yacc, bison. In *2023 13th International Conference on Software Technology and Engineering (ICSTE)*, pages 7–13. IEEE.

Gary T Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M Zimmerman, and 1 others. 2008. Jml reference manual.

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. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. Specgen: Automated generation of formal program specifications via large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 16–28. IEEE.

Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

Gemma Team Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, L. Sifre, Morgane Riviere, Mihir Kale, J Christopher Love, Pouya Dehghani Tafti, L'eonard Hussenot, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Am'elie H'eliou, Andrea Tacchetti, and 88 others. 2024. Gemma: Open models based on gemini research and technology. *ArXiv*, abs/2403.08295.

Frederic P Miller, Agnes F Vandome, and John McBrewster. 2010. *Apache Maven*. Alpha Press.

Hoan Anh Nguyen, Robert Dyer, Tien N Nguyen, and Hridesh Rajan. 2014. Mining preconditions of apis in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 166–177. ACM.

Jeremy W Nimmer and Michael D Ernst. 2002. Automatic generation of program specifications. *ACM SIGSOFT Software Engineering Notes*, 27(4):229–239.

OpenAI. 2023. Code interpreter. https://platform.openai.com/docs/guides/tools-code-interpreter. OpenAI API Documentation. Accessed: 2026-01-06.

OpenAI. 2025. Gpt-5 is here. OpenAI.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with SWE-gym. In *Forty-second International Conference on Machine Learning*.

Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, pages 815–825. IEEE Press.

Parquery. 2025. icontract. GitHub repository.

Parquery AG. 2019. icontract documentation: Async. https://icontract.readthedocs.io/en/latest/async.html. Accessed: 2026-01-05.

pytest-dev. 2025. pytest-cov: Coverage plugin for pytest. https://github.com/pytest-dev/pytest-cov. GitHub repository.

python-poetry. 2026. Poetry: Python packaging and dependency management made easy. https://github.com/python-poetry/poetry. GitHub repository.

Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *ACM SIGPLAN Notices*, pages 123–134. ACM.

Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis, II. 1977. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581.

Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. 2007. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34:651–666.

Colin Frank Snook. 2001. *Exploring the barriers to formal specification*. Ph.D. thesis, University of Southampton.

Bo Wang, Mingda Chen, Youfang Lin, Mike Papadakis, and Jie M Zhang. 2024. An exploratory study on using large language models for mutation testing. *CoRR*.

Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. Docter: documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–188.

Danning Xie, Byoungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S Lee. 2025. How effective are large language models in generating software specifications? In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE.

Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. 2024. Llm4fin: Fully automating llm-powered test case generation for fintech software acceptance testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1643–1655.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

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*, pages 1–12.

Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2s: translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 25–37.

Shiyu Zhang, Juan Zhai, Bu Lei, Wang Linzhang, and Xuandong Li. 2020. Automated geration of ltl specifications for smart home iot using natural language. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.

A Discussion

Insufficiency for Real Bugs. We use mutants rather than real bugs made in development to assess the completeness. The reason is that mutation lets us generate many defective variants for the same method. For real bugs (Just et al., 2014a; Jimenez et al., 2024), each fixed version is typically paired with a very limited number (usually only one) of corresponding buggy versions, which makes it insufficient to assess how completely a specification constrains the method's behavior.

Moreover, prior work shows that mutants can effectively couple with real faults (Just et al., 2014b). Therefore, although PostCondBench is designed to evaluate the quality of formal postconditions, the reported mutant discrimination can be a prerequisite capability for downstream uses such as verification and debugging.

B Future Work

Live Benchmark against Data Leakage. "Live" benchmarks are continuously updated with newer data, reducing the risk of data leakage (Jain et al., 2024). They are often designed to support efficient updates. For PostCondBench, most stages of the construction pipeline are automated, and the benchmark can be re-run on newly released repositories with limited manual effort; the expert effort for maintaining a refreshed version is expected to be less than 50 hours per year. It is practical to maintain a periodically refreshed benchmark.

C Comparison with Existing Work

Table 4 positions PostCondBench relative to prior work on postcondition generation. While existing studies primarily propose generation approaches and evaluate them under task-specific datasets or execution settings, they do not release a reusable benchmark with curated ground truths or an explicit evaluation methodology for completeness. PostCondBench fills this gap by providing a benchmark that jointly supports multilingual, repository-level programs, advanced postcondition syntax, and automatic evaluation of both correctness and completeness.

D Experimental Setup

D.1 Models

We chose five state-of-the-art large language models (LLMs) for experiments: GPT-5 (OpenAI, 2025), Claude-4.5 (Sonnet) (Anthropic, 2025), LLaMA-4 (Maverick) (AI, 2025), Qwen3-32B (Yang et al., 2025), Gemma-3-27B (Mesnard et al., 2024). For GPT-5, we use the OpenAI batch API. For Claude-4.5 and LLaMA-4, we use the batch inference pipeline in Amazon Bedrock. For Qwen and Gemma models, we locally deploy Vllm (Kwon et al., 2023) services for experiments. We apply the reasoning mode for GPT-5 and Claude 4.5, setting `reasoning.effort = "medium"` for GPT-5 and `thinking.budget_tokens = 2048` for Claude 4.5. Besides the reasoning effort configurations, we follow each provider's default configurations and use consistent prompts across models for fair comparison.

```
# Code Context
{code_context}
# Target Method
{target_method}
# Guideline
Above is the {Python|Java} code context and
target method. You should generate {icontract|JML}
format postconditions ...
```

Figure 7: Prompt template. The placeholders are shown in green.

D.2 Prompt Construction

PostCondBench supports runnable repositories and automatic evaluation; we therefore formulate postcondition generation as prompting an LLM with a target method plus its surrounding context. All settings share the prompt template in Figure 7, which contains two main fields: `{code_context}` (in-file context) and `{target_method}` (the method to validate).

We instantiate three input settings:

- **F2P.** We provide the method signature, the NL method-level comment, and the full implementation in `{target_method}` (cf. Figure 1). We fill `{code_context}` with the method-located file content to provide in-file context.
- **C2P.** We provide the method signature and implementation in `{target_method}`, removing any code comments inside the implementation. We also remove all code comments from `{code_context}`.
- **N2P.** We provide the method signature and its NL method-level comment in `{target_method}`. In `{code_context}`, we

Table 4: Comparison of evaluation features in Python/Java postcondition generation.

Work	C2S (Zhai et al., 2020)	Endres et al. (2024) (HumanEval+)	Endres et al. (2024) (Defects4J)	SpecGen (Ma et al., 2025)	Xie et al. (2025)	Ours
Multi-Language	✗	✗	✗	✗	✗	✓
Repository-level	✓	✗	✓	✗	✓	✓
Ground Truths Included	✗	✗	✗	✓	✓	✓
Advanced Spec. Syntax	✓	✗	✗	✓	✗	✓
Auto. Completeness Eval.	✗	✓	✗	✗	✗	✓

Endres et al. (2024) report postcondition generation results under two benchmarks: `HumanEval+` (Liu et al., 2023) and `Defects4J` (Just et al., 2014a).

keep the method-located file but remove method bodies, so the model can leverage declarations and surrounding structure without seeing implementations.

These settings isolate the effects of code, comments, and their combination on correctness and bug-completeness.

D.3 Evaluation Metrics

We evaluate generated postconditions along two dimensions: correctness and completeness. Completeness is evaluated only for test-correct postcondition sets; candidates that fail correctness are treated as incomplete.

Correctness. A generated *postcondition set* is considered *correct* if, when instrumented into the original method, all postconditions in the set pass all provided test cases. Test-based correctness reflects whether the generated specification is consistent with observed program behaviors.

Completeness. A generated *postcondition set* is considered *complete* if it not only passes all test cases, but also rejects all automatically generated mutants of the target method. Since mutants introduce buggy behaviors that should be ruled out by a correct specification, completeness measures whether a postcondition set sufficiently constrains the method's behavior. Completeness is evaluated only for test-correct postcondition sets; candidates that fail correctness are treated as incomplete.

• `@k` aggregation. Let k denote the number of independently generated postcondition sets for a given method. **Corr@k** indicates whether at least one of the k generated postcondition sets is test-correct. **Comp@k** indicates whether at least one of the k generated postcondition sets is both test-correct and complete.

• Correctness–completeness gap. To quantify the discrepancy between correctness and completeness, we define two complementary measures. The *absolute gap* is defined as

$$\Delta@k = \text{Corr}@k - \text{Comp}@k, \tag{1}$$

which captures the extent to which test-based correctness overestimates completeness. We also report the *conditional completeness rate*,

$$\rho@k = \frac{\text{Comp}@k}{\text{Corr}@k}, \tag{2}$$

measuring the fraction of test-correct postconditions that are also complete. Unless otherwise specified, gap analyses in the main paper focus on `@1`, while multi-sample results are reported in the appendix.

E Correctness–Completeness Gap: Case study

Figure 8 gives concrete examples. For `distance` (left), the correct-but-incomplete set mainly checks boundary/special cases (e.g., `x==y`) and the return type, yet fails to kill a mutant that initializes `sum` to one when `x != y`; the complete set additionally checks the expected value (line 003), which rules out the mutant. The Java example (right) shows a similar pattern: the correct-but-incomplete set includes a redundant constraint, while the complete set adds a stricter condition beyond the special case.

F Ablation on Mutation Schemes

F.1 False Discovery Rate Across Mutation Schemes

We ablate our two mutation schemes—rule-based and LLM-based—to assess their complementarity. Table 6 reports the false discovery rate (FDR) for each scheme. Concretely, for a given scheme S, we

Table 5: Postcondition generation evaluation results from Large Language Models. The **bold** font emphasizes the highest-performing LLM for each metric within each language.

	C2P (Code-only)				N2P (NL-only)				F2P (NL + Code)			
Metric	**Corr @1**	**Comp @1**	**△@1**	**Comp/ Corr**	**Corr @1**	**Comp @1**	**△@1**	**Comp/ Corr**	**Corr @1**	**Comp @1**	**△@1**	**Comp/ Corr**
GPT-5	0.493	**0.290**	0.203	0.588	0.434	**0.177**	0.257	0.408	0.520	**0.298**	0.222	0.573
Claude-4.5	**0.645**	0.234	0.411	0.362	**0.593**	0.150	0.443	0.253	**0.650**	0.237	0.413	0.364
LLaMA-4	0.224	0.105	0.119	0.469	0.180	0.062	0.118	0.344	0.239	0.116	0.122	0.487
Qwen3-32B	0.141	0.062	0.079	0.441	0.092	0.043	0.049	0.469	0.120	0.060	0.061	0.494
Gemma-3-27B	0.117	0.054	0.063	0.459	0.097	0.026	0.071	0.266	0.117	0.053	0.064	0.453

```python
def distance(x,y):
    """...calculates the (eulidean) distance between vector x and y...
    result = ()
-   sum = 0
+   sum = 1
    for i in range(len(x)):
        result += (x[i] -y[i],)
    for component in result:
        sum += component**2
    return math.sqrt(sum)
```

```python
@icontract.ensure(lambda result: isinstance(result, float))
@icontract.ensure(lambda result: result >= 0.0)
@icontract.ensure(lambda x, y, result:
    math.isclose(result, math.sqrt(
        sum((x[i] - y[i]) ** 2 for i in range(len(x)))),
        rel_tol=1e-12, abs_tol=0.0))
@icontract.ensure(lambda x, y, result: (x == y) == (result == 0.0))
```

```python
@icontract.ensure(lambda result: result >= 0)
@icontract.ensure(lambda result, x, y: (result == 0) == (x == y))
@icontract.ensure(lambda result: isinstance(result, float))
```

```java
/** Returns true if any of the Element's child nodes should indent...
static boolean hasChildBlocks(Element el) {
    Element child = el.firstElementChild();
-   for (int i = 0; i < maxScan && child != null; i++) {
+   for (int i = 0; i >= maxScan && child != null; i++) {
        if (child.isBlock() || !child.tag.isKnownTag()) return true;
        child = child.nextElementSibling();
    }
    return false;
}
```

```java
//@ ensures \result <==> el.children().stream().limit(maxScan).anyMatch(
    child -> child.isBlock() || !child.tag.isKnownTag());
//@ ensures \result ==> el.firstElementChild() != null;
```

```java
//@ ensures el.firstElementChild() == null ==> \result == false;
//@ ensures \result == true ==> el.firstElementChild() != null;
```

Figure 8: Left: A Python instance from `keon/algorithms` (Keon, 2025). Right: A Java instance from `jhy/jsoup` (jhy, 2025). Top: Target methods. For each method, the "+" and "-" leading lines represent the differences from the original code to an example mutant. Middle: Complete postcondition sets (both are GPT-5 generated from the F2P task). Bottom: Correct but incomplete postcondition sets (Claude-4.5 generated from the F2P task). The green box enclosed the key postcondition that make the set more complete.

first evaluate completeness using only mutants generated by S and collect postcondition sets that are classified as *complete* under this restricted mutant set. We then re-evaluate these same postcondition sets on mutants from the other scheme $\bar{S}$ and compute the fraction that fail to kill all $\bar{S}$-mutants; this fraction is the FDR. A higher FDR indicates that a single scheme is insufficient in isolation, and that the other scheme contributes additional mutants that expose incompleteness.

As shown in Table 6, both schemes exhibit nontrivial FDRs, suggesting meaningful complementarity. For example, the Avg row shows an FDR of 0.206 for LLM-based mutation on Python–F2P, and an FDR of 0.203 for rule-based mutation on Java–N2P. In other words, if we rely on only one mutation scheme, a noticeable fraction of postcondition sets deemed "complete" would still be incomplete and would be revealed by mutants from the other scheme.

Overall, rule- and LLM-based mutation complement each other for completeness evaluation.

F.2 Complementarity Under Scheme Exclusion

We further recompute Comp@1 after excluding either all LLM-based mutants or all operator-based mutants. Table 7 shows that scores increase after removing either mutation family, indicating that each family exposes defects the other does not cover well. For example, when LLM-based mutants are excluded, Claude-4.5 increases from 0.2068 to 0.2944; when operator-based mutants are excluded, LLaMA-4 increases from 0.0944 to 0.1281. This provides a direct complement to the FDR analysis above: high completeness requires handling both mutation families well.

F.3 Mutant-Budget Stability

We also vary the overall mutant budget by randomly sampling a fixed fraction of the full mutant set for each method. As shown in Table 8, completeness approaches saturation as the budget increases. Between 80%, 90%, and 100% of mutants, the mean Comp@1 changes only slightly for all five models, and the standard deviations are small once the budget reaches 80% or above. This sug-

Table 6: Mutation FDR (False Discovery Rate) (↓) of different mutation schemes. I.e., if we only use one scheme to identify the complete postconditions, the ratio of the discovered "complete" postcondition that cannot kill all mutants from another scheme.

	Python						Java					
	F2P		C2P		N2P		F2P		C2P		N2P	
	#Rule	#LLM	#Rule	#LLM	#Rule	#LLM	#Rule	#LLM	#Rule	#LLM	#Rule	#LLM
GPT-5	0.155	**0.154**	**0.127**	0.188	**0.132**	0.169	0.146	**0.049**	0.111	**0.051**	0.183	**0.036**
Claude-4.5	**0.164**	0.201	**0.153**	0.205	0.209	**0.152**	0.220	**0.064**	0.210	**0.049**	0.250	**0.061**
LLaMA-4	0.186	**0.174**	**0.103**	0.158	**0.143**	0.167	0.148	**0.050**	0.187	**0.126**	0.203	**0.150**
Qwen3-32B	0.400	**0.000**	**0.000**	**0.000**	-	-	0.181	**0.144**	0.227	**0.077**	0.108	**0.042**
Gemma-3-27B	0.667	**0.500**	0.833	**0.000**	-	-	0.252	**0.097**	0.215	**0.167**	0.270	**0.062**
Avg	0.314	**0.206**	0.243	**0.110**	**0.161**	0.162	0.189	**0.081**	0.190	**0.094**	0.203	**0.070**

Table 7: Comp@1 after excluding one mutation family from completeness evaluation. Higher values after exclusion indicate that the removed mutation family exposes defects not covered well by the retained family.

Condition	GPT-5	Claude-4.5	LLaMA-4	Qwen3-32B	Gemma-3-27B
w/ all mutants	0.2551	0.2068	0.0944	0.0551	0.0441
wo/ LLM-based	0.3094	0.2944	0.1046	0.0606	0.0481
wo/ operator-based	0.2773	0.2289	0.1281	0.0690	0.0622

gests that completeness is not dominated by a small number of unusually influential mutants.

F.4 Operator-Level Ablation

To test whether the metric is overly sensitive to a particular rule-based operator, we remove each operator in turn and average the resulting Comp@1 across models. Table 9 shows that most removals have only minor effects. In Python, dropping any single operator changes Comp@1 by at most 0.0038; in Java, the largest change is 0.0063 after removing `null_returns`. This supports that completeness is not driven by one critical operator.

F.5 Mutation-Strength Ablation

Finally, we test robustness under large random perturbations of the mutant set. We either remove 5 of the 11 rule-based operators at random or remove 50% of LLM-based mutants, and then recompute Comp@1 over 50 trials. As shown in Table 8, the mean changes are small and the standard deviations remain low for all models. These results indicate that the completeness metric is stable under substantial changes in both mutant composition and mutant strength.

G Discussion on Bias from LLM-Generated Ground Truths

As described in Section 3.2.3, we introduce an LLM (GPT-5-mini) for ground truth generation. 66% of the final benchmark ground-truths are LLM-generated. In this section, we discuss whether the LLM-generated ground truths will introduce bias in the benchmark evaluation.

Importantly, PostcondBench does not use ground-truth postconditions during evaluation. Corr@1 and Comp@1 are computed solely via test execution and mutant discrimination, independent of any reference specification. Therefore, adopting LLM-generated outputs as ground truth does not affect how models are scored. During dataset construction, LLM was used only to reduce manual effort by proposing candidate ground-truths. Target methods are filtered solely when human experts judge that writing a complete postcondition set is infeasible under the current framework (Appendix H). LLM assists in identifying solvable cases but does not influence evaluation criteria or introduce model-specific bias.

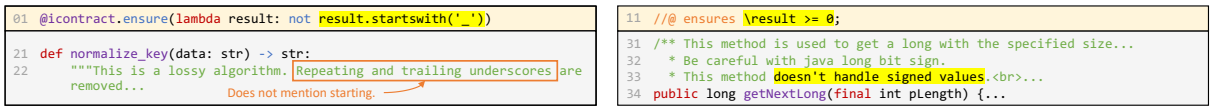
H Expert Refinement Statistics/Analysis

As Section 3.2.5 mentioned, during domain expert refinement, some methods/mutants are found inappropriate for inclusion in the benchmark. We analyze and attribute them in this section.

Overall, filtering affects a minority of the sampled methods rather than most of them. Among 297 sampled Python methods, 87 are filtered (29.3%); among 234 sampled Java methods, 24 are filtered (10.3%). Most filtered methods arise from limitations of the current specification frameworks

Table 8: Comp@1 under two forms of large mutant-set perturbation. Except for the baseline column, each cell reports the mean and the standard deviation over 50 trials.

Model	Baseline	Ex. 5 Operators	Ex. 50% LLM Mutants	Mutant-Budget Fractions 0.1	0.2	0.8	0.9	1.0
GPT-5	0.255	0.265 (0.004)	0.265 (0.002)	0.331 (0.007)	0.307 (0.006)	0.262 (0.003)	0.259 (0.003)	0.255 (0.000)
Claude-4.5	0.207	0.215 (0.003)	0.216 (0.002)	0.298 (0.010)	0.261 (0.008)	0.212 (0.003)	0.209 (0.002)	0.207 (0.000)
LLaMA-4	0.094	0.098 (0.002)	0.098 (0.001)	0.129 (0.006)	0.116 (0.004)	0.097 (0.002)	0.096 (0.002)	0.094 (0.000)
Qwen3-32B	0.055	0.056 (0.001)	0.057 (0.001)	0.074 (0.003)	0.068 (0.003)	0.056 (0.001)	0.056 (0.001)	0.055 (0.000)
Gemma-3-27B	0.044	0.046 (0.001)	0.045 (0.001)	0.059 (0.004)	0.053 (0.003)	0.045 (0.001)	0.045 (0.001)	0.044 (0.000)



(a) icontract example (b) JML example

Figure 9: Two examples of incorrect postconditions generated due to LLMs' hallucination on semantics. The left side is the generated postconditions from the N2P task and the target method from `kellyjonbrazil/jc`. The documentation states that repeating and trailing underscores are removed, but makes no claim about leading underscores. The generated postcondition nevertheless enforces the absence of leading underscores, introducing an unsupported constraint. The right side target method is from `devnied/Bit-lib4j`. The postcondition is generated from the F2P task. The method reads a `long` value of a given bit length. Although the comment notes that signed values are "not handled" this implies no special treatment of the sign bit rather than non-negativity. The generated postcondition incorrectly enforces `\result >= 0`, thereby ruling out valid behaviors. Both postconditions are generated by Claude-4.5.

rather than the complexity of the methods: 45 of the 87 filtered Python methods and 13 of the 24 filtered Java methods fall into framework-limited categories, such as iterator-related or concurrency-related. For the remaining filtered cases, domain experts determined that writing a complete postcondition set would still be non-trivial even for humans under the current evaluation setup.

H.1 Python Language

297 Python methods sampled by farthest-first traversal. We ensure that the sampled methods have at least five mutants.

Among them, 135 target methods have complete postconditions generated by GPT-5-mini. These methods have 2670 mutants, all of which are killed by auto-generated postconditions.

We manually checked the remaining 162 Python methods and attempted to write complete postconditions for them.

Of the 162 Python methods, 75 have manually written postconditions that are identified as complete. The remaining 87 methods are identified as inappropriate for inclusion. The reasons for which they are filtered are analyzed in the following section.

These 75 methods have 2957 mutants. Among them, 43 mutants were identified as inappropriate for inclusion and were filtered. 10 methods filtered the mutants. The reasons for their filtration are analyzed in the following section.

Overall, we have 135 + 75 = 210 Python methods in our benchmark.

H.1.1 Method-level Filtering Analysis

After manual checking, 87 methods are filtered. Under the current formal specification framework, writing complete postcondition expressions for them is non-trivial even for human engineers. We analyze the reason for filtering.

Iterator related (18). By limitation of the API, Iterator/Generator objects can only be iterated over once. Validating them by postcondition expressions has side effects. Therefore, methods have key variables that in Iterator/Generator types, and the key feature relies on accessing elements from the

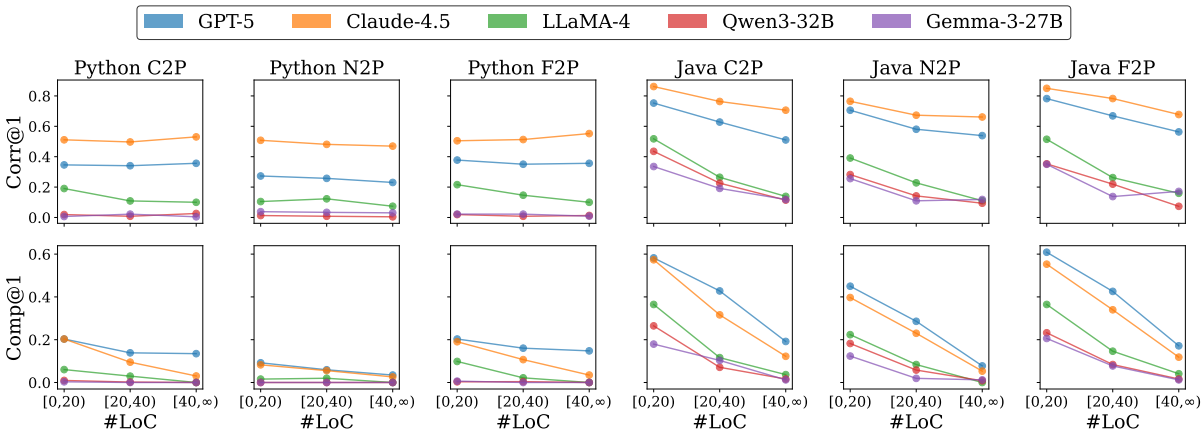


Figure 10: Corr@1 (top) and Comp@1 (bottom) across target methods grouped by lines of code (LoC).

Table 9: Operator-level ablation for rule-based mutation, averaged across models.

Python	
Excluded operator	**Comp@1**
No exclusion (baseline)	0.0516
asymmetric_str_method_swap	0.0516
augassign_to_assign	0.0516
numeric_increments	0.0516
symmetric_str_method_swap	0.0516
unary_op_removal	0.0516
boolean_constant_flip	0.0519
arg_removal	0.0521
string_perturbation	0.0523
keyword_rewrite	0.0525
operator_replacement	0.0531
assignment_nullification	0.0554
Java	
Excluded operator	**Comp@1**
No exclusion (baseline)	0.2106
conditionals_boundary	0.2106
false_returns	0.2106
invert_negatives	0.2106
true_returns	0.2107
increments	0.2108
math	0.2109
primitive_returns	0.2110
negate_conditionals	0.2110
empty_returns	0.2111
void_method_call	0.2152
null_returns	0.2169

iterator objects that have been filtered. Figure 13 shows the `prefix_lines` method from project miyuchina/mistletoe.

Validating these methods requires access to the actual string elements. However, both the input parameters and return value have Iterator objects. Accessing elements in `@icontract.snapshot` or `@icontract.ensure` decorators leads to side-effects. The current icontract does not provide an effective path to deal with these cases.

(a)
```
@icontract.ensure(lambda result:
    all(result[i] <= result[i + 1]
        for i in range(len(result) - 1)))
```

(b)
```
@icontract.ensure(lambda result:
    all(a <= b for a, b in zip(result, result[1:])))
```

(c)
```
@icontract.ensure(lambda result:
    result == sorted(result))
```

(d)
```
@icontract.ensure(lambda result:
    all(a <= b for a, b in itertools.pairwise(result)))
```

(e)
```
@icontract.ensure(lambda result:
    list(itertools.accumulate(result, max)) == result)
```

```
def sort(arr: List[int]) -> List[int]:
    ...
```

Figure 11: An example for sorting algorithm validation. (a) to (e) are postconditions written in icontract, which are semantically equivalent but syntactically different.

Timeout (17). The postcondition running also requires time. For these cases, complete validations lead to timeouts. Figure 14 shows the `load_inventory` method from the project hynek/doc2dash. Note that the return value is directly related to the content in the `objects.inv` file. Therefore, a complete validation requires accessing the file content, leading to timeouts.

Concurrency related (13). Some methods' key feature is related to obtaining results through concurrency mechanisms (starting sub-processes, sub-threads, etc.), which are identified as inappropriate for inclusion. Figure 15 shows method `execute_code` in vndee/llm-sandbox. In this case, the results depend on running the inputted code inside the session, which launches a sandbox using Docker, Kubernetes, Podman, etc. This leads to a non-trivial validation, since it would require analyzing the code's semantics or launching a new sandbox within the postcondition.

Table 10: Postcondition generation performance on non-standalone (**Dep.**) and standalone (**Solo.**) methods. **Dep.** requires repository-specific dependencies, while **Solo.** is runnable with only built-ins. Bold indicates the higher score within each comparison.

Language		Python				Java			
Model / Setting		Corr@1		Comp@1		Corr@1		Comp@1	
		Dep.	Solo.	Dep.	Solo.	Dep.	Solo.	Dep.	Solo.
GPT-5	C2P	0.315	**0.419**	0.135	**0.210**	0.614	**0.768**	0.385	**0.600**
	N2P	0.211	**0.365**	0.038	**0.126**	0.591	**0.708**	0.245	**0.503**
	F2P	0.335	**0.419**	0.139	**0.245**	0.662	**0.768**	0.393	**0.578**
Claude-4.5	C2P	0.470	**0.600**	0.070	**0.216**	0.782	**0.784**	0.332	**0.459**
	N2P	0.441	**0.597**	0.018	**0.152**	0.699	**0.703**	0.216	**0.368**
	F2P	0.484	**0.603**	0.072	**0.223**	0.775	**0.805**	0.335	**0.459**
LLaMA-4	C2P	0.120	**0.158**	0.024	**0.052**	0.301	**0.395**	0.160	**0.265**
	N2P	0.092	**0.142**	0.004	**0.039**	0.232	**0.351**	0.083	**0.232**
	F2P	0.143	**0.190**	0.023	**0.081**	0.302	**0.405**	0.173	**0.281**
Qwen3-32B	C2P	0.012	**0.023**	0.001	**0.010**	0.250	**0.351**	0.116	**0.146**
	N2P	**0.009**	0.006	0.000	0.000	0.149	**0.303**	0.055	**0.232**
	F2P	**0.015**	0.006	0.003	**0.003**	0.215	**0.292**	0.108	**0.157**
Gemma-3-27B	C2P	**0.016**	0.006	**0.001**	0.000	0.209	**0.276**	0.090	**0.184**
	N2P	**0.035**	0.032	0.000	0.000	**0.170**	0.108	0.046	**0.076**
	F2P	0.019	**0.019**	**0.003**	0.000	0.197	**0.297**	0.083	**0.200**
Avg		0.181	**0.239**	0.035	**0.090**	0.410	**0.488**	0.188	**0.316**

Table 11: Effect of adding grammar guidance on the F2P task for two strong models.

Model	w/ guidance		w/o guidance	
	Corr@1	Comp@1	Corr@1	Comp@1
GPT-5	0.767	0.400	0.520	0.298
Claude-4.5	0.814	0.283	0.629	0.207

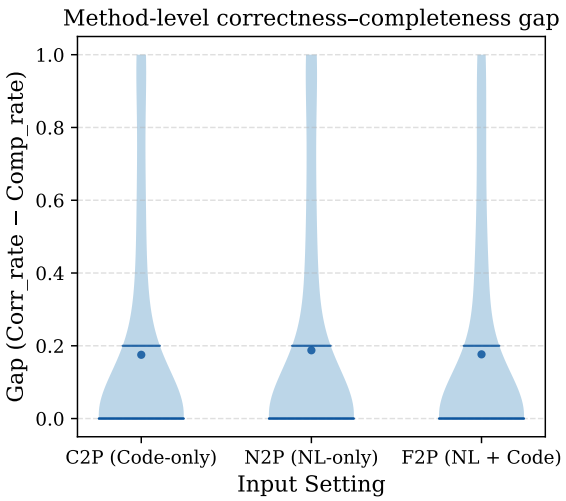


Figure 12: Distribution of method-level correctness–completeness gaps under different input settings, without methods with Corr@k = 0 excluded. The gaps are relatively smaller than Figure 3 due to generally low Corr_rates (e.g., Corr@1 = 0.224 in Table 5 for LLaMA-4 under C2P setting)

Moreover, this category also includes several coroutine-using methods. Figure 16 shows `get_task` in a2aproject/a2a-python. Since the return value directly relies on the results from `self.task_store.get`. If `get` is a sync method, we can call it and validate on its return value inside the postcondition. However, since `get` is an async method and Python does not support async lambda. With icontract, defining async conditions (and snapshots) is indeed tedious (Parquery AG, 2019).

Language entities related (8). Some methods return a function object, edit a class object, etc. Validation on these language entities is generally nontrivial. Figure 17 shows `while_until_true` in pypyr/pypyr, which returns a function object. Moreover, Figure 18 shows `make_sync` in run-llama/llama_deploy, which edits and returns a class object.

Ordinary (9). Some methods' key features (logging, printing to stdout, sending email, etc.) are not related to changes in the accessible code context status and cannot be caught by the specification framework. Figure 19 shows `rich_format_error` in fastapi/typer. This method's key feature is to print error logs under different conditions. Buggy

mutants that disturb log printing can be caught by the test suite since the test cases use mock util to validate logging outputs. However, this method is not feasible to validate with only the formal specification framework.

Specified error process (5). For Python, we need the program output `icontract.errors.ViolationError` in stderr to know that the postcondition can kill the mutant. However, in some cases, errors do not output an error log. Figure 20 shows the test case format for `_make_command_help` in langchain-ai/langchain-mcp-adapters. These test cases would simply fail due to assertion error (`exit_code` is not 0), and the error message will not specify whether the error is from icontract violation error.

Others (17). Not included above, and non-trivial for human engineers to write complete postconditions. These methods are generally lengthy or exist within complex code contexts, increasing the costs of understanding. The human engineer failed to write complete postconditions for these. Figure 21 shows method `merge` in pypyr/pypyr. This 81-line method is complicated, and recursion is used. The human engineer failed to write complete postconditions for it.

H.1.2 Mutant-level Filtering Analysis

Of the 210 Python methods in the benchmark, 10 have mutants filtered after manual checking. 43 mutants are filtered. Note that we ensure the main feature of the methods can still be validated after some of the mutants are filtered for these methods.

Ordinary (29 mutants; five methods involved). The feature that mutant disturbs (logging, process sleeping length, etc.) is not related to accessible code context status change and cannot be caught by the specification framework. Figure 22 shows `getprops` in `google/mobly`.

The main feature of this method is validatable. However, a few mutants disturb the line of `time.sleep`, e.g., replacing it by `time.sleep(0)`. We filter out these mutants for a similar reason to the way we filter "ordinary" category methods.

Specified error process (nine mutants; two methods involved). These mutants triggered errors that do not output error types. Figure 23 shows the triggered test case format for two mutants on method `_parse_ignore_file` in project `coderamp-labs/gitingest`.

These test cases would simply fail due to assertion error (`exit_code` is not 0), and the error message will not specify whether the error is from `icontract` violation error. For this case, since only a few mutants (seven of 34) are affected by this issue, we keep the method but filter out affected mutants.

Equivalent but not the same results (two mutants; one method involved). Sometimes, the algorithm has multiple possible results, the test cases are over-validated, and fail on mutants that are actually equivalent to the correct code. We filter out these mutants since we only keep buggy mutants. Figure 24 shows the original method `k_closest(points, k, origin)` from `keon/algorithms`, the equivalent mutant, and the over-constrained test case.

While the mutant's return value `[(-2, -2), (-1, 0), (1, 1), (1, 0)]` is equivalent to the expected result, the test case failed.

Therefore, we filter out mutants under this category since we only keep buggy mutants.

Timeout (three mutants; two methods involved). The postcondition running also requires time. Post-conditions that are complete enough to kill these mutants lead to a timeout. Figure 25 shows the `partition_by_spaces` method in the `frostming/marko` project together with the downstream call path that amplifies the failure into a timeout.

However, one mutant replaces the `break` in `partition_by_spaces` with `return`, directly returning nothing. That will cause an error in `break_paragraph`. Further, it leads to more loop running in the `parse` method, and finally causes a timeout. Since only one mutant triggers this timeout issue, and the main feature of `partition_by_spaces` is still validatable, we keep this target method and filter out this mutant.

H.2 Java Language

We sampled 234 Java methods by farthest-first traversal. We ensure that the sampled methods have at least five mutants.

Among them, 141 target methods have complete postconditions generated by GPT-5-mini. These methods have 2,530 mutants, all of which are killed by auto-generated postconditions.

We manually checked the remaining 93 Java methods, trying to write complete postconditions for them.

From the 93 Java methods, 69 methods have manually-written postconditions that are identified as complete. The remaining 24 methods are identified as inappropriate for inclusion. The reasons that they are filtered are analyzed in the following section.

These 69 methods have 2,075 mutants. Among them, two mutants were identified as inappropriate to be included and filtered. One method has mutants being filtered. The reasons that they are filtered are analyzed in the following section.

Overall, we have $141 + 69 = 210$ Java methods in our benchmark.

H.2.1 Method-level Filtering Analysis

After manual checking, 24 methods are filtered. Under the current formal specification framework, writing complete postcondition expressions for them is non-trivial even for human engineers. We analyze the reason for filtering.

Timeout (5). The postcondition running also requires time. For these cases, complete validations lead to timeouts. Figure 26 shows the `compare` method from `red6/pdfcompare`.

This method applies a PDF loading and comparison. Therefore, validating this method generally requires accessing two PDF files, leading to timeouts.

Iterator related (4). By limitation of the API, Iterator/Generator objects can only be iterated over once. Validating them by postcondition expressions has side effects. Therefore, methods have key variables that in Iterator/Generator types, and the key feature relies on accessing elements from the iterator objects that have been filtered. Figure 27 shows the `iterator` method from project `decorators-squad/eo-yaml`.

Validating the return value of this method requires access to the actual string elements of the return value. However, the return value has Iterator objects. Accessing elements in postconditions leads to side-effects.

Inaccessible code context (4). In Java, only public cross-class fields/methods can be accessed. This category refers to the case where the target method modifies some write-only code context. The postcondition cannot perform validation due to inaccessible code contexts. Figure 28 shows `drawRectanglesOfDifferences` in `romankh3/image-comparison`.

The `java.awt.Graphics2D` object is created from `java.awt.image.BufferedImage` by `BufferedImage::createGraphics()` API. However, due to the privacy limitation, the buffered image cannot be accessed by the created `Graphics2D` object. This method `drawRectanglesOfDifferences` draws rectangles onto a `BufferedImage` object using a `Graphics2D` object `graphics`, while the modified buffered image cannot be accessed for validation.

Random related (3). Some methods rely on random algorithms. For defects within these methods, test cases can be called multiple times, and bugs can be distinguished by observing the statistics, while a postcondition that runs each time after the method calls cannot catch the bugs. Figure 29a shows `changeMess` in `TheAlgorithms/Java`.

`ber` is a field. This method randomly gets a pseudorandom `x` between 0 and 1 and modifies `message` when `x < ber`. Figure 29b shows one representative test case that covers this method. After 1,000 calls of `changeMess`, it is nearly certain that the `wrongMess` field is larger than zero. However, the postcondition validation is called each time after the method runs. It is possible that after a method call, the `wrongMess` is still zero. Therefore, it is non-trivial to validate `wrongMess` due to this randomness.

Ordinary (2). Some methods' key feature is not related to accessible code context status change and cannot be caught by the specification framework. Figure 30 shows `execute` in `antkorwin/xsync`.

As the key feature of this `execute` method, we need to validate whether the `runnable` has been run, which is non-trivial to validate since the implementation of `runnable` can be various.

Others (6). Not included above, and non-trivial for human engineers to write complete postconditions. These methods are generally long or within complex code contexts, increasing the understanding costs. Figure 31 shows `douglasPeucker` in `dyn4j/dyn4j`.

This 85-line method is complicated, and recursion is used. The human engineer failed to write complete postconditions for it.

H.2.2 Mutant-level Filtering Analysis

Of the 210 Java methods in the benchmark, one has mutants filtered after manual checking. Two mutants are filtered. Note that we ensure the main

feature of the methods can still be validated after part of the mutants are filtered for these three methods.

All mutants are filtered due to **nameless implementation**, which happens in method `accumulate` from `dyn4j/dyn4j`: Figure 32a shows the target method, and Figure 32b shows a representative `isComplete` implementation from the test suite.

This method has two mutants that replace `if (force.isComplete(elapsedTime))` and `if (torque.isComplete(elapsedTime))` with `if (elapsedTime > 0)`. If `isComplete` has no side-effect, we can call it in the postconditions and validate that after the method calls, no element in list `this.forces` and `this.torques` satisfies `isComplete`.

However, in particular, `isComplete` can be various, defined by inner classes, and has side effects. For example, Figure 32b shows the `Force` class implementation used in the test case `applyTimed`.

For the `Torque` class, the situation is similar.

Therefore, killing the mutant that disturbs the `isComplete`-related feature is non-trivial. It cannot be called directly. Without calling it, validating is also non-trivial due to the various implementations with inner classes involved.

```
@classmethod
def prefix_lines(
    cls,
    lines: Iterable[str],
    first_line_prefix: str,
    following_line_prefix: str = None,
) -> Iterable[str]:
    ...
    for line in lines:
        ...
        yield prefixed if not prefixed.isspace() else ""
```

Figure 13: Python iterator-related filtered method example: `prefix_lines` from miyuchina/mistletoe.

```
def load_inventory(source: Path) -> Mapping[str, Mapping[str, InventoryEntry]]:
    ...
    with (source / "objects.inv").open("rb") as fp:
        ...
```

Figure 14: Python timeout-related filtered method example: `load_inventory` from hynek/doc2dash.

```
def execute_code(
    code: str,
    language: str = "python",
    libraries: list[str] | None = None,
    timeout: int = 30,
) -> list[ImageContent | TextContent]:
    ...
    try:
        ...
        with session_cls(
            ...
        ) as session:
            result = session.run(
                code=code,
                libraries=libraries or [],
                timeout=timeout,
            )
    ...
    else:
        return results
```

Figure 15: Python concurrency-related filtered method example: `execute_code` from vndee/llm-sandbox.

```
async def get_task(self) -> Task | None:
    ...
    self._current_task = await self.task_store.get(
        self.task_id, self._call_context
    )
    ...
    return self._current_task
```

Figure 16: Python coroutine-related filtered method example: `get_task` from a2aproject/a2a-python.

```
def while_until_true(interval, max_attempts):
    def decorator(f):
        ...

    return decorator
```

Figure 17: Python language-entity filtered method example: `while_until_true` from pypyr/pypyr.

```
def make_sync(_class: type[T]) -> Any:
    """Wraps the methods of the given model class so that they can be called without
    ↪  `await`."""

    class ModelWrapper(_class):  # type: ignore
        _instance_is_sync: bool = True
    ...
    return ModelWrapper
```

Figure 18: Python language-entity filtered method example: `make_sync` from run-llama/llama_deploy.

```
def rich_format_error(self: click.ClickException) -> None:
    ...
    if ctx is not None:
        console.print(ctx.get_usage())

    if ctx is not None and ctx.command.get_help_option(ctx) is not None:
        console.print(
            ...
        )

    console.print(
        ...
    )
```

Figure 19: Python ordinary-category filtered method example: `rich_format_error` from fastapi/typer.

```
def test_help_create():
    result = runner.invoke(app, ["create", "--help"])
    assert result.exit_code == 0
    ...
```

Figure 20: Python specified-error-process example: test case covering `_make_command_help` from langchain-ai/langchain-mcp-adapters.

```
def merge(self, add_me):
    """Merge add_me into context and applies interpolation.
    ... (27 lines of comments)
    """
    def merge_recurse(current, add_me):
        ... (47 lines)
    ... (2 lines)
    merge_recurse(self, add_me)
```

Figure 21: Python other filtered method example: `merge` from pypyr/pypyr.

```
def getprops(self, prop_names):
    ...
    if attempt < attempts - 1:
        time.sleep(DEFAULT_GETPROPS_RETRY_SLEEP_SEC)
    return results
```

Figure 22: Python mutant-level ordinary example: `getprops` from google/mobly.

```
result = _invoke_isolated_cli_runner(["./", "--output", "-", "--exclude-pattern", "tests/"])
...
assert result.exit_code == 0
```

Figure 23: Python mutant-level specified-error-process example: triggered test case for `_parse_ignore_file` from coderamp-labs/gitingest.

```
def k_closest(points, k, origin=(0, 0)):
    ...
    heapify(heap)
    ...
    for point in points[k:]:
        ...
    return [point for nd, point in heap]
```

(a) Original method.

```
def k_closest(points, k, origin=(0, 0)):
    ...
    heap.sort()
    ...
    for point in points[k:]:
        ...
    return [point for nd, point in heap]
```

(b) Equivalent mutant.

```
self.assertEqual(
    [(-2, -2), (1, 1), (1, 0), (-1, 0)], k_closest(...)
)
```

(c) Over-constrained test case.

Figure 24: Python mutant-level equivalent-result example: original method, equivalent mutant, and test case for `k_closest` from keon/algorithms.

```
def partition_by_spaces(text, spaces) -> tuple[str, str, str]:
    ...
    for i, c in enumerate(text):
        ...
        elif start >= 0:
            end = i
            break
    ...
```

(a) Target method.

```
def parse(...) -> list[str] | SetextHeading:
    ...
    while ...:
        if cls.break_paragraph(source):
            break
        ...

def break_paragraph(...) -> bool:
    try:
        str1, str2, str3 = parse_leading(...)
        if ...:
            return True
        return False
    finally:
        ...
```

(b) Simplified downstream call path.

Figure 25: Python mutant-level timeout example: `partition_by_spaces` from frostming/marko and a simplified downstream call path.

```
public T compare() throws IOException, RenderingException {
    ...
    try (PDDocument expectedDocument = Loader.loadPDF(...)) {
        try (PDDocument actualDocument = Loader.loadPDF(...)) {
            compare(expectedDocument, actualDocument);
        }
    }
    ...
    return compareResult;
}
```

Figure 26: Java timeout-related filtered method example: `compare` from red6/pdfcompare.

```
public Iterator<YamlLine> iterator() {
    Iterator<YamlLine> iterator = this.yamlLines.iterator();
    if (iterator.hasNext()) {
        final List<YamlLine> docsStart = ...
        ...
        iterator = docsStart.iterator();
    }
    return iterator;
}
```

Figure 27: Java iterator-related filtered method example: `iterator` from decorators-squad/eo-yaml.

```
private void drawRectanglesOfDifferences(List<Rectangle> rectangles, Graphics2D graphics) {
    List<Rectangle> rectanglesForDraw;
    ...
    draw(graphics, rectanglesForDraw);
    if (fillDifferenceRectangles) {
        fillRectangles(graphics, ...);
    }
}
```

Figure 28: Java inaccessible-code-context filtered method example: `drawRectanglesOfDifferences` from romankh3/image-comparison.

```
public void changeMess() {
    for (int y : message) {
        double x = randomGenerator.nextDouble();
        ...
        if (x < ber) {
            messageChanged = true;
            ...
            message.set(...);
        }
    }
    if (messageChanged) {
        wrongMess++;
    }
}
```

(a) Target method.

```
@Test
void testIntermediateBER() {
    CRCAlgorithm c = new CRCAlgorithm("1101", 4, 0.5);
    c.generateRandomMess();
    for (int i = 0; i < 1000; i++) {
        ...
        c.changeMess();
        ...
    }
    assertTrue(c.getWrongMess() > 0);
    ...
}
```

(b) Representative test case.

Figure 29: Java random-related filtered method examples for `changeMess` from TheAlgorithms/Java.

```
public void execute(KeyT firstKey, KeyT secondKey, Runnable runnable) {
    ...
    if (...) {
        synchronized (firstMutex) {
            synchronized (secondMutex) {
                runnable.run();
            }
        }
    } else {
        synchronized (globalLock) {
            synchronized (firstMutex) {
                synchronized (secondMutex) {
                    runnable.run();
                }
            }
        }
    }
}
```

Figure 30: Java ordinary-category filtered method example: `execute` from antkorwin/xsync.

```
/**
 * Recursively ... (8 lines of comments)
 */
private final List<Vector2> douglasPeucker(...) {
    ... (22 lines)
    if (...) {
        // sub-divide and run the algo on each half
        List<Vector2> aReduced = this.douglasPeucker(...);
        List<Vector2> bReduced = this.douglasPeucker(...);
        ... (4 lines)
    } else {
        ... (41 lines)
    }

    return result;
}
```

Figure 31: Java other filtered method example: `douglasPeucker` from dyn4j/dyn4j.

```
protected void accumulate(double elapsedTime) {
    ...
    if (...) {
        Iterator<Force> it = this.forces.iterator();
        while(it.hasNext()) {
            Force force = it.next();
            ...
            if (force.isComplete(elapsedTime)) {
                it.remove();
            }
        }
    }
    ...
    if (...) {
        Iterator<Torque> it = this.torques.iterator();
        while(it.hasNext()) {
            Torque torque = it.next();
            ...
            if (torque.isComplete(elapsedTime)) {
                it.remove();
            }
        }
    }
}
```

(a) Target method.

```
@Test
public void applyTimed() {
    ...
    Force f = new Force(1, 0) {
        private double time = 0;
        public boolean isComplete(double elapsedTime) {
            time += elapsedTime;
            if (time >= 2.0 / 60.0) {
                return true;
            }
            return false;
        }
    };
    ...
}
```

(b) Representative test implementation.

Figure 32: Java mutant-level nameless-implementation examples from dyn4j/dyn4j.

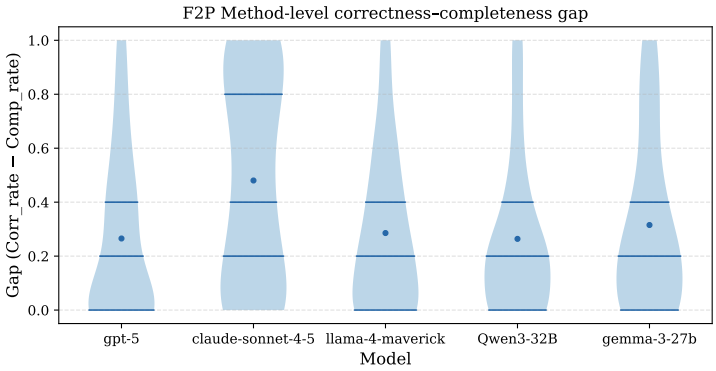


Figure 33: Distribution of method-level correctness–completeness gaps with F2P setting under different models. For each method m, the gap is computed as $\Delta(m) = \text{corr\_rate}(m) - \text{comp\_rate}(m)$, where corr_rate and comp_rate denote the fraction of generated postcondition sets that are test-correct and complete, respectively. Each violin aggregates gap values across methods for which at least one test-correct postcondition is generated; methods with Corr@k = 0 are excluded.

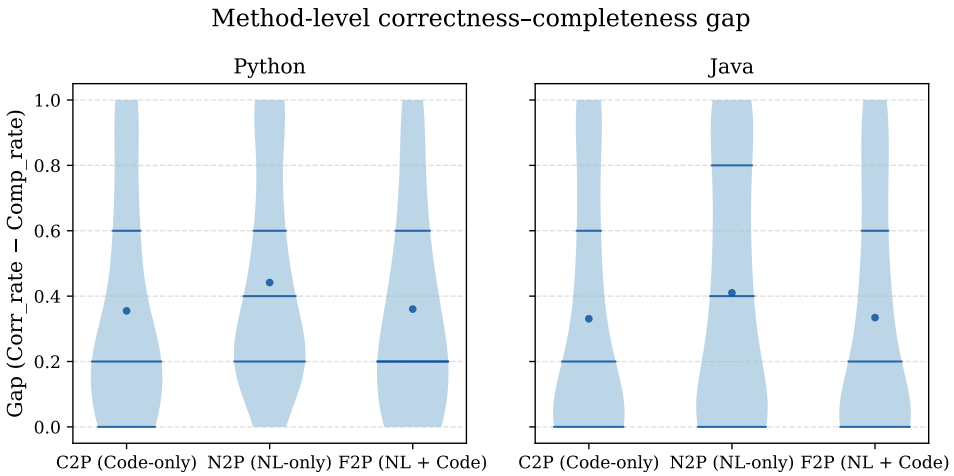


Figure 34: Distribution of method-level correctness–completeness gaps under different input settings and programming languages. For each method m, the gap is computed as $\Delta(m) = \text{corr\_rate}(m) - \text{comp\_rate}(m)$, where corr_rate and comp_rate denote the fraction of generated postcondition sets that are test-correct and complete, respectively. Each violin aggregates gap values across methods for which at least one test-correct postcondition is generated; methods with Corr@k = 0 are excluded.