MacCode Lab: A Reproducible Single-Machine Harness for Execution-Guided Code Agents, and When Repair Actually Helps

Mayank Sharma
Indian Institute of Technology Jodhpur
Jodhpur, India
b23es1023@iitj.ac.in

Rohit Kumar Mourya
Indian Institute of Technology Jodhpur
Jodhpur, India
b23es1029@iitj.ac.in

ABSTRACT

Code-agent evaluation is increasingly repeated as models, prompts, and agent components change, rather than performed only once as a leaderboard exercise. Practitioners need evaluation that runs economically on local hardware and identifies which parts of an agent loop contribute measurable value. We present MacCode Lab, a fully local, resume-safe evaluation harness for execution-guided code agents that runs end to end on a single Apple Silicon workstation using quantized open-weight models under MLX. The harness ingests cached LiveCodeBench and HumanEval tasks, selects among an eight-style prompt palette with a difficulty-conditioned UCB1 bandit, generates and executes candidate programs in isolated subprocesses, classifies failures into a compact taxonomy, and feeds error-specific hints into an execution-guided repair loop. It persists per-task records, run status, and bandit state, allowing an unattended run to be inspected during execution and re-scored after completion.

We run two open-weight models through the harness: a 3-bit quantization of a general mixture-of-experts model (Qwen3.6-35B-A3B) and a 4-bit quantization of a code-specialized model (Qwen2.5-Coder-7B). The resulting findings concern evaluation reliability rather than model ranking. First, the value of execution-guided repair is model-dependent, not only workload-dependent. On the larger model, repair raises LiveCodeBench from 91 to 113 solved tasks on public tests; on the smaller model it produces no net gain. Second, public-test pass rates overestimate full-protocol performance by an amount that varies sharply across models. When we re-score every accepted solution against the full LiveCodeBench protocol including hidden tests, the larger model loses only three to seven solutions while the smaller model loses twenty-three to twenty-six. Public-only scoring assigns a substantially higher score to the smaller model; under the full protocol that gap nearly closes and, with repair, the observed ordering reverses. Our contribution is not a leaderboard claim but a reproducible instrument and a measured account of when repair helps, when it does not, and how public-test scores differ from full-protocol results.

1 INTRODUCTION

In practical development workflows, code-agent evaluation is repeated when a model provider ships an update, a prompt template is revised, or an engineering team changes the agent loop. This regime emphasizes different properties from one-time leaderboard evaluation. Evaluation must be economical enough to run frequently, robust enough to run unattended, and sufficiently instrumented to explain why a score changed rather than merely report that it changed. Those properties are the subject of this paper.

A local code-agent evaluation workflow is therefore a systems artifact, not merely a model invocation. It must load a model, manage benchmark data, generate code, execute it safely, classify failures, attempt repair, persist state, and run long enough to expose operational failure modes. These requirements are now tractable on current Apple Silicon workstations. A single machine with large unified memory can host a strong open-weight model and execute thousands of candidate programs without a cloud cluster, while long-running workloads still require careful orchestration. Compact mixture-of-experts models with relatively few active parameters make this a practical evaluation setting.

This paper documents MacCode Lab, an evaluation harness and agent loop for that setting, together with frozen local runs across two models and an independent re-verification of every reported number. The agent loop is deliberately simple. The harness samples a task from a cached benchmark stream, selects prompt styles with a difficulty-conditioned bandit, generates candidates, executes them against the task's tests, classifies failures, and supplies the failure signal to an error-aware repair prompt. Local JSONL caching, per-task records, a live status file, snapshot export, and watchdog supervision turn this loop into an instrument that a single researcher can run, inspect during execution, and reproduce within one day. We study two locally hosted open-weight models because repeated, incremental evaluation on a single workstation directly represents the operational setting considered here.

Two observations motivate the work. First, execution-based code benchmarks such as LiveCodeBench [1] establish what to measure, but not how a local agent should allocate effort over time on a real machine or how much of a reported gain survives strict evaluation. Second, recent benchmarks covering whole-program reconstruction [2], context retrieval [3], skill injection [4], and structured action spaces [5] shift attention from code emission alone toward context management, failure recovery, and efficiency over long horizons. MacCode Lab integrates these concerns into a single, auditable, Mac-native loop. We ask whether an execution-guided repair gain persists across models and how public-test scoring affects comparative conclusions.
Our contributions are the following.

- **A fully local, resume-safe evaluation harness** for execution-guided code agents on Apple Silicon. It uses MLX [7] on the GPU, a typed execution layer that dispatches stdin, functional,

and call-style tasks, a failure taxonomy, and long-running experiment orchestration with status tracking, snapshot export, and watchdog supervision.
- **Evidence that repair is model-dependent, not only workload-dependent.** Across two models on the same harness and tasks, execution-guided repair substantially improves the larger general model on LiveCodeBench but provides no net improvement for the smaller code-specialized model.
- **A public-versus-hidden trustworthiness analysis with a change in observed model ordering.** We re-run every accepted solution against the full LiveCodeBench protocol. The two models lose very different fractions of their public passes to hidden tests, and a comparison that looks decisive on public tests narrows or reverses under the full protocol.
- **A difficulty-conditioned prompt bandit** over an eight-style palette with error-specific repair hints, whose learned per-difficulty preferences we read directly off its persisted state.

2 SYSTEM DESIGN

2.1 Overview

Figure 1 summarizes the loop. A benchmark cache supplies tasks. The prompt bandit selects styles conditioned on task difficulty. The local MLX model generates candidates. The execution layer runs each candidate against the task's tests. The observed failure class drives the next repair prompt. A per-task record, the run status, and the bandit state are then written back to disk, and the loop repeats until the configured budget is exhausted. The principal design focus is the closed, persisted loop rather than the model itself. Every agent decision is logged in a form that can be replayed and re-scored after execution.

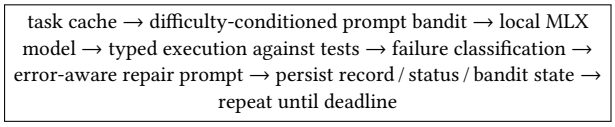

Figure 1: The MacCode Lab control loop. Every step is persisted so that runs can be inspected during execution and re-scored offline.

2.2 Benchmark Data Pipeline

The loader supports cached LiveCodeBench slices and a local HumanEval cache, and writes JSONL to disk so that repeated experiments do not re-download benchmark inputs. We use LiveCodeBench release v6 as the primary execution-based workload and HumanEval as a compact functional-correctness reference. For each task the cache stores the prompt, any starter code, the public test cases, the task difficulty, and the encoded hidden test cases. Keeping the hidden cases on disk is what later allows a completed run to be re-scored under the full evaluation protocol without re-querying the dataset.

Because the hidden cases are cached locally, their isolation deserves an exact statement. Hidden tests are used only for post-hoc re-scoring. They are never included in the generation prompt or the repair prompt, never used as bandit reward, never used for prompt or style selection, and never read by any code path that executes while candidates are being generated. During a run the worker sees only the task statement, any starter code, and the public tests. The full-protocol score is computed after generation has ended, by decoding the cached hidden cases and replaying accepted solutions against them.

2.3 Prompt Selection and Execution-Guided Repair

MacCode Lab uses an eight-style prompt palette: **direct** (the shortest correct answer), **careful** (reason through edge cases), **structured** (a clean implementation with explicit handling), **debugger** (test-driven repair behavior), **invariant** (identify and preserve the key invariant), **counterexample** (search for a failing edge case before coding), **template** (helper functions plus a direct main path), and **pragmatic** (robust, locally verifiable code).

Style choice is not static. The harness maintains a UCB1 bandit per difficulty bucket and selects the top-k styles for the current task. The budget is itself difficulty-aware, so easy tasks receive fewer initial styles and fewer repair rounds while hard tasks receive more aggressive exploration. In the experimental configuration, the worker generates several candidates in parallel rather than evaluating them strictly sequentially, thereby using the workstation's CPU and GPU concurrently. Section 3.4 reports the resulting preferences.

Repair prompts reuse the original task statement, the failing source, a failure summary, and a small set of error-specific hints. Syntax failures prompt a check of indentation and brackets, name errors prompt a check that symbols are defined, timeouts prompt a reduction in asymptotic complexity, and wrong-answer failures prompt a re-reading of the specification and its edge cases. The hints are intentionally lightweight. They form a feedback channel that opens after the execution trace has already exposed a flaw, not a hand-crafted solver.

2.4 Evaluation Protocol and Execution Layer

Correct execution dispatch is a prerequisite for the validity of every reported result. The layer dispatches on task type. Stdin and stdout tasks, such as Codeforces and AtCoder style problems, run the candidate as a program, feed each case on standard input, and compare output after a normalization that strips trailing whitespace on each line and surrounding blank lines. Functional LiveCodeBench tasks, such as LeetCode style problems, are wrapped in a generated harness that parses the encoded argument list, resolves either a free function or a `Solution` method as the entry point, invokes it once per case, and compares the normalized return value. Call-style tasks, used by HumanEval, append the dataset's own `check` function and assert against it. In every case the candidate runs in a temporary directory under a subprocess timeout, and a shared import preamble is prepended so that solutions relying on common standard-library names are not penalized for a missing import. This dispatch currently covers single-file Python tasks in these three shapes; repository-level, multi-file, and non-Python workloads are outside the harness's present scope.

Failures are classified by pattern over standard error and standard output into a compact taxonomy: syntax, assertion, name,

type, value, index, key, attribute, import, recursion, overflow, timeout, runtime, wrong answer, and pass. The taxonomy serves two purposes. It gives each run a compact failure profile, and it selects the repair hint without a separate learned diagnosis model.

A key evaluation distinction is that the harness scores candidates against visible public tests during generation, and such pass rates can be optimistic relative to hidden-test performance. Because the encoded hidden cases remain available for post-hoc evaluation, we re-score every accepted solution under the full protocol in Section 3.2. We recommend this separation for local harnesses that report execution-based scores.

2.5 Long-Running Experiment Orchestration

The experiment runner repeats the cached stream until a wall-clock budget elapses. The worker writes a live status file after every task, and a summary exporter can emit a fresh snapshot without stopping the worker, allowing inspection without interruption. A watchdog optionally restarts a stalled run and exports a final snapshot at the budget boundary. We evaluate two models under this configuration: Qwen3.6-35B-A3B, a general mixture-of-experts model, under local MLX 3-bit quantization, and Qwen2.5-Coder-7B-Instruct [8], a code-specialized model, under 4-bit quantization. Both use MLX on the Apple GPU as the backend, HumanEval and LiveCodeBench release v6 as workloads, greedy decoding (temperature 0, top-p 1.0), and resume-safe JSONL logs for tasks, bandit state, and summaries.

3 EXPERIMENTAL EVALUATION

We report frozen local run artifacts and an independent re-verification of every count (Section 7). Both models use the same harness, greedy decoding, public-test execution during generation, and JSONL record format. Table 1 records the machine, software, model provenance, and run configuration. LiveCodeBench runs cover the first 200 tasks of release v6, and HumanEval runs cover all 164 tasks. The 200-task slice is a deterministic prefix rather than a random sample: it can be cached once, resumed after interruption, and re-scored without a seed-dependent dataset view. This choice supports deterministic single-machine reproduction at the cost of statistical generality, as discussed in Section 6.

Table 1: Execution environment, model provenance, and run configuration for the frozen runs. Section 7 lists the exact commands.

Item	Value
Machine	Apple MacBook Pro, M5 Max (18-core CPU), 128 GB unified memory
OS at run time	macOS 26.4.1
Software	Python 3.12, `mlx` 0.31.2, `mlx-lm` 0.31.3
35B model	`andrevp/Qwen3.6-35B-A3B-3bit-MLX` (3-bit MLX)
35B revision	`794408c4b69a90c9a7b96d3fce6949cccf08df85`
7B model	`mlx-community/Qwen2.5-Coder-7B-Instruct-4bit` (4-bit MLX)
7B revision	`019cc73c45c770444708a6dd8690c66243cc5c80`
Quantization	pre-quantized MLX conversions used as released; no local conversion
Decoding	greedy: temperature 0.0, top-p 1.0
Max new tokens	512 (LiveCodeBench); 384 (HumanEval)
Repair budget	3 initial styles; 2 repair rounds, beam 2 (repair); 0 rounds (no repair)
KV cache	MLX maximum KV size 8192
Execution timeout	6 s per candidate execution
Scoring during run	public tests only
Scoring post hoc	accepted solutions replayed on public and hidden tests

The 35B LiveCodeBench repair run completed 200 tasks in roughly 2.4 hours of wall-clock time on a single workstation. The two models were not evaluated as a single preplanned batch. The smaller model was run first and only in part, the larger model was evaluated next, and its LiveCodeBench repair result motivated returning to the smaller model and evaluating it to completion. The harness's persisted, resumable state supported this directly, since the smaller model's runs resumed from their earlier checkpoints rather than restarting. We recomputed the resulting aggregate counts from the per-task records, and our re-verification recomputes them independently.

3.1 Public-Test Results

Table 2 reports the public-test counts for both models. On HumanEval, repair changes the score by at most one task in either direction for either model, consistent with a benchmark already near saturation. On LiveCodeBench the models respond differently. For the 35B, repair improves the final public-test score from a within-run initial of 89 to 113. For the 7B, repair increases the within-run score from 119 to 125, matching the independent no-repair result of 125 and providing no net advantage over plain sampling.

Table 2: Main results on public tests for both models. "Initial" is the first-attempt pass within each run. No-repair and repair are independent runs, so their initial counts differ slightly under greedy decoding because of state-dependent style selection.

Model	Benchmark	Variant	Tasks	Initial	Final
35B	HumanEval	No repair	164	151	151
35B	HumanEval	Repair	164	150	150
35B	LiveCodeBench	No repair	200	91	91
35B	LiveCodeBench	Repair	200	89	113
7B	HumanEval	No repair	164	152	152
7B	HumanEval	Repair	164	152	153
7B	LiveCodeBench	No repair	200	125	125
7B	LiveCodeBench	Repair	200	119	125

3.2 Public-Test and Full-Protocol Evaluation

The central robustness question is whether accepted solutions pass only the visible public tests or the full hidden suite. We re-ran every accepted solution against the full LiveCodeBench protocol, combining the public cases with the encoded hidden cases, which average roughly 13 to 14 per task. Table 3 and Figure 2 show marked differences between the two models.

For the 35B, attrition from public to hidden scoring is small: three solutions without repair and seven with repair. The repair advantage is preserved, moving from 88 to 106 under the full protocol. For the 7B, attrition is larger: twenty-three solutions without repair and twenty-six with repair, so a public score of 125 corresponds to 102 and 99 full-protocol passes. Two consequences matter for evaluation. First, repair improves the 35B under the full protocol (a gain of 18) but slightly reduces the 7B score (a loss of 3), so repair effectiveness must be evaluated for a specific model. Second,

the public-test comparison favors the 7B (125 against 91 to 113), whereas the full-protocol gap narrows without repair (102 against 88) and the observed ordering reverses with repair (99 against 106). Public-only and full-protocol evaluation therefore produce different comparative results.

Because each configuration is a single 200-task run, the estimates remain uncertain. Wilson 95% intervals on the headline rates span roughly ±7 percentage points; for the full-protocol repair comparison, 106/200 is 53.0% with interval [46.1, 59.8] against 99/200 at 49.5% with [42.6, 56.4]. The intervals overlap, so we do not claim a statistically significant cross-model ordering under the full protocol. The more robust observation is the attrition asymmetry: on the same task set, one model loses 3 and 7 of its publicly accepted solutions to hidden tests while the other loses 23 and 26.

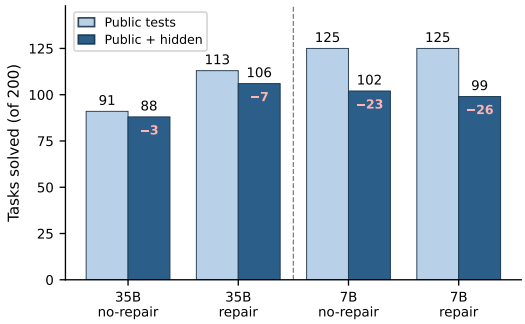

Figure 2: LiveCodeBench tasks solved under public-test scoring against the full public-and-hidden protocol. The larger model loses three to seven solutions to hidden tests; the smaller model loses twenty-three to twenty-six. Repair helps the 35B and not the 7B, and the cross-model ordering changes under the full protocol.

Table 3: Public-test against full-protocol (public and hidden) scoring on LiveCodeBench release v6. The smaller model loses far more of its public passes to hidden tests, and the cross-model comparison changes under the full protocol.

Model	Variant	Public	Full (hidden)	Loss
35B	No repair	91	88	−3
35B	Repair	113	106	−7
7B	No repair	125	102	−23
7B	Repair	125	99	−26
35B	Repair lift (full)		88 → 106	+18
7B	Repair lift (full)		102 → 99	−3

These results do not establish that either model is generally superior. The models differ in parameter count, quantization (4-bit against 3-bit), specialization (a code model against a general model), and likely exposure to benchmark-adjacent data. These differences make a single public-test score an insufficient basis for comparison; the public-to-hidden gap provides additional information about each evaluated system.

3.3 Performance by Task Difficulty

Table 4 breaks the LiveCodeBench runs down by difficulty under both scoring protocols. For the 35B, repair gains are largest on medium tasks and remain meaningful on hard tasks, and this pattern persists under hidden-test scoring. For the 7B, repair changes a few easy-task outcomes but loses ground on medium and hard tasks under hidden scoring. This behavior is consistent with its larger public-to-hidden gap: additional regeneration produces more solutions that pass visible cases without passing the full protocol.

Table 4: LiveCodeBench difficulty breakdown. "Init" and "Final" are public-test counts within each run, and "Full" is the final count under public and hidden scoring.

Model/Variant	Difficulty	Total	Init	Final	Full
35B No repair	Easy	69	49	49	49
35B No repair	Medium	97	41	41	39
35B No repair	Hard	34	1	1	0
35B Repair	Easy	69	49	53	53
35B Repair	Medium	97	37	52	48
35B Repair	Hard	34	3	8	5
7B No repair	Easy	69	50	50	49
7B No repair	Medium	97	63	63	48
7B No repair	Hard	34	12	12	5
7B Repair	Easy	69	51	51	51
7B Repair	Medium	97	60	60	44
7B Repair	Hard	34	14	14	4

3.4 Prompt-Selection Policy Analysis

Because the bandit state is persisted, its accumulated style preferences can be inspected directly. Figure 3 reports total reward per style within each difficulty bucket for the 35B LiveCodeBench repair run. The preferences vary by difficulty. On easy tasks the terse **direct** and **careful** styles dominate, while on medium tasks the more scaffolded **structured** and **template** styles receive greater reward. On hard tasks no style accumulates substantial reward, consistent with the low absolute solve rate. Total reward reflects both per-attempt success and selection frequency because UCB1 explores stronger arms more often. The qualitative ranking persists when reward is normalized by trials (for example, on medium tasks the scaffolded styles reach a per-attempt success rate near 0.25 against roughly 0.13 for the terse styles), so the figure should be interpreted as a directional signal rather than a precise success rate. This analysis demonstrates that the learned prompt-selection policy remains inspectable through persisted logs.

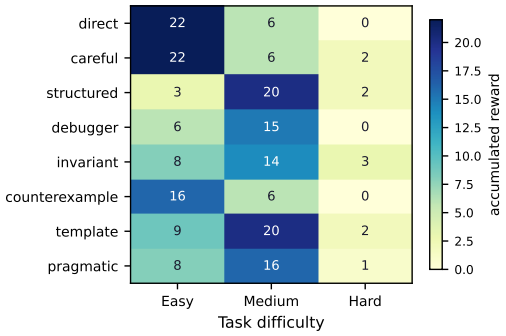

Figure 3: Accumulated bandit reward per prompt style by difficulty for the 35B LiveCodeBench repair run. The preferred styles shift with difficulty: terse styles on easy tasks, scaffolded styles on medium tasks, and little signal on hard tasks.

3.5 Failure-Mode Analysis

Table 5 shows the most common per-attempt outcomes in the 35B LiveCodeBench repair run. Syntax failures dominate the failed attempts because the model sometimes returns fenced code, partial explanations, or incomplete stdin programs. The extractor removes ordinary code fences, but failed repair attempts can still be syntactically invalid. This never inflates the final score, since candidates are always executed before being counted as solved, but it does identify a concrete engineering target. A stricter output contract or a structured edit interface would convert a large share of these wasted attempts into useful ones.

Table 5: Most common per-attempt outcomes in the 35B LiveCodeBench repair run, counted over attempts rather than tasks.

Outcome	Count
syntax	562
pass	207
wrong_answer	122
runtime	90
timeout	3
value	1

4 DISCUSSION

The snapshot supports four practical lessons for evaluating code agents after deployment.

First, a single workstation is a viable evaluation instrument. Both models run locally, the benchmark cache resides on disk, and the worker executes thousands of candidate programs without a cloud cluster, completing 200 LiveCodeBench tasks with a multi-style repair loop in a few hours. A single operator can therefore run, inspect, and reproduce this evaluation within one day.

Second, repair is not uniformly beneficial; its value depends on the model as well as the workload. On the near-saturated HumanEval benchmark, it changes either model's score by at most one task. On LiveCodeBench it adds 22 to 24 final public-test solves for the 35B and 18 under the full protocol, whereas for the 7B it provides no public-test gain and reduces the full-protocol score by three tasks. Repair utility should therefore be measured for each model-workload pairing rather than assumed to transfer.

Third, public-test scores should be interpreted alongside full-protocol re-scoring, and the size of the resulting gap is model-specific. The 35B loses three to seven tasks moving from public to hidden scoring, while the 7B loses twenty-three to twenty-six. A public-test comparison favors the 7B, whereas under the full protocol the gap narrows and, with repair, the observed ordering reverses. This does not establish that either model is generally superior. It shows that an execution-based evaluation should report how public-test scores change under the full protocol because public-only scoring can introduce different biases across systems.

Fourth, the current bandit is a baseline policy rather than a final design. Choosing among text styles yields interpretable per-difficulty preferences, but work on context retrieval [3] and structured actions [5] suggests extending the policy beyond text-only prompt variation. Relevant questions include which context to retain, which snippet to edit, and which intermediate state to preserve across retries. MacCode Lab provides a reproducible platform for evaluating these extensions, including why repair helps one model but not another, whether structured edits outperform free-form regeneration on hard tasks, and whether explicit reasoning about unseen cases reduces the public-to-hidden gap.

5 RELATED WORK

LiveCodeBench introduced contamination-free, execution-based code evaluation and foregrounded capabilities such as self-repair and code execution, making it a suitable workload for long-running local experiments and hidden-test analysis [1]. HumanEval remains the canonical compact functional-correctness benchmark and a useful local smoke test for a code-generation pipeline [6]. Beyond single-shot generation, recent work advances process-aware evaluation. ProgramBench measures holistic program reconstruction from documentation and behavior [2]. ContextBench argues that intermediate signals such as context recall and precision can be more informative than final success [3]. SWE-Skills-Bench finds that static skill injection often helps far less than expected, motivating adaptive context and skill routing [4]. CodeStruct shows that structured action spaces reduce brittleness and token waste relative to free-form text [5]. The failure profile in Table 5, dominated by malformed free-form output, supports that direction.

Two workshop themes are especially relevant. The first is post-market monitoring: persisted, replayable run records and live snapshot export allow behavior to be inspected and re-scored over time rather than only at submission. The second is trustworthy metrics. Our public-versus-hidden analysis validates an execution-based metric against a known source of discrepancy, and the observed change in model ordering shows that this discrepancy need not affect all systems uniformly. Although agent evaluation increasingly uses LLM-as-judge scoring, execution against hidden tests provides a grounded, falsifiable signal; we quantify its difference from the less costly public-only signal. Taken together, these works suggest that

code-agent evaluation is moving from single-shot generation toward process-aware systems, and MacCode Lab supports that shift by treating process auditability as a first-class artifact.

6 THREATS TO VALIDITY

The following limitations delimit the interpretation of the results.

Construct validity, that is, what "solved" means. During a run, "solved" means passing the public tests. We mitigate the optimism of that measure by re-scoring every accepted solution against the full hidden protocol in Section 3.2, and we recommend the full-protocol numbers as the headline. We do not claim these are comparable to leaderboard numbers produced under different scaffolds, decoding settings, or task subsets.

Cross-model comparison. The two models differ in parameter count, quantization level, and specialization, so we do not present their absolute scores as a controlled study of model scale. We use the comparison only to show that repair benefit and public-to-hidden attrition vary by model, which is a statement about the evaluation rather than a ranking of the models.

Internal validity, that is, run-to-run variation. No-repair and repair are independent runs, and the bandit's initial style selection is state-dependent, so initial-attempt pass counts differ slightly even under greedy decoding, for example 91 against 89 for the 35B and 125 against 119 for the 7B. Minor MLX nondeterminism may also contribute. We report each run's own initial and final counts and treat the within-run lift and the cross-run comparison as distinct quantities. Results are single-seed, so a small difference such as the one-task HumanEval gaps should not be over-interpreted. The repair condition also changes information and compute at once: it receives execution feedback and extra attempts. Without a paired design that fixes the first-attempt candidates and an equal-compute no-feedback baseline, we cannot separate the causal contributions of error-specific hints, additional samples, and bandit state. Neither ablation was run for this snapshot; the result establishes that the complete repair loop helps one model in this setting, and isolating each component is future work.

External validity, that is, scope. Results are specific to this machine, these quantizations, this MLX version, these decoding settings, and the first 200 release-v6 tasks. The study covers exactly two models on one hardware configuration, the LiveCodeBench slice is a deterministic prefix rather than a stratified random sample, and HumanEval is close to saturation for both models, so most of the informative signal comes from one benchmark. The counts should be read as a reproducible checkpoint on this slice, not as an estimate of full-benchmark performance and not as leaderboard scores. Beyond the Wilson intervals in Section 3.2 we report raw task counts; repeated seeds and bootstrap intervals over tasks would strengthen the repair-lift and attrition estimates and are future work. The harness itself currently evaluates single-file Python tasks through functional, stdin, and call-style dispatch, so nothing here speaks to repository-level or non-Python agent workloads. Wall-clock timing was affected by unrelated local processes, so pass and fail counts are stronger evidence than elapsed-time comparisons.

Contamination. LiveCodeBench mitigates contamination by release windowing. We use release v6, but we cannot rule out that either open-weight model has seen some tasks, and the two models may differ in this respect, which is a further reason to read the public-to-hidden gap rather than the public score. The repair against no-repair contrast is internal to each model and its tasks, so it is robust to a fixed level of contamination even where absolute scores are not.

7 CONCLUSION

We presented MacCode Lab, a reproducible single-machine harness for execution-guided code agents on Apple Silicon, and used it to evaluate whether repair gains persist across models. Running a 3-bit general mixture-of-experts model and a 4-bit code-specialized model through the same harness, we find that repair is approximately neutral on near-saturated HumanEval runs, substantially improves the larger model on LiveCodeBench, and provides no net public-test gain for the smaller model. Under the full hidden-test protocol, the larger model loses few accepted solutions while the smaller model loses substantially more; consequently, a comparison that appears decisive on public tests narrows and, with repair, reverses. The broader contribution is methodological. A local workstation can serve as an auditable, replayable instrument for process-aware code-agent evaluation, and model-specific full-protocol re-scoring should accompany public-test results. We release the harness, frozen run records for both models, and re-verification scripts to support reproducible experiments on structured edits, context caching, and adaptive repair policies.

REPRODUCIBILITY APPENDIX

Every count in this paper was re-derived from the released run records by an independent script. It recomputes initial and final pass counts from `records.jsonl`, re-executes every accepted solution against the cached public tests using the harness's own typed dispatch, and decodes the persisted hidden test cases to re-score every accepted solution under the full protocol. The same script runs against both models by pointing it at the corresponding run directories. The verification script treats hidden cases as a read-only scoring resource: it decodes and executes them only after generation is complete, so hidden outcomes cannot influence prompt choice, repair content, bandit state, or stopping decisions. The exact commands used to produce the frozen runs follow. The supplement contains the package, the run records (`records.jsonl`, `bandit.json`, `status.json`, `summary.json`, and accepted solutions) for both models, and the verification script.

Artifact checklist. Reproducing every count requires: (i) the harness package `mac_code_lab/` and the script `reverify_all.py`; (ii) the cached task files `data/livecodebench/release_v6_200.jsonl` and `data/humaneval/humaneval_all.jsonl`; and (iii) eight run directories: the four 35B runs under the `qwen36_35b_a3b_` prefix and the four 7B runs under `qwen25_7b_completed_runs/`, covering HumanEval and LiveCodeBench in no-repair and repair variants (the artifact manifest lists each directory by exact name). Each directory contains `records.jsonl`, `status.json`, `summary.json`, and the accepted solutions under `solutions/`; the 35B runs additionally persist `bandit.json`, which Figure 3 reads. Hidden test cases are stored base64- and zlib-encoded inside each task's record metadata, so the full-protocol re-score needs no network access.

Model provenance is pinned in Table 1: both models are pre-quantized MLX conversions used exactly as released, at the revisions listed there. Software versions are those of the archived run workstation.

- dataset prefetch: `python scripts/download_datasets.py --include-humaneval`
- overnight run: `python scripts/overnight_run.py`
- LiveCodeBench repair run (per model):
  `python scripts/run_benchmark.py --source livecodebench --release release_v6 --model-name MODEL --backend mlx --max-tasks 200 --no-repeat-until-deadline --initial-styles 3 --repair-rounds 2 --repair-beam 2 --mlx-max-kv-size 8192 --max-new-tokens 512 --temperature 0.0 --top-p 1.0 --timeout-s 6 --output-dir DIR`
- LiveCodeBench no-repair run: as above with `--repair-rounds 0`.
- HumanEval runs: as above with `--source humaneval --max-tasks 164 --max-new-tokens 384` (repair adds `--repair-rounds 2 --repair-beam 2`; no-repair uses `--repair-rounds 0`). `MODEL` is the 3-bit Qwen3.6-35B-A3B or 4-bit Qwen2.5-Coder-7B-Instruct path.
- re-verification (counts, public re-exec, hidden re-score), run from the repository root:
  `python reverify_all.py .`

REFERENCES

[1] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. *LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code.* arXiv:2403.07974, 2024.

[2] John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. *ProgramBench: Can Language Models Rebuild Programs From Scratch?* arXiv:2605.03546, 2026.

[3] Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. *ContextBench: A Benchmark for Context Retrieval in Coding Agents.* arXiv:2602.05892, 2026.

[4] Tingxu Han, Yi Zhang, Wei Song, Chunrong Fang, Zhenyu Chen, Youcheng Sun, and Lijie Hu. *SWE-Skills-Bench: Do Agent Skills Actually Help in Real-World Software Engineering?* arXiv:2603.15401, 2026.

[5] Myeongsoo Kim, Joe Hsu, Dingmin Wang, Shweta Garg, Varun Kumar, and Murali Krishna Ramanathan. *CODESTRUCT: Code Agents over Structured Action Spaces.* arXiv:2604.05407, 2026.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. *Evaluating Large Language Models Trained on Code.* arXiv:2107.03374, 2021.

[7] Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. *MLX: Efficient and Flexible Machine Learning on Apple Silicon.* https://github.com/ml-explore/mlx, 2023.

[8] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, et al. *Qwen2.5-Coder Technical Report.* arXiv:2409.12186, 2024.