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

Unit tests remain the gold standard for evaluating code correctness, yet creating comprehensive test suites is resource-intensive. This work investigates whether subjective evaluation methods—LLM-as-a-Judge and Agent-as-a-Judge—can serve as viable alternatives to objective unit testing for programming tasks. We formalize three evaluation paradigms (J_{unit}, J_{llm}, J_{agent}) and measure their alignment using LiveCodeBench and EvoEval. Our preliminary results show that while LLM judges often align with unit test pass rates, they sometimes provide lower scores because of considerations other than correctness. For upcoming experiments on agent judges, we hypothesize that agents with terminal access will demonstrate superior evaluation outcome compared to pure LLMs, due to its additional ability to write, execute, and analyze files. Ultimately, insights from our study can demonstrate whether agent and/or LLM judges could act as cost-efficient alternatives when unit tests are not available or accessible for software development.

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, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction and Problem Statement

For coding tasks where code solutions are expected, deterministically verifiable unit tests have long been the reliable gold standard for assessing correctness. However, curating correct and comprehensive unit tests is both labor-intensive and difficult. As Large Language Model (LLM) evolves, LLM-as-a-Judge [6] has emerged as a computationally efficient alternative to human evaluators across various tasks such as coding, writing, and instruction following.

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 [5] as a successor to LLMs, 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 codes, and interpret outputs. Studies have shown that with extensive tool access, agents are better selectors of debugging patches than pure LLMs [3].

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

Therefore, the core problem we are investigating is: **when evaluating solutions for programming tasks, can subjective judges be a reliable alternative to objective, gold-standard unit tests?** Our investigation examines two distinct judging paradigms.

First, we evaluate **LLM-as-a-Judge**, which assesses code correctness based purely on the model's understanding of the solution without any execution capabilities. This paradigm tests whether language models can identify logical errors, algorithmic correctness, and edge case handling through static analysis alone.

Second, we explore **Agent-as-a-Judge**, which extends beyond pure language understanding by providing the judge with terminal access and code execution capabilities. With these tools, agent judges can autonomously generate test cases, execute solutions, and observe runtime behavior before making their assessment. We hypothesize that this execution-based approach will yield evaluation results that align more closely with unit test outcomes, as agents can verify their judgments through empirical testing rather than relying solely on code comprehension.

By comparing unit tests, LLM judges, and agent judges, we measures the reliability of the subjective, automated evaluation tools (LLM/Agent judges) on coding tasks, where objective assessment theoretically exist. Our findings reveal insight into how much subjective judges align or disagree with unit tests and what assessment can subjectivity offers beyond correctness check.

2 Related Work

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

Zheng et al. [6] demonstrated that large language models can effectively replace human evaluators across various tasks by leveraging their understanding of code semantics and programming patterns to assess solution correctness through static analysis alone. As tool use and agents become popular and effective in a wide range of tasks, Agent-as-a-Judge [5] extends the evaluation capabilities by equipping judges with tool-use abilities. This enhanced paradigm posits that agents with access to execution environments can provide more nuanced and accurate evaluations by dynamically verifying code behavior, running test cases, and observing runtime outputs rather than relying solely on code comprehension.

Table 1: Comparison of Three Evaluation Paradigms of Coding Task Solutions

Method	Input	Output	Method Description
Unit Test (J_{unit})	Solution	Objective score J_{unit}	Run deterministic test cases to compute pass rate. The evaluator parses execution results and calculates the proportion of passed tests.
LLM-as-a-Judge (J_{llm})	(Question, Solution)	Subjective score J_{llm}	Prompt the model to act as an expert code reviewer and provide a correctness score. The model has no terminal access and cannot observe unit test outputs.
Agent-as-a-Judge (J_{agent})	(Question, Solution)	Score J_{agent} + trajectory	Prompt the agent to act as an expert evaluator. The agent can use terminal tools to generate and run tests, analyze outputs, and form an informed evaluation of solution correctness.

3 Methodology

In this section, we introduce the pipeline and details for our dataset curation process, experiment design, and evaluation metrics.

3.1 Data

For coding tasks where manually-labeled ground-truth unit tests are provided, we selectly sample problems from LiveCodeBench [2], EvoEval [1], and CodeJudge-Eval [4] to form our dataset. These tasks usually follows a LeetCode-style task input, where it requires a Python function to pass the given unit tests.

Since provided reference solutions are trivial as they should reflect perfect pass rate regardless of assessment approach, we need non-trivial solutions to these tasks to allow effective downstream evaluation on judges. Therefore, we collect inference results by prompting Claude Code + Claude-4.5-Haiku to solve these coding problems.

This process generates a structured dataset containing multiple tasks, where each task consists of a (Problem, Solution) pair along with its corresponding unit test results. It will later serve as the standardized input for both the J_{llm} and J_{agent} evaluation tracks, ensuring consistent and comparable assessments across different judging methods.

3.2 Evaluation Paradigms

As listed in Table 1, for a given coding task T (problem description) and a candidate solution S (code), we aim to compare the outputs of three evaluation approaches, all of which output a score in the $[0.0, 1.0]$ range.

Unit Test ($J_{\text{unit}}(S,T) \rightarrow [0.0, 1.0]$) serves as our objective ground truth, parsing test output logs from the inference phase to extract deterministic results such as "TASK FAILED: X out of Y tests failed!" or "All X tests passed!", with the final score calculated as the test case pass rate $\frac{\text{passed\_tests}}{\text{total\_tests}}$.

LLM-as-a-Judge ($J_{\text{llm}}(S,T) \rightarrow [0.0, 1.0]$) provides subjective evaluation through static code analysis, where an LLM receives only the problem description and solution without execution capabilities, acting as an expert code reviewer to assess correctness and edge case handling.

Agent-as-a-Judge ($J_{\text{agent}}(S,T) \rightarrow [0.0, 1.0]$) extends subjective evaluation with tool-use capabilities, allowing the agent to autonomously generate and execute test cases through terminal access, observing runtime behavior to inform its evaluation score.

Our goal is to measure the alignment between these subjective judges and the objective baseline, quantifying how well J_{llm} and J_{agent} approximate J_{unit}.

3.3 Evaluation Metrics

To quantify the alignment between subjective and objective evaluation methods, we focus on measuring prediction errors and their magnitudes.

We define the **Prediction Error** as $J_{\text{llm}} - J_{\text{unit}}$ or $J_{\text{llm}} - J_{\text{unit}}$ for LLM- or Agent-Judges respectively, which represents the difference between the subjective score assigned by the judge and the objective unit test pass rate. This signed error helps us understand whether the judge tends to overestimate or underestimate code correctness.

We then compute two aggregate metrics from these errors: the **Mean Absolute Error (MAE)**, which measures the average magnitude of prediction errors regardless of direction, and the **Root Mean Squared Error (RMSE)**, which penalizes larger errors more heavily by squaring the differences before averaging.

Together, these metrics provide a comprehensive view of how well subjective evaluation aligns with objective ground truth, with MAE offering interpretability and RMSE highlighting cases where the judge significantly misjudges code quality.

4 Setup for Preliminary Results

Our overall project methodology is divided into three phases: inference, data preparation, and judging.

In the **inference phase**, we use Claude 4.5 Haiku to solve LeetCode problems from the LiveCodeBench dataset[2]. Each problem description is provided to the model, which outputs a code solution that becomes the object of our evaluation.

The **data preparation phase** uses inference data from LiveCodeBench[2] for cost saving. We extract problem-solution pairs from the dataset and pair them with their corresponding unit test results. This process generates a structured JSON dataset containing (Problem, Solution) pairs along with their pre-calculated unit test pass rates, serving as the ground truth for all subsequent evaluations.

The **judging phase** forms the core of our project, where we run three parallel evaluation tracks on the same structured dataset. For unit testing (J_{unit}), we execute deterministic test cases and compute the pass rate as our objective accuracy score.

For **LLM-as-a-Judge** (J_{llm}), we use GPT-4o-mini for this milestone with the following prompt: `"You are an expert code reviewer. Evaluate the correctness and quality of the following code solution."` The model provides a subjective correctness score without access to terminal or unit test outputs.

We leave the development of **Agent-as-a-Judge** (J_{agent}) for future steps, where we want to we prompt an agent to act as an expert evaluator with terminal access and equip it with containerized environments when judging the code solutions. We plan to try both vanilla (i.e., same prompt as LLM-as-a-Judge) and instructional (i.e., prompt it to write, execute, and interpret tests) modes to fully investigate the potentials of agent judges.

5 Preliminary Results

In this section, we demonstrate and analyze our preliminary results by comparing LLM judge scores to unit test pass rates on a sample of LiveCodeBench tasks.

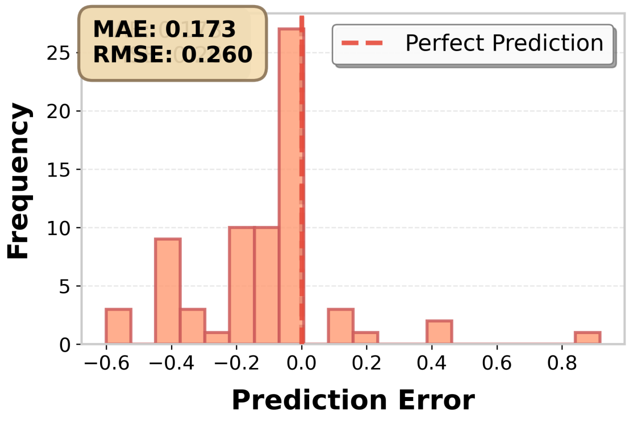

Figure 1: The histogram shows prediction errors calculated as LLM judge scores minus actual unit test pass rates across 70 LiveCodeBench coding tasks evaluated using GPT-4o-mini. Negative values (left side) indicate underestimation, while positive values (right side) indicate overestimation. The red dashed line marks zero error, representing perfect alignment between judge scores and test results.

As shown in Fig 1, the distribution exhibits a leftward skew with the majority of errors concentrated in the negative region, revealing a systematic conservative bias in LLM evaluation. This pattern indicates that GPT-4o-mini consistently tends to underestimate solution quality compared to empirical test results, with a mean error of −0.104 demonstrating an average 10.4% gap favoring caution over overconfidence. The concentration of predictions within ±0.2 of actual accuracy (MAE = 0.173) suggests reasonable calibration at fine-grained levels, while the broader spread (RMSE = 0.260) reflects occasional larger discrepancies where the judge identifies code quality issues beyond functional correctness. Notably, despite this conservative bias in continuous scoring, the binary classification accuracy reaches 88.6%, indicating that the judge reliably distinguishes between fundamentally passing and failing solutions at a coarse granularity.

Through detailed analysis of judge reasoning, we observe that LLM judges can effectively reflect unit test rigorousness while offering additional insights beyond functional correctness. In scenarios where judge scores closely align with unit test pass rates, we find that both evaluations converge on identical performance bottlenecks through different mechanisms—the LLM via static complexity analysis and unit tests via empirical timeout behavior. For instance, in the **Alternating String problem** [1] with constraints $N, Q \leq 5 \times 10^5$, unit tests evaluated a solution achieving 64.3% accuracy (9/14 tests), with all 5 failures due to timeouts on large inputs where $N, Q \approx 5 \times 10^5$. The LLM judge assigned 0.6, precisely identifying the root cause: "significant performance issues due to the O(R-L) complexity of flipping characters for each type 1 query, which can lead to timeouts." Both evaluations converged on the same diagnosis—the naive $O(Q \times N)$ approach correctly handles small inputs but becomes prohibitive at scale, requiring $O(2.5 \times 10^{11})$ operations in worst cases. The judge accurately recommended advanced data structures (segment trees, lazy propagation) to achieve the necessary $O(Q \log N)$ complexity. The 4.3-point gap (0.6 vs 0.643) demonstrates strong alignment, with the LLM's static analysis successfully predicting the empirical runtime failures.

In cases where LLM judges provide lower scores than unit test pass rates, we observe that judges evaluate dimensions of code quality that unit tests cannot directly measure, such as algorithmic efficiency and implementation elegance. These underestimations often reflect valid criticisms where solutions achieve functional correctness but employ suboptimal approaches. As shown in the **1D Eraser problem**[2], unit tests evaluated a solution achieving 100% accuracy (13/13 tests passed), confirming functional correctness. However, the LLM judge assigned 0.6, correctly identifying an efficiency flaw: after whitening k consecutive cells, the code increments the pointer by only 1, unnecessarily checking each newly-whitened cell rather than jumping forward by k positions. While the solution produces correct results with acceptable runtime, this represents $O(nk)$ complexity versus the optimal $O(n)$. The judge's reasoning was accurate: "it does not correctly skip over already white cells after performing an operation, which may lead to unnecessary

[1]https://github.com/WindyCall/final_project/blob/main/2025-11-04__15-32-48/abc341_e__dCoaVdt/agent/command-0/command.txt
[2]https://github.com/WindyCall/final_project/blob/main/2025-11-04__15-32-48/1873_d__ewd5mxi/agent/command-0/command.txt

operations.” This 40-point gap demonstrates the LLM’s ability to evaluate code quality beyond correctness—a dimension that unit tests cannot measure, as they only verify input-output correctness regardless of implementation efficiency.

Therefore, our analysis reveals that LLM judges provide complementary value to traditional unit testing: they achieve reasonable alignment with empirical correctness (correlation = 0.259, binary accuracy = 88.6%) while simultaneously evaluating algorithmic sophistication and code quality. The conservative bias (mean error = -0.104) proves beneficial in code evaluation contexts, as underestimation of suboptimal implementations is preferable to overconfidence. While unit tests provide definitive correctness verification, LLM judges offer rapid, scalable assessment that captures both functional correctness and implementation quality, making them valuable for preliminary screening in educational and development workflows.

6 Next Steps

To complete our study, we plan to first develop the Agent-as-a-Judge workflow to allow comparative analysis among unit tests, LLM judges, and Agent judges. Then we will benchmark our full dataset tasks with multiple LLM judges to explore Judge-level variances and offer practical insights (e.g., state-of-the-art models/judges are not necessary or do not always provide the best evaluation, depending on the task complexity). Later, if we have time, we will further investigate internal biases of these judges (e.g., consistency bias, position bias, verbosity bias, self-enhancement bias) to more comprehensively "judge the judges".

References

[1] Yuxuan Cui, Jiaxin Wang, Zongjie Zhang, Chen Qian, Binyuan Wang, Yu Zhang, Yong Liu, and Ge Xu. 2024. EvoEval: A Multi-level Benchmark for Evaluating Code Generation of Large Language Models. *arXiv preprint arXiv:2403.19114* (2024). https://arxiv.org/abs/2403.19114

[2] 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

[3] 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

[4] Juyoung Song, Sang-eun Lee, Dongmin Kim, Jiseong Kim, Hyung-Kyu Shin, Min-Hwan Jeon, and Yunjey Jang. 2024. CodeJudge-Eval: A Benchmark for Evaluating Reasoning Capabilities of LLMs in Code Understanding and Generation. *arXiv preprint arXiv:2408.10718* (2024). https://arxiv.org/abs/2408.10718

[5] 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

[6] 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