MANUFACTURING A PROGRESS SIGNAL: SELF-EVALUATING CODING AGENTS FOR LONG-HORIZON PROGRAM SYNTHESIS FROM SCRATCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are increasingly undertaking long-horizon software development tasks that require hours of work. Program synthesis from scratch, as studied in PROGRAMBENCH, is particularly challenging when agents must infer behavior from usage documentation and an executable reference. Existing agents may lack concrete repair targets, lose track of explored and untested behaviors, and struggle to obtain trustworthy and timely progress feedback. We introduce CODEMAPS, a framework that enables coding agents to manufacture and update an explicit progress signal from executable tests grounded in reference behavior. The framework preserves tests in a persistent corpus to provide reproducible repair targets and regression checks, while an obligation ledger guides further probing toward untested documented behaviors. A new test selection strategy controls redundancy and corpus growth through behavioral classes, separate quotas for divergent and regression tests, and explicit admission and corpus limits. When results are averaged equally across the four model configurations, CODEMAPS achieves a mean hidden-test pass rate of 44.49%, compared with 35.58% for MSA-full, 32.63% for Claude Code, and 39.17% for SPECFIRST. These correspond to relative improvements of 25.02%, 36.35%, and 13.58%, respectively. Ablation studies support the contributions of corpus selection and the obligation ledger. Retrospective evaluations of candidate snapshots further present positive correlations between corpus and hidden-test pass rates, each averaged across tasks. These results indicate that the manufactured signal tracks improvements in average held-out performance during program synthesis. The code is available at https://anonymous.4open.science/r/codemaps-4F5B/.

1 INTRODUCTION

Coding agents based on large language models (LLMs) have evolved from single-file completion to sustained workflows spanning hours or even days, using tools to read, edit, compile, execute, and revise code within a live environment (Yang et al., 2024; Wang et al., 2025; Yang et al., 2026). The duration of tasks that frontier agents can complete autonomously at a 50% success rate, measured by the time required by human experts, has been doubling approximately every few months (Kwa et al., 2025). Evaluation has expanded in parallel, from resolving individual issues (Jimenez et al., 2024) to completing terminal-based workflows (Merrill et al., 2026) and constructing entire software libraries (Zhao et al., 2025). These longer task horizons are particularly relevant to practical software development, where meaningful changes typically require more than a single edit. Automating such work requires agents to sustain reliable performance across long sequences of interdependent decisions.

Among long-horizon software development tasks, building complete programs from scratch presents a substantial challenge (Adamczewski et al., 2026; Yang et al., 2026; Chen et al., 2026; Ding et al., 2025). Agents must infer required behavior, make architectural decisions, and implement and validate components. We study this setting using PROGRAMBENCH (Yang et al., 2026), which provides usage documentation and an execute-only reference program and requires agents to build a new implementation from scratch that matches its observable behavior. During synthesis, agents develop their own checks by probing the reference and using the resulting observations as expected results.

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.

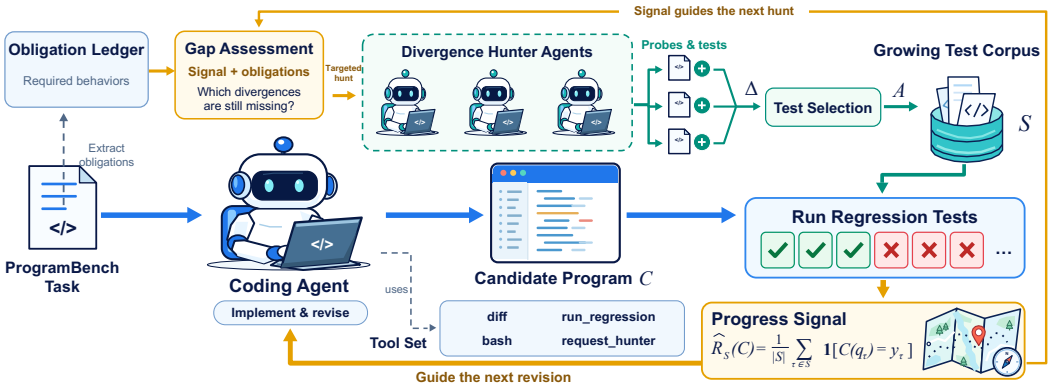

Figure 1: Overview of CODEMAPS. Given a PROGRAMBENCH task, a coding agent iteratively revises a candidate program C. The progress signal and obligation ledger jointly identify gaps and guide divergence hunter agents to synthesize a set of new tests Δ. The test selection strategy admits a subset $A \subseteq \Delta$ into the test corpus S. Evaluating C on S yields an updated progress signal $\widehat{R}_S(C)$ that guides subsequent code revisions and divergence hunting. The coding agent interacts with the environment through a tool set comprising `diff`, `run_regression`, `bash`, and `request_hunter`.

Hidden tests provide an independent evaluation of the resulting programs. Even the strongest frontier models fully resolve at most 4.5% of instances (ProgramBench, 2026).

Prior work has explored tool interface design and code localization for coding agents (Yang et al., 2024; Xia et al., 2025), trajectory search (Antoniades et al., 2025), and, more broadly, test-time compute allocation (Snell et al., 2025; Wu et al., 2025). MetaGPT and ChatDev organize software design, implementation, and testing through specialized agents working from user requirements (Hong et al., 2024; Qian et al., 2024). More directly, SpecFirst introduces a dedicated behavioral exploration phase before implementation: a specification agent probes the reference binary and records its findings in a structured natural-language specification for the coding agent (Chen et al., 2026). However, these methods either rely on existing repository tests, overlook or misinterpret program requirements, generate tests for a specific issue rather than maintain a continually expanding corpus for evaluating a program throughout a long synthesis run, or rely on a single initial prompt that leaves coding agents to decide what to do on their own. In particular, a natural-language specification records expected behavior but does not itself provide executable checks for assessing subsequent revisions or detecting regressions. Although SpecFirst permits further probing during implementation, omissions or misinterpretations in the specification may continue to mislead the coding agent unless they are identified and corrected.

Specifically, three limitations remain. First, agents have limited guidance for sustained refinement. Documentation describes intended functionality but does not identify which behaviors the current candidate implements incorrectly. Without reproducible evidence of unresolved divergences, agents may struggle to select concrete repair targets and verify whether revisions resolve them. They may consequently stop after implementing basic functionality or continue editing without clear evidence of progress.

Second, agents may struggle to retain exploration evidence and systematically broaden behavioral coverage. Observations gathered during development may remain scattered across the interaction history and become difficult to recover as the context evolves. Without a persistent record of previously explored behaviors and documented behaviors that remain untested, agents may repeatedly revisit familiar cases while overlooking other features, input conditions, and behavioral interactions.

Third, internal evaluation may not provide trustworthy and timely evidence of sustained progress. Redundant tests can cause frequently probed behaviors to dominate the corpus pass rate, allowing improvements on a narrow range of behaviors to have a disproportionate effect on the signal. A revision may also resolve some defects while breaking previously verified behaviors. Meanwhile,

evaluating a growing corpus increases execution costs and delays feedback. Assessing sustained progress therefore requires controlled test redundancy and replayable regression checks within a manageable evaluation budget.

To address these limitations, we introduce CODEMAPS, a framework that enables coding agents to manufacture an explicit progress signal for evaluating and improving their own candidate programs throughout long-horizon program synthesis from scratch. We instantiate the framework in the PROGRAMBENCH setting.

As illustrated in Figure 1, CODEMAPS constructs the signal from executable tests whose expected observations are obtained from the reference program. Reproducible divergences provide concrete repair targets, and replaying the corresponding tests allows the agent to check whether subsequent revisions resolve them. The framework retains these tests in a persistent corpus and tracks untested documented behaviors in an obligation ledger, preserving exploration evidence and guiding further probing. It periodically launches divergence hunter subagents to address unmet obligations, while the coding agent can request additional probing when it needs further repair targets. To improve the usefulness of the accumulated evidence, we introduce a new test selection strategy that prioritizes tests by behavioral class weight and imposes separate quotas for divergent and regression tests within each class. These quotas control redundancy and limit the influence of repeatedly probed behaviors on the progress signal. Regression tests help detect when revisions break previously verified behaviors, while limits on test admission and total corpus size control corpus growth and help keep evaluation manageable. Together, these mechanisms provide concrete feedback for continued refinement while preserving and broadening the behavioral evidence available to the agent.

We evaluate CODEMAPS on PROGRAMBENCH (Yang et al., 2026). Our baselines include `msa-full`, which resumes `mini-swe-agent` after early termination until either the time or step limit is reached, Claude Code prompted to use a team of collaborating agents, and SPECFIRST. When results are averaged equally across four base models, CODEMAPS achieves a mean hidden-test pass rate of 44.49%, compared with 35.58%, 32.63%, and 39.17% for these baselines, respectively. The improvements hold across all four models and all three difficulty groups. The relative improvement over SPECFIRST increases from 10.54% on easy tasks to 23.87% on hard tasks. Ablation studies support the contributions of the test selection strategy and the obligation ledger. Retrospective evaluations of candidate snapshots present strong positive correlations between corpus and hidden-test pass rates, each averaged across tasks. These results indicate that the manufactured signal tracks improvements in held-out performance on average.

In summary, our contributions are as follows:

- We introduce CODEMAPS, a framework that enables coding agents to evaluate their own candidate programs by manufacturing and updating an explicit progress signal to help identify repair targets and guide further exploration throughout long-horizon program synthesis from scratch. Experiments indicate that the held-out performance of the candidate continuously grows as the signal grows.
- We propose a test corpus selection approach that prioritizes tests by behavioral class weight under separate quotas for divergent and regression tests, and a total corpus capacity. The strategy promotes behavioral diversity, controls redundancy, and bounds corpus growth to support repeated evaluation throughout development. We prove that greedy selection maximizes the selection objective under the specified budget and quota constraints.
- We evaluate CODEMAPS on all PROGRAMBENCH tasks across four base models and demonstrate consistent improvements over `msa-full`, the multi-agent Claude Code baseline, and SPECFIRST. Ablation studies assess the contributions of corpus selection and the obligation ledger. We further analyze corpus composition and examine the correlation between corpus and hidden-test pass rates, each averaged across tasks.

2 METHODOLOGY

We next describe how the coding agent evaluates its own candidate program and manufactures a trustworthy progress signal grounded in observations from the reference program. The agent updates this signal throughout the run using accumulated test evidence and uses it to guide further refinements of the candidate program.

2.1 THE SELF-MANUFACTURED PROGRESS SIGNAL

Let $\mathcal{O}$ be the reference program, accessible only through execution, and let C be the candidate program. A query $q = (s, x) \in \mathcal{Q}$ consists of a pre-execution state s and an invocation x. The state specifies the execution conditions controlled by the evaluation protocol, including the file system, symlink structure, environment variables, working directory, and terminal configuration. The invocation specifies the argument vector and standard input, or a sequence of commands. Executing a program on q produces an observation $y = (y_{\text{out}}, y_{\text{err}}, y_{\text{exit}}, s') \in \mathcal{Y}$, consisting of standard output, standard error, exit status, and the observed post-execution state.

We model a program P using set-valued semantics:

$$[\![P]\!] : \mathcal{Q} \to 2^{\mathcal{Y}}, \qquad [\![P]\!](q) = \{y \in \mathcal{Y} \mid P \text{ may produce } y \text{ on } q\}. \tag{1}$$

For a deterministic program, repeated executions of the same query yield the same observation, such as $\mathcal{O}(q) = y$. A nondeterministic program may admit multiple observations, such as $[\![\mathcal{O}]\!](q) = \{y_1, y_2\}$. These semantics describe possible observations without assigning them probabilities.

A query q is *settled* if $|[\![\mathcal{O}]\!](q)| = 1$. We screen queries by repeatedly executing the reference under the same pre-execution state and retaining those whose observations are identical. Repeated execution helps identify flaky queries. Agreement across these executions provides empirical evidence of stability, but does not rule out the possibility that the reference program may produce a different observation on the same query.

A *test case* is a pair $\tau = (q, y)$, where q is settled and y is an observation obtained by executing $\mathcal{O}$. The agent proposes the query, while the expected observation is measured from $\mathcal{O}$.

Let S be a finite, nonempty corpus of test cases. The `diff` tool executes $\mathcal{O}$ and C on the same query $q = (s, x)$, with both executions starting from the same pre-state s. It compares the observations and records reproducible divergences as candidate tests $\tau = (q, y)$, where y is produced by $\mathcal{O}$. Selected tests are then added to S. We define the empirical progress signal as

$$\widehat{R}_S(C) \;=\; \frac{1}{|S|} \sum_{\tau \in S} \mathbf{1}\big[C(q_\tau)) = y_\tau\big], \tag{2}$$

Equality requires agreement across all components of the observation, such as standard output, standard error, exit status, and observed post-state.

The external objective is $\text{score}_T(C)$, the pass rate on a hidden test suite T inaccessible to the agent. The environment permits execution queries to $\mathcal{O}$ but provides neither a correctness nor an equivalence oracle for C. We use $\widehat{R}_S(C)$ as an observable surrogate for guiding program construction from scratch. The goal is not to infer the particular queries in T, but to construct S so that it captures a broad and diverse portion of $\mathcal{O}$'s behavior. We therefore aim for improvements in $\widehat{R}_S$ to transfer to improvements in score_T. The `run_regression` tool replays the current corpus and reports the pass rate, failed queries, and observation divergences, providing concrete evidence for the next repair.

To make the progress signal more trustworthy, we broaden behavioral coverage (§2.2) and control test redundancy (§2.3), aiming to better align improvements in $\widehat{R}_S(C)$ with gains in $\text{score}_T(C)$.

2.2 BREADTH FROM AN OBLIGATION LEDGER AND TEST CASE SYNTHESIS

The coding agent synthesizes queries to identify behavioral divergences between the candidate and reference programs. To guide further test synthesis, we use a checklist of documented behaviors to track which probing requirements remain unmet. Based on these unmet requirements, we launch a batch of divergence hunter subagents to generate additional queries targeting the corresponding behaviors. The subagents execute both programs on the same queries and use the observed divergences to construct candidate test cases.

Extracting documented obligations. We obtain documentation from the reference program's own help output by invoking `-help` and recursively requesting help for each documented subcommand. We use regular expressions to extract documented subcommands, options, and explicitly listed option values. We then construct a set of *obligations* Φ. For each option with explicitly listed values, we

create one obligation per value. For each remaining option, we create an obligation. We also create one obligation for each documented subcommand. For example, `-color [always|never|auto]` produces three obligations, one for each listed value. We construct Φ once and keep it fixed throughout the run.

Each obligation specifies a behavior to probe and an execution requirement for counting it as tested. For a subcommand obligation, the reference execution must reach the subcommand's functionality rather than stop at argument parsing. Once this requirement is met, the obligation is satisfied even if the candidate fails the test. Obligation coverage records which behaviors have been tested, whereas test results indicate whether the candidate reproduces the reference on those queries.

Tracking unexplored documented behaviours. We define the remaining obligation set as

$$D(S) = \{\phi \in \Phi : \phi \text{ is not satisfied by any test in } S\}.$$

We recompute $D(S)$ from the current retained corpus. Unmet obligations provide directions for further probing.

With Φ and the execution requirement for each obligation fixed, any obligation satisfied by the current corpus remains satisfied when more tests are added. The set of unmet obligations therefore satisfies

$$S \subseteq S' \quad \Longrightarrow \quad D(S') \subseteq D(S).$$

Thus, the number of unmet obligations, $|D(S)|$, can only decrease or remain unchanged as tests are added. When removing tests, we retain at least one qualifying test for each obligation already satisfied, ensuring that previously established coverage is preserved. An empty $D(S)$ indicates that all extracted obligations have been satisfied, but does not imply that all behaviors of the reference program have been tested.

CODEMAPS periodically launches divergence hunter subagents to address obligations that remain unmet by the current test corpus. In addition to the divergence hunter subagents launched to address unmet obligations, the main coding agent can invoke the `request_hunter` tool to launch a dedicated divergence hunter subagent when it needs to identify additional divergences. Each subagent operates within a short time budget and a limited number of turns. It synthesizes queries and executes both the reference and candidate programs on the same queries to identify behavioral divergences. The resulting test cases are submitted to the selection procedure for possible inclusion in the corpus.

2.3 CORPUS SELECTION AS COVERAGE MAXIMIZATION

Unconstrained test accumulation can make repeatedly probed behaviors dominate the signal and increase regression costs. For example, perturbing one flag may yield many distinct inputs that exercise the same behavior. We select tests by behavioral class under quotas and budgets, promoting diversity while bounding corpus growth.

Behavioral classes. We group test cases into *behavioral classes* using a signature function σ that captures interaction structure while abstracting away selected argument values and execution-specific details. Tests share the same *behavioral class* iff their signatures are equal. We write $c(\tau) := \sigma(\tau)$ for the class label of test case τ. For $\tau = ((s, x), y)$, define

$$\sigma(\tau) = \begin{cases} (\text{subcommand}, \text{flags}, y_{\text{exit}}, \text{skel}(y_{\text{err}})) & \text{if the reference rejects } q, \\ (\text{subcommand}, \text{flags}, \text{form}(q), \text{modality}(y)) & \text{otherwise.} \end{cases} \tag{3}$$

Both branches include subcommand, the subcommand specified by the query, and flags, the set of option names used, excluding their values. For rejected queries, skel abstracts variable diagnostic details; otherwise, form summarizes input structure and modality describes how the reference result is expressed. The branches distinguish rejection patterns from successful behaviors that may share an exit status and an empty error stream. Appendix C provides further details on the behavioral signatures.

The test selection problem. Let Δ be a finite set of candidate tests disjoint from the current corpus S. Partition it into divergent tests Δ_{div}, on which the candidate disagrees with the reference, and regression tests Δ_{reg}, on which their observations agree. Both types use reference observations.

Admission types remain fixed after repair, so a repaired divergent test can detect regressions without changing quota counts.

Let $\mathcal{C}_{\text{sel}} = \{c(\tau) : \tau \in S \cup \Delta\}$. For each class c and type $t \in \{\text{div}, \text{reg}\}$, let $n_{c,t}$ count retained tests of that class and type, and $\text{cap}_t(c)$ be their quota. For $A \subseteq \Delta$, define $A_{c,t} = \{\tau \in A \cap \Delta_t : c(\tau) = c\}$. The admission budget B limits new tests per round; M bounds the total corpus. Assuming S satisfies all quotas and $|S| \le M$, select A to maximize its total class weight:

$$\begin{aligned} \max_{A \subseteq \Delta} \quad & F(A) = \sum_{c \in \mathcal{C}_{\text{sel}}} w_c\big(|A_{c,\text{div}}| + |A_{c,\text{reg}}|\big) \\ \text{s.t.} \quad & |A| \le B, \qquad |S| + |A| \le M, \\ & n_{c,t} + |A_{c,t}| \le \text{cap}_t(c), \quad \forall c \in \mathcal{C}_{\text{sel}},\ t \in \{\text{div}, \text{reg}\}. \end{aligned} \tag{4}$$

Here $w_c > 0$ is fixed during selection; higher weights favor newly discovered classes and output modalities beyond standard streams. The two types have independent quotas, while both count toward B and M. All quotas and budgets are nonnegative integers.

Proposition 1 (Optimality of greedy selection)**.** *Assume that the current corpus S satisfies all class quotas and $|S| \le M$. Each test belongs to exactly one behavioral class, and its classification as divergent or regression remains fixed during selection. The corpus S, all limits, and class weights remain fixed during selection, and all class weights are positive. Greedy selection processes all candidates in decreasing order of class weight. Among tests with the same class weight, divergent tests are considered before regression tests. A test is admitted if and only if its addition satisfies the class quotas, the admission budget B, and the total corpus capacity M. This procedure maximizes the objective in Equation 4. Appendix D.2 gives the full proof.*

Selection stops when B tests have been admitted, capacity M is reached, or no feasible candidate remains. Appendix D.1 expands the notation and quota semantics.

3 Evaluation

This section presents the experimental evaluation of CodeMaps. We evaluate its performance across different base models, examine the contributions of the test selection strategy and obligation ledger, and analyze the correlation between the manufactured progress signal and held-out pass rates.

3.1 Experimental Setup

Models. We evaluate CodeMaps and all baselines on the same four base models: Qwen-3.7-Plus (high reasoning effort), Seed-2.1-Turbo (medium reasoning effort), GPT-5.4-mini (medium reasoning effort), and Qwen-3.5-35B-A3B (reasoning enabled). These models differ in model family, parameter scale, open-source availability, reasoning effort, and data cutoff, providing a basis for assessing the robustness of our results to model choice. We access all models through official provider APIs and set $\text{temperature} = 0$ throughout to reduce sampling variability when comparing methods.

Environment and budget. We follow the official ProgramBench container configuration. Each program synthesis run takes place in a network-isolated cleanroom container with 20 CPUs and 60 GB of memory. Evaluation containers have 10 CPUs and 20 GB of memory. All agent runs receive the same per-task budget of 6 hours of wall-clock time and 1,000 agent turns, with one graded submission. To balance the number of retained tests against regression-testing cost, we set the per-round admission budget to 15 test cases and the total corpus capacity to 300 test cases. For the sub agents, we use 4 sub agents as divergence hunters.

Evaluation benchmark and grading. We use the official ProgramBench grader without modification and report the mean hidden-test pass rate across all 200 tasks. The benchmark has a median of 770 tests per task and a mean line coverage of 79.7%. Submissions that fail to build receive a score of zero and remain included in the overall average. The hidden test suites are never exposed to the agent, preventing their direct use in constructing the progress signal.

Table 1: Mean hidden-test pass rates (%) on PROGRAMBENCH, grouped by task difficulty. For each model and method, we average the pass rates across the N tasks in the corresponding difficulty group. The Overall rows report the mean across all 200 tasks. Tasks with invalid submissions and failed evaluations are assigned a pass rate of zero and remain included in the averages.

Model	Difficulty	N	MSA-full	Claude Code	SpecFirst	CODEMAPS (Ours)
Qwen-3.7-Plus	Easy	28	47.87	41.97	62.05	65.78
	Medium	143	36.58	35.81	41.65	45.95
	Hard	29	20.18	21.32	20.59	26.39
	Overall	200	35.78	34.57	41.45	45.89
Seed-2.1-Turbo	Easy	28	54.16	42.52	69.51	75.08
	Medium	143	44.70	36.97	48.84	54.54
	Hard	29	24.66	23.27	26.06	31.91
	Overall	200	43.12	35.76	48.43	54.13
Qwen-3.5-35B-A3B	Easy	28	30.21	29.14	33.76	43.14
	Medium	143	22.68	18.18	25.61	32.36
	Hard	29	14.46	12.06	14.51	18.67
	Overall	200	22.54	18.83	25.14	31.88
GPT-5.4-Mini	Easy	28	52.80	58.17	54.41	58.88
	Medium	143	42.56	42.48	42.98	47.38
	Hard	29	21.14	19.46	22.79	27.02
	Overall	200	40.89	41.34	41.65	46.04

Baselines. We compare CODEMAPS with three baselines. The first, `MSA-full`, is based on `mini-swe-agent` under the official PROGRAMBENCH configuration. We additionally instruct the agent to continue working until either the six-hour time budget or the 1,000-step limit is reached. If it stops early, we resume it within the remaining budget until either limit is reached. The second baseline uses Claude Code (2.1.227), which we prompt to follow the N-agent team paradigm described in the Claude Opus 5 system card (Anthropic, 2026). The third baseline is SPECFIRST (Chen et al., 2026), which separates behavioral specification elicitation from code synthesis. Since its original implementation is not publicly available, we reimplement it following the paper. Our reimplementation reproduces the reported behavior for models such as GPT-5.4-mini.

3.2 MAIN RESULTS: EFFECTIVENESS OF CODEMAPS

Table 1 presents the results of comparing CODEMAPS with `msa-full`, Claude Code, and SPECFIRST across four base models on all 200 PROGRAMBENCH tasks. Averaged equally across the four models, CODEMAPS achieves a mean hidden-test pass rate of 44.49%, compared with 35.58% for `msa-full`, 32.63% for Claude Code, and 39.17% for SPECFIRST. Across individual models, CODEMAPS improves over `msa-full` by 5.15 to 11.01 percentage points, with relative improvements ranging from 12.59% to 41.44%. Compared with Claude Code multi agent, the absolute gains range from 4.70 to 18.37 percentage points, with relative improvements of 11.37% to 69.30%. Compared with SPECFIRST, the absolute gains range from 4.39 to 6.74 percentage points, with relative improvements of 10.54% to 26.81%. The highest overall pass rate is achieved with Seed-2.1-Turbo at 54.13%, followed by GPT-5.4-Mini at 46.04%. These consistent improvements across all four models indicate that the benefits of CODEMAPS are not limited to a particular base model.

Performance by task difficulty. CODEMAPS outperforms all three baselines across every combination of base model and difficulty level. Averaged equally across the four base models, CODEMAPS achieves a mean pass rate of 60.72% on easy tasks, exceeding SPECFIRST by 5.79 percentage points, corresponding to a relative improvement of 10.54%. On medium tasks, CODEMAPS achieves a mean pass rate of 45.06%, outperforming SPECFIRST by 5.29 percentage points, corresponding to a relative improvement of 13.30%. On hard tasks, CODEMAPS achieves a mean pass rate of 26.00%, exceeding SPECFIRST by 5.01 percentage points, corresponding to a relative improvement of 23.87%. Although absolute performance declines with task difficulty, the gains over SPECFIRST remain between 5 and 6 percentage points and correspond to larger relative improvements on harder tasks.

Table 2: Ablation study of test selection and the obligation ledger. The W/O SELECTION variant replaces the proposed test-selection procedure with random test selection. The W/O SELECTION & LEDGER variant additionally removes the obligation ledger. We report mean hidden-test pass rates (%) across all 200 tasks. The highest pass rate for each base model is shown in bold.

Model	CODEMAPS	w/o Selection	w/o Selection & Ledger
Qwen-3.7-Plus	**45.89**	43.74	42.53
Seed-2.1-Turbo	**54.13**	52.21	50.81
Qwen-3.5-35B-A3B	**31.88**	29.97	28.32
GPT-5.4-Mini	**46.04**	43.78	42.27

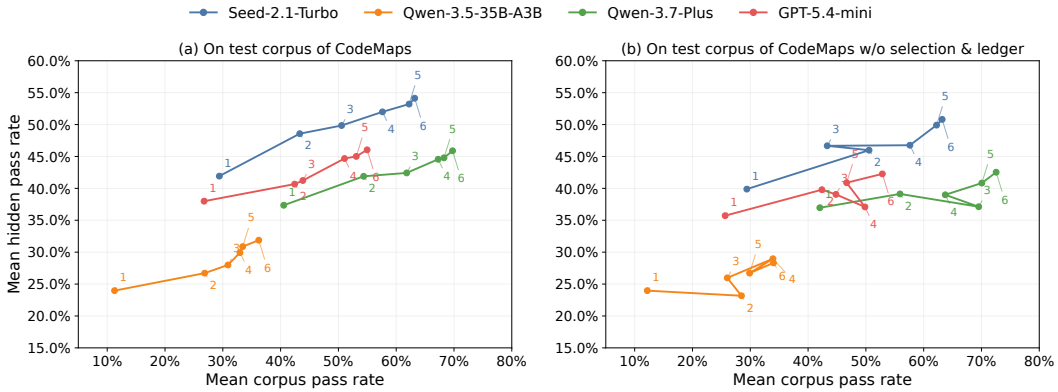

Figure 2: Correlation between the manufactured progress signal and held-out performance. The left and right panels present results for CODEMAPS and the *w/o Selection & Ledger* variant, respectively. For each base model, corpus and hidden-test pass rates are averaged separately across tasks at each checkpoint. We uniformly select five intermediate candidate snapshots from each run and include the final candidate as the sixth checkpoint. Each color represents a base model. Indices 1–6 indicate chronological order, with larger indices denoting later snapshots.

3.3 ABLATION: CORPUS CONSTRUCTION AND OBLIGATION LEDGER (RQ2)

We evaluate the contributions of test selection and the obligation ledger through two ablation studies. The W/O SELECTION variant replaces the proposed test selection procedure with random selection of candidate tests. The W/O SELECTION & LEDGER variant additionally removes the obligation ledger that guides probing toward unmet documented requirements. For all variants, we fix the total corpus capacity at $M = 300$ and the admission budget at $B = 15$. All other hyperparameters remain unchanged, including the number of divergence hunter subagents.

Table 2 presents the mean hidden-test pass rates across all 200 tasks for each of the four base models. Replacing the proposed selection procedure with random selection reduces pass rates by 1.91 to 2.26 percentage points, with an average decrease of 2.06 percentage points across models. The decline occurs with every base model, indicating the effectiveness of the proposed test selection strategy.

Removing the obligation ledger from the random selection variant further reduces pass rates by 1.21 to 1.65 percentage points, with an average decrease of 1.44 percentage points across models. This additional decline across all four models indicates the contribution of probing guided by unmet documented requirements under random test selection. Together, these results present that the full framework achieves higher pass rates than either ablated variant across all evaluated base models. We further analyze the breadth of the resulting test corpora in subsection E.1.

3.4 DOES THE MANUFACTURED SIGNAL ALIGN WITH HELD-OUT PERFORMANCE?

We next measure the correlation between the manufactured progress signal $\widehat{R}_S(C)$ and the hidden-test pass rate $\mathrm{score}_T(C)$. For each run, we select five intermediate candidate snapshots uniformly across

the run and include the final candidate as the sixth checkpoint. We retrospectively evaluate all six snapshots on the constructed test corpus S and hidden PROGRAMBENCH test suite T. The hidden suite and its evaluation results are never exposed to the agent.

For each base model and checkpoint, we separately average the corpus pass rates and hidden-test pass rates across the evaluated tasks. We then compute Pearson's correlation coefficient (Pearson, 1896) between the two resulting sequences of six checkpoint means. We also examine whether the two mean pass rates change in the same direction between consecutive checkpoints. A coefficient close to 1 indicates a strong positive linear association between the two sequences.

Figure 2 presents that both mean pass rates increase together under CODEMAPS across all four base models. Pearson correlations range from 0.934 to 0.988, and both scores increase in every consecutive checkpoint interval for each model. From the first checkpoint to the final candidate, mean corpus pass rates increase by 24.95 to 33.80 percentage points, while mean hidden-test pass rates increase by 7.91 to 12.21 percentage points. These results indicate that the manufactured signal tracks improvements in average held-out performance during program synthesis.

This alignment does not imply that the corpus pass rate accurately estimates the hidden-test pass rate. For Seed-2.1-Turbo, the mean corpus pass rate at the first checkpoint is 29.40%, compared with a mean hidden-test pass rate of 41.92%. At the final checkpoint, the corresponding rates are 63.20% and 54.13%. Thus, corpus pass rates can be either lower or higher than hidden-test pass rates, even when their changes consistently follow the same direction.

The W/O SELECTION & LEDGER variant exhibits less consistent alignment, with Pearson correlations ranging from 0.620 to 0.936. Changes in the two mean pass rates agree in direction in 12 of the 20 consecutive checkpoint intervals. Increases in corpus pass rate sometimes coincide with decreases in hidden-test performance. For example, between the second and third checkpoints with GPT-5.4-Mini, the mean corpus pass rate increases from 42.39% to 44.82%, while the mean hidden-test pass rate decreases from 39.79% to 39.05%. Compared with this combined ablation, the full framework exhibits stronger correlations and more consistent agreement in the direction of score changes. The comparison indicates that the proposed selection procedure produces corpora whose scores more consistently track candidate improvements across the runs.

4 RELATED WORK

Program synthesis using large language models. Program synthesis has been extensively studied over the past few decades. Classical program synthesis constructs programs from formal specifications through symbolic search and constraint solving (Solar-Lezama et al., 2006; Alur et al., 2013; Shen & Rinard, 2019; Li et al., 2025b). Several studies on program synthesis using LLMs have explored the capabilities of these models in solving simple programming tasks (Chen et al., 2021; Austin et al., 2021; Li et al., 2026b;a). Subsequent coding agents incorporated tools and execution feedback, enabling models to inspect repositories, edit files, execute tests, and iteratively revise their implementations (Yang et al., 2024; Wang et al., 2025). Frameworks such as ChatDev organize software development through specialized agents responsible for design, implementation, and testing (Qian et al., 2024). These developments have expanded the scope of synthesis to constructing complete codebases through sustained cycles of implementation, testing, and refinement (Zhao et al., 2025; Yang et al., 2026; Adamczewski et al., 2026; Ding et al., 2025; Chen et al., 2026). Such tasks extend beyond generating individual functions to making architectural decisions, integrating components, and sustaining implementation and validation over long sequences of interactions.

5 CONCLUSION

We introduce CODEMAPS, a framework for manufacturing progress signals during long-horizon program synthesis from scratch. It combines tests grounded in reference behavior, an obligation ledger, and test selection under quotas to guide continued refinement. Experiments on PROGRAMBENCH across four model configurations show improvements over baselines. Retrospective snapshot analyses further reveal positive correlations across checkpoints between corpus and hidden-test pass rates averaged across tasks.

AI USE STATEMENT

In this work, we used generative AI tools to assist with translation. We have not used generative AI tools for any other tasks requiring disclosure. Additionally, we used generative AI tools to polish the writing and to create or modify software code. We have reviewed all AI-assisted work, including the translated and polished text and the generated or modified code. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Tom Adamczewski, David Owen, David Rein, Florian Brand, Giles Edkins, Allen Hart, and Daniel O'Connell. Mirrorcode: Ai can rebuild entire programs from behavior alone. *arXiv preprint arXiv:2606.30182*, 2026.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pp. 1–8. IEEE, 2013.

Anthropic. Claude Opus 5 System Card. System card, Anthropic, July 2026. URL https://www-cdn.anthropic.com/b514064af1408018e64b1ad24e7d5e75850b4ffd/Claude%20Opus%205%20System%20Card.pdf.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-search: Enhancing software agents with monte carlo tree search and iterative refinement. In *International Conference on Learning Representations (ICLR)*, 2025.

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

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2207.10397.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Mouxiang Chen, Zhongxin Liu, He Tao, Yusu Hong, David Lo, Xin Xia, and Jianling Sun. B4: Towards optimal assessment of plausible code solutions with plausible tests. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1693–1705, 2024.

Yihao Chen, Shi Chang, Feng Lin, Khaled Chawa, Boyuan Chen, Shaowei Wang, and Ahmed E Hassan. Specfirst: Behavioral specification elicitation as a first-class step in agent-based program synthesis from scratch. *arXiv preprint arXiv:2607.27167*, 2026.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, et al. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.

Lishui Fan, Mouxiang Chen, Tingwei Zhu, Kui Liu, Xin Xia, Shanping Li, and Zhongxin Liu. Zerocoder: Can llms improve code generation without ground-truth supervision? *arXiv preprint arXiv:2604.07864*, 2026.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.01798.

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 (ICLR)*, 2024. URL https://arxiv.org/abs/2310.06770.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, et al. Measuring AI ability to complete long software tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2503.14499.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP*, 2025a. doi: 10.18653/v1/2025.findings-emnlp.865.

Shangyu Li, Zhaoyang Zhang, Sizhe Zhong, Diyu Zhou, and Jiasi Shen. A sound static analysis approach to i/o api migration. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2): 584–614, 2025b.

Shangyu Li, Juyong Jiang, Meibo Ren, Sizhe Zhong, Huiri Tan, Yunhao Gou, Xu Han, Chun Yong Chong, Yun Peng, and Jiasi Shen. Codepivot: Bootstrapping multilingual transpilation in llms via reinforcement learning without parallel corpora. *arXiv preprint arXiv:2604.18027*, 2026a.

Shangyu Li, Juyong Jiang, Tiancheng Zhao, and Jiasi Shen. Osvbench: Benchmarking llms on specification generation tasks for operating system verification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 31698–31707, 2026b.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2306.09896.

Karl Pearson. Vii. mathematical contributions to the theory of evolution.—iii. regression, heredity, and panmixia. *Philosophical Transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character*, (187):253–318, 1896.

ProgramBench. ProgramBench leaderboard. https://programbench.com/, 2026. Accessed September 26, 2026.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 15174–15186, 2024.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with AlphaCodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024. URL https://arxiv.org/abs/2401.08500.

Jiasi Shen and Martin C Rinard. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 269–285, 2019.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. In *International Conference on Learning Representations (ICLR)*, 2025. Oral.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pp. 404–415, 2006.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. On the self-verification limitations of large language models on reasoning and planning tasks. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2402.08115.

Leitian Tao, Baolin Peng, Haorui Wang, Hang Wang, Hao Cheng, Wenlin Yao, Qianhui Wu, Tao Ge, Sharon Li, and Jianfeng Gao. Execcritic: Learn to test, test to improve for coding agents. *arXiv preprint arXiv:2609.09133*, 2026.

Boyang Wang, Yunhan Wang, and Yalun Wu. The unreliable progress bar: Can llm agents reliably report task progress throughout execution? *arXiv preprint arXiv:2609.08589*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2407.16741.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. In *International Conference on Learning Representations (ICLR)*, 2025.

Yitao Wu, Si Shen, Rui Yang, Hong Peng, and Bin Hu. Verify, repair, repeat, or stop? robust stopping for noisy verify-repair loops in llm agents. *arXiv preprint arXiv:2607.17641*, 2026.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, et al. Programbench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026.

Bingchen Zhao, Dhruv Srikanth, Yuxiang Wu, and Zhengyao Jiang. Specbench: Measuring reward hacking in long-horizon coding agents. *arXiv preprint arXiv:2605.21384*, 2026.

Wenting Zhao, Nan Jiang, Celine Lee, Justin Chiu, Claire Cardie, Matthias Gallé, and Alexander Rush. Commit0: Library generation from scratch. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2412.01769.

A Discussion

Limitations. Our evaluation is limited to currently available large language models, and our findings may not generalize to future model generations. To reduce dependence on the characteristics of any single model, we evaluate models from multiple providers, spanning different parameter scales and including both proprietary models and models with publicly available weights. Moreover, the recorded obligations and behavioral classes show which documented features and interaction types have corresponding tests in the corpus. A corpus may satisfy all extracted obligations while leaving undocumented behaviors and interactions untested. The progress signal therefore reflects performance on accumulated test evidence and does not guarantee correctness on unseen inputs.

Future work. Future work could incorporate the execution time of individual tests into corpus selection, aiming to preserve broad behavioral coverage while reducing the time required for repeated evaluation. Extending the framework to tasks where access to a reference program is limited would further broaden its applicability.

B Additional Related Work

Test generation for program synthesis and repair. Generated tests have been used to select, refine, and learn to produce code. CodeT and B4 use execution results from generated tests to rank candidate solutions to individual programming problems (Chen et al., 2023; 2024). ZeroCoder uses interactions between generated code and tests to jointly train a coder and a tester (Fan et al., 2026). Concurrent work on ExecCritic generates tests independently of the repair agent and holds qualified tests fixed while the agent revises patches for issues in existing repositories (Tao et al., 2026). S^* first generates multiple candidate programs and iteratively refines each using feedback from public tests. It then generates test inputs that expose behavioral differences between candidates and uses their execution results to guide candidate selection (Li et al., 2025a). AlphaCodium generates additional tests before code refinement and retains passed tests as anchors during iterative debugging (Ridnik et al., 2024). Our approach focuses on continuously expanding and managing the test corpus as the candidate program evolves. Agents generate new queries guided by unmet documented requirements and observed divergences, while reference executions determine the expected observations. We further select tests by behavioral class under explicit budgets and quotas to broaden coverage and control redundancy. These mechanisms support the ongoing construction of a progress signal throughout the development of a complete program.

Reliability of self-evaluation. LLMs struggle to assess the correctness of their own outputs in reasoning and program repair (Huang et al., 2024; Olausson et al., 2024; Stechly et al., 2025). In long-horizon coding, improvements on visible tests do not necessarily translate into better performance on held-out tests. The gap between visible and held-out test pass rates tends to be larger for tasks with larger reference implementations (Zhao et al., 2026). Wu et al. (2026) study when agents should stop revising their outputs by assessing whether further repair has positive expected gain while accounting for possible errors in the verifier's judgments. Our approach constructs and maintains a test corpus throughout program development. We broaden behavioral coverage and control test redundancy, aiming to better align improvements in the resulting progress signal with gains in held-out performance. Separately, Wang et al. (2026) train reasoning models to predict their progress toward completing a reasoning sequence. Our signal measures the candidate program's performance on accumulated tests by comparing its observed behavior with observations obtained from reference executions.

C Details on the Behavioral Signatures

This section expands the signature components in Equation 3.

Both branches include $\mathrm{subcommand}$, the subcommand specified by the query, and flags, the set of option names used, excluding their values. For rejected test cases, the signature additionally includes the exit status and a diagnostic template $\mathrm{skel}(y_{\mathrm{err}})$. This template is obtained by removing variable details from the error message, including addresses, argument values, and reported input sizes or lengths. Within each subcommand and option combination, rejections with the same exit status and diagnostic template belong to the same class.

For other test cases, the signature instead includes $\mathrm{form}(q)$ and $\mathrm{modality}(y)$. The former summarizes the structure of the arguments, standard input, and input files using features such as input type, file extension, and size category, while omitting their concrete contents. The latter describes how the reference result is expressed, including modifications to existing files, newly created files, changes to file-system structure, structured or plain text, or an exit status alone.

We use separate branches because exit status and diagnostic patterns characterize rejection behavior, whereas successful executions may share the same exit status and an empty error stream, making input structure and output modality more informative for distinguishing their behaviors.

D Test Selection Details and Optimality Proof

This section expands the notation and constraints of the selection problem in Equation 4 and proves Proposition 1 from §2.3.

D.1 Notation and Constraint Details

Let Δ denote the finite set of candidate test cases considered for admission to the corpus S, with $\Delta \cap S = \varnothing$. We partition Δ into divergent tests Δ_{div}, on which the candidate program disagrees with the reference, and regression tests Δ_{reg}, on which their observations are the same. Both types use expected observations obtained from reference executions and enter the corpus through the selection procedure. Their classifications remain fixed during selection. Each admitted test continues to count toward the quota for its original type, even if its pass/fail status later changes. Once the candidate passes a divergent test, that test also helps detect later regressions without being reclassified as a regression test.

The class labels considered during selection are $\mathcal{C}_{\mathrm{sel}} = \{c(\tau) : \tau \in S \cup \Delta\}$, including newly discovered classes. For each class label c, let $n_{c,\mathrm{div}}$ and $n_{c,\mathrm{reg}}$ denote the numbers of tests in S previously admitted as divergent and regression tests, respectively. We assign separate corpus quotas $\mathrm{cap}_{\mathrm{div}}(c)$ and $\mathrm{cap}_{\mathrm{reg}}(c)$ to these two types. Each quota limits only its corresponding type within class c.

We additionally impose a total corpus capacity M, which limits the combined number of divergent and regression tests across all classes. The admission budget B limits the number of new tests admitted in a single selection. All class quotas, M, and B are nonnegative integers and remain fixed during selection.

For any subset $A \subseteq \Delta$, define

$$A_c = \{\tau \in A : c(\tau) = c\}, \qquad A_{c,\mathrm{div}} = A_c \cap \Delta_{\mathrm{div}}, \qquad A_{c,\mathrm{reg}} = A_c \cap \Delta_{\mathrm{reg}}.$$

Assuming that the current corpus satisfies all class quotas and $|S| \leq M$, we select at most B new test cases to maximize their total class weight, formally expressed in Equation 4. Since $\Delta \cap S = \varnothing$, admitting A increases the corpus size to $|S| + |A|$.

In Equation 4, $w_c > 0$ denotes the weight assigned to class c and remains fixed during selection. We assign higher weights to newly discovered classes and to classes with output modalities beyond the standard streams, such as generated files and filesystem structure. The separate class quotas limit repeated tests of each type within a behavioral class. Unused capacity for one type cannot be used to exceed the quota of the other type. Both test types count toward the total corpus capacity M and the admission budget B. The former bounds the overall corpus size, while the latter limits its growth in a single selection. These constraints impose upper bounds and do not require a minimum number of tests of either type.

D.2 Proof of Greedy Optimality

This section gives the proof of Proposition 1 for the fixed-weight admission objective in Equation 4.

Proof. Since each test belongs to exactly one class and one type, the objective can be written as

$$F(A) = \sum_{\tau \in A} w_{c(\tau)}.$$

We prove optimality by an exchange argument, showing that an optimal set can be transformed into the greedy set through replacements that keep all constraints satisfied and do not decrease the objective value. Let $G \subseteq \Delta$ be the set of candidate tests selected so far by the greedy algorithm, and suppose an optimal set $A^\star$ contains G. This condition holds initially when $G = \varnothing$. Consider the next test τ admitted by greedy selection. If $\tau \in A^\star$, then $G \cup \{\tau\} \subseteq A^\star$, so the enlarged greedy set is still contained in an optimal set.

Now suppose $\tau \notin A^\star$. For any test $u \in A^\star \setminus G$, we have $G \cup \{u\} \subseteq A^\star$. Since $G \cup \{u\} \subseteq A^\star$, its total number of tests does not exceed that of $A^\star$. Within each class, its counts of divergent and regression tests also do not exceed the corresponding counts in $A^\star$. Since S is fixed and all constraints impose upper bounds on test counts, adding u to G satisfies all constraints.

At any earlier point, the tests already admitted by the greedy algorithm formed a subset of G, so admitting u at that point would also have satisfied all constraints. Therefore, if u had already been

considered, it would have been admitted and would belong to G. Since $u \notin G$, it has not yet been considered. The algorithm processes candidates in decreasing order of class weight, so

$$w_{c(u)} \leq w_{c(\tau)} \qquad \text{for every } u \in A^\star \setminus G.$$

Suppose adding τ to $A^\star$ would exceed the quota for its type within its class. Since $A^\star$ satisfies all constraints, the tests of that class and type in S and $A^\star$ must already fill the corresponding quota.

By definition, τ is the next test admitted by the greedy algorithm, which admits a test only when adding it to G satisfies all constraints. Therefore, $G \cup \{\tau\}$ satisfies this quota. The tests of that class and type in S and G must consequently leave at least one available position. Thus, $A^\star$ contains more tests of that class and type than G does.

Since $G \subseteq A^\star$, there must be a test $u \in A^\star \setminus G$ with the same class and type as τ. Replacing u with τ leaves the count for that class and type unchanged and does not affect the counts for any other class or type. Therefore, all class quotas remain satisfied.

The replacement also leaves the number of selected tests unchanged. Since the existing corpus S remains unchanged, the total corpus size also remains unchanged. Thus, both the admission budget B and the total corpus capacity M remain satisfied.

If adding τ to $A^\star$ would satisfy all class quotas but exceed the admission budget B or the total corpus capacity M, remove any test $u \in A^\star \setminus G$ and add τ. Such a test exists because $G \cup \{\tau\}$ satisfies both limits, whereas $A^\star \cup \{\tau\}$ violates at least one of them. Since S is fixed, both limits depend only on the number of selected tests. Therefore, $A^\star$ contains more tests than G. Since $G \subseteq A^\star$ and $|A^\star| > |G|$, there exists a test $u \in A^\star \setminus G$ that can be removed without removing any member of G.

The replacement leaves the number of selected tests and the resulting corpus size unchanged. Both the admission budget and the total corpus capacity therefore remain satisfied. All class quotas were satisfied after adding τ by assumption, and removing u cannot violate them.

If adding τ to $A^\star$ would satisfy every constraint, then its positive class weight would give

$$F(A^\star \cup \{\tau\}) = F(A^\star) + w_{c(\tau)} > F(A^\star),$$

contradicting the optimality of $A^\star$.

In each replacement case, define

$$A' = (A^\star \setminus \{u\}) \cup \{\tau\}.$$

The set A' satisfies all constraints and contains $G \cup \{\tau\}$ because $u \notin G$. Moreover, the class-weight inequality established above gives

$$F(A') - F(A^\star) = w_{c(\tau)} - w_{c(u)} \geq 0.$$

Since $A^\star$ is optimal, A' is also optimal. Thus, after admitting τ, the enlarged greedy set remains contained in an optimal set.

Repeating this argument shows that the final greedy set G is contained in an optimal set. Every candidate outside the final set G was rejected because adding it would have exceeded a class quota, the admission budget B, or the total corpus capacity M. At that point, the corresponding limit had already been reached. The existing corpus S remains fixed, and the greedy algorithm never removes previously selected tests. The tests occupying those positions therefore remain, and the limit does not increase. Consequently, adding the rejected candidate to the final set G would still exceed that same limit.

If an optimal set containing G included an additional test u, then $G \cup \{u\}$ would be a subset of that optimal set and would satisfy all constraints. This contradicts the fact that no further test can be added to G. Hence, an optimal set containing G must equal G, and greedy selection is optimal. □

The resulting procedure sorts Δ by decreasing class weight, with divergent tests placed before regression tests when their class weights are equal. A test is admitted if and only if its addition satisfies the quota for its type within its class, the admission budget B, and the total corpus capacity M. Selection stops when B tests have been admitted, the total corpus capacity M is reached, or no further candidate can be admitted under these constraints.

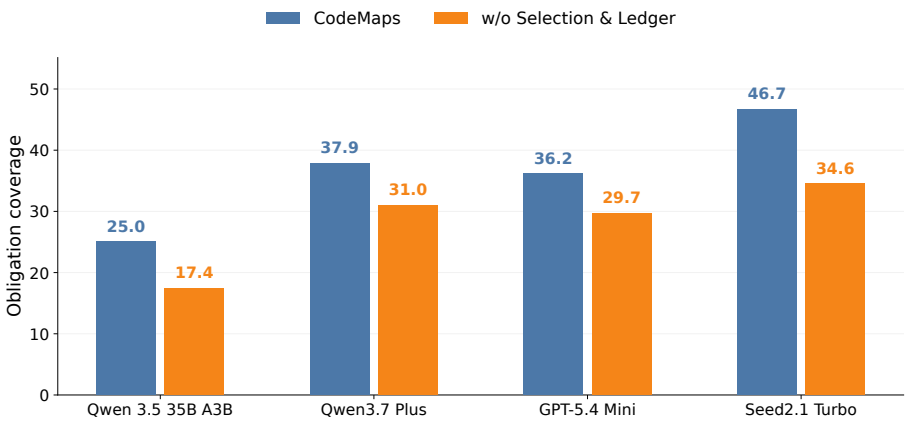

Figure 3: Mean obligation coverage (%) of the test corpora constructed by CODEMAPS and its *w/o Selection & Ledger* variant, presented separately for each base model.

E ADDITIONAL EXPERIMENTAL ANALYSES

E.1 BREADTH OF THE TEST CORPUS

We analyze the breadth of the retained test corpora by counting the distinct documented obligations associated with at least one retained test. Repeated tests associated with the same obligation do not increase this count. For each base model, we average these counts across the evaluated tasks and normalize them by 93.28, the average number of extracted obligations per task across PROGRAMBENCH. The resulting percentages represent normalized mean obligation counts.

Figure 3 compares CODEMAPS with the W/O SELECTION & LEDGER variant. CODEMAPS achieves higher values across all four base models, with 25.0% versus 17.4% for Qwen-3.5-35B-A3B, 37.9% versus 31.0% for Qwen-3.7-Plus, 36.2% versus 29.7% for GPT-5.4-Mini, and 46.7% versus 34.6% for Seed-2.1-Turbo. The largest difference is observed for Seed-2.1-Turbo.

These results suggest that the full framework retains test evidence associated with a broader range of documented requirements than the variant without the proposed selection strategy and obligation ledger. However, these associations do not establish that the corresponding behaviors have been fully tested or correctly implemented.