LITTLE ENGINES OF EVIDENCE: CODIFICATION OF TASK-SPECIFIC NATURAL-LANGUAGE RUBRICS

Anonymous authors
Paper under double-blind review

ABSTRACT

Language-model-based evaluators can grade outputs of complex tasks using natural-language rubrics as specifications. High-quality evaluators often use agentic workflows that make multiple LLM and tool calls. While effective, such evaluators are expensive, noisy, and difficult to run at high cadence. We study *rubric codification*: converting each natural-language rubric into a small executable program that servers as the evaluator. We focus on spreadsheet tasks where an input workbook is transformed into an output workbook based on a user query, and codify the NL rubrics associated with such tasks. A codification agent receives a rubric, training examples consisting of initial and final workbooks plus verdicts from an (expensive) agentic evaluator, and it emits Python code. We evaluate the effectiveness of the codification approach on a finance-spreadsheet benchmark set with 36 queries and 194 rubric criteria.

1 INTRODUCTION

Modern benchmarks for LLM-based agents increasingly require evaluators that go beyond exact string matching. Spreadsheet agents, for example, may add worksheets, create formulas, build pivot tables, update cell formats, and generate explanations. Multiple spreadsheet outputs can satisfy the same task requirements, making exact-match evaluation both brittle and inadequate. Consequently, benchmark designers increasingly rely on LLM-based judges that assess outputs according to natural-language rubrics.

LLM-based judges provide the flexibility needed to evaluate open-ended agent behavior, but they incur substantial inference cost and latency. Evaluating a single benchmark often requires multiple model calls and multiple tool calls. These costs are particularly burdensome in settings such as continuous regression testing and hill-climbing optimization loops, where the same evaluation procedure must be executed repeatedly across large numbers of agent runs. While natural-language rubrics are convenient for humans to author, repeatedly invoking expensive LLM judges to apply those rubrics creates a significant evaluation bottleneck.

This observation raises a natural question: can rubric-based grading be codified? That is, can we automatically synthesize a program that reproduces the behavior of an expensive LLM judge and thereby amortize evaluation cost across many benchmark runs? Such codification would reduce evaluation latency, improve scalability, and provide deterministic rewards for optimization and regression testing.

At first glance, the answer appears to be no. Consider a rubric requiring that a spreadsheet agent compute MOIC using a ratio rather than a difference. Writing a general-purpose program that determines whether an arbitrary spreadsheet transformation satisfies such a rubric appears difficult. However, benchmark evaluation differs from the general setting in an important way. The grader does not receive arbitrary pairs of spreadsheets. Instead, it receives spreadsheet outputs produced by a particular benchmark task. Let x denote the benchmark's initial spreadsheet and y the spreadsheet generated by the system under test. Because the task fixes x and the system under test constrains the space of plausible outputs y, the grading code only needs to operate over a much narrower distribution. A grading procedure that is infeasible to generate for the general distribution may be quite feasible to generate when restricted to the distribution constrained by the benchmark and the system under test.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

This observation motivates our central formulation of task-specific rubric codification and its view as a programming-by-example (PBE) problem. Suppose an expensive grader evaluates a rubric r on a benchmark output pair (x, y) and produces a score s. Each tuple $((x, y, r), s)$ serves as an input-output example describing the grader's behavior. Given a collection of such examples, the goal is to synthesize a program that predicts grader outputs for future benchmark runs. In effect, the synthesized program acts as a reusable surrogate for the expensive grader.

Realizing this idea requires overcoming several challenges. First, benchmark rubrics often contain nuanced requirements that are difficult to infer from scalar grades alone. We therefore leverage the natural-language explanations produced by the expensive grader as additional supervision during synthesis. Second, individual synthesized programs can overfit particular examples or capture only part of a rubric. To address this issue, we use an re-codification loop guided by failures on a cegis-development-set of examples. We also use an alternate mechanism of generating a committees of rubric-checking programs and aggregate their outputs, but the two approaches produce similar gains. Finally, because codification is not always possible, especially for ambiguous or novel cases, we use a simple fallback mechanism to get hybrid strategies that combine synthesized programs with selective LLM-based grading.

Building on these ideas, we introduce an agentic workflow for task-specific rubric codification. Our approach combines program synthesis by example, explanation-guided learning, multiple-example supervision, and failure-guided feedback loop to generate code that approximates the behavior of an expensive LLM grader. The framework can be viewed as a counterexample-guided inductive synthesis process in which grader outputs and explanations iteratively guide the construction of rubric-checking programs.

Our contributions are:

- We formulate task-specific rubric codification as a programming-by-example problem and argue that benchmark-specific constraints make codification substantially more tractable than general rubric evaluation.
- We propose an agentic codification framework that synthesizes rubric-checking programs from grader-produced examples and explanations.
- We use a failure-guided re-codification loop to improve robustness of the generated code.
- We show that the codified rubrics are effective, and when they disagree with the LLM-based grader, it is often the case that the LLM-based grader is itself uncertain on that particular grading task.
- We show that synthesized codifiers closely match the behavior of an expensive agentic grader, with disgreement falling below 10% on grading instances where the agentic grader does not flip between pass and fail grades in 3 independent runs.

1.1 Illustration

Consider a task: "Calculate Ending MRR on the given workbook. Also add a column that compares your calculated Ending MRR to the next month's starting MRR and verify that it's correct." The input workbook here has a table with column headers such as, "Month", "Starting MRR" and "Churned MRR" (which is the rightmost column). The task-specific rubric to be graded is: "Layout-Placement: Added Ending MRR and check/comparison fields should be placed directly to the right of the existing MRR data." This task-specific rubric is an specialization of a general task-agnostic rubric "Layout, Placement: How clean is the placement of content?" to this task. Using specific low-level rubrics has been recognized as an effective way to use LLM-as-a-judge (Biyani et al., 2024). It is difficult to turn the general rubric to code, but the task-specific rubric above is more amenable to codification. Our codification agent produces code that works as follows: (1) Find the correct sheet in the output workbook (2) Identify the rightmost column (3) Confirm that Ending MRR is added to the right (4) Ensure comparison is added. For Step (1), the code extracts the sheet named "MRR" from the output workbook. This works because the sheet named "MRR" is already present in the input workbook that the task wants to edit. Similarly, the code for Step (2) finds a row that contains the strings "Month", "Starting MRR" and "Churned MRR", which is deemed as the header row and the column of the cell where "Churned MRR" is found is deemed the rightmost column. Again, these constant strings are directly pulled from the input workbook. Next comes the challenging part:

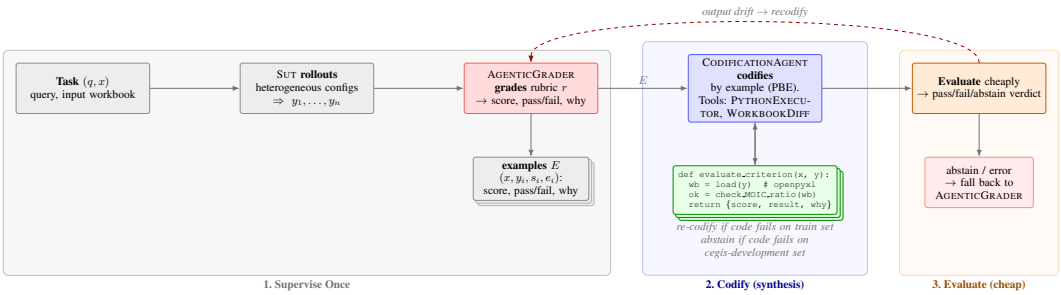


Figure 1: **Rubric codification at a glance.** *(1)* An LLM grader (AGENTICGRADER) labels each natural-language rubric r on a handful of system-under-test rollouts $y_i = \text{SUT}(q, x)$, yielding scored, explained pass/fail *examples*. *(2)* A codification agent (CODIFICATIONAGENT) treats these labels as a programming-by-example problem and (using a Python executor (`openpyxl`) and a workbook-diff tool) synthesizes a small, deterministic evaluator `evaluate_criterion`; the agent can implicitly or explicitly loop until code works on train set; codification is successful only if code passes the cegis-development set. *(3)* At evaluation time, the code grades new outputs cheaply, falling back to AGENTICGRADER if codification had failed or the code throws an exception. Codification amortizes expensive supervision across repeated (e.g., nightly) benchmark runs, and output drift triggers periodic recodification.

Step (4) requires finding the new column that contains comparison of Starting MRR and Ending MRR. This column may be placed anywhere to the right (with some intermediate columns) and with any header. Here the codification agent relies on the sample output workbooks it gets in our approach: it uses the comparison column name in these samples to produce a vocabulary of common column headers, [“check”, “comparison”, “compare”, “difference”, “variance”, ...], and this list is used in the code to detect the comparison column. While this may not work always, it will work on most of the output workbooks that a spreadsheet agent is likely to create for this task. This is the basic intuition behind our work.

2 RELATED WORK

LLM-as-a-judge systems replace hand-coded metrics with model-generated scores or preferences. G-Eval introduced a form-filling paradigm with reasoning steps for NLG evaluation and reported stronger human alignment than classical reference metrics (Liu et al., 2023b), while GPTScore framed evaluation as conditional likelihood scoring under task and aspect prompts (Fu et al., 2024). MT-Bench and Chatbot Arena popularized strong-model judging and pairwise comparison for open-ended assistant evaluation (Zheng et al., 2023). A growing body of work hardens this paradigm: Prometheus 2 is an open evaluator model that performs both direct assessment and pairwise ranking against user-defined criteria (Kim et al., 2024); Self-Taught Evaluators train judges from synthetic contrasting outputs without human labels (Wang et al., 2024); and Agent-as-a-Judge extends judging to agentic systems that inspect intermediate steps rather than only final artifacts (Zhuge et al., 2024), mirroring the expensive agentic grader we distill from. Recent surveys catalog the design space and reliability pitfalls of these evaluators (Gu et al., 2024). Importantly, the judges are themselves imperfect: JudgeBench shows that even strong models barely exceed chance on challenging response pairs (Tan et al., 2025), and RuVerBench finds that even frontier judges stay noisy when verifying rubrics in agentic deep-research and coding scenarios (Peng et al., 2026); this motivates offloading the stable, mechanical part of a rubric to inspectable code while reserving the judge for genuinely ambiguous cases. Our supervision is closest in spirit to HealthBench, whose tens of thousands of physician-authored, conversation-specific rubric criteria show that instance-specific rubrics can drive open-ended evaluation at scale (Arora et al., 2025); we take such per-instance rubrics as input and compile them into executable checks.

Decomposing evaluation into small, explicit criteria reduces ambiguity and variance. CheckList reframed model testing as targeted behavioral tests rather than a single accuracy number (Ribeiro et al., 2020), and CheckEval applies checklist decomposition to LLM judges to combat low agreement and high variance (Lee et al., 2024). Closest to our cost argument, RocketEval recasts judging

as answering an instance-specific checklist with a lightweight model and reweights checklist items against reference labels, matching a frontier judge at a fraction of the cost (Wei et al., 2025). Our results echo this literature (tight per-rubric criteria are far more codifiable than broad, combined rubrics) but we replace repeated LLM checklist-grading with synthesized programs that execute deterministically and can be cached across benchmark runs.

A parallel thread uses criteria not to evaluate but to *supervise* training. Reinforcement Learning with Verifiable Rewards (RLVR) yields large reasoning gains when correctness is programmatically checkable, as in mathematics and code (Lambert et al., 2024a; DeepSeek-AI, 2025). To extend reward signals to open-ended tasks, Rubrics as Rewards turns instance-specific rubrics into structured rewards (Gunjal et al., 2025), and Reinforcement Learning from Checklist Feedback extracts per-instruction checklists and scores responses with both AI judges and *specialized verifier programs* (Viswanathan et al., 2025). These verifier programs are the closest prior idea to ours, but they are produced inside a training-time reward loop; we instead synthesize reusable, auditable evaluators, validate them against a reference judge, and deploy them as an amortized evaluation layer with explicit LLM fallback. This connects to reward-model and verifier research more broadly, e.g., outcome and process verifiers for reasoning (Cobbe et al., 2021; Lightman et al., 2023; Setlur et al., 2024), reward-model benchmarking (Lambert et al., 2024b), and principle-based AI feedback (Bai et al., 2022), but our object is deployable evaluation infrastructure rather than a policy-optimizing reward.

Programmatic evaluation is strongest when a correctness predicate can be executed. HumanEval, MBPP, and APPS run generated code against tests (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021); EvalPlus shows that weak suites overstate quality and that stronger tests expose hidden failures (Liu et al., 2023a); and CodeT uses generated tests and execution agreement to select among candidate programs (Chen et al., 2022). Executable checking increasingly anchors agent benchmarks as well: SWE-bench validates repository edits with the projects' own test suites (Jimenez et al., 2024), and τ-bench verifies tool-agent episodes by comparing the final database state against an annotated goal state (Yao et al., 2024) (a state-comparison predicate directly analogous to checking an output workbook against a rubric). Recent analysis of coding-agent reward design further argues that no fixed verifier stays reliable as agents improve, so verification must co-evolve with the generator (Wang et al., 2026). In all of these the checker is hand-authored; the object we synthesize *is* the checker, learned from a reference judge's labels, so a held-out set of judge-labeled rollouts plays the role of a hidden test suite.

Programming-by-example systems such as Flash Fill (Gulwani, 2011) and FlashFill++ (Cambronero et al., 2023) synthesize spreadsheet transformations from small example sets shaped by a carefully designed DSL. Rubric codification differs in that examples label *correctness* rather than demonstrate a transformation, yet the machinery is shared: counterexample-guided synthesis and generalizing from examples. Self-consistency motivates our committee-of-programs aggregation (Wang et al., 2023), while Self-Refine, Self-Debugging, and Reflexion (Madaan et al., 2023; Chen et al., 2024; Shinn et al., 2023) underpin the iterative execution-feedback loop that we find substantially improves codification quality.

3 Rubric Codification

Consider a task specified with a natural language query q and an input x. We assume that there is a base system, the *system-under-test* SUT, that generates samples y from the probability distribution $Pr(y \mid q, x)$ over likely outputs given (q, x). In order to grade SUT, one needs a task specification; a popular choice is a set of natural-language rubrics.

A rubric r specifies when a tuple (q, x, y) is deemed correct. Assuming a 5-point correctness scale, let (oracle) grader G_r map (q, x, y) to a pair (s, e) where $s \in \{0, 1, 2, 3, 4\}$ is a score and e is a natural-language explanation for that score. Let $Pass((s, e)) =$ "$pass''$ if $s \geq 3$ and $Pass((s, e)) =$ "$fail''$ if $s < 3$. The rubric r can be task-dependent; that is, it can change when q and x change.

Rubrics are designed to evaluate SUT. An evaluation benchmark consists of tuples (q, x, r), where (q, x) is a task and r is a rubric. Given a benchmark (q, x, r), the system SUT gets the grade $r(q, x, \text{SUT}(q, x))$.

The goal of codification is to synthesize a program p_r that approximates G_r for a *fixed* task. Because the query q, the input workbook x, and the rubric r are all fixed at codification time, p_r needs only the agent output y (and the fixed input workbook x, which it reads to compare against y); we therefore generate p_r as a function of (x, y). It returns a score and explanation (s', e'), and we want the pass-fail verdict to match the same verdict from G_r.

Definition 1 *Given query q, fixed input x, and a rubric r, the task-specific rubric codification problem seeks to generate a program p_r that maximizes the expectation $E_{y\sim \textsc{Sut}(q,x)}[Pass(p_r(x, y)) = Pass(G_r(q, x, y))]$ that p_r correctly judges the rubric r* for y drawn from $\textsc{Sut}(q, x)$.

The main difference between general-purpose codification and task-specific rubric codifition is that we only care for the code p_r to work on a very restricted set of pairs (x, y). We assume we have a reference grader AGENTICGRADER that approximates the oracle grader $G_r(q, x, y)$ using expensive LLM calls in an agentic loop, and our operational codification objective tries to maximize the expectation $E_{y\sim \textsc{Sut}(q,x)}[Pass(p_r(x, y)) = Pass(\textsc{AgenticGrader}(q, x, y))]$.

3.1 CODIFICATION AS PROGRAMMING-BY-EXAMPLE

Our main idea is to view codification as a programming-by-example (PBE) problem. In programming-by-example, we are given a small number of input-output pairs, say $(i_1, o_1), (i_2, o_2), \ldots$, where i's are the inputs and the o's the corresponding outputs, and the task is to synthesize a program p such that, for any input i in the provided set (and in a held-out test set), running p on i returns the corresponding output o. Codification is naturally a PBE problem: for the fixed rubric r, the expensive AGENTICGRADER labels any rollout y with an output (s, e), yielding input–output examples $\big((x, y), (s, e)\big)$ for synthesis.

We codify each rubric criterion independently. Starting from a system-under-test SUT and a benchmark (q, x, r), our overall approach for generating the codified rubric p_r is as follows (see also Figure 1):

1. **Generate rollouts.** Starting with (q, x), we use SUT to generate a sample set Y from $\textsc{Sut}(q, x)$. In practice, SUT is not a fixed system but has tunable hyperparameters and the set Y is picked by randomly sampling the hyperparameters and using them to generate $y \in Y$.
2. **Grade rubrics individually.** Run AGENTICGRADER to generate a score s and explanation e for the rubric r for each $y \in Y$ to get the set $\{(y, s, e) \mid (s, e) = \textsc{AgenticGrader}(q, x, y), y \in Y\}$ of so-called *examples* E.
3. **Split train/val/test examples.** Split the set E into train, cegis-development and test sets. We use 15 training examples, 5 cegis-development examples and 30 test examples.
4. **Synthesize code.** Run a new codification agent CODIFICATIONAGENT to generate code p_r. The agent receives the rubric r, the input x and the collection of tuples $(y, s, e) \in E_{\texttt{train}}$. (It never receives labels or artifacts from $E_{\texttt{test}}$. In the CEGIS variant, it can query limited feedback computed over $E_{\texttt{dev}}$ as described in Appendix C." The agent is given *the same tools* that the AGENTICGRADER was given. The agent can internally re-codify if it finds that the generated code fails the training examples. This loop can also be forced explicitly.
5. **Validate and select.** Codification succeeds if the generated code passes the cegis-development set, and fails otherwise. By pass we mean that the code should generate the matching pass/fail verdict.

The examples produced by the AGENTICGRADER can be provided to the CODIFICATIONAGENT as in-context examples or as artifacts that can be accessed through tools.

The above procedure yields a program p_r that automates the check for rubric r. At test time, it is used as follows:

1. **Evaluate code.** When SUT is being evaluated on a benchmark (q, x, r), the system is first run to generate $y = \textsc{Sut}(q, x)$, and then the output y is evaluated by executing $p_r(x, y)$. Fatal API errors and uncaught exceptions are treated as abstention, i.e., the program p_r assigns no verdict in those cases.

2. **Fallback mechanism.** If codification had failed to produce p_r or if p_r abstained from grading, we fall back to AGENTICGRADER to produce a score and explanation.

4 EVALUATIONS

We evaluate rubric codification on tasks related to spreadsheet manipulation where a user issues a query q to perform that task on their input workbook x, and expects to get back an output workbook y. The system SUT is any complex multi-agent system that performs this task.

4.1 BENCHMARK

We use a collection of 36 finance-related queries as the main benchmark to evaluate codification. Associated with these 36 tasks, we also have 194 rubric criteria – about 5-6 rubrics per task – that are used to grade the output of SUT on the tasks. The system SUT is in constant development and it evolves daily through prompt changes, skill or tool changes, underlying model changes, and so on. The latest version of the system SUT is evaluated on the 36 queries daily. We emphasize here that the held-out evaluation set (30 per rubric) was temporally separated from codification by more than a month, during which sys continued to evolve.

4.2 DOES CODIFICATION WORK?

The foremost question is: does codification work? We followed our aforementioned PBE-based codification agent to codify the 194 rubrics described above. We then picked 30 evaluation runs of the system SUT and collected both (a) the outputs y generated by SUT and (b) the grades, score-exaplanation pair (s, e) generated by the expensive LLM-based AGENTICGRADER. We used the generated code to grade the (q, x, y) tuples. We compared the Boolean pass-fail outcomes. If the code gives a "pass" verdict, then we say we get a *PassMatch* if AGENTICGRADER also gives a "pass" verdict for that case and we get a *PassMismatch* if AGENTICGRADER gives a "fail" for that rubric. Similarly, if the code gives a "fail" verdict, then we say we get a *FailMatch* if AGENTICGRADER also gives a "fail" verdict for that case and we get a *FailMismatch* if AGENTICGRADER gives a "pass". If the code abstains, then we classify the case as a "fallback".

Since AGENTICGRADER is LLM-based, it can change its verdict on the same grading task, especially when the rubric is at the borderline of pass/fail boundary for the output being graded. Code is completely deterministic. We run AGENTICGRADER three times to grade each output. We compare the verdict generated by code with the verdict generated by the 3 AGENTICGRADER runs.

Figure 2 shows the percentage of test cases that fall into one of these five categories. We use three ways to aggregate the 3 scores from AGENTICGRADER. On the left plot, we use majority voting, on the middle plot, we use the minimum aggregation function, and on the right plot we move the cases where the AGENTICGRADER flip-flops (between Pass and Fail) to the "Open" category. There is less than 10% mismatch when we ignore the grading instances where AGENTICGRADER flip-flops.

RUBRICS DIFFER IN THEIR CODIFIABILITY

We next investigate whether rubrics differ in how amenable they are to codification; in particular, is it a smooth curve from "not amenable to codification" at one end to "very amenable to codification" at the other, or a more multimodal distribution with clear peaks? We took the results from the experiment described above, but instead of aggregating everything into matches and mismatches, we partitioned the data by rubric and, for each rubric, computed its *match rate*. The match rate for a rubric r is defined as the ratio of the total number of matches to the total number of all grading tasks for that rubric: if a rubric r gets a match rate of 0 then it means the generated code p_r produced a pass-fail verdict that never matched the verdict produced by AGENTICGRADER and if a rubric r gets a match rate of 1 then it means p_r produced a pass-fail verdict that always matched the verdict produced by AGENTICGRADER. Figure 3 shows the distribution of different match rates: on the x-axis we have the match rate, and on the y-axis we have the percentage of rubrics that had that match rate.

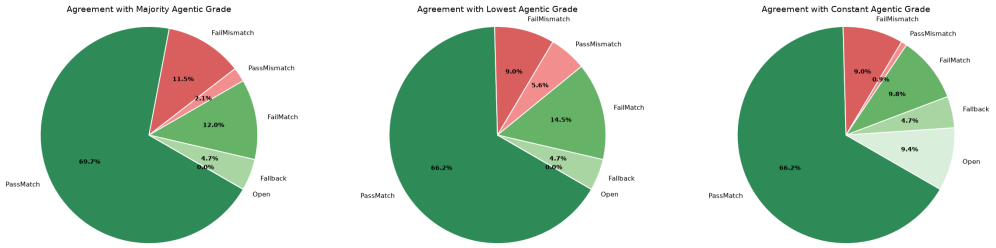


Figure 2: Percentage of grading instances where the code and aggregated-AGENTICGRADER both said "pass" (PassMatch), both said "fail" (FailMatch), code said "pass" and grader said "fail" (PassMismatch) and vice-versa (FailMismatch). Fallback refers to the code abstaining. (Left) Code verdict is compared with majority-verdict from 3 AGENTICGRADER-runs. (Middle) Code-verdict is compared with the most-strict (least) score among the 3 AGENTICGRADER scores. (Right) Code-verdict is compared with the stable-verdict of the 3 AGENTICGRADER-runs; if the 3 pass-fail verdicts were not identical, that grading instance is classified as "Open" since code-verdict can be considered correct. There is less than 10% mismatch when we ignore the grading instances where AGENTICGRADER flip-flops.

The distribution in Figure 3 shows that most rubrics had a high match rate, which indicates that they were amenable to codification and our CODIFICATIONAGENT produced code that matched the verdicts given by AGENTICGRADER when evaluating those rubrics. Looking at the other end, we see that a few rubrics have a very low match rate. This indicates that there definitely exist rubrics that were difficult for our codification procedure. As the plots in Figure 2 showed, a significant chunk of this set of rubrics are the ones where AGENTICGRADER itself is inconsistent in grading them – possibly indicating that either the NL rubrics are ambiguous or their satisfaction on the given outputs is at the borderline of pass and fail.

A natural question is why the code for the rubrics on the left of the plot not fail the validation check. This is because our validation set was small, and furthermore, the runs used for evaluation were taken more than a month after those used for codification, and the underlying SUT had been evolving continuously in this period.

POOR CODIFICATION HAS CONSISTENT ERROR SIGNATURE

We next try to understand the kind of errors that codified rubrics make. In other words, when code p_r corresponding to a rubric r is not performing well (in terms of matching AGENTICGRADER), does it produce mismatched "pass" and "fail" verdicts uniformly, or is there a skew toward one of them when it errs?

We answer that question by computing the ratio PassMismatch/TotalMismatch for each rubric, where PassMismatch is the count of grading instances where p_r gives a "pass" and AGENTICGRADER gives a "fail", FailMismatch is the count of grading instances where p_r gives a "fail" and AGENTICGRADER gives a "pass", and TotalMismatch is the sum PassMismatch + FailMismatch. For programs p_r that do not err, this measure is undefined and we filter them out. For the remaining programs, we plot the distribution of this ratio in Figure 4 (the y-axis is the percentage of programs). The figure shows an interesting phenomenon: most rubrics either have very high PassMismatch and very small FailMismatch or vice-versa, but it is uncommon for both to be nonzero and approximately equal. This means that a code p_r produces a mismatched verdict either when it generates a "pass" verdict (right bar) or when it generates a "fail" verdict (left bar), but rarely both (bars in between). Thus, errors tend to be strongly one-sided at the rubric level. It is possible to exploit this phenomenon to improve our codification approach, which we leave for future work.

4.3 ABLATION: CEGIS

A central technique in PBE is counterexample-guided inductive synthesis (CEGIS): the generated program is executed on the provided examples and, whenever its output does not match the expected output, this feedback is returned to the CODIFICATIONAGENT. Although the ability to perform CEGIS is built into the CODIFICATIONAGENT harness, we also externally enforce the CEGIS loop.

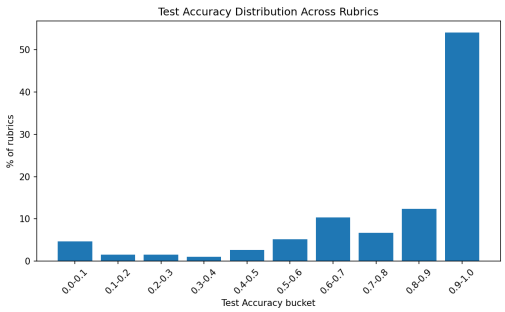


Figure 3: Distribution of match rate: Most rubrics had a match rate close to 1, indicating they were amenable to codification and their codified versions matched the verdict given by AGENTICGRADER.

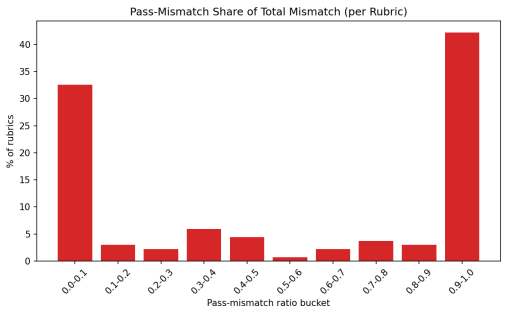


Figure 4: The distribution of the ratio of the number of mismatching "pass" grade to the total number of mismatching grades. Most rubrics fall on the two ends of the range.

Variant	RM	ASM
+CEGIS	**0.817**	**0.836**
-CEGIS	0.694	0.712
+Code Artifacts	0.694	0.704
+Diffs in Signatures	0.692	0.711
+Diff Summaries	0.646	0.656

Table 1: Effect of different codification variants: adding a CEGIS loop substantially improves quality, whereas providing grader code artifacts or workbook diffs does not.

#Input Examples	RM	ASM
1	0.407	0.402
5	0.597	0.610
10	0.683	0.704
15	**0.694**	**0.712**
15 (no explanation)	0.441	0.429

Table 2: Effect of changing number of examples given to CODIFICATIONAGENT: giving more examples improves the quality of codification.

We performed an ablation by removing the external loop. Table 1 shows that CEGIS is important and removing it significantly degrades the quality of the generated code – paralleling the observations made in the PBE literature. The table shows two metrics: RM, shorthand for Result-Match, is the fraction of grading instances where the pass-fail verdicts match (between the program p_r and the LLM-based grader AGENTICGRADER); and ASM, shorthand for Approximate-Score-Match, is the fraction of grading instances where the numerical score generated by p_r is within ± 1 of the score given by AGENTICGRADER.

We further experimented with providing more artifacts to the ablated CODIFICATIONAGENT to see if it helps with compensating for the loss from removing CEGIS. Specifically, we tried providing code that was generated by the AGENTICGRADER during its run, but it did not help (Row Code Artifacts in Table 1). We also considered a variant where we provided the difference between some outputs y's (that are part of the examples given to the CODIFICATIONAGENT, with the expectation that seeing the difference may help the agent generate better code, but that was not the case (Row Diff Summaries in Table 1). Finally, in another variant, we prompted the agent to generate code with a signature $(x, y, \mathit{diff}) \mapsto (s, e)$, rather than the signature $(x, y) \mapsto (s, e)$, where diff was the difference between y and x. In our case, since x and y are spreadsheets, diff is a representation of the artifacts (cell values, tables, charts) that were deleted from x and those that were added to x (to reach y). The idea here was that it may be easier to write code that works on the pre-computed diff; however, we did not see any significant gains from this variant. The primary reason some of these ideas do not work is that the CODIFICATIONAGENT has a natural way of doing things, and any variant that makes the task unnatural, unusual, or uncommon usually does not help.

4.4 ABLATION: NUMBER AND TYPE OF EXAMPLES TO USE FOR CODIFICATION

We investigate the effect of changing the number of examples we give to the CODIFICATIONAGENT. Since CEGIS requires examples, we conducted this experiment by turning off external CEGIS loop. We experiment with giving just 1 example, 5 examples, 10 examples, and 15 examples for the

CODIFICATIONAGENT to consult while producing code. Table 2 reports the results. The main finding is that examples play an important role in the codification process and giving more examples leads to better code. The reason may simply be that more examples reveal more diversity in the grading tasks, which helps produce generalizable code.

To study the contribution of the explanation e, we drop e from the input to the CODIFICATIONAGENT and give it only the workbooks (x, y) and the score s for each example. This ablation is the last row of Table 2: removing explanations degrades performance, indicating that the CODIFICATIONAGENT relies on the explanation text to generate code for a rubric r. This also makes intuitive sense since the explanation adds semantic context to the raw number s being shown to the CODIFICATIONAGENT.

4.5 SANITY CHECK

A reader may notice that the grading data we used for evaluating codification is skewed toward a "pass". This suggests that a naive program that always returns "pass" will perform extremely well in our evaluation. For sanity checking the programs we generated, we created two classes of "synthetic" grading tasks: (1) for each original task, we passed a non-existant filename as the output workbook, and (2) for each original task, we passed the original workbook as the output workbook too. In the first experiment, codified checks produced 3.2% of "fail" verdicts and 96.8% of "incomplete" verdicts, and no "pass" verdicts. An "incomplete" verdict, which corresponds to a score of -1, can be explicitly returned by the program or it can be returned when the code throws a runtime exception. In the second experiment, codified checks produced 26% "pass", 48.9% "fail" and the rest "incomplete" verdicts. The reason why the pass rate is not close to zero is that many rubrics are conditional checks on the edits made by the agent – they are vacuously true if there are no edits. These sanity checks show that the synthesized evaluators inspect their input and are not trivial constant-pass programs. Note that because we report the complete breakdown of pass/pass, pass/fail, fail/pass, and fail/fail outcomes, the class imbalance and any standard binary classification metric can be directly recovered from the reported results.

5 DISCUSSION AND CONCLUSION

Reference labels inherit grader variance. The codifier is trained and evaluated against AGENTICGRADER, not an oracle. As a result, disagreements may reflect poor codification, AGENTICGRADER variance, or genuine rubric ambiguity. Nevertheless, AGENTICGRADER is a deployed carefully-engineered grader, and is known to generate good hillclimbing signals.

Dependence on rubric quality and domain. As with prior work on LLM-as-a-judge, codification depends on the availability of high-quality rubrics. This limits our evaluation to a finance-oriented spreadsheet benchmark, and performance may differ in other domains. While the quantitative results may not transfer directly, we expect the central insight—that benchmark-specific rubric codification is feasible because grading is restricted to a task-specific output distribution—to generalize more broadly.

Cost. Any reduction in calls to AGENTICGRADER is a significant reduction in cost. The cost for codification is a one-time cost that is negligible compared to the cost incurred by AGENTICGRADER on nightly evaluations of deployed agentic systems: the 20 LLM calls (each making one tool call) for codification of a rubric are equivalent on average to grading of 10 grading instances using AGENTICGRADER and we evaluate on average 6 SUTvariants a day, resulting in more than a 1000 rubric-grading instances daily.

Conclusion We introduced task-specific rubric codification, a framework for converting expensive LLM-based rubric evaluation into lightweight executable graders. By formulating codification as a programming-by-example problem and leveraging an agentic synthesis workflow, many spreadsheet-grading rubrics can be translated into small Python evaluators that closely approximate an expensive agentic grader. The resulting evaluators substantially reduce evaluation cost and latency while supporting large-scale regression testing and optimization. More broadly, our results suggest that benchmark-specific grading procedures often exhibit enough structure to be codified, enabling hybrid evaluation systems that reserve LLM reasoning for only the most ambiguous cases.

AI USE STATEMENT

In this work, we used generative AI tools for none of the subtasks in the required categories. We have not used generative AI tools for: help develop theoretical models or conceptual frameworks, formulate mathematical claims, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, interpret results. The rest of the required disclosure tasks, generate synthetic data sets, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, are not applicable to this work.

Additionally, we used generative AI tools for create or modify scientific figures or images, creation of artifacts, draft parts of a research paper, summarize or analyse existing literature, discover research topics or identify gaps, sourcing/searching for information, edit a research paper to improve readability, identify relevant literature. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Rahul K. Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, and Karan Singhal. HealthBench: Evaluating large language models towards improved human health. *arXiv preprint arXiv:2505.08775*, 2025.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Scott Ringer, Shauna Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Param Biyani, Yasharth Bajpai, Arjun Radhakrishna, Gustavo Soares, and Sumit Gulwani. RUBICON: rubric-based evaluation of domain-specific human AI conversations. In *Proc. 1st ACM Intl. Conf. on AI-Powered Software, AIware 2024*. ACM, 2024.

José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. FlashFill++: Scaling programming by example by cutting to the chase. In *Proceedings of the ACM on Programming Languages, POPL*, 2023. doi: 10.1145/3571226.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

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, Joshua 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. GPTScore: Evaluate as you desire. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 6556–6576. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.naacl-long.365.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, and Shengjie Ma. A survey on LLM-as-a-judge. *arXiv preprint arXiv:2411.15594*, 2024.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 317–330. ACM, 2011. doi: 10.1145/1926385.1926423.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. *arXiv preprint arXiv:2507.17746*, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. Prometheus 2: An open source language model specialized in evaluating other language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, and Alisa Liu. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024a.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. RewardBench: Evaluating reward models for language modeling. *arXiv preprint arXiv:2403.13787*, 2024b.

Yukyung Lee, Joonghoon Kim, Jaehee Kim, Hyowon Cho, Jaewook Kang, Pilsung Kang, and Najoung Kim. CheckEval: A reliable LLM-as-a-judge framework for evaluating text generation using checklists. *arXiv preprint arXiv:2403.18771*, 2024.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023a.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2511–2522. Association for Computational Linguistics, 2023b. doi: 10.18653/v1/2023.emnlp-main.153.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Yangda Peng, Yunjia Qi, Hao Peng, Haotian Xia, Guanzhong He, Xintong Shi, Richeng Xuan, Songyuanyi Lu, Yixian Liu, Zhichao Hu, Yuhong Liu, Lei Hou, Bin Xu, and Juanzi Li. Can LLM-as-a-judge reliably verify rubrics in agentic scenarios? *arXiv preprint arXiv:2606.29920*, 2026.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4902–4912, 2020. doi: 10.18653/v1/2020.acl-main.442.

Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. Rewarding progress: Scaling automated process verifiers for LLM reasoning. *arXiv preprint arXiv:2410.08146*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Sijun Tan, Siyuan Zhuang, Kyle Montgomery, William Y. Tang, Alejandro Cuadron, Chenguang Wang, Raluca Ada Popa, and Ion Stoica. JudgeBench: A benchmark for evaluating LLM-based judges. In *International Conference on Learning Representations*, 2025.

Vijay Viswanathan, Yanchao Sun, Shuang Ma, Xiang Kong, Meng Cao, Graham Neubig, and Tongshuang Wu. Checklists are better than reward models for aligning language models. In *Advances in Neural Information Processing Systems*, 2025.

Binghai Wang, Chenlong Zhang, Dayiheng Liu, Jiajun Zhang, Jiawei Chen, Mingze Li, Mouxiang Chen, Rongyao Fang, Siyuan Zhang, Xuwu Wang, Yuheng Jing, Zeyao Ma, and Zeyu Cui. The verification horizon: No silver bullet for coding agent rewards. *arXiv preprint arXiv:2606.26300*, 2026.

Tianlu Wang, Ilia Kulikov, Olga Golovneva, Ping Yu, Weizhe Yuan, Jane Dwivedi-Yu, Richard Yuanzhe Pang, Maryam Fazel-Zarandi, Jason Weston, and Xian Li. Self-taught evaluators. *arXiv preprint arXiv:2408.02666*, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023.

Tianjun Wei, Wei Wen, Ruizhi Qiao, Xing Sun, and Jianghong Ma. RocketEval: Efficient automated LLM evaluation via grading checklist. In *International Conference on Learning Representations*, 2025.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

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. Judging LLM-as-a-judge with MT-Bench and chatbot arena. *arXiv preprint arXiv:2306.05685*, 2023.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*, 2024.

A APPENDIX

RUBRIC CODIFICATION: SUPPLEMENTARY MATERIAL

B DETAILS ON THE AGENTIC GRADER

In our spreadsheet task, we have a Natural Language query q and an input workbook x. The output consists of two things: an output workbook and any text response generated while solving q. The AGENTICGRADER used for grading spreadsheet tasks uses GPT-5 series model to judge whether a criterion, specified as a rubric in Natural Language, is met on the output spreadsheet given the input spreadsheet, query, and text response. The AGENTICGRADER has access to a python tool that can execute any python code (generated by the agent). The python tool returns the content printed on stdout by that code. The agent is informed that it can use python library openpyxl to read the workbooks. Additionally, the AGENTICGRADER also has access to the diff between the input and the output workbook. This diff is precomputed and provided to AGENTICGRADER in files (that the grading agent can ready and process using python). The prompt for the AGENTICGRADER just sets up the grading task as a teacher grading the work of a student. The prompt also contains details on the (1) tools (2) the available artifacts (i.e., the workbooks, the diffs) and (3) instructions on how to use openpyxl on large workbooks efficiently – by loading workbooks lazily – and the common pitfalls when using openpyxl to work on workbooks that have been lazily loaded.

The AGENTICGRADER returns a grade in the following form:

```
{
   "score": x ∈ {0,1,2,3,4},
   "result": score > 2,
   "comments": <NL explanation for the score>
}
```

C DETAILS ON CEGIS LOOP

In the main paper, we reported results based on using a CEGIS loop. Here are more details on this variant.

In the CEGIS Loop variant, the base setup is identical in terms of tools. An additional tool, which allows the model to get limited feedback from executing the predicted code on a cegis-development set, is added. The feedback has the following format for each example

```
{
   "givenScore": x ∈ {0,1,2,3,4},
   "givenResult": givenScore > 2,
   "givenComments": Explanation
   "expectedScore": x ∈ {0,1,2,3,4},
   "expectedResult": expectedScore > 2,
   "expectedComments": AGENTICGRADER Explanation
}
```

The first three lines here are the output of the synthesized code (for the rubric) and the last three lines are the output of the AGENTICGRADER. The codifier is not given eyes-on access to the input and output workbooks underlying these grades to prevent overfitting. The codifier only gets the information shown above, which it can use to refine the code it generates next. Note that the AGENTICGRADER is LLM-based, so it can have some inconsistency in the scores it gives, and generating code to exactly match AGENTICGRADER output may not be feasible. The codifer decides when it is done improving the code using this additional tool and it terminates by submitting a program.

D EXAMPLES OF TASK-SPECIFIC RUBRICS

Here are some example task-specific rubrics we have used.

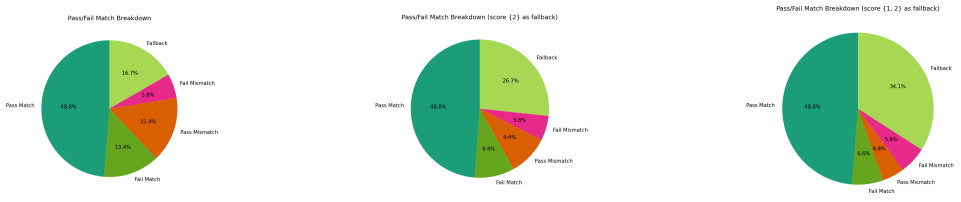


Figure 5: Percentage of grading instances where the program generated with 15 examples and AGENTICGRADER both said "pass" (PassMatch), both said "fail" (FailMatch), program said "pass" and grader said "fail" (PassMismatch) and vice-versa (FailMismatch). (Left) Fallback refers to the program abstaining. (Middle) Fallback also includes cases where the program gives a borderline score of 2. (Right) Fallback also includes the program giving a score of 1 or 2. The red mismatches shrink and cost (fallback) increases as we move right.

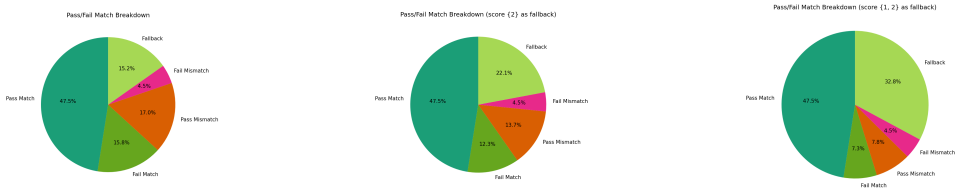


Figure 6: Percentage of grading instances where the program generated with 10 examples and AGENTICGRADER both said "pass" (PassMatch), both said "fail" (FailMatch), program said "pass" and grader said "fail" (PassMismatch) and vice-versa (FailMismatch). (Left) Fallback refers to the program abstaining. (Middle) Fallback also includes cases where the program gives a borderline score of 2. (Right) Fallback also includes the program giving a score of 1 or 2. The red mismatches shrink and cost (fallback) increases as we move right.

- **MOIC metric clearly added and labeled**: MOIC appears as a clearly named investor-return metric (e.g., (*@"MOIC"@*)) in the delivered output (workbook and/or final reported tables), and it is clearly associated with OSF scenario results (not hidden/ambiguous).
- Acceptable implementations: new column in a scenario table, a new row/metric in a summary block, or a separate MOIC table as long as each scenario can be matched unambiguously.
- **Correct MOIC formula structure**: MOIC is calculated as **OSF Exit Value (*@÷@*) OSF Investment** (ratio, not difference), with correct numerator/denominator orientation (no inversion).
- Accept either Excel formulas referencing cells/ranges or explicitly computed values shown in the output.
- **Correct input sourcing (OSF-specific + unit consistency)**: The MOIC calculation uses the **OSF** Exit Value and **OSF** Investment (not other similarly named totals like total exit value, enterprise value, equity value, sponsor cashflows, etc.), and the numerator/denominator are in consistent units (e.g., both in $orbothin$mm) so the resulting multiple is scale-correct.

E A COMPLETE EXAMPLE

Consider the task: "Calculate Ending MRR on the given workbook. Also add a column that compares your calculated Ending MRR to the next month's starting MRR and verify that it's correct." The input workbook for this task is shown in Figure 9, and it has a table with column headers such as, "Month", "Starting MRR" and "Churned MRR" (which is the rightmost column). The task-specific rubric to be graded is: "Layout-Placement: Added Ending MRR and check/comparison fields should

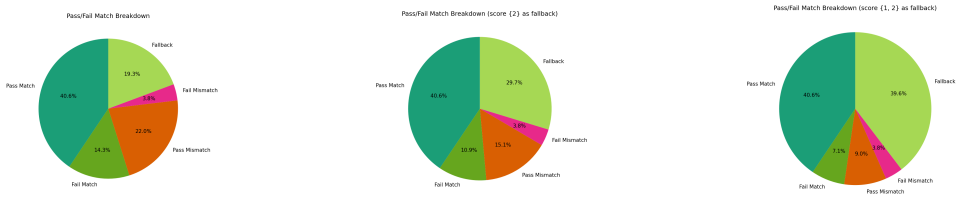


Figure 7: Percentage of grading instances where the program generated with 5 examples and AGENTICGRADER both said "pass" (PassMatch), both said "fail" (FailMatch), program said "pass" and grader said "fail" (PassMismatch) and vice-versa (FailMismatch). (Left) Fallback refers to the program abstaining. (Middle) Fallback also includes cases where the program gives a borderline score of 2. (Right) Fallback also includes the program giving a score of 1 or 2. The red mismatches shrink and cost (fallback) increases as we move right.

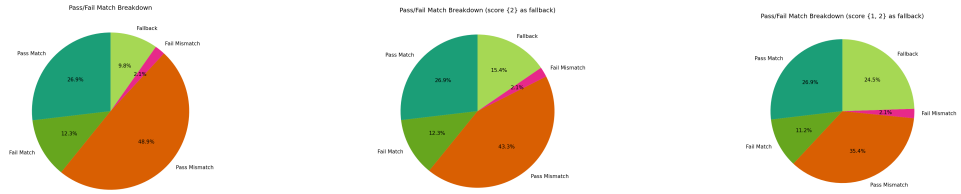


Figure 8: Percentage of grading instances where the program generated with 1 example and AGENTICGRADER both said "pass" (PassMatch), both said "fail" (FailMatch), program said "pass" and grader said "fail" (PassMismatch) and vice-versa (FailMismatch). (Left) Fallback refers to the program abstaining. (Middle) Fallback also includes cases where the program gives a borderline score of 2. (Right) Fallback also includes the program giving a score of 1 or 2. The red mismatches shrink and cost (fallback) increases as we move right.

be placed directly to the right of the existing MRR data." This task-specific rubric is an specialization of a general task-agnostic rubric "Layout, Placement: How clean is the placement of content?" to this task. Using specific low-level rubrics has been recognized as an effective way to use LLM-as-a-judge (Biyani et al., 2024). It is difficult to turn the general rubric to code, but the task-specific rubric above is more amenable to codification. The codification agent takes some sample output workbooks. One such sample output workbook is shown in Figure 10. Different outputs may use different column names for the newly added column. Our codification agent produces code that works as follows:

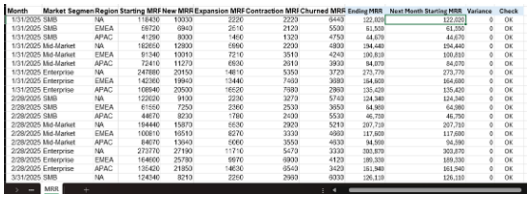
1. Find the correct sheet in the output workbook
2. Identify the rightmost column
3. Confirm that Ending MRR is added to the right
4. Ensure comparison is added.

For Step (1), the code extracts the sheet named "MRR" from the output workbook. This works because the sheet named "MRR" is already present in the input workbook that the task wants to edit. Similarly, the code for Step (2) finds a row that contains the strings "Month", "Starting MRR" and "Churned MRR", which is deemed as the header row and the column of the cell where "Churned MRR" is found is deemed the rightmost column. Again, these constant strings are directly pulled from the input workbook. Next comes the challenging part: Step (4) requires finding the new column that contains comparison of Starting MRR and Ending MRR. This column may be placed anywhere to the right (with some intermediate columns) and with any header. Here the codification agent relies on the sample output workbooks it gets in our approach: it uses the comparison column name in these samples to produce a vocabulary of common column headers, ["check", "comparison", "compare", "difference", "variance", ...], and this list is used in the code to detect the comparison column. While this may not work always, it will work on most of the output workbooks that a spreadsheet agent is likely to create for this task. This is the basic intuition behind our work. The code generated by the codifer agent is shown below:



Month	Market Segmen	Region	Starting MRR	New MRR	Expansion MRR	Contraction MRR	Churned MRR
1/31/2025	SMB	NA	118430	10030	2220	2220	6440
1/31/2025	SMB	EMEA	59720	6940	2510	2120	5500
1/31/2025	SMB	APAC	41290	8000	1450	1320	4750
1/31/2025	Mid-Market	NA	182650	12800	5990	2200	4800
1/31/2025	Mid-Market	EMEA	91340	10010	7210	3510	4240
1/31/2025	Mid-Market	APAC	72410	11270	6930	2610	3930
1/31/2025	Enterprise	NA	247880	20150	14810	5350	3720
1/31/2025	Enterprise	EMEA	142360	19940	13440	7460	3680
1/31/2025	Enterprise	APAC	108940	20500	16520	7680	2860
2/28/2025	SMB	NA	122020	9100	2230	3270	5740
2/28/2025	SMB	EMEA	61550	7250	2360	2530	3650
2/28/2025	SMB	APAC	44670	8230	1780	2400	5530
2/28/2025	Mid-Market	NA	194440	15870	5530	2920	5210

Figure 9: Input workbook for example



Month	Market Segmen	Region	Starting MRR	New MRR	Expansion MRR	Contraction MRR	Churned MRR	Ending MRR	Next Month Starting MRR	Variance	Check
1/31/2025	SMB	NA	118430	10030	2220	2220	6440	122,020	122,020	0	OK
1/31/2025	SMB	EMEA	59720	6940	2510	2120	5500	61,550	61,550	0	OK
1/31/2025	SMB	APAC	41290	8000	1450	1320	4750	44,670	44,670	0	OK
1/31/2025	Mid-Market	NA	182650	12800	5990	2200	4800	194,440	194,440	0	OK
1/31/2025	Mid-Market	EMEA	91340	10010	7210	3510	4240	100,810	100,810	0	OK
1/31/2025	Mid-Market	APAC	72410	11270	6930	2610	3930	84,070	84,070	0	OK
1/31/2025	Enterprise	NA	247880	20150	14810	5350	3720	273,770	273,770	0	OK
1/31/2025	Enterprise	EMEA	142360	19940	13440	7460	3680	164,600	164,600	0	OK
1/31/2025	Enterprise	APAC	108940	20500	16520	7680	2860	135,420	135,420	0	OK
2/28/2025	SMB	NA	122020	9100	2230	3270	5740	124,340	124,340	0	OK
2/28/2025	SMB	EMEA	61550	7250	2360	2530	3650	64,980	64,980	0	OK
2/28/2025	SMB	APAC	44670	8230	1780	2400	5530	46,750	46,750	0	OK
2/28/2025	Mid-Market	NA	194440	15870	5530	2920	5210	207,710	207,710	0	OK
2/28/2025	Mid-Market	EMEA	100810	10510	8270	3330	4900	117,600	117,600	0	OK
2/28/2025	Mid-Market	APAC	84070	13940	6060	3550	4930	94,590	94,590	0	OK
2/28/2025	Enterprise	NA	273770	27190	11710	5470	3330	303,870	303,870	0	OK
2/28/2025	Enterprise	EMEA	164600	25780	9970	6900	4120	189,330	189,330	0	OK
2/28/2025	Enterprise	APAC	135420	21850	14630	6540	3420	161,940	161,940	0	OK
3/31/2025	SMB	NA	124340	8210	2260	2990	6030	126,110	126,110	0	OK

Figure 10: Output workbook for example

```
from typing import Optional, Dict, Any
import openpyxl

def evaluate_criterion(initial_state_path: str,
    final_state_path: str) -> Dict[str, Any]:
    try:
        wbi = openpyxl.load_workbook(initial_state_path,
            data_only=False)
        wbf = openpyxl.load_workbook(final_state_path, data_only
            =False)
    except Exception as e:
        return {"result": "incomplete", "score": -1, "comments":
             f"Could not open workbook: {e}"}

    sheet = "MRR" if "MRR" in wbi.sheetnames and "MRR" in wbf.
        sheetnames else None
    if sheet is None:
        common = [s for s in wbi.sheetnames if s in wbf.
            sheetnames]
        sheet = common[0] if common else None
    if sheet is None:
        return {"result": "fail", "score": 0.0, "comments": "No
            common worksheet found."}

    wsi, wsf = wbi[sheet], wbf[sheet]
    header_row = right_col = None

    for r in range(1, min(wsi.max_row, 20) + 1):
        vals = [str(wsi.cell(r, c).value or "").strip().lower()
            for c in range(1, wsi.max_column + 1)]
        if any(v == "month" for v in vals) and any("starting mrr
            " in v for v in vals) and any("churned mrr" in v for
            v in vals):
            header_row = r
            for c in range(1, wsi.max_column + 1):
                if "churned mrr" in str(wsi.cell(r, c).value or
                    "").strip().lower():
                    right_col = c
            break

    if header_row is None:
        return {"result": "incomplete", "score": -1, "comments":
             "Could not identify the existing MRR header row."}

    [...]

    norm = [str(wsf.cell(header_row, c).value or "").strip().
        lower() for c in range(right_col + 1, wsf.max_column +
        1)]
```

```
    if not norm or "ending" not in norm[0] or "mrr" not in norm
        [0]:
       return {"result": "fail", "score": 0.0, "comments": "
           Ending MRR header is not placed directly to the right
            of existing MRR columns."}

    comp_words = ("check", "comparison", "compare", "difference
        ", "variance", "verification", "validation", "next month
        ")
    if not any(any(w in h for w in comp_words) for h in norm
        [1:]):
       return {"result": "fail", "score": 0.0, "comments": "No
           comparison/check field is aligned to the right of
           Ending MRR."}

    [...]

    if end_nonempty < threshold or comp_nonempty < threshold:
       return {"result": "fail", "score": 1.0, "comments": "
           Added columns are not filled/aligned down the
           existing data rows."}

    return {"result": "pass", "score": 4.0, "comments": "Added
        Ending MRR and comparison/check columns are directly to
        the right of existing MRR data and aligned with header/
        data rows."}
```

F Detailed Plots for Example Ablations

We present the same kinds of plots that we presented in Section 4 but for the case when the program was generated in 4 different settings: by giving only n=1 example, n=5 examples, n=10 examples, and n=15 examples. Figure 5 shows the plots for codification using n=15 examples, Figure 6 shows the plots for codification using n=10 examples, Figure 7 shows the plots for codification using n=5 examples, and Figure 8 shows the plots for codification using n=1 example. In each figure, the leftmost pie chart is the default fallback scheme – we fallback only when the program fails by raising an exception or explicitly returns a -1 score. The middle pie chart is the case where we also fallback when the code returns score=2. The right pie char is the case where we also fallback when the code returns either 1 or 2.

The main observation is that we see a similar trend as reported in the main paper. The trend toward the mismatches shrinking as the fallback increases (with more lenient fallback condition) can also be seen here.

We also notice that the pass mismatch percentage increases as the number n of examles decreases. This indicates that the increase in the number n of training examples improves held-out agreement.

Trading Time Saving for Accuracy

We envision codified rubrics as an optimization whose goal is to reduce (though not necessarily eliminate) the number of grading tasks that are eventually routed to the expensive AgenticGrader. The default logic for routing to AgenticGrader is simple: we route to AgenticGrader when the generated code abstains (because the program fails to complete successfully in some stipulated time or gives an explicit -1 score). However, we can modify the routing logic to trade some of the cost savings (from using the code aggressively whenever possible) for better agreement with AgenticGrader. A natural choice is to extend the base routing logic and also route to AgenticGrader when the code gives a borderline score of 2 (Fallback Strategy 1). Recall that the code produces a score in the range [0,...,4]. An even more liberal policy is to also route to AgenticGrader when the code gives a score of 1 or 2 (Fallback Strategy 2).

The middle and right panels of Figure 5 repeat this analysis under Fallback Strategy 1 and Fallback Strategy 2, respectively. The mismatch percentage decreases as we move from the base strategy to Fallback Strategy 1 to Fallback Strategy 2. We pay for these gains through an increase in the percentage of grading tasks routed to the expensive AGENTICGRADER.

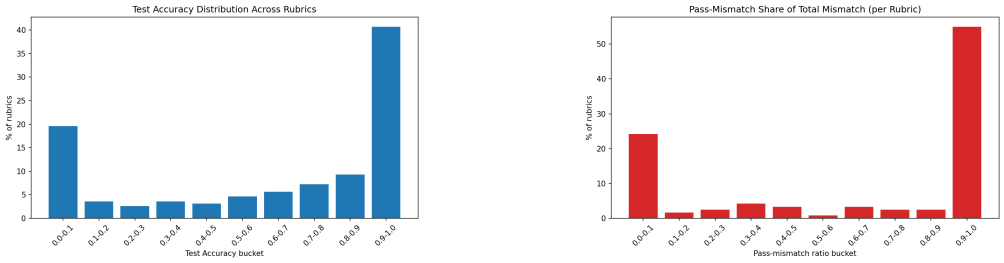


Figure 11: (Left) The distribution of pass rates across rubrics (Right)The distribution of the ratio of number of mismatching "pass" grade (PassMismatch) to the total number of mismatching grades (TotalMismatch) for the program generated with 15 training examples

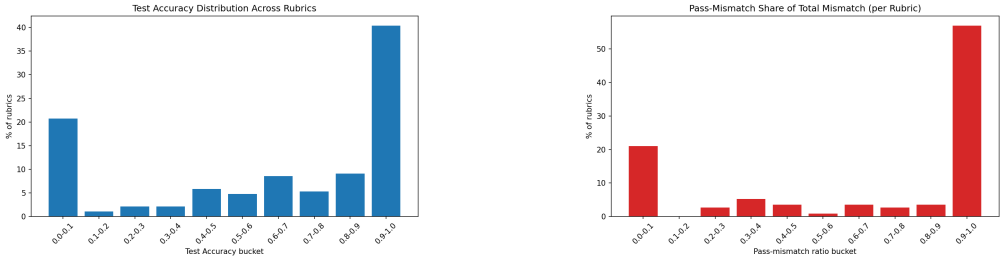


Figure 12: (Left) The distribution of pass rates across rubrics (Right)The distribution of the ratio of number of mismatching "pass" grade (PassMismatch) to the total number of mismatching grades (TotalMismatch) for the program generated with 10 training examples

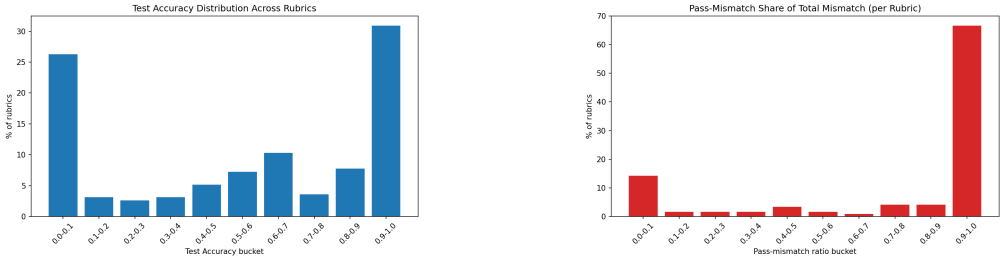


Figure 13: (Left) The distribution of pass rates across rubrics (Right)The distribution of the ratio of number of mismatching "pass" grade (PassMismatch) to the total number of mismatching grades (TotalMismatch) for the program generated with 5 training example

Figure 11 shows the distribution of pass rates across rubrics on the left and on the right, it shows the pass-mismatch ratio as defined in the main paper. Figure 12, Figure 13, and Figure 14, show the same plot for the n=10, n=5 and n=1 case.

TARGETED VS COMPOSED RUBRICS

A central question in rubric codification is its efficacy for different kinds of rubrics. In particular, it has been established by several studies that when using LLM-as-a-judge, it helps to start with rubrics that are sufficiently detailed and targeted. However, combining multiple small rubrics r into a combined rubric R can save cost. The combined rubric R is simply an NL rubric obtained by

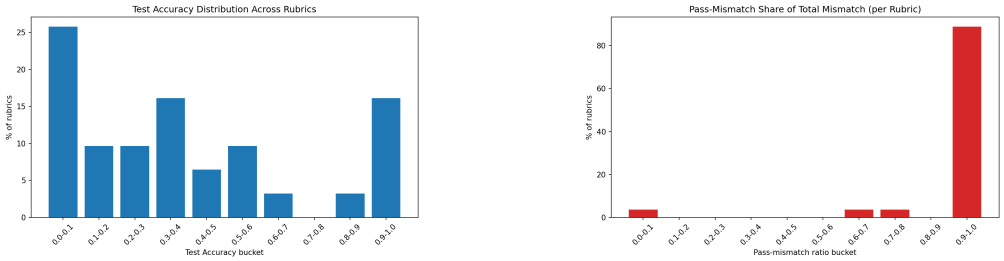


Figure 14: (Left) The distribution of pass rates across rubrics (Right)The distribution of the ratio of number of mismatching "pass" grade (PassMismatch) to the total number of mismatching grades (TotalMismatch) for the program generated with 1 training example

Variant	RM	ASM
Basic Rubrics	0.694	0.712
Combined Rubrics	0.230	0.433

Table 3: Effect of combining rubrics on codification: Codifying small targeted rubrics is better.

concatenating the strings for its constituent simple rubrics r. Rather than codifying individual simple rubrics r, one can wonder what happens if we directly codify the complex rubric R. Table 3 shows that such a variant, called Combined Rubrics, performs poorly than the baseline. Here we are using the version with explicit CEGIS turned off. The results show the importance of having rubrics at the right granularity to enable effective codification. Note that in this table, we are comparing codification of different rubrics by comparing their verdict and score match rates.

G Summary Statistics for Single Program Runs

The summary statistics for the runs are shown in Table 4. As expected, the number of tool calls and messages increase with the number of examples, with the CEGIS version having more than twice as many tool calls on average. Interestingly, the programs also appear to increase in length from 1 to 5 examples, plateau at 5-15 examples and again increase in the CEGIS version. The CEGIS version on average generates 5.87 candidate programs including it's final submission.

Cohort	Messages	Tool Calls	Code Chars	Code Lines	Non-blank Lines
1 Example	9.23	2.61	4719.19	103.94	91.90
5 Examples	16.06	6.03	6249.63	160.16	138.18
10 Examples	20.79	8.41	6508.62	166.54	144.03
15 Examples	20.89	8.44	6554.32	165.65	143.30
CEGIS Version	42.32	19.16	8703.55	233.97	201.87

Table 4: Synthesis process metrics per run

H Hyperparameters and Prompts

The models are sampled with a temperature of 1.0 and token limit of 100000 tokens. All experiments are performed with the reasoning variant of OpenAI's GPT-5 model. The models are allowed to make tool calls upto 25 times, with the CEGIS variant being allowed 25 calls each for a total of 6 candidate programs.

The prompt for the codifier is shown below.

```
System Prompt:
Your task is to synthesize deterministic Python grading code
   by learning from multiple labeled evaluation runs.
```

```
You may need to use tools to inspect workbooks, compare diffs,
    and read approach-detail artifacts before writing code.

# AVAILABLE TOOLS

- `run_python`: Execute Python code snippets to inspect
   workbook contents and compare values across artifacts.
  Use print statements to emit findings. State is not carried
     from one run to the next.
- `calculate_workbook_diff`: Compute additions and deletions
   between source and target workbooks.
  Use this to understand which workbook edits correlate with
     pass/fail labels.

# IMPORTANT

1. Work iteratively in small steps. Gather evidence from the
   provided runs before deciding on grading logic.
2. Learn what the grader is actually rewarding or penalizing
   from the labeled examples, not just what the criterion text
    suggests in isolation.
3. The grader output is noisy and not perfectly reliable so
   expect some inconsistency, resolve it as you see fit.
4. Use the available tools only to inspect the provided
   artifacts and infer reusable grading logic.
5. Your output must be executable Python code, not an
   explanation.
6. Prefer robust grading logic that generalizes across the
   observed runs rather than memorizing a single case.
7. You may use functions from the standard library or from the
    `openpyxl` library to implement your grading code. Keep
   the code minimal and simple, avoid writing functions
   already implemented in these libraries.
8. The entry point for the generated code should be a function
    named `evaluate_criterion` that takes paths to the initial
    and final workbooks and returns a dictionary with keys `
   result`, `score`, and `comments`.
9. Do not spend too much time trying to match the grader
   output perfectly since it may have inconsistent grading for
    formulae that look very similar. The goal is to learn the
   underlying intent and produce grading code that captures
   that intent in a generalizable way, not to reverse-engineer
    the idiosyncrasies of the specific runs provided.
User Prompt:
Here are the artifacts and labels available for synthesis.

Criterion:
{
  **MOIC metric clearly added and labeled**: MOIC appears as a
     clearly named investor-return metric (e.g., "MOIC") in the
      delivered output (workbook and/or final reported tables),
      and it is clearly associated with OSF scenario results (
     not hidden/ambiguous).
}

Runs:
[
  {
    "run_id": "run_001",
```

```
    "agent_result": agent_score >= 3,
    "agent_score": x ∈ {0,1,2,3,4},
    "agent_explanation": "Explanation from Grader",
    "artifacts": {
       "initial_state": path,
       "final_state": path,
       "grader_conversation": path,
       ...
    }
  }
  ...
]

You are generating replacement grading code by learning from
   multiple labeled runs.
Use the previously provided criterion and run artifacts (
   including run_id and temp paths).
Infer grader behavior and output only executable Python code.

Output requirements:
1) Implement evaluate_criterion(initial_state_path: str,
   final_state_path: str, artifacts: Optional[Dict[str, Any]]
   = None) -> Dict[str, Any].
2) Return keys: result, score, comments.
3) result in {pass, fail, incomplete}.
4) pass should map to score > 2; fail to 0..2; incomplete to
   -1.
5) Base decisions on evidence found via tool calls, not
   assumptions.
Respond with code only.
Final Instructions - CEGIS Version:
You are generating replacement grading code by learning from
   multiple labeled runs.
Use the previously provided criterion and run artifacts (
   including run_id and temp paths).
You are also given access to a test_execution tool that
   evaluates your candidate code on a hidden examples and
   returns only result/score comparisons.
The cegis-development set artifacts are intentionally hidden;
   use only the returned metrics and comments to iteratively
   improve your code.
Infer grader behavior and output only executable Python code.

Output requirements:
1) Implement evaluate_criterion(initial_state_path: str,
   final_state_path: str, artifacts: Optional[Dict[str, Any]]
   = None) -> Dict[str, Any].
2) Return keys: result, score, comments.
3) result in {pass, fail, incomplete}.
4) pass should map to score > 2; fail to 0..2; incomplete to
   -1.
5) Base decisions on evidence found via tool calls, not
   assumptions.
6) You decide when to stop refining. When you are ready to
   terminate, respond with a line containing only
   FINAL_EVALUATOR, then the code.
Until then, continue refining using tools as needed.
```

I DISCUSSION

Task-specific codification. Codification is feasible because we target a fixed benchmark task rather than a rubric that must generalize across tasks. Consider the rubric "Ensure MOIC is calculated as a ratio and not a difference." In isolation, this rubric is difficult to codify. However, for a specific task q and input workbook x, the space of plausible output workbooks y is much smaller. A codifier only needs to grade most outputs generated by SUT for that task, not arbitrary (x, y) pairs. This restriction makes rubric codification tractable.

Codification as caching. Codification can be viewed as an enhanced form of caching and the fallback to the agentic grader is akin to a cache miss. An expensive LLM judge labels examples, and those judgments are compressed into an executable program that can be reused across evaluations. This upfront cost is amortized over repeated benchmark runs, making codification particularly valuable for nightly regression testing and reinforcement learning workloads. However, codifiers may become stale as SUT evolves and the distribution of generated outputs changes, requiring periodic auditing and re-codification.

Committee of Programs The codification process can be modified to generate not one, but multiple programs for a given rubric. One could then use the committee of programs to each judge the output and the scores from the committee could be aggregated. We performed experiments to evaluate if using committee of programs would improve the quality of grading done by code. We used a committee of 3 programs. We needed a mechanism to aggregate the 3 grading verdicts that the committee generates. We tested different aggregation functions, including min, max, mean, median, and mode, for aggregating the grades produced by the committee members. Our main finding was that committee improves the quality of grading when the committee members are generated using a simple process – specifically, without using external CEGIS – but there is diminishing returns from using committees as we make the process of codification more complete; for example, by increasing the number of examples used and the use of CEGIS. Thus, while using committee of programs generally helps, it does not add value when we use our most favorable setting of hypoerparameters in the codification agent.