Subjectivity Always Worse? Evaluating Agent-as-a-Judge, LLM-as-a-Judge, and Unit Tests on Coding Tasks

Lin Shi
ls2282@cornell.edu
Cornell Tech
New York, NY, USA

Yan Xiao
yx689@cornell.edu
Cornell Tech
New York, NY, USA

Tongjia Rao
tr426@cornell.edu
Cornell Tech
New York, NY, USA

Abstract

Large language models (LLMs) and code agents are now widely used to *generate* programs, but in most benchmarks they are still *evaluated* through human-written unit tests. Unit tests are reliable but costly to design, and they mostly reflect a single view of quality: whether a solution passes a fixed test suite. In this work, we ask whether LLM and agent judges can act as flexible, low-friction alternatives or complements to unit tests when grading competitive-programming solutions. We study this question on LiveCodeBench, a dynamic benchmark of Python coding problems with hidden unit tests collected from LeetCode, AtCoder, and Codeforces. For each task, we generate a solution and then evaluate it using six settings: the official unit tests, two text-only LLM judges (correctness-only and multi-aspect), two agent judges with tool use (correctness-only and multi-aspect), and one agent that is explicitly prompted to write and run its own tests. We compare these settings in terms of **numerical correctness alignment**, **binary agreement** on pass/-fail decisions, and **multi-aspect scores** covering style, simplicity, and robustness, and we also log agent behaviour and evaluation cost. Our experiments show that LLM and agent judges broadly track unit-test correctness and achieve similar levels of agreement, but they exhibit distinct biases: non-test-writing judges tend to be slightly conservative, while the test-writing agent often becomes optimistic and over-trusts its own tests. Multi-aspect judges surface style and robustness issues that tests alone cannot see, and there are clear case studies where they either warn about likely performance problems beyond the tests or miss simple specification errors that tests catch. We conclude that AI judges can serve as **practical, multi-perspective alternatives to unit tests** when tests are limited, but they work best when used together with unit tests rather than as a full replacement.

ACM Reference Format:

Lin Shi, Yan Xiao, and Tongjia Rao. 2025. Subjectivity Always Worse? Evaluating Agent-as-a-Judge, LLM-as-a-Judge, and Unit Tests on Coding Tasks. In *Proceedings of Deep Learning Final Project (CS5787).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

CS5787, Cornell Tech
2025. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction and Problem Statement

As code generation with Large Language Models (LLMs) and coding agents becomes increasingly accessible and scalable, verifying the correctness of generated code and maintaining evaluation pipelines has emerged as a key challenge. For programming tasks with well-defined behavior, deterministically verifiable unit tests have long been the reliable gold standard for assessing correctness. However, curating high-quality, comprehensive unit tests is both labor-intensive and difficult, especially at the scale of modern code-generation workloads. This raises a natural question: **can automated AI evaluators serve as reliable, cost-effective substitutes for human-authored unit tests in evaluating code solutions?**

LLM-as-a-Judge [7] has recently emerged as a computationally efficient alternative to human evaluators across various tasks such as coding, writing, and instruction following. In this paradigm, an LLM inspects the problem description and the candidate solution, and then produces an assessment without running the code. More recently, agentic systems have become increasingly capable of reasoning and tool use to manage more complex tasks, motivating researchers to explore Agent-as-a-Judge [6] as a successor to pure LLM judges, often achieving better evaluation outcomes while introducing new challenges. In 2025, coding and terminal agents (e.g., Claude Code, Codex, Gemini CLI) are emerging as state-of-the-art tools for coding tasks, with access to terminal environments that allow them to write files, execute code, and interpret outputs. Studies have shown that with extensive tool access, agents are better selectors of debugging patches than pure LLMs [4].

Beyond reproducing the objective notion of correctness encoded by unit tests, AI evaluators also introduce *subjectivity*: they can comment on style, clarity, robustness to edge cases, or maintainability from a human-centric perspective. This leads to a two-part question that guides our work. First, **can LLM and agent judges faithfully preserve the correctness signal of human-authored unit tests, so that they are usable as objective evaluators when high-quality tests are unavailable?** Second, **conditional on this, does their additional subjectivity become an advantage by providing richer evaluation lens that go beyond pass/fail correctness without sacrificing reliability?** If both conditions hold, automated AI judges would not only reduce the cost of constructing unit tests, but also offer more comprehensive and insightful assessments of generated code.

To investigate these questions, we start from a setting where objective ground truth is available. We select 70 LeetCode-style functional coding tasks from LiveCodeBench [3], each equipped with human-annotated reference unit tests that define task-level correctness. On top of this dataset, we evaluate five judge configurations spanning two paradigms: two **LLM-as-a-Judge** settings (a correctness-only judge and a multi-aspect judge that also scores style, simplicity, and robustness) and three **Agent-as-a-Judge** settings (a correctness-only agent, a multi-aspect agent with terminal access, and an explicitly unit-test–writing agent that is prompted to construct and run its own tests). This design lets us first validate how well each AI judge approximates the outcomes of human-authored unit tests, and then examine what additional value their subjective criteria provide.

Our quantitative analysis, combining accuracy metrics, confusion matrices, and statistical tests, shows that both LLM and agent evaluators largely reflect the correctness evaluation of ground-truth unit tests: except for the explicitly unit-test–writing agents, all settings disagree with unit tests on roughly 20% of tasks under a binary all-pass vs. fail criterion, and true decisions consistently outnumber false ones. This suggests that AI evaluators can act as effective proxies when high-quality tests are missing, although non-trivial disagreement remains. We further observe that agents only rarely and spontaneously write unit tests when not instructed to do so, with this behavior more common when they focus solely on correctness than when they are asked to judge a wide spectrum of perspectives. When we directly prompt agents to write unit tests while also evaluating other dimensions, their correctness scores become the closest to human-authored unit tests, making this the most effective setting for faithfully preserving unit-test behavior while offering multi-perspective feedback—at the cost of higher time and monetary expenditure. At the same time, these unit-test–writing agents exhibit a tendency to assign higher correctness scores than the ground-truth tests, suggesting that their self-constructed tests are less comprehensive (e.g., weaker on edge cases), whereas other AI evaluator settings tend to be more conservative by underrating correctness based solely on code inspection. Complementary qualitative trajectory analysis, performed by humans with LLM assistance, further highlights how different judge configurations trade off evaluation fidelity, subjectivity, and computational efficiency.

In the remainder of this paper, we first review related work on LLM- and agent-based evaluation in Section 2. We then describe our dataset construction and task selection from LiveCodeBench in Section 3, followed by the judge configurations, prompting strategies, and evaluation metrics in Section 4. Section 5 presents our quantitative and qualitative experimental results, including statistical analyses and trajectory-level case studies. Finally, Section 6 summarizes our key findings and discusses implications and directions for future work.

2 Related Work

Our research builds upon two emerging evaluation paradigms that we apply to the professional coding domain: **LLM-as-a-Judge** and **Agent-as-a-Judge**.

Zheng et al. [7] demonstrated that large language models can effectively replace human evaluators across various tasks by leveraging their understanding of semantics and patterns to assess solution quality through static analysis alone. In the coding domain, LLM judges have been used to approximate human ratings of correctness, readability, and style without executing code, offering a scalable but inherently subjective alternative to unit-test–based evaluation. Parallel work on large code benchmarks such as HumanEval [2], MBPP [1], and LiveCodeBench [3] has shown that deterministically verifiable unit tests remain the dominant approach for measuring functional correctness of generated code, but constructing and maintaining such test suites is costly and often benchmark-specific.

As tool use and agents become popular and effective in a wide range of tasks, Agent-as-a-Judge [6] extends evaluation capabilities by equipping judges with tool-use abilities. In this setting, agents are granted access to execution environments where they can write files, run programs, and inspect runtime behavior before making decisions. Coding and terminal agents (e.g., Claude Code, Codex, Gemini CLI, OpenHands [5]) exemplify this trend, and recent work has shown that agents with extensive tool access are better selectors of debugging patches than pure LLMs [4]. This enhanced paradigm posits that agents with execution access can provide more nuanced and accurate evaluations by dynamically verifying code behavior, synthesizing test cases, and observing outputs rather than relying solely on code comprehension.

Our work connects these lines of research by directly comparing LLM and agent judges against human-authored unit tests on a shared set of LiveCodeBench tasks. Unlike prior work that relies solely on unit tests or solely on subjective AI judgments, we treat unit tests as ground-truth correctness and systematically study (1) how closely different LLM and agent configurations reproduce unit-test outcomes, and (2) how their additional, subjective dimensions of evaluation (e.g., style, simplicity, robustness) interact with objective correctness when agents are allowed or instructed to write, execute, and analyze their own tests.

3 Data

Our experiments are conducted on tasks derived from LiveCodeBench, a holistic and contamination-free benchmark for evaluating LLMs on code-related tasks [3]. LiveCodeBench continuously collects new problems from recent competitive programming contests on platforms such as LeetCode, AtCoder, and CodeForces, with precise release timestamps that support contamination-aware evaluation. Each problem is accompanied by a reference implementation and unit tests, and the benchmark covers multiple scenarios beyond code generation, including self-repair, code execution, and test output prediction.

In this work, we focus on the functional-level, LeetCode-style problems within LiveCodeBench that provide human-authored unit tests for Python solutions. From this pool, we sample 70 representative tasks that balance *task length*, *difficulty*, and *problem diversity*. Concretely, we select problems whose descriptions are neither trivial nor excessively long, span a range of difficulty levels (e.g., easy, medium, hard), and cover a diverse set of algorithmic patterns

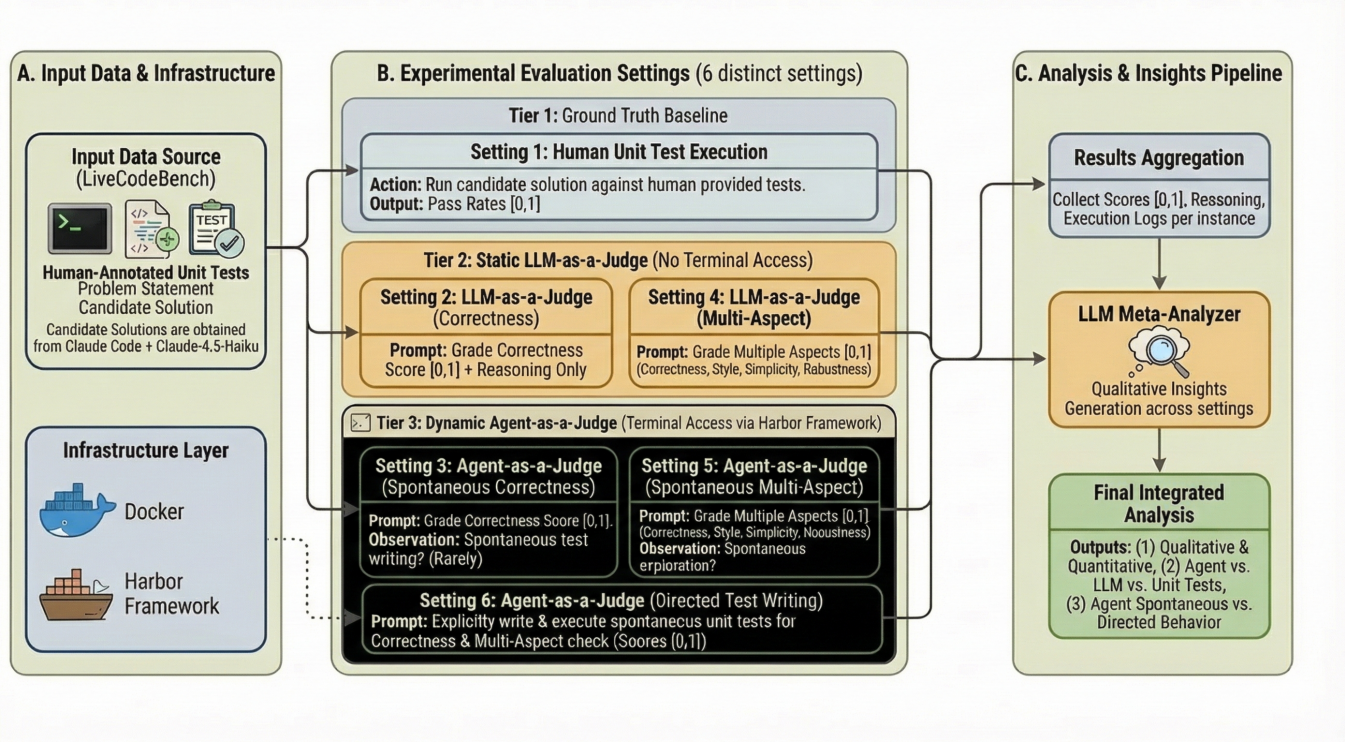

Figure 1: Overview of the pipeline architecture.

and data structures (such as arrays, strings, graphs, dynamic programming, and greedy methods). For each selected problem, we retain the natural language description, function signature, and the accompanying human-written unit tests, which we treat as the ground-truth definition of correctness.

To obtain candidate solutions for evaluation, we use Claude Code with Claude-4.5-Haiku as the underlying agent and model. Given each problem description and the required function signature, Claude Code is prompted to produce a complete Python solution, resulting in one *(problem, solution)* pair for each of the 70 tasks. These generated solutions, together with the reference unit tests from LiveCodeBench, form the core dataset on which we apply and compare our LLM and agent judges.

4 Methods

Figure 1 provides an overview of our evaluation pipeline. Starting from 70 functional coding tasks from LiveCodeBench (Section 3), we first generate a single candidate Python solution for each task using Claude Code with Claude-4.5-Haiku. The human-authored unit tests that accompany each task are then used to establish ground-truth correctness labels. On top of this shared set of (problem, solution) pairs, we apply six evaluation settings: one unit-test baseline, two LLM-based judges, and three agent-based judges. Finally, we conduct both quantitative and qualitative analyses to compare these AI evaluators against unit tests and to investigate how different judge configurations trade off fidelity, subjectivity, and cost.

4.1 Evaluation Paradigms

We consider six evaluation paradigms in total: human-authored unit tests (as an objective baseline), two *LLM-as-a-Judge* settings, and three *Agent-as-a-Judge* settings. All evaluators take as input the same problem description and candidate solution; agent-based judges additionally have access to a terminal environment via our agent stack (e.g., openhands + gpt-5-mini) that allows them to write files, execute code, and inspect outputs. Table 1 summarizes the methods, their inputs and outputs, and their intended behaviors.

4.2 Evaluation and Analysis Framework

We analyze the behavior of these evaluators using a combination of quantitative metrics and qualitative trajectory inspection. Our goal is twofold: (1) to measure how closely different AI judges reproduce the objective correctness signal from human-authored unit tests, and (2) to understand what additional, inherently subjective information they provide and at what computational cost.

4.2.1 Quantitative Analysis. Our quantitative analysis focuses on three main aspects: (i) correctness alignment between AI evaluators and unit tests, (ii) behavior of multi-aspect scores across LLM and agent judges, and (iii) cost efficiency of agent-based evaluators.

Method	Output	Description
Unit Tests (UT)	Pass rate in $[0, 1]$	Human-written unit tests in LiveCodeBench, served as ground-truth correctness score baseline.
LLM-Corr	Correctness score in $[0, 1]$ + reasoning	LLM-as-a-Judge configuration that performs static analysis of the candidate code and outputs a single correctness score, focusing only on functional correctness without execution or tool access.
LLM-Multi	Multi-aspect scores in $[0, 1]$ + reasoning	LLM-as-a-Judge configuration that outputs separate scores for correctness, style, simplicity, and robustness based solely on code inspection, without running the program.
Agent-Corr	Correctness score in $[0, 1]$ + agent trajectories	Agent-as-a-Judge configuration with identical instructions to LLM-Corr, but with access to a terminal environment that can write, execute, and inspect code.
Agent-Multi	Multi-aspect scores in $[0, 1]$ + agent trajectories	Agent-as-a-Judge configuration with identical instructions to LLM-Multi, augmented with access to a terminal environment that can write, execute, and inspect code.
Agent-UT	Multi-aspect scores in $[0, 1]$ + agent trajectories	Agent-as-a-Judge configuration that extends Agent-Multi with explicit prompts directing the agent to write unit tests, run them, and base its correctness assessment primarily on the resulting test outcomes.

Table 1: Introduction and comparison of six evaluation settings in this study. Detailed prompts to LLMs and Agents are described in Appendix B.

First, we examine correctness alignment under both a binary and a continuous setting. In the binary setting, we apply an identity-style criterion across all evaluators: a solution is considered *resolved* if and only if its score is exactly 1 (i.e., passes all human-authored unit tests or receives a perfect correctness score from an AI judge), and *unresolved* otherwise. This all-pass vs. fail view captures the practical scenario where one often only cares whether a solution is entirely correct or not. Using this binarization, we compute accuracy and confusion matrices for each evaluation setting, and directly compare how LLM and agent judges behave against the ground-truth unit tests.

Since both unit-test pass rates and AI-judge scores are continuous, we also analyze correctness alignment in the numerical setting. Here we compare the raw correctness scores produced by each judge with the unit-test pass rates using scatter plots and distribution summaries. This continuous view reveals more nuanced trends that are invisible under threshold-based binary decisions, such as systematic under- or over-estimation of correctness and the degree of spread around the unit-test baseline.

Second, we study the behavior of multi-aspect assessments. For the LLM-Multi, Agent-Multi, and Agent-UT settings, we compare the distributions of style, simplicity, and robustness scores to examine how LLM and agent judges align with each other on these subjective dimensions. We are particularly interested in whether terminal access changes how agents rate these aspects compared to purely static LLM judges, and whether explicitly prompting agents to write unit tests (Agent-UT) affects their multi-aspect outputs. By analyzing pairwise comparisons across these settings, we assess how much additional subjectivity the agent configurations introduce beyond correctness and how consistent these subjective evaluations are across judge types.

Third, we quantify the cost efficiency of agent-based evaluators. For each problem and each agent setting, we log the number of reasoning steps, the number of tool calls (e.g., terminal invocations), wall-clock time, and approximate API cost. We then compare these metrics across Agent-Corr, Agent-Multi, and Agent-UT to understand the overhead introduced by multi-aspect evaluation and explicit unit-test writing. This analysis allows us to characterize trade-offs between evaluation richness (more aspects, more grounded checking) and resource consumption, using descriptive statistics and non-parametric tests to assess whether observed differences are statistically meaningful.

4.2.2 Qualitative Trajectory Analysis. To complement the quantitative results, we perform qualitative analysis of agent trajectories, combining LLM-assisted summarization with human inspection. For a subset of representative tasks and judge configurations, we examine the full action sequences of the agents, including intermediate reasoning, tool invocations, code executions, and test construction. This trajectory-level view helps explain how particular correctness judgments and multi-aspect scores arise, and why they sometimes diverge from unit-test outcomes.

We focus on three types of qualitative phenomena. First, we study when and how agents *spontaneously* decide to write and run tests in the absence of explicit instructions, and how this behavior differs between correctness-only and multi-aspect prompts. This sheds light on the natural test-writing tendencies of agents and how prompt design modulates them. Second, we identify common failure modes, such as agents over-trusting shallow or poorly designed tests, missing important edge cases, misinterpreting error messages, or entering unproductive tool-use loops that do not improve evaluation quality. Third, we analyze illustrative cases where AI evaluators provide more comprehensive assessments than unit tests

(e.g., flagging style or robustness issues that are not captured by pass/fail criteria), as well as cases where agent-written tests are clearly less thorough than the human-authored suites, leading to overly optimistic correctness scores.

By aligning these qualitative observations with our quantitative findings, we obtain a more complete picture of how LLM and agent evaluators behave in practice: not only how often they agree or disagree with human-authored unit tests, but also how subjectivity, tool use, and prompt design shape the evaluations they ultimately produce.

5 Experiments

This section compares the six evaluation settings introduced in Section 4 on the 70 LiveCodeBench tasks. For each problem, we start from the same candidate Python solution generated by the base code model and obtain a ground-truth correctness score from the human-authored unit tests. We then collect scores from five automated evaluators: an LLM judge prompted only for correctness (LLM-Corr), a multi-aspect LLM judge (LLM-Multi), an agentic judge with terminal access focusing on correctness (Agent-Corr), an agentic multi-aspect judge (Agent-Multi), and an agent that is explicitly instructed to write and run unit tests (Agent-UT). Unless otherwise noted, unit tests are treated as the main reference for correctness, and we view AI judges as alternative ways to approximate those outcomes.

5.1 Correctness Alignment with Unit Tests

A key question in this work is how well LLM and agent judges can match the correctness decisions implied by unit tests. This matters especially when high-quality unit tests are missing or too expensive to write. If AI judges behave similarly to tests, they can act as cheaper and more flexible substitutes; if they systematically disagree, we need to understand where and why.

5.1.1 Numerical Correctness. We first treat correctness as a continuous score in $[0, 1]$. Figure 2 shows, for each evaluation setting, the distribution of per-task correctness scores as jittered scatter points, together with their means marked by purple diamonds.

The first finding is that **all AI judges roughly follow the overall shape of the unit-test correctness distribution**. In every setting, most tasks receive scores close to either 0 or 1, and there are relatively few mid-range scores. The scatter clouds for LLM-Corr, LLM-Multi, Agent-Corr, and Agent-Multi largely overlap with the unit-test column. In other words, these evaluators tend to call the same solutions "clearly correct" or "clearly wrong" as the human-written tests. This is especially true on simple problems such as AtCoder "Three Threes,"[1] where both tests and AI judges give a correctness score very close to 1.0 for a short, straightforward implementation.

The second finding is that **LLM-Corr, LLM-Multi, Agent-Corr, and Agent-Multi are slightly more conservative than unit tests**. Their mean correctness scores are lower than the unit-test mean, and many solutions that pass all unit tests receive scores in the 0.7–0.9 range rather than a perfect 1.0. When we read their explanations, they often mention missing edge cases or possible performance problems and then reduce the score a bit, even without a concrete failing example. For example, on algorithmic tasks where time complexity grows quickly with input size, these judges sometimes remark that the code "may struggle on the largest inputs" and respond with a slightly lower correctness score. In practice, this means these settings sometimes "under-score" solutions compared to what the official tests would suggest, but they may still be capturing real risks that the test suite does not cover.

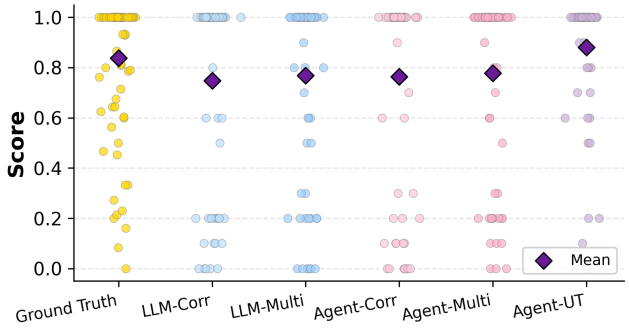

Figure 2: Distribution of correctness scores across all 70 tasks for unit tests and the five AI judge settings. Diamonds denote per-setting means.

The third finding is that **the test-writing agent (Agent-UT) is noticeably more optimistic than unit tests**. Its mean correctness score is higher than that of the ground-truth tests and many points are clustered near 1.0. Once Agent-UT writes a small set of tests and all of them pass, it often treats the solution as almost fully correct, even in cases where the official benchmark tests still fail on hidden corner cases. This suggests that the tests produced by Agent-UT are useful but less comprehensive than the curated LiveCodeBench test suites. On tasks with tricky logic, the agent tends to encode the main examples and a few variants, but it does not always explore rare or adversarial cases. As a result, its scores can be overconfident.

Looking slightly more closely at the scatter plot, we see that **large disagreements tend to cluster around "borderline" tasks and tasks with tricky specifications**. On easy problems such as "Three Threes," all evaluators agree and assign scores near 1.0. On harder problems, such as "Subsequence Pairs with Equal GCD,"[2] the unit tests sometimes give a score of 1.0 while the AI judges give much lower scores because they are worried about performance at the upper limits. There are also cases like "Wrong Answer,"[3] where the unit tests give a very low score due to a clear logic bug, but the AI judges give a high score because the code looks clean and matches the samples. These extreme points are rare, but they

[1] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/task_abc333_a__2hkhkhp__7x3ZUjB.html

[2] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/task_3608__sywfwab__4QQjp8T.html

[3] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/task_abc343_a__jgfl92c__xJJe2bT.html

highlight where numerical correctness scores differ the most and guide the qualitative analysis later in this section.

Overall, Figure 2 shows that AI judges can capture the broad pattern of unit-test correctness, but they come with different biases: **text-only and non-test-writing agents lean conservative, while the test-writing agent leans optimistic**. This tension between under-estimation and over-estimation is a recurring theme in the rest of our analysis.

5.1.2 Binary Agreement with Unit Tests. We next look at a strict pass–fail view and focus on agreement with the unit tests. A solution is counted as correct if and only if its correctness score equals 1.0; otherwise it is counted as incorrect. Unit tests are binarized in the same way by checking whether all tests pass. For each AI evaluator, we then compute how often its binary decision matches the unit-test decision (agreement), and how often it disagrees.

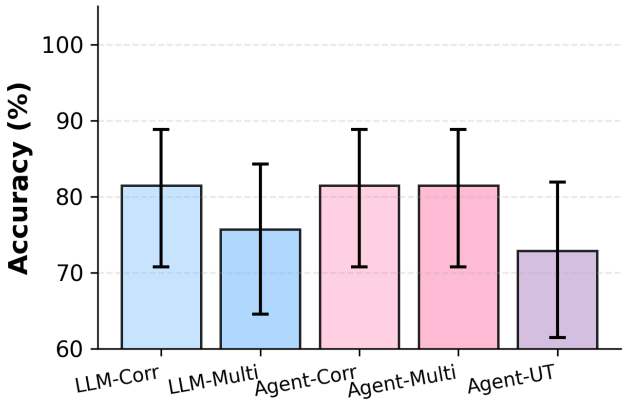

Figure 3: Binary agreement with unit tests for each AI evaluator. A solution is counted as correct only if its correctness score equals 1.0. Error bars show 95% bootstrap confidence intervals.

Figure 3 shows that **all five AI evaluators reach a similar level of agreement with the unit tests, roughly in the 72–82% range**. This holds even though their numerical scores show different conservative or optimistic tendencies. Correctness-only and agent-based judges sit near the top of this range, while LLM-Multi and Agent-UT are slightly lower, but none of the differences are dramatic. From a high-level perspective, a user who cares only about whether a solution passes or fails would see broadly comparable agreement rates across all five AI settings.

The confusion-style plot in Figure 4 helps explain what happens in the remaining 20–30% of tasks where AI judges and unit tests disagree. For LLM-Corr, LLM-Multi, Agent-Corr, and Agent-Multi, the disagreements are mostly **false negatives**: the judge labels a solution as not fully correct even though the unit tests pass. This is the binary reflection of the conservative scoring behaviour we saw earlier. Whenever a judge is unsure about edge cases or scalability, it tends to withhold a perfect score and thus disagrees with unit tests in the direction of being harsh. For some applications,

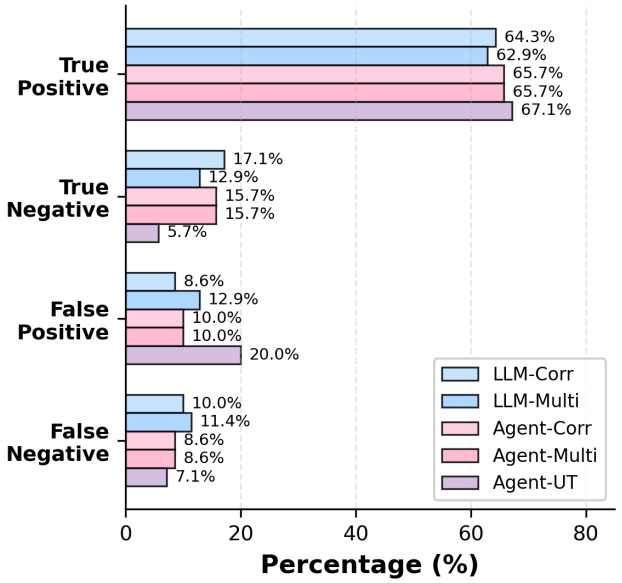

Figure 4: Confusion-style breakdown of binary decisions, expressed as a percentage of tasks for each evaluator. Each group of bars sums to 100%.

such as grading, these harsh decisions might mean that a student loses points despite passing all tests; for safety-critical settings, this conservative bias may actually be desirable.

Agent-UT has a different profile, with many more **false positives**. It sometimes labels a solution as fully correct after its own tests pass, while the official unit tests still fail. In these cases, the disagreement arises because the agent's self-constructed test suite misses certain patterns that the human test writers thought were important. A typical example is a corner case or a hidden constraint that does not appear in the samples. The unit tests capture this case and fail the solution, while Agent-UT's tests do not. In other words, Agent-UT agrees with its own tests but not necessarily with the stronger benchmark tests.

Across all settings, the total share of disagreements is around one fifth of the tasks, especially for the non-UT configurations. Manual inspection suggests that these disagreements fall into a few main categories. Some tasks have hard-to-see edge cases, such as off-by-one errors in interval problems, where AI judges sometimes spot or miss details that the tests do not. Other tasks have heavy algorithmic requirements, such as "Subsequence Pairs with Equal GCD," where judges consider worst-case complexity while tests mainly cover small and medium inputs. Finally, some tasks have short but subtle specifications, such as "Wrong Answer," where the key phrase "not equal to" is easy to miss. **The roughly 20% disagreement region is exactly where the problem is hard or the test suite is incomplete, and it is also the region where combining AI judgments with tests is most informative.**

In short, the binary view confirms the story from numerical scores. **Agreement with unit tests is fairly high for all settings, but**

the type of disagreement differs. Non-test-writing judges tend to disagree by being cautious and under-calling correctness, while the test-writing agent tends to disagree by over-calling correctness when its own tests pass but the benchmark tests do not.

5.2 Spontaneous and Directed Agent Behavior

The agent-based judges run inside the openhands framework and can open files, run Python programs, and use a terminal. This allows them to look at code behaviour in a way that pure LLM judges cannot. We have validated that AI evaluators can largely preserve human-written unit tests with minor disagreement. A natural question then is how often they choose to write tests on their own under different prompts, and what changes when we directly instruct them to write tests.

Table 2 summarizes the mean number of reasoning steps per trajectory and how often each agent configuration writes and runs tests without being asked to do so.

Agent Settings	Number of Steps	Spontaneous Unit Tests (%)
Agent-Corr	13.4 ± 18.1	7.1%
Agent-Multi	10.4 ± 5.3	2.9%
Agent-UT	29.7 ± 50.2	100.0%

Table 2: Agent behavior: mean step counts and frequency of spontaneous test-writing (the agent chooses to write and run tests without being instructed to do so).

The first finding is that **spontaneous test writing exists but is rare**. Agent-Corr writes tests on only about 7% of tasks, and Agent-Multi on fewer than 3%. Most of the time, these agents judge correctness from the code alone. We guess that this spontaneous behavior is a result and evidence of agent's reasoning capabilities. When they do choose to write tests, they usually do so because they are uncertain about boundary cases or about how the code behaves on atypical inputs. In Codeforces problem "1D Eraser,"[4] for example, Agent-Multi explicitly comments that the interval logic for removing black cells is tricky. It opens a new script, writes a handful of oracle test cases against brute-force solution, runs them, and then bases its final correctness score on whether they pass. Similar behaviour appears on LeetCode "Minimum Average of Smallest and Largest Elements,"[5] AtCoder "Count Bracket Sequences,"[6] "Minimize Abs 2,"[7] and "Strictly Increasing?,"[8] where the agent briefly steps into a test-writing loop when it feels that pure inspection is not enough.

In these spontaneous cases, the agents almost always create small and friendly test suites. The tests often mirror the examples in the problem statement, plus a few simple variations. Very few of them probe extreme values, long arrays, or carefully constructed corner cases. As a result, **spontaneous tests tend to increase the agent's confidence, and often increase its scores, but they do not fully match the coverage of the official unit tests**. This helps explain why Agent-Corr and Agent-Multi can be slightly optimistic on the specific runs where they choose to write tests, while still being conservative on average.

The second finding is that **explicitly asking the agent to write tests completely changes its behaviour**. In Agent-UT, the prompt tells the agent to construct and run tests as part of the evaluation. This leads to unit-test construction on every task and roughly doubles the average number of steps and tool calls compared to Agent-Corr and Agent-Multi. Some tasks are still handled quickly, with a single short test suite, while others lead to long sequences where the agent writes tests, runs them, observes failures, and modifies either the tests or its expectations before running again. Even though the tests generated by Agent-UT are not as strong as the benchmark test suites, the act of writing and running them makes its correctness scores numerically closer to the unit-test scores overall, and its disagreements with unit tests tend to have clear reasons grounded in the tests it wrote.

A third, more subtle, observation is that **the prompt design affects how much "energy" the agent spends on testing**. When the agent is prompted only for correctness (Agent-Corr), it writes tests more frequently than when it is also asked to rate style, simplicity, and robustness (Agent-Multi), even though both have the same tool access. This aligns with the raw percentages in Table 2. When we ask the agent to think only about correctness, it has more capacity to question its own judgment and decide to run code. When we ask it to think about many aspects at once, it seems less likely to invest extra steps in generating tests and instead leans more on static reasoning. This suggests that prompts that narrow the agent's focus might encourage more active testing, while prompts that broaden its focus may encourage more holistic but less test-heavy evaluation.

Taken together, this section shows that agents rarely decide to test code on their own, and that when they do, they tend to write small, human-like sanity tests that can over-trust a solution. When test-writing is required by the prompt, agents invest much more effort, their scores align better with formal unit tests, and the costs increase accordingly. **Prompting agents to write tests appears to be the most faithful way to imitate unit-testing behaviour, but it is also the most expensive.**

5.3 Multi-Aspect Subjective Evaluation

In many real applications, we care not only about whether code is correct, but also about how it is written and how robust it is. The multi-aspect LLM and agent judges (LLM-Multi, Agent-Multi, and Agent-UT) attach scores for correctness, style, simplicity, and robustness. Figure 5 summarizes these aspect scores across all tasks, with unit-test correctness shown for comparison.

[4] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/1873_d__ewd5mxi__x9Nv7BA.html
[5] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/3194__xqwbn3k__SSRJCHv.html
[6] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/abc312_d__qe6grrc__EjN7MmB.html
[7] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/abc330_c__vzsqbdi__bg8J3YL.html
[8] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/abc395_a__yyzhzs8__QA3cf4p.html

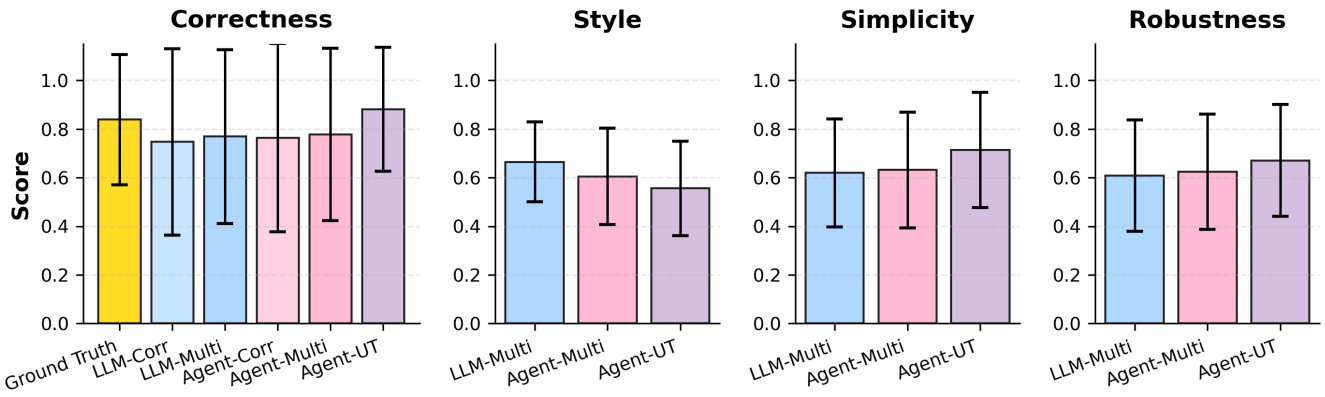

Figure 5: Multi-aspect evaluation across judge configurations. The left panel compares correctness scores to unit tests. The remaining panels show style, simplicity, and robustness scores for the multi-aspect LLM and agent judges. Bars denote means; error bars indicate standard deviations across tasks.

Looking at the left panel, we again see that **multi-aspect LLM and agent judges sit slightly below the unit-test baseline on correctness, while Agent-UT tends to give somewhat higher scores**. This repeats the conservative-versus-optimistic pattern we saw earlier, now in the context of multi-aspect prompts. The differences in mean correctness between LLM-Multi and Agent-Multi are small, which suggests that simply giving the agent tool access does not radically change how it rates correctness in aggregate. The main change comes when we shift from non-test-writing settings to the test-writing Agent-UT.

On style and simplicity, the scores are moderate and fairly similar across settings. LLM-Multi is slightly more generous about style, often praising descriptive names and clear structure. Agent-Multi and Agent-UT are slightly more positive about simplicity, especially when the solution uses direct algorithms and avoids unnecessary complexity. The overall impression is that **AI judges have a reasonably consistent sense of what "clean code" looks like, and this sense does not depend strongly on whether the judge has tool access**. It is also encouraging that solutions with high correctness scores tend, on average, to have higher style and simplicity scores, which suggests that good solutions are usually written in a reasonably clean way.

Robustness scores tell a different story. Across all three multi-aspect settings, robustness scores are lower and more spread out. Judges frequently mention missing input checks, fragile handling of corner cases, or possible performance problems at large inputs, even when unit tests pass. **All multi-aspect judges are stricter on robustness than on style or simplicity**, and they sometimes give a moderately robust solution a noticeably lower robustness score because it lacks explicit error handling or because its runtime may degrade badly in worst-case scenarios. This again reflects the tendency of AI judges to reason beyond the exact behaviour exercised by the tests.

The task-level examples highlight these patterns. In AtCoder "Three Threes,"[9] the program must print the digit N exactly N times for $1 \le N \le 9$. The solution is a two-line implementation with virtually no edge cases. The unit tests report a correctness score of 1.0. Both LLM-Multi and Agent-Multi give a correctness score of 1.0 and high style, simplicity, and robustness scores, with only minor comments about missing comments. This is a case where everything aligns, and it supports the view that easy problems are handled well by all evaluation methods.

The LeetCode problem "Subsequence Pairs with Equal GCD"[10] presents a more interesting case. The candidate solution passes all 43 unit tests and thus has a unit-test score of 1.0. However, both LLM-Multi and Agent-Multi give much lower correctness and robustness scores. Their explanations point out that the algorithm's state space grows quickly with input size and that its worst-case time and memory usage is likely too large when n and the value range are at their maximum. Since the benchmark tests do not include these worst-case inputs, they do not expose this weakness. Here, **the AI judges are more cautious than the unit tests in a way that seems reasonable**, and their lower scores highlight a real concern about complexity that is not reflected in the test suite.

The opposite happens in AtCoder "Wrong Answer."[11] The problem asks for any digit between 0 and 9 that is not equal to $A + B$. The candidate solution instead prints $A+B$. The unit tests, which include a range of inputs, quickly reveal this bug and give a correctness score of 0.08. In contrast, both multi-aspect judges rate the solution as nearly perfect on correctness and robustness. They praise the code for being short and clear and for matching the example pattern, but they miss the key "not equal to" condition. This shows that **unit tests can catch simple logical mistakes that AI judges**

[9] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/task_abc333_a__2hkhkhp__7x3ZUjB.html
[10] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/task_3608__sywfwab__4QQjp8T.html
[11] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/task_abc343_a__jgfl92c__xJJe2bT.html

sometimes overlook, especially when the code looks clean and the error is in a small but crucial phrase of the problem statement.

For a simple end-to-end demonstration of our pipeline, consider AtCoder "Take ABC."[12] The task gives a string over the alphabet {A, B, C} and asks us to repeatedly remove the substring "ABC" from left to right until it no longer appears, then print the result. The candidate solution uses a standard stack-based idea: it scans the string, appends each character to a list, and whenever the last three characters form "ABC", it deletes them. The unit tests give a correctness score of 1.0. Both LLM-Multi and Agent-Multi explain this algorithm in their own words and also assign a perfect correctness score with high simplicity and fairly high robustness scores. Because the problem is easy to describe but still involves a real algorithmic idea, this task shows how the full evaluation pipeline works on a realistic but not too complex example.

Across these problems, the main lesson is that **multi-aspect LLM and agent judges provide extra views on code quality that unit tests alone do not give**. They are especially good at surfacing potential robustness and style issues, though they can still miss simple logical bugs. When they disagree with unit tests, sometimes they reveal gaps in the test suite (as in the GCD example), and sometimes they simply make mistakes (as in "Wrong Answer"). For practical use, this suggests that combining multi-aspect AI judgments with unit tests can give a richer and more balanced view of solution quality than either alone.

5.4 Task Complexity and Agent Expense

Finally, we look at how expensive the different agent settings are. For each evaluation run of Agent-Corr, Agent-Multi, and Agent-UT, we log the number of reasoning steps, the number of tool calls (mostly terminal runs), the wall-clock time, and an approximate API cost in USD based on the token usage of `gpt-5-mini`. Figures 6–9 show these metrics as box-and-scatter plots and include statistical tests (Kruskal–Wallis) that compare the three settings.

The plots for steps and tool calls show a clear separation. **Agent-Corr and Agent-Multi are relatively light-weight**, usually finishing an evaluation in around ten steps with only a few tool calls. Their distributions are fairly tight, which means their cost is predictable across tasks. **Agent-UT is much heavier**: it has a higher median step count, many more tool calls, and a long tail of tasks where the agent writes and runs tests multiple times. The statistical tests reported in the figures indicate that these differences are unlikely to be due to random noise, especially for steps and tool calls.

Agent-Corr and Agent-Multi look very similar to each other in these plots. Adding style, simplicity, and robustness commentary on top of correctness does not noticeably change how many steps or tool calls the agent uses. For our setup, whether the agent is asked to write tests is far more important than how many textual aspects it has to comment on. This supports the idea that multi-aspect judgment is "cheap" relative to test-writing: generating more text is inexpensive compared to running code many times.

[12] https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/abc328_d__phkhhz8__c2UPmVz.html

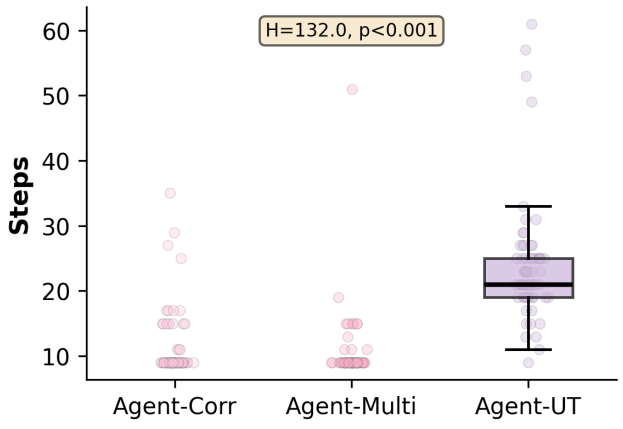

Figure 6: Number of reasoning steps per task for the three agent settings.

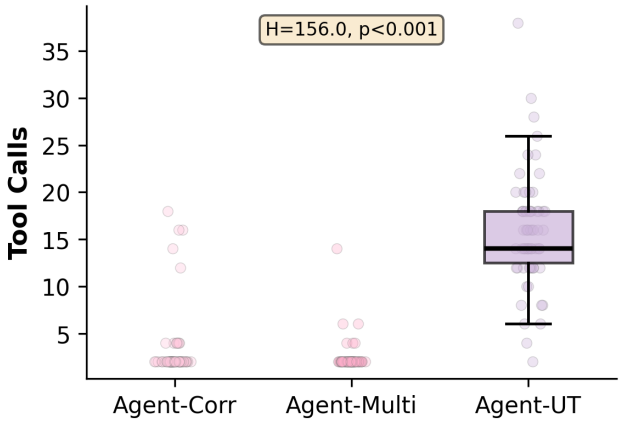

Figure 7: Number of tool (terminal) calls per task under each agent configuration.

The patterns for wall-clock time and cost match the patterns for steps and tool calls. **Agent-Corr and Agent-Multi are fast and cheap**, usually finishing quickly with a low cost per task. **Agent-UT is clearly slower and more expensive**, often taking two to three times as long and costing about two to three times as much on average. The variability in time and cost is also higher for Agent-UT, because some tasks trigger extensive test-writing and debugging while others do not. The statistical tests in Figures 8 and 9 again show strong differences between settings.

These cost results tie back to the correctness and agreement analysis. Agent-UT offers slightly better numerical alignment with unit tests and richer multi-aspect commentary, but it does so at a noticeably higher cost. Agent-Corr and Agent-Multi are cheaper and still show strong agreement with unit tests, with the expected conservative bias. LLM-only judges are cheaper still, since they do not call tools at all, but they also miss the chance to catch runtime errors or check their understanding with actual execution.

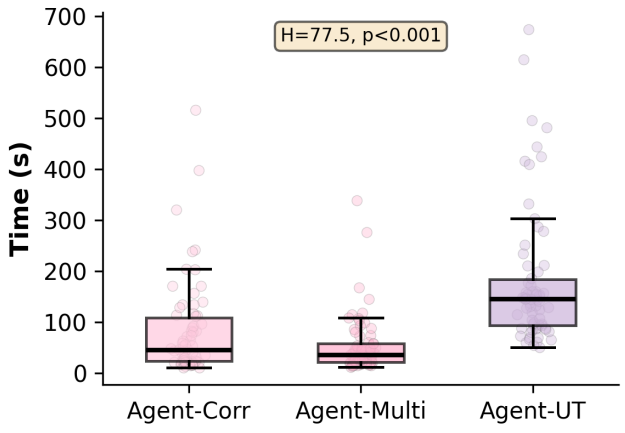

Figure 8: Wall-clock evaluation time per task for different agent settings.

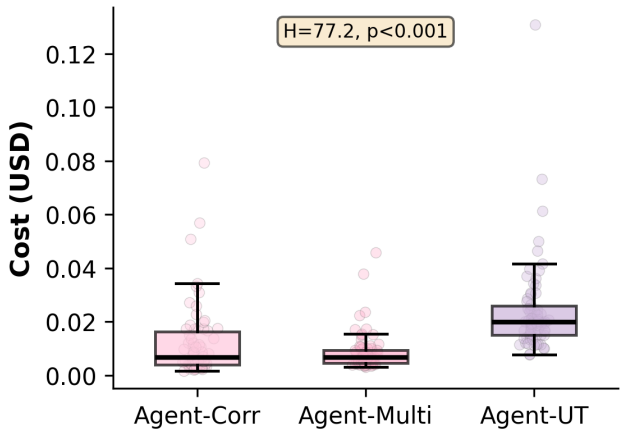

Figure 9: Approximate API cost per task (USD) under each agent setting.

Putting everything together, we view these three agent configurations as forming a simple spectrum. **LLM-only judges are the cheapest but rely purely on static reasoning**. **Agent-Corr and Agent-Multi add tool use and occasionally write simple tests, giving a stronger signal at a moderate extra cost**. **Agent-UT gives the closest match to human-written unit tests and the richest multi-aspect commentary, but it is the most expensive option**. In practice, this means that large-scale, low-stakes evaluation may prefer LLM or non-test-writing agent judges, while smaller, high-stakes settings may justify the cost of a test-writing agent, especially when real unit tests are sparse or unavailable. In all cases, understanding the trade-off between agreement with unit tests and evaluation cost is essential for choosing the right configuration.

6 Conclusion

In this paper we studied how LLM and agent judges behave when grading code solutions on LiveCodeBench, using human-written unit tests as a reference. Our results show that **LLM and agent judges can act as cost-effective alternatives when unit tests are unavailable or incomplete**, since they preserve a large portion of the unit-test correctness judgments while also providing **multi-aspect subjective evaluations** of style, simplicity, and robustness. This makes them especially attractive in modern code-based systems, where subjective factors and explanation often matter as much as raw pass/fail outcomes.

At the same time, we find meaningful differences between text-only LLM judges and tool-using agents. Agents only rarely write tests on their own, but **explicitly prompting an agent to write and run unit tests leads to correctness scores that are more closely aligned with human-annotated tests** than other AI settings. However, these agent-written tests are not as comprehensive as the benchmark tests, can **overestimate the correctness of some solutions**, and require many more steps and higher evaluation cost. Overall, our study suggests that LLM and agent judges are promising, practical tools for code evaluation, but they should be used with an understanding of their biases and trade-offs, and ideally **combined with human-designed unit tests** rather than used as a complete replacement.

7 Limitations and Future Work

Our study has several important limitations. First, the scope of tasks is relatively narrow: we focus on 70 single-file Python problems from LiveCodeBench, which is a competitive-programming style benchmark with unit tests drawn from LeetCode, AtCoder, and Codeforces. These tasks are useful and diverse, but still much smaller and more self-contained than real-world, multi-file software projects. Second, most of our analysis is based on a single agent stack, OpenHands with `gpt-5-mini`, even though many other agent frameworks and base models now exist. As a result, our findings may reflect the behaviour of this particular combination more than general properties of all LLM and agent judges.

These limitations suggest clear directions for future work. A natural next step is to **expand the evaluation to more and richer datasets**, including more LiveCodeBench tasks, other code benchmarks, synthetic stress tests, and harder scenarios such as repository-level or system-level coding tasks. Benchmarks that involve navigating and modifying full repositories, such as those proposed for repository-aware agents, would be especially relevant for testing judge capabilities beyond single files. In parallel, we plan to **benchmark a broader range of LLMs and agent scaffolds**, varying both model families and agent designs, to see how robust our observations are across architectures. Ultimately, we hope to move toward hybrid evaluation pipelines that combine unit tests, cheap LLM judges, and more expensive test-writing agents, and that work across both small coding puzzles and large, realistic software systems.

References

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

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[3] Le Du, Ybo Li, Qiang Gu, Yekun Shi, Zhengyu Li, Zixin Wang, Ke Qian, Jun Zhao, and Ming Yan. 2024. LiveCodeBench: Holistic and Contamination-Free Evaluation of Large Language Models for Code. *arXiv preprint arXiv:2403.07974* (2024). https://arxiv.org/abs/2403.07974

[4] Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, Yun Lin, Yingfei Xiong, Chao Peng, and Xia Liu. 2025. Trae Agent: An LLM-based Agent for Software Engineering with Test-time Scaling. *arXiv preprint arXiv:2507.23370* (2025). https://arxiv.org/abs/2507.23370

[5] Young-Jae Jo, Jin-Man Heo, Han-Ul Kim, Jae-Hyeok Park, Gye-Rae Song, Gye-Hyeon Kim, Joon-Sung Kim, and Joon-Young Choi. 2024. OpenHands: A Hand-Language Dataset for Open-Set Hand-Object Interaction Recognition. *arXiv preprint arXiv:2407.16741* (2024). https://arxiv.org/abs/2407.16741

[6] Quan Wang, Zhendong Liu, Yebowen Zhang, Yutong Wang, Jiacheng Yi, Chen Luo, Weize Wang, Ziyi Wang, Jiatong Shi, PENG Yin, and et al. 2024. Agent-as-a-Judge: Agent-Based Evaluation of Agentic System. *arXiv preprint arXiv:2410.10934* (2024). https://arxiv.org/abs/2410.10934

[7] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and et al. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *arXiv preprint arXiv:2306.05685* (2023). https://arxiv.org/abs/2306.05685

A Code

All code used in this work is available at:

https://github.com/WindyCall/CS5787_Final_Project/

The repository includes our full pipeline, covering data preparation and preprocessing (task extraction and cleaning), prompt construction for LLM and agent judges, experiment scripts for running evaluations, analysis notebooks for quantitative and qualitative results, and the visualization code used to generate all figures in this paper. **We used GenAI to help us implement and optimize system design, pipeline construction, results analysis, and visualization.**

B Prompt Templates

We implement all prompt templates as Python utilities in the following directory of our codebase:

https://github.com/WindyCall/CS5787_Final_Project/tree/main/src/utils/prompts

Each file in this directory defines the natural-language template that is used to build the corresponding evaluation prompts for a specific setting. The file `correctness_prompt_generator.py` constructs prompts that ask the judge to focus only on correctness; these prompts are used for both the text-only LLM correctness judge (LLM-Corr) and the agent correctness judge (Agent-Corr). The file `multi_aspect_prompt_generator.py` constructs prompts that ask for four scores (correctness, style, simplicity, robustness) and is used for the multi-aspect LLM judge (LLM-Multi) and the multi-aspect agent judge (Agent-Multi). Finally, the file `agent_unit_test_prompt_generator.py` defines the prompt used in our test-writing agent setting (Agent-UT), where the agent is instructed to generate and execute its own unit tests and then produce multi-aspect scores based on the test results.

Together, these three prompt generator scripts fully specify the evaluation instructions for all six judge configurations studied in this paper, and they can be inspected or reused directly from the linked codebase.

C Visualization

To facilitate judgment comparison, we built a visualization tool to list problems, candidate solutions, unit test logs, LLM judge reasoning, and agent trajectories. See our website below for more detail:

https://github.com/WindyCall/CS5787_Final_Project/blob/main/visualization/harbor_viz_site/index.html

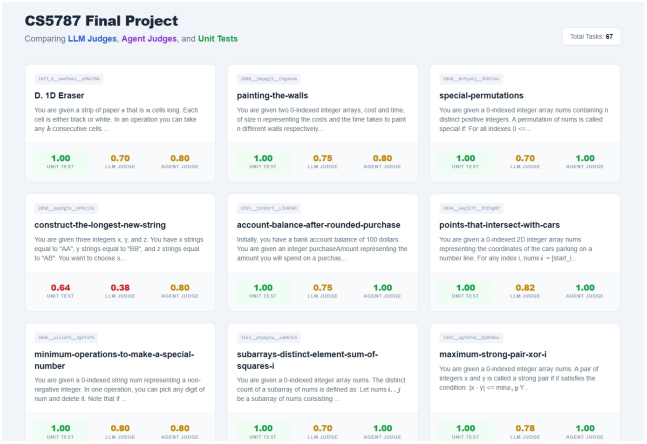

Figure 10: Entry Page of our fantastic visualization tool.

Figure. 10 demonstrates an entry page for our visualization tool that we used to inspect comparative judgments and conclude analysis findings.