# When Postconditions Help or Mislead: An Empirical Study of LLM Code Review

Shirshak Jung Thapa
Independent Researcher
Orlando, FL, USA
shirshakjungthapa@gmail.com

## Abstract

LLM code reviewers see what code does but not what it is supposed to do, so they flag correct code and miss real bugs. Postconditions state intended behavior explicitly, and LLMs can now infer them, but LLM-inferred postconditions are often incomplete. We ask two questions: do the benchmark's validated reference postconditions help LLM code reviewers, and what happens when LLM-inferred postconditions are used instead? Using PostcondBench, we review 210 real-world Java methods, each in its bug-free original form and as three mutants. The reviewer sees the method alone, the whole source file, or the method with either the reference or the LLM-inferred postconditions. Three frontier reviewers, Claude Opus 5.5, Gemini 4 Argon and GPT-6.1 Sol, produce 17,640 reviews, which we score against the benchmark. For Claude and Gemini, adding the reference postconditions cuts false alarms by more than half, whereas adding the whole source file does not significantly change them. In our main prompt, the LLM-inferred postconditions come with the instruction "Treat them as the specification". With it, GPT-6.1 Sol finds 16 percentage points fewer bugs; on bugs the postconditions do not cover, its rate falls from 82% to 21%. Claude's and Gemini's false alarms fall by 6.2 and 4.8 points, but not significantly. A follow-up experiment shows that the instruction causes GPT-6.1 Sol's drop: without it, the same postconditions help GPT-6.1 Sol find more bugs than with no postconditions (89.5% vs. 86.3%), and its false alarms stay about the same. Some of what the benchmark counts as GPT-6.1 Sol's false alarms are real: in 19 of the 23 cases we could check, the code breaks its reference postconditions on an input the benchmark's tests never try. Postconditions should therefore be given to a reviewer as information, not with an instruction to treat them as the specification.

## CCS Concepts

• **Software and its engineering** → **Software verification and validation**; • **Computing methodologies** → *Artificial intelligence.*

## Keywords

code review, large language models, postconditions, formal specifications, JML, false alarms

## 1 Introduction

LLMs are now routinely used to review code, both as assistants in pull requests and as automated checkers in industrial pipelines [1, 14]. A reviewer of this kind faces a basic difficulty: it sees what the code does but not what the code is supposed to do. Without a statement of intent, it must guess, and its guesses produce two kinds of error. It flags correct code whose behavior it did not expect, and it accepts buggy code whose behavior looks plausible. Both errors are costly, and false alarms have a well-documented cost of their own. False positives are a main reason developers avoid static analysis tools [4], and in an industrial deployment of an LLM reviewer, pull requests took longer to close and developers reported faulty and irrelevant comments as drawbacks [1].

Postconditions state intent explicitly. A postcondition says what a method guarantees when it returns, so a reviewer who has it no longer has to guess what the method is supposed to do. Postconditions are costly to write by hand and rarely written in practice, but LLMs can now infer them [2, 10, 16], which makes it practical to give a reviewer postconditions for every method. LLM-inferred postconditions, however, are often incomplete: PostcondBench, a recent benchmark for postcondition inference, finds that they often hold on the original code yet fail to rule out many bugs [17]. A reviewer that trusts them may stop looking for exactly the bugs they miss. Whether postconditions help or mislead an LLM reviewer has, to our knowledge, not been measured.

We study this question on PostcondBench. Its 210 Java methods from 60 open-source projects come with bug-free originals, mutants that fail the project's tests, reference postconditions and LLM-inferred postconditions of known quality. Three reviewers from three vendors, Claude Opus 5.5, Gemini 4 Argon and GPT-6.1 Sol, review each method in its original form and as three mutants. The reviewer sees the method alone, the whole source file without postconditions, or the method with either the reference or the LLM-inferred postconditions; the LLM-inferred ones are introduced either with the instruction to treat them as the specification or with a warning that they may be incomplete or wrong. A follow-up experiment adds two more conditions: the LLM-inferred postconditions without the instruction to treat them as the specification, and the reference postconditions with a warning that they may be incomplete or wrong. We score every review against the benchmark: a warning on an original is a false alarm, and a warning that points at a mutant's changed line localizes the bug.

We ask two questions, each broken into research questions.
*Do postconditions help?*

- **RQ1** Do postconditions reduce false alarms? (§5.1)
- **RQ2** Do postconditions help reviewers find and localize bugs? (§5.2)

*When do they mislead, and why?*

- **RQ3** What happens when the postconditions are incomplete or incorrect? (§5.3)
- **RQ4** How does the framing of the postconditions change reviewer behavior? (§5.4)
- **RQ5** Why do reviewers differ? (§5.5)

Our main findings are:

(1) **Reference postconditions reduce false alarms.** For Claude Opus 5.5 and Gemini 4 Argon, they cut false alarms by more than half; the whole source file does not.
(2) **One instruction makes GPT-6.1 Sol miss bugs.** Told to treat LLM-inferred postconditions as the specification, GPT-6.1 Sol finds 16 percentage points fewer bugs, mostly by reporting nothing, and misses most of the bugs the postconditions do not cover. The follow-up shows that the instruction, not the postconditions, causes this: presented neutrally, the same postconditions help GPT-6.1 Sol find more bugs than the method alone. We call this *silent over-trust*: a reviewer that trusts incomplete postconditions stops warning about the bugs they do not cover, so its errors appear as missing warnings rather than wrong ones. Figure 1 shows a real case.
(3) **Most of the GPT-6.1 Sol false alarms we could check are accurate.** When we run the inputs its warnings name, the code breaks its reference postconditions in 19 of 23 cases, on inputs the benchmark's tests never try.

This paper makes the following contributions:

- A controlled study of how postconditions of different origin, quality and framing affect three LLM code reviewers, on 210 real Java methods and 630 mutants.
- A judge-free scoring protocol for LLM code review that uses only the benchmark's ground truth.
- Evidence that the same postconditions affect different reviewers differently, that over-trust in incomplete postconditions is silent, showing up as missing warnings rather than wrong ones, and, in a follow-up experiment, that the instruction to treat them as the specification, not the postconditions themselves, causes it.
- A replication package with all prompts, reviews, scoring code and analysis scripts.

```
public static String decrypt() {
  StringBuilder wordDecrypted = new StringBuilder();
   for (int i = 1; i < table.length - 1; i++) {
    for (Object item : table[i]) {
      wordDecrypted.append(item);
    }
  }
  return wordDecrypted.toString()
      .replaceAll(ENCRYPTION_FIELD, "");
}
```

*LLM-inferred postconditions (incomplete):*

```
//@ ensures \result != null;
//@ ensures !\result.contains(ENCRYPTION_FIELD);
//@ ensures \result.length() <= (table.length - 1) * table[0].length;

//@ ensures table != null ==> \result.length() >= 0;
```

| Prompt | GPT-6.1 Sol's reply |
|---|---|
| Method only | Flags the loop line: "The loop skips the final row of the table, omitting any plaintext characters stored there from the decrypted result." |
| Authoritative | `{"bugs": []}` |
| Neutral | Flags the same line, with the same explanation |

**Figure 1: A real case from our data (TheAlgorithms/-Java, `ColumnarTranspositionCipher.decrypt`; Javadoc omitted). The mutant changes the loop bound from `i < table.length` to `i < table.length - 1`, dropping the last row of the table. The LLM-inferred postconditions bound the result's length from above but never require every row to appear, so they do not reject this mutant. Without postconditions, GPT-6.1 Sol reports the changed line. Told to treat the postconditions as the specification, it reports no bug. Given the same postconditions without that instruction, it reports the line again, with the same explanation. Claude Opus 5.5 and Gemini 4 Argon report the line in all three conditions. This pattern is typical: 118 of the 137 buggy versions whose detection changed lost every warning.**

## 2 Background and Related Work

*Postconditions and their inference.* A postcondition states what a method guarantees when it returns: a relation between its inputs, its result and the state it changes. In Java, postconditions are commonly written in the Java Modeling Language (JML) [8] as `ensures` clauses that can be checked at run time. Writing them by hand is costly, so a long line of work infers them, first dynamically from execution traces, as in Daikon [3], and more recently with LLMs. Endres et al. introduced the task of turning natural-language intent into postconditions with LLMs and evaluated the results by correctness on tests and by their power to reject buggy code [2]. SpecGen [10] and AutoSpec [16] generate specifications that a verifier can prove, and Lahiri studies how to evaluate whether an LLM-written specification captures user intent [7]. PostCond-Bench [17] evaluates inferred postconditions on real methods from open-source projects, scoring them by correctness on the project's tests and by completeness, the ability to reject every mutant of the method. It reports a substantial gap between the two: LLM-inferred postconditions often pass the correctness check yet fail to reject many mutants. All of this work evaluates specifications on their own. We ask what happens when postconditions are handed to a downstream consumer, an LLM code reviewer, and in particular whether incomplete postconditions do harm.

*LLM-based code review.* Automated review moved from models trained on review data, such as CodeReviewer [9], to general-purpose LLMs deployed in industry. At Google, AutoCommenter flags violations of coding practices, and its usefulness depended on suppressing comments developers found unhelpful [14]. In an industrial study of an LLM review tool, Cihan et al. found that most comments were acted on, but the tool lengthened pull-request closure times, and developers reported faulty and irrelevant comments as drawbacks [1]. These findings echo earlier work on static analysis, where false positives are a main reason developers avoid the tools [4]. Our focus on false alarms follows from this: a reviewer that flags correct code imposes a cost even when it also finds real bugs.

*How LLMs use the context they are given.* The input an LLM receives beyond the task can mislead it. Irrelevant context degrades reasoning [12], and models tend to agree with what the user appears

to believe [11]. For code, the evidence is mixed. Comments can substantially change how well LLMs fix bugs [15], and LLM-generated test oracles tend to encode a program's actual behavior rather than its intended behavior [6]. Yet adversarial comments had little effect on LLM security reviewers in a recent large study [13]. A postcondition differs from a comment: it is formal, and the prompt can present it as authoritative. We show that whether postconditions mislead depends on the reviewer and on how they are framed.

*Evaluating reviewers.* Review quality is often scored by another LLM acting as a judge [18], which introduces a second model's errors and biases into the measurement. Mutation-based evaluation offers a ground truth that requires no judge: mutants are a reasonable proxy for real faults in testing [5], and PostcondBench's mutants come with a known location. We score reviews against that location alone (§3.4), and report the judge-based analysis only for comparison.

## 3 Methodology

We measure two things: how often a reviewer flags correct code (false alarms), and how often it finds an injected bug (localization). Three reviewers review every method under several conditions (§3.2, §3.3), and a fixed rule scores each review against the benchmark (§3.4).

### 3.1 Benchmark Items

We use the 210 Java methods of PostcondBench [17], taken from 60 open-source repositories (median length 25 lines). For each method, PostcondBench provides the original code, which passes the project's tests, and a set of mutants, each of which fails at least one of those tests. We sample three mutants per method with a fixed seed, drawing from both rule-based and LLM-generated mutants (251 rule-based and 379 LLM-generated in total), and, in 142 mutants, remove comments that the mutation added to its changed lines and that name the injected change (e.g., `// Always null, causing incorrect behavior`); the method's own comments are kept. Each method thus appears in four versions, the original and three buggy versions, for 840 versions in total.

*Ground truth.* Following the benchmark, an original is bug-free, and a buggy version contains one bug, located at the lines where it differs from the original.

### 3.2 Review Conditions

Table 1 lists what each condition shows the reviewer besides the method. No condition combines the whole file with postconditions.

**Method only.** The method, including its Javadoc comment. This is the baseline.

**Whole file.** The whole source file and no postconditions; the reviewer is told to review only the method. This adds context but no statement of intent, so it separates the effect of postconditions from the effect of more input.

**Reference postconditions.** PostcondBench's reference JML postconditions. They were drafted with the method's mutants in view and refined by experts until they reject every mutant, so they show what postconditions can do when they cover the bugs under review.

**Table 1: What each condition shows the reviewer besides the method under review. The last two rows are the follow-up experiment.**

| Condition | Rest of file | Postconditions (framing) |
|---|---|---|
| Method only | – | none |
| Whole file | ✓ | none |
| Reference | – | reference (authoritative) |
| LLM-inferred, authoritative | – | LLM-inferred (authoritative) |
| LLM-inferred, hedged | – | LLM-inferred (hedged) |
| LLM-inferred, neutral | – | LLM-inferred (neutral) |
| Reference, hedged | – | reference (hedged) |

**LLM-inferred postconditions.** The JML postconditions that Claude Sonnet 4.5 inferred for the method, as released with PostcondBench (first of five samples, in PostcondBench's F2P setting, where the model sees the method's code, its comment and the enclosing file). Using PostcondBench's evaluation, we label each set *complete* (correct on the tests and rejecting every one of the method's mutants in PostcondBench, not only the three we sample; 73 methods), *incomplete* (correct but missing at least one of those mutants; 90) or *incorrect* (failing the correctness check; 47).

*Framing.* One sentence introduces the postconditions, and it is the only text that differs between framings:

**Authoritative:** *"The intended behavior of this method is specified by these JML postconditions. Treat them as the specification:"*

**Hedged:** *"These JML postconditions were generated automatically and may be incomplete or wrong. Use them as a hint about the intended behavior, not as ground truth:"*

**Neutral:** *"JML postconditions for this method:"*

The main study presents reference postconditions with the authoritative framing and LLM-inferred postconditions with the authoritative and the hedged framing. A follow-up experiment adds the neutral framing for LLM-inferred postconditions and the hedged framing for reference postconditions. Appendix A gives the full prompts.

### 3.3 Reviewers and Prompt

We study three reviewers from three vendors: Claude Opus 5.5 (Anthropic), Gemini 4 Argon (Google) and GPT-6.1 Sol (OpenAI). All three receive the same system prompt:

```
You are an expert Java code reviewer. Review the method for
      FUNCTIONAL bugs only: cases where the code does not do what it
      is supposed to do. Ignore style, naming, performance, and
      missing documentation. Only report a bug if you are reasonably
      confident it is real. Respond with JSON only: {"bugs": [{"line
      ": "<exact code line>", "explanation": "<one sentence>"}]} Use
      {"bugs": []} if you find no functional bug.
```

Each item is a single request with no history, examples or tools, at each provider's default sampling and reasoning settings; current Claude models reject a temperature setting. The Claude API requires an output limit, which we set to 1,500 tokens. A reply that is not the JSON object above counts as a review with no warning; this

happened for 7 of the 12,600 main-study reviews and none of the 5,040 follow-up reviews.

### 3.4 Judge-Free Scoring

Let $O$ be the reviews of originals in a condition, $M$ the reviews of buggy versions, and $R(x)$ the lines reported in review $x$:

$$\text{FA} = \frac{1}{|O|} \sum_{o \in O} \mathbf{1}\big[\, |R(o)| > 0 \,\big], \tag{1}$$

$$\text{Det} = \frac{1}{|M|} \sum_{m \in M} \mathbf{1}\big[\, |R(m)| > 0 \,\big], \tag{2}$$

$$\text{Loc} = \frac{1}{|M|} \sum_{m \in M} \mathbf{1}\big[\, \exists\, r \in R(m),\ t \in T(m) :\ \text{match}(r, t) \,\big]. \tag{3}$$

The *false-alarm rate* is the share of originals with at least one warning; a false alarm is thus a warning on an original that the benchmark treats as bug-free. The *detection rate* is the share of buggy versions with at least one warning, and the *localization rate* the share of buggy versions with a warning at the bug.

*Bug location.* $T(m)$ holds the lines of buggy version $m$ that differ from the original; for a pure deletion, it also holds the deleted text and the neighboring lines. Lines holding only braces or similar tokens are dropped. A reported line matches a bug line if, ignoring whitespace, the two are equal, or, when both are longer than eight characters, one contains the other.

*Why this rule.* Localization is our main bug-finding measure, because reviewers warn on almost every buggy version (§5.2). The rule checks where a warning points, not its explanation. Claude Opus 5.5's and GPT-6.1 Sol's reviews were also scored by LLM judges, each model judging the other's reviews; the line rule agrees with them on 93% of mutant reviews (Appendix C). Unlike a judge, the rule is deterministic and can be recomputed from the released reviews.

## 4 Experimental Setup

*Runs.* All reviews were collected through the vendors' official APIs from one fixed item file, so every reviewer sees identical prompts; each stored review records the hash of its prompt. The reviewers are `claude-opus-5-5`, `gemini-4-argon` and `gpt-6.1-sol`, as named by their APIs, run at each provider's default sampling and reasoning settings (§3.3). The main study has 4,200 reviews per reviewer (840 versions × 5 conditions; 12,600 in total) and the follow-up 1,680 per reviewer (840×2; 5,040). To measure run-to-run variation, we reviewed every item of 25 methods a second time; repeating a review changes its outcome for at most 2.1% of items.

*Statistics.* The method is the unit of analysis, since the versions of one method are not independent. Rates come with 95% confidence intervals from a method-level bootstrap (2,000 resamples). To compare two conditions, we pair reviews of the same version and apply a two-sided sign-flip test over methods (exact when at most 20 methods differ, otherwise 200,000 random sign assignments; seed 0). We correct for multiple comparisons with Holm's method, over eight tests per reviewer in the main study (Table 5) and four in the follow-up (Table 4). Results on detection, on specification quality and on the reasons for reviewer differences are exploratory, and their $p$-values (Table 6) are not adjusted.

## 5 Results

Tables 2 and 3 give every rate, and Appendix B every test. Differences are in percentage points (pp). All conclusions hold when replies that are not valid JSON are excluded, and repeating the reviews changes their outcome for at most 2.1% of items. Sections 5.1–5.2 ask whether postconditions help; Sections 5.3–5.5 ask when they mislead, and why.

### 5.1 Do Postconditions Reduce False Alarms?

**Yes for Claude Opus 5.5 and Gemini 4 Argon; no for GPT-6.1 Sol.** Without postconditions, the reviewers flag 15.7% (Claude Opus 5.5), 19.0% (Gemini 4 Argon) and 29.0% (GPT-6.1 Sol) of correct methods (Table 2).

*Reference postconditions.* Adding the reference postconditions cuts these rates by more than half for Claude Opus 5.5 (to 5.2%) and Gemini 4 Argon (to 8.1%); both drops are significant. GPT-6.1 Sol's rate does not fall (36.2%; the change is not significant).

*Whole file.* Giving the whole source file instead, without postconditions, does not significantly change any reviewer's rate. The drop therefore comes from the postconditions, not from extra context.

*LLM-inferred postconditions.* With the instruction to treat them as the specification, LLM-inferred postconditions lower the rates less: to 9.5% for Claude Opus 5.5 and 14.3% for Gemini 4 Argon, while GPT-6.1 Sol stays at 29.5%. Where the inferred postconditions are complete (73 methods), they work about as well as the reference ones on the same methods: false alarms are 0.0%, 4.1% and 16.4% for Claude Opus 5.5, Gemini 4 Argon and GPT-6.1 Sol, against 0.0%, 2.7% and 20.5% with reference postconditions. These methods also get fewer false alarms without postconditions (5.5%, 8.2% and 23.3%).

> **Finding.** Postconditions cut false alarms for Claude Opus 5.5 and Gemini 4 Argon, by more than half when they are the reference ones; the whole file does not. GPT-6.1 Sol's false alarms do not fall, but many of them are accurate (§5.5).

### 5.2 Do Postconditions Help Reviewers Find and Localize Bugs?

**Reference postconditions slightly improve bug finding; with the authoritative instruction, LLM-inferred postconditions cost GPT-6.1 Sol 16 pp.** Almost every review of a buggy version contains some warning (95.9–99.0%), even without postconditions. But the same reviewers also warn on 16–29% of correct methods, so a warning alone does not show that the reviewer found the bug. We therefore measure whether a warning points at the line that contains the bug (localization). Without postconditions, this happens for 86.3% (GPT-6.1 Sol) to 91.7% (Claude Opus 5.5) of buggy versions (Table 3).

*Reference postconditions.* Localization rises for all three reviewers, significantly for Gemini 4 Argon (+3.5 pp) and GPT-6.1 Sol (+5.9 pp); Claude Opus 5.5, already at 91.7%, gains 1.3 pp. For GPT-6.1 Sol, the whole file helps almost as much (+4.4 pp).

**Table 2: False-alarm rate: share of the 210 bug-free methods flagged as buggy (%). Δ: paired change in percentage points vs. method only. †For the hedged row, Δ is against the authoritative LLM-inferred postconditions, not method only. ‡Follow-up conditions; their tests are in Table 4. Holm-adjusted method-clustered sign-flip test over eight tests per reviewer: $^{*}p < .05$, $^{**}p < .01$, $^{***}p < .001$.**

| | Claude Opus 5.5 | | Gemini 4 Argon | | GPT-6.1 Sol | |
|---|---|---|---|---|---|---|
| Condition | Rate | Δ | Rate | Δ | Rate | Δ |
| Method only | 15.7 | | 19.0 | | 29.0 | |
| Whole file | 13.8 | −1.9 | 21.0 | +1.9 | 36.2 | +7.1 |
| Reference | 5.2 | $-10.5^{***}$ | 8.1 | $-11.0^{**}$ | 36.2 | +7.1 |
| LLM-inferred, authoritative | 9.5 | −6.2 | 14.3 | −4.8 | 29.5 | +0.5 |
| LLM-inferred, hedged† | 10.5 | +1.0 | 17.6 | +3.3 | 28.6 | −1.0 |
| LLM-inferred, neutral‡ | 8.6 | | 12.4 | | 29.5 | |
| Reference, hedged‡ | 8.1 | | 13.3 | | 30.0 | |

**Table 3: Bug finding on the 630 buggy method versions (%). Det.: any warning reported. Loc.: a warning points at the benchmark's changed line. ΔLoc.: paired change in localization (percentage points), as in Table 2; †: change vs. LLM-inferred, authoritative, not vs. method only; ‡: follow-up conditions, tested in Table 4.**

| | Claude Opus 5.5 | | | Gemini 4 Argon | | | GPT-6.1 Sol | | |
|---|---|---|---|---|---|---|---|---|---|
| Condition | Det. | Loc. | ΔLoc. | Det. | Loc. | ΔLoc. | Det. | Loc. | ΔLoc. |
| Method only | 99.0 | 91.7 | | 97.3 | 89.4 | | 95.9 | 86.3 | |
| Whole file | 100.0 | 92.7 | +1.0 | 99.0 | 90.5 | +1.1 | 99.4 | 90.8 | $+4.4^{***}$ |
| Reference | 99.8 | 93.0 | +1.3 | 99.8 | 92.9 | $+3.5^{**}$ | 99.8 | 92.2 | $+5.9^{***}$ |
| LLM-inferred, authoritative | 98.9 | 92.1 | +0.3 | 97.8 | 91.1 | +1.7 | 80.2 | 70.3 | $-16.0^{***}$ |
| LLM-inferred, hedged† | 99.4 | 92.7 | +0.6 | 97.9 | 90.3 | −0.8 | 97.5 | 88.3 | $+17.9^{***}$ |
| LLM-inferred, neutral‡ | 99.2 | 92.5 | | 98.1 | 91.1 | | 98.9 | 89.5 | |
| Reference, hedged‡ | 99.8 | 92.5 | | 98.7 | 91.9 | | 99.0 | 90.2 | |

*LLM-inferred postconditions.* With the authoritative instruction, Claude Opus 5.5 and Gemini 4 Argon are unaffected, but GPT-6.1 Sol's localization falls from 86.3% to 70.3%. This is silent overtrust: of the 137 buggy versions whose detection changes, 118 lose every warning. When the inferred postconditions are complete, localization rises for all three reviewers instead (Figure 2).

*Warning counts.* Reviewers give about one warning per buggy version (0.9–1.2) in every condition, so localization does not reward scattering warnings.

**Finding.** Reference postconditions modestly improve localization. With the authoritative instruction, LLM-inferred postconditions cost GPT-6.1 Sol 16 pp, mostly because it reports nothing; Claude Opus 5.5 and Gemini 4 Argon are unaffected.

## 5.3 What Happens When the Postconditions Are Incomplete or Incorrect?

**Under the authoritative instruction, only GPT-6.1 Sol is misled, and only when the postconditions are incomplete or incorrect.** Figure 2 splits the buggy versions by the quality of their LLM-inferred postconditions. Claude Opus 5.5 and Gemini 4 Argon stay within 3 pp of their baseline in every group. GPT-6.1 Sol's localization rises with complete postconditions (93.6% to 97.7%) but falls with incomplete ones (81.1% to 48.5%) and incorrect ones (85.1% to 69.5%). With the hedged framing, it returns to its baseline.

False alarms split the same way. With incorrect postconditions, Claude Opus 5.5's and Gemini 4 Argon's false alarms fall (25.5% to 12.8% for both), while GPT-6.1 Sol's rise (27.7% to 42.6%). These groups are small (47–90 methods), so we report all results in this section without tests.

**Finding.** Under the authoritative instruction, incomplete and incorrect postconditions mislead GPT-6.1 Sol; Claude Opus 5.5's and Gemini 4 Argon's bug finding is unaffected by postcondition quality. §5.4 shows that the instruction, not the postconditions, is responsible.

## 5.4 How Does the Framing of the Postconditions Change Reviewer Behavior?

**The instruction "Treat them as the specification" causes GPT-6.1 Sol's loss.** Figure 3 shows the same LLM-inferred postconditions under all three framings.

*Hedged.* Replacing the instruction with a warning that the postconditions may be incomplete or wrong restores GPT-6.1 Sol's localization to 88.3%, slightly above its baseline. Claude Opus 5.5 and Gemini 4 Argon barely change.

*Neutral (follow-up).* The hedged sentence both removes the instruction and adds a warning; the follow-up separates the two. With neither, GPT-6.1 Sol's localization is 89.5%: 19.2 pp above the authoritative framing and 3.2 pp above the method alone, both significant

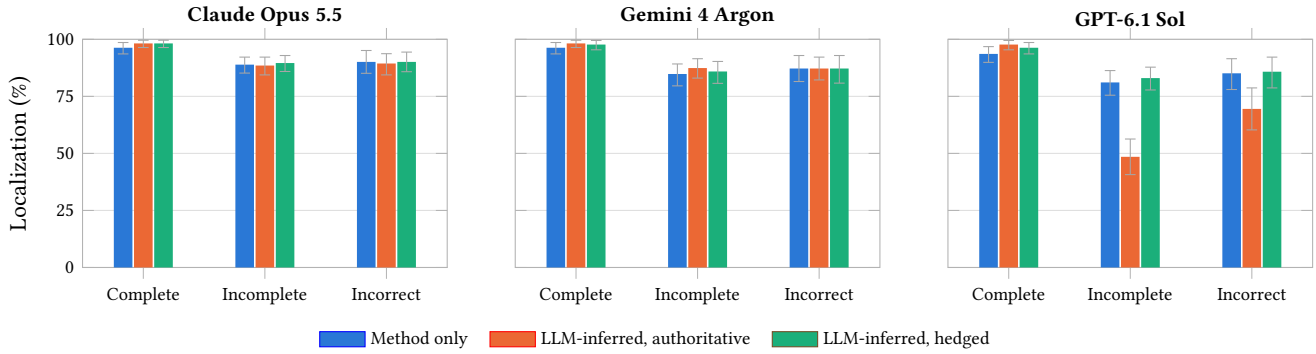


**Figure 2: Localization on buggy versions, grouped by the quality of the LLM-inferred postconditions (labels from PostcondBench's evaluation, §3.2: 73 complete, 90 incomplete, 47 incorrect methods; 3 buggy versions each). Error bars: method-bootstrap 95% intervals. Only GPT-6.1 Sol drops, and only when the postconditions are presented as authoritative and are incomplete or incorrect.**

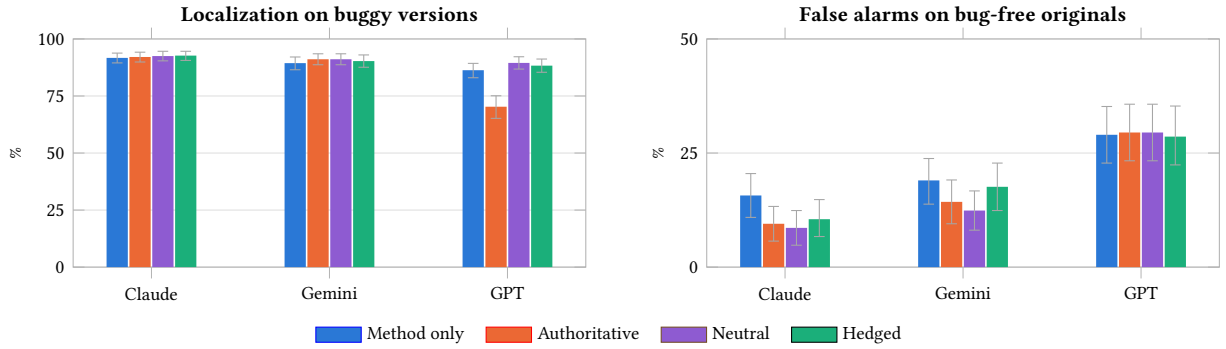


**Figure 3: The same LLM-inferred postconditions under three framings: authoritative ("Treat them as the specification"), neutral ("JML postconditions for this method:") and hedged ("may be incomplete or wrong"), against the method alone. Error bars: method-bootstrap 95% intervals. Only the authoritative framing hurts, and only for GPT-6.1 Sol.**

(Table 4). Its false alarms stay about the same (Table 2), and Claude Opus 5.5 and Gemini 4 Argon again barely change from the authoritative framing. The loss therefore comes from the instruction, not from the postconditions.

*Hedged reference postconditions.* Adding the warning to the reference postconditions gives back part of their false-alarm benefit for Claude Opus 5.5 (5.2% to 8.1%) and Gemini 4 Argon (8.1% to 13.3%), though both stay well below their baselines and neither change is significant. For GPT-6.1 Sol, it significantly lowers localization, by 2.1 pp.

**Finding.** The instruction, not the postconditions, causes GPT-6.1 Sol's loss: presented neutrally, LLM-inferred postconditions raise its localization above its baseline. Hedging also works, but with correct postconditions it gives back some of their benefit, significantly only for GPT-6.1 Sol's localization.

## 5.5 Why Do Reviewers Differ?

**Under the authoritative instruction, GPT-6.1 Sol checks the code against the postconditions.** The analyses in this section are exploratory.

*Bugs the postconditions miss.* For methods whose LLM-inferred postconditions are correct, we use PostcondBench's evaluation to split the buggy versions by whether those postconditions reject them (336) or not (153). On the versions they do not reject, GPT-6.1 Sol's localization falls from 82.4% to 20.9% under the authoritative instruction, and it reports nothing on 66% of them (silent over-trust); under each of the neutral and hedged framings, it localizes 84.3%. Claude Opus 5.5 and Gemini 4 Argon stay near their baselines. The comparison is paired within each buggy version.

*Warnings that cite the postconditions.* Under the reference postconditions, 79% of GPT-6.1 Sol's false alarms mention them, against 55% for Claude Opus 5.5 and 24% for Gemini 4 Argon. Under the authoritative LLM-inferred postconditions, the shares are 79% (GPT-6.1 Sol), 50% (Claude Opus 5.5) and 40% (Gemini 4 Argon); presented with the hedged framing, the same postconditions are cited in 8% or less of each reviewer's false alarms.

*Most of the warnings we could check are accurate.* These warnings often name an input on which, the reviewer claims, the code breaks a postcondition. Of the 70 false alarms that cite the reference

**Table 4: Follow-up experiment. Δ: paired change in percentage points with method-bootstrap 95% CI; +/−: versions (Loc.) or methods (FA) gained / lost; Holm over the four tests of each reviewer. Inferred: LLM-inferred postconditions; Ref.: reference postconditions; FA: false alarms; Loc.: localization.**

| Hypothesis | Outcome | Δ | 95% CI | +/− | $p$ | Holm $p$ |
|---|---|---|---|---|---|---|
| *Claude Opus 5.5* | | | | | | |
| H1 Inferred: neutral vs. authoritative | Loc. | +0.5 | [+0.0, +1.1] | 3/0 | .250 | .750 |
| H2 Inferred: neutral vs. method only | Loc. | +0.8 | [−0.5, +2.1] | 10/5 | .337 | .750 |
| H3 Ref.: hedged vs. authoritative | FA | +2.9 | [+0.0, +6.2] | 9/3 | .146 | .584 |
| H4 Ref.: hedged vs. authoritative | Loc. | −0.5 | [−1.1, +0.2] | 1/4 | .375 | .750 |
| *Gemini 4 Argon* | | | | | | |
| H1 Inferred: neutral vs. authoritative | Loc. | +0.0 | [−1.0, +1.0] | 6/6 | 1.000 | 1.000 |
| H2 Inferred: neutral vs. method only | Loc. | +1.7 | [+0.2, +3.5] | 17/6 | .051 | .206 |
| H3 Ref.: hedged vs. authoritative | FA | +5.2 | [+0.5, +10.0] | 19/8 | .052 | .206 |
| H4 Ref.: hedged vs. authoritative | Loc. | −1.0 | [−2.4, +0.3] | 4/10 | .244 | .487 |
| *GPT-6.1 Sol* | | | | | | |
| H1 Inferred: neutral vs. authoritative | Loc. | +19.2 | [+15.2, +23.5] | 124/3 | <.001 | <.001 |
| H2 Inferred: neutral vs. method only | Loc. | +3.2 | [+1.4, +5.1] | 25/5 | <.001 | .002 |
| H3 Ref.: hedged vs. authoritative | FA | −6.2 | [−13.8, +1.4] | 27/40 | .142 | .142 |
| H4 Ref.: hedged vs. authoritative | Loc. | −2.1 | [−3.5, −0.8] | 2/15 | .005 | .010 |

postconditions, we could run the 28 whose methods build without the full project (25 from GPT-6.1 Sol, 2 from Gemini 4 Argon, 1 from Claude Opus 5.5), checking the reference postconditions at run time with PostcondBench's own translator. For GPT-6.1 Sol, 23 could be decided. In 19, the code did break the postcondition, on edge inputs such as `NaN` values and integer overflow, or because the postcondition itself is too strict. The benchmark's tests never try these inputs, so its labels count these warnings as false alarms.

**Finding.** Under the authoritative instruction, GPT-6.1 Sol reviews the code against the postconditions: it misses bugs they do not cover, and it finds inputs on which the code breaks them. Claude Opus 5.5 and Gemini 4 Argon behave as if they treat postconditions as a hint about intent.

# 6 Discussion

## 6.1 Postconditions, Not Context, Reduce False Alarms

For Claude Opus 5.5 and Gemini 4 Argon, reference postconditions removed more than half of the false alarms, while the whole file did not significantly change them. A reviewer without a statement of intent has to guess what the code should do, and some of its warnings flag a guess the code does not match; a postcondition replaces the guess. This matters because false alarms are one of the main reasons developers ignore a review tool [4]. LLM-inferred postconditions gave a smaller reduction, which was not significant, and complete ones about the same on matched methods.

## 6.2 The Same Postconditions, Read Two Ways

The three reviewers received identical prompts, yet used the postconditions differently. Claude Opus 5.5 and Gemini 4 Argon behave as if they treat them as a hint about intent: they flag fewer correct methods, but still report bugs the postconditions do not cover. GPT-6.1 Sol, under the authoritative instruction, behaves as if it treats them as the definition of correctness. This cuts both ways. It is why GPT-6.1 Sol misses bugs the postconditions do not cover, often without any warning, and also why it finds inputs on which the code breaks the postconditions, inputs the tests never try. With complete postconditions it helps; with incomplete ones, which made up 90 of our 210 LLM-inferred sets, it hurts. The effect of postconditions is therefore a property of the reviewer and the postconditions together, and a pipeline built on one model should be tested with the model it will use.

## 6.3 One Sentence of Framing

The follow-up locates GPT-6.1 Sol's problem in one instruction, "Treat them as the specification". Without it, the same postconditions help GPT-6.1 Sol find more bugs than the method alone. Telling it that the postconditions may be incomplete or wrong also removes the loss, but with correct postconditions it gave back part of their benefit. In our study, neutral framing did not significantly lower any reviewer's localization or raise its false alarms. We therefore recommend giving a reviewer postconditions as information, not with an instruction to treat them as the specification.

## 6.4 Are All False Alarms False?

We count every warning on an original as a false alarm, because the originals pass their tests. Tests, however, do not try every input. Our execution check shows that most of GPT-6.1 Sol's warnings we could check are correct (19 of 23): on the input they name, the code breaks its reference postconditions. Some point at fragile code, others at postconditions that are stricter than the code's intended behavior. Our false-alarm rates therefore likely overstate GPT-6.1 Sol's errors under the reference postconditions. Benchmarks that use test-validated specifications as ground truth should expect a careful reviewer to find cases the tests miss.

## 6.5 Implications

*For tool builders.* Give the reviewer postconditions when good ones are available, as information rather than as an instruction to follow. Test the reviewer on incomplete postconditions before relying on it, and check whether a sudden drop in warnings after postconditions are added reflects silent over-trust rather than correct code.

*For specification inference.* Inferred postconditions are usually scored by correctness and completeness. Our results show that an incomplete postcondition is not merely weaker: given to a reviewer with the wrong instruction, it can suppress warnings the reviewer would otherwise give. Evaluating inferred postconditions by their effect on the tools that use them would complement the existing measures.

# 7 Conclusion

We measured how postconditions change three LLM code reviewers on 210 Java methods from PostcondBench. Reference postconditions cut false alarms by more than half for Claude Opus 5.5 and Gemini 4 Argon, which the whole source file did not. LLM-inferred postconditions, however, can mislead: told to treat them as the specification, GPT-6.1 Sol shows silent over-trust and misses most of the bugs they do not cover. A follow-up experiment traces this to the instruction itself; presented neutrally, the same postconditions help GPT-6.1 Sol find more bugs than it does without them. Some of what the benchmark counts as false alarms are real: in 19 of the 23 cases we could check, the code does break its postconditions. Postconditions are therefore neither safe nor harmful in themselves: their effect depends on the reviewer and on how they are presented. We recommend giving reviewers postconditions as information rather than as instructions, and testing each reviewer on incomplete postconditions before relying on it. Extending the study to the Python tasks of PostcondBench is a natural next step.

# Limitations

*Ground truth.* Following the benchmark, an original is bug-free because it passes its tests, and a mutant has one bug at the lines it changes. As PostcondBench's authors note, high coverage "does not necessarily imply high test quality" [17], and our execution check confirms that the tests do not cover every input. The check covers only the 28 warnings we could run without building the project, which favors simple methods (§5.5). Absolute false-alarm rates are therefore relative to the benchmark's labels. Every condition reviews the same methods, so the comparisons between conditions are less affected. One exception: under reference postconditions, some of GPT-6.1 Sol's false alarms are real (§5.5), so its false-alarm rate in that condition is overstated more than in others. The mutants are synthetic and need not resemble the bugs developers make.

*Scope.* We study 210 Java methods, reviewed one method at a time, whereas real reviews concern diffs across methods, files and languages.

*Postconditions.* The reference postconditions were written with the mutants in view, so they show the best case, not a typical one. The LLM-inferred postconditions come from a single model, sample and prompt setting. That model, Claude Sonnet 4.5, is from the same vendor as one of our reviewers, Claude Opus 5.5. Claude Opus 5.5 does not benefit more from these postconditions than Gemini 4 Argon does (6.2 and 4.8 percentage points fewer false alarms; neither drop is significant). Each framing uses a single wording.

*Reviewers and prompt.* We evaluate three proprietary models with one prompt at their default settings, through APIs whose models may change. The methods are public and may appear in the models' training data; renaming the parameters of the 516 mutants whose methods take parameters changes method-only localization by at most 0.4 pp, but cannot rule out memorization of other code.

*Scoring.* Localization checks which line a warning points at, not whether its explanation is right, so it misses a correct warning that quotes a nearby line. Compared with the LLM judges (Appendix C), it therefore undercounts bug finding: almost all disagreements are bugs a judge credits and the rule does not. Neither measure is a human verdict.

*Analysis.* The follow-up uses the same 210 methods, so it is not a replication on new methods. The analyses in §5.5 are exploratory, and their count of warnings that cite the postconditions is approximate (keyword match on the explanation).

# Data Availability

We release everything needed to reproduce our numbers: the item-building script with the PostcondBench version and seed it uses, the prompts, the raw reviews of all three reviewers including the repeat runs, the scoring protocol and its implementation, the scripts that produce every table, figure and number in this paper, the LLM-judge comparison, the parameter-renaming check, the sensitivity tables, and the harnesses and results of the execution check. The artifact is available at https://doi.org/10.5281/zenodo.XXXXXXX (to be added).

# 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.

# References

[1] Umut Cihan, Vahid Haratian, Arda İçöz, Mert Kaan Gül, Ömercan Devran, Emircan Furkan Bayendur, Baykal Mehmet Uçar, and Eray Tüzün. 2025. Automated Code Review in Practice. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 425–436. doi:10.1109/ICSE-SEIP66354.2025.00043 arXiv:2412.18531.

[2] 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 (2024), 1889–1912. doi:10.1145/3660791

[3] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming* 69, 1–3 (2007), 35–45.

[4] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[5] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. 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 (FSE)*. ACM, 654–665.

[6] Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. 2024. Do LLMs generate test oracles that capture the actual or the expected program behaviour? arXiv:2410.21136.

[7] Shuvendu K. Lahiri. 2024. Evaluating LLM-driven User-Intent Formalization for Verification-Aware Languages. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*. 142–147. doi:10.34727/2024/isbn.978-3-85448-065-5_19

[8] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *ACM SIGSOFT Software Engineering Notes* 31, 3 (2006), 1–38.

[9] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 1035–1047. doi:10.1145/3540250.3549081

[10] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 16–28. doi:10.1109/ICSE55347.2025.00129

[11] Mrinank Sharma, Meg Tong, Tomek Korbak, David Duvenaud, Amanda Askell, Sam Bowman, Esin Durmus, Zac Hatfield-Dodds, Scott Johnston, Shauna Kravec, Timothy Maxwell, Sam McCandlish, Kamal Ndousse, Oliver Rausch, Nicholas Schiefer, Da Yan, Miranda Zhang, and Ethan Perez. 2024. Towards Understanding Sycophancy in Language Models. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*.

[12] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. 2023. Large Language Models Can Be Easily Distracted by Irrelevant Context. In *Proceedings of the 40th International Conference on Machine Learning (ICML) (PMLR, Vol. 202)*. 31210–31227.

[13] Scott Thornton. 2026. Can Adversarial Code Comments Fool AI Security Reviewers – Large-Scale Empirical Study of Comment-Based Attacks and Defenses Against LLM Code Analysis. arXiv:2602.16741.

[14] Manushree Vijayvergiya, Małgorzata Salawa, Ivan Budiselić, Dan Zheng, Pascal Lamblin, Marko Ivanković, Juanjo Carin, Mateusz Lewko, Jovan Andonov, Goran Petrović, Daniel Tarlow, Petros Maniatis, and René Just. 2024. AI-Assisted Assessment of Coding Practices in Modern Code Review. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware)*. ACM, 85–93. doi:10.1145/3664646.3665664

[15] Antonio Vitale, Emanuela Guglielmi, Simone Scalabrino, and Rocco Oliveto. 2026. On the Impact of Code Comments for Automated Bug-Fixing: An Empirical Study. In *Proceedings of the 34th IEEE/ACM International Conference on Program Comprehension (ICPC)*. doi:10.1145/3794763.3794794

[16] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting Program Specification Synthesis by Large Language Models using Static Analysis and Program Verification. In *Computer Aided Verification (CAV 2024) (Lecture Notes in Computer Science, Vol. 14682)*. Springer, 302–328.

[17] Gehao Zhang and Juan Zhai. 2026. POSTCONDBENCH: Benchmarking Correctness and Completeness in Formal Postcondition Inference. In *Findings of the Association for Computational Linguistics: ACL 2026*. San Diego, CA, USA, 35478–35507.

[18] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems 36 (NeurIPS), Datasets and Benchmarks Track*.

# A Prompts

All reviewers receive the system prompt in §3.3. The user prompt is built from the templates below, where `{method}` is the method name, `{code}` the method, `{file}` the enclosing source file with the method in place, and `{postconds}` the JML postconditions, one `//@ ensures` clause per line. Code is enclosed in Markdown code fences.

*Method only (also the start of every postcondition prompt).*

````
Method under review:
```java
{code}
```
````

*Whole file.*

````
Review ONLY the method `{method}`. The rest of its source file is
      shown as context.
Source file:
```java
{file}
```
Method under review:
```java
{code}
```
````

*Conditions with postconditions.* The method-only prompt, followed by one of these blocks. The authoritative framing uses the first, the hedged framing the second and the neutral framing the third.

````
The intended behavior of this method is specified by these JML
      postconditions. Treat them as the specification:
```
{postconds}
```
````

````
These JML postconditions were generated automatically and may be
      incomplete or wrong. Use them as a hint about the intended
      behavior, not as ground truth:
```
{postconds}
```
````

````
JML postconditions for this method:
```
{postconds}
```
````

# B Full Test Results

Table 5 lists every primary test and Table 6 the secondary detection tests. Excluding replies that are not valid JSON (§3.3) changes no Holm-significant result; the sensitivity tables are in the artifact.

# C Comparison with LLM Judges

For Claude Opus 5.5 and GPT-6.1 Sol, we also scored bug finding with LLM judges. Each review was judged by the other reviewer model as primary judge (GPT-6.1 Sol judged Claude Opus 5.5's reviews and Claude Opus 5.5 judged GPT-6.1 Sol's) and by the reviewer model itself as second judge, used only to measure agreement. On a mutant, the judge decided whether some warning matched the injected bug, given the diff and the warnings. We tested six comparisons per reviewer: recall and the rate of flagged originals, each for the whole file, the reference postconditions and the authoritative LLM-inferred postconditions against method only, with Holm correction over the six. The flag rate is the same quantity as our false-alarm rate.

Table 7 gives the results. The judges agreed with each other on mutants (Cohen's $\kappa = 0.89$ on Claude Opus 5.5's reviews and 0.95 on GPT-6.1 Sol's), but only weakly on whether a flag on an original was a false alarm ($\kappa = 0.20$ for both), which is why we do not use judge verdicts on originals. The line rule and the primary judge agree on 93.4% (Claude Opus 5.5) and 93.0% (GPT-6.1 Sol) of the 3,150 mutant reviews per reviewer. Judge recall is higher than our localization (for example 98.1% against 91.7% for Claude Opus 5.5 without postconditions). Nearly all disagreements between the two measures (207 of 207 for Claude Opus 5.5, 218 of 219 for GPT-6.1 Sol) are mutants that the judge credits and the line rule does not,

**Table 5: All primary tests (Holm family of eight per reviewer). Δ: paired difference in percentage points with method-bootstrap 95% CI; +/−: versions (localization) or methods (false alarm) where the outcome went from no to yes / yes to no; $p$: method-clustered sign-flip test.**

| Comparison | Outcome | Δ | 95% CI | +/− | $p$ | Holm $p$ |
|---|---|---|---|---|---|---|
| *Claude Opus 5.5* | | | | | | |
| whole file vs. method only | localization | +1.0 | [+0.0, +1.9] | 8/2 | .109 | .547 |
| whole file vs. method only | false alarm | −1.9 | [−5.7, +2.4] | 9/13 | .524 | 1.000 |
| reference vs. method only | localization | +1.3 | [+0.2, +2.5] | 10/2 | .056 | .334 |
| reference vs. method only | false alarm | −10.5 | [−15.2, −5.7] | 3/25 | <.001 | <.001 |
| LLM-inferred (auth.) vs. method only | localization | +0.3 | [−1.0, +1.6] | 9/7 | .807 | 1.000 |
| LLM-inferred (auth.) vs. method only | false alarm | −6.2 | [−11.0, −1.4] | 8/21 | .024 | .170 |
| LLM-inferred (hedged) vs. LLM-inferred (auth.) | localization | +0.6 | [−0.2, +1.6] | 6/2 | .289 | 1.000 |
| LLM-inferred (hedged) vs. LLM-inferred (auth.) | false alarm | +1.0 | [−3.3, +5.2] | 12/10 | .832 | 1.000 |
| *Gemini 4 Argon* | | | | | | |
| whole file vs. method only | localization | +1.1 | [−0.2, +2.5] | 13/6 | .167 | .668 |
| whole file vs. method only | false alarm | +1.9 | [−3.8, +7.6] | 21/17 | .626 | .846 |
| reference vs. method only | localization | +3.5 | [+1.7, +5.4] | 26/4 | <.001 | .002 |
| reference vs. method only | false alarm | −11.0 | [−16.2, −5.2] | 7/30 | <.001 | .002 |
| LLM-inferred (auth.) vs. method only | localization | +1.7 | [+0.2, +3.3] | 18/7 | .043 | .259 |
| LLM-inferred (auth.) vs. method only | false alarm | −4.8 | [−9.5, +0.0] | 9/19 | .087 | .436 |
| LLM-inferred (hedged) vs. LLM-inferred (auth.) | localization | −0.8 | [−2.1, +0.5] | 5/10 | .337 | .846 |
| LLM-inferred (hedged) vs. LLM-inferred (auth.) | false alarm | +3.3 | [−1.9, +8.6] | 19/12 | .282 | .846 |
| *GPT-6.1 Sol* | | | | | | |
| whole file vs. method only | localization | +4.4 | [+2.5, +6.5] | 31/3 | <.001 | <.001 |
| whole file vs. method only | false alarm | +7.1 | [+1.4, +12.4] | 25/10 | .017 | .066 |
| reference vs. method only | localization | +5.9 | [+3.7, +8.3] | 39/2 | <.001 | <.001 |
| reference vs. method only | false alarm | +7.1 | [−1.0, +15.2] | 45/30 | .107 | .320 |
| LLM-inferred (auth.) vs. method only | localization | −16.0 | [−21.0, −11.6] | 23/124 | <.001 | <.001 |
| LLM-inferred (auth.) vs. method only | false alarm | +0.5 | [−7.1, +8.6] | 37/36 | 1.000 | 1.000 |
| LLM-inferred (hedged) vs. LLM-inferred (auth.) | localization | +17.9 | [+13.8, +22.4] | 126/13 | <.001 | <.001 |
| LLM-inferred (hedged) vs. LLM-inferred (auth.) | false alarm | −1.0 | [−9.0, +7.1] | 34/36 | .905 | 1.000 |

**Table 6: Secondary tests: detection (any warning on a mutant), Δ in percentage points, versions gained/lost, and sign-flip $p$. No multiplicity adjustment.**

| | Claude Opus 5.5 | | | Gemini 4 Argon | | | GPT-6.1 Sol | | |
|---|---|---|---|---|---|---|---|---|---|
| Comparison | Δ | +/− | $p$ | Δ | +/− | $p$ | Δ | +/− | $p$ |
| whole file vs. method only | +1.0 | 6/0 | .062 | +1.7 | 14/3 | .018 | +3.5 | 23/1 | <.001 |
| reference vs. method only | +0.8 | 5/0 | .062 | +2.5 | 16/0 | <.001 | +4.0 | 25/0 | <.001 |
| LLM-inferred (auth.) vs. method only | −0.2 | 3/4 | 1.000 | +0.5 | 9/6 | .581 | −15.7 | 19/118 | <.001 |
| LLM-inferred (hedged) vs. method only | +0.3 | 3/1 | .625 | +0.6 | 9/5 | .424 | +1.6 | 11/1 | .011 |
| LLM-inferred (hedged) vs. LLM-inferred (auth.) | +0.5 | 3/0 | .250 | +0.2 | 4/3 | 1.000 | +17.3 | 119/10 | <.001 |

typically a warning that describes the bug while quoting a nearby line.

Every comparison has the same direction under both analyses, and the large effects agree: Claude Opus 5.5's drop in false alarms with reference postconditions, and GPT-6.1 Sol's loss with authoritative LLM-inferred postconditions (−19.0 pp in judge recall, −16.0 pp in localization). Three small effects differ in significance. Claude Opus 5.5's gains from the whole file (+1.7 pp) and from reference postconditions (+1.6 pp) are significant under the judges but not in localization (+1.0 and +1.3 pp), and GPT-6.1 Sol's increase in flags with the whole file reaches Holm $p = .050$ in the six-test family but not in our eight-test family ($p = .066$). For the exploratory hedged condition, judge recall for GPT-6.1 Sol rises from 74.8% to 95.7%.

**Table 7: Judge-based analysis (Claude Opus 5.5 and GPT-6.1 Sol only). Recall: share of mutants for which the primary LLM judge accepted a warning as matching the injected bug. Flags: share of originals with any warning (identical to our false-alarm rate). Δ vs. method only in percentage points; Holm over the six tests per reviewer.**

| | Recall | | | Flags | | |
|---|---|---|---|---|---|---|
| Condition | Rate | Δ | Holm $p$ | Rate | Δ | Holm $p$ |
| *Claude Opus 5.5* | | | | | | |
| Method only | 98.1 | | | 15.7 | | |
| Whole file | 99.8 | +1.7 | .010 | 13.8 | −1.9 | 1.000 |
| Reference | 99.7 | +1.6 | .016 | 5.2 | −10.5 | <.001 |
| LLM-inferred (auth.) | 98.4 | +0.3 | 1.000 | 9.5 | −6.2 | .073 |
| *GPT-6.1 Sol* | | | | | | |
| Method only | 93.8 | | | 29.0 | | |
| Whole file | 98.6 | +4.8 | <.001 | 36.2 | +7.1 | .050 |
| Reference | 99.5 | +5.7 | <.001 | 36.2 | +7.1 | .213 |
| LLM-inferred (auth.) | 74.8 | −19.0 | <.001 | 29.5 | +0.5 | 1.000 |