PLANNING TO EXPLORE: CURIOSITY-DRIVEN EXPLORATION FOR LLM TEST GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

The use of LLMs for code generation has naturally extended to code testing and evaluation. As codebases grow in size and complexity, so does the need for automated test generation. Current approaches for LLM-based test generation rely on strategies that maximize immediate coverage gain, a greedy approach that plateaus on code where reaching deep branches requires setup steps that individually yield zero new coverage. Drawing on principles of Bayesian exploration, we treat the program's branch structure as an unknown environment and maintain an exploration state built from execution, represented by a coverage map and a per-function posterior over which parts of the program the agent can reach. Our method, *CovQ-Value*, uses this posterior to target candidate test plans at unexplored code, asks the LLM which functions each plan will execute and enable, and selects the plan with the highest estimated Q-value. We show that the major gains come from using the exploration state to decide what to explore next. Our method outperforms greedy selection on TestGenEval Lite, achieving 103–150% higher branch coverage and winning on 83–88% of targets. In addition, we build a benchmark for iterative test generation, *RepoExploreBench*, where CovQValue achieves 129–157% higher branch coverage than greedy selection across three LLMs, winning on 94–97% of targets. These results show the potential of curiosity-driven exploration for LLM agents, enabling more effective discovery with sequential interaction.

1 INTRODUCTION

LLMs are increasingly used not only to generate code, but also to test it (Chen et al., 2021; Schäfer et al., 2023). However, generating tests that reach deep program behavior often requires discovering the right sequence of steps and branches, where some steps can yield zero immediate coverage gain. Most existing LLM-based test generation methods are effectively greedy. They generate tests independently, without learning from previous executions or reasoning about what remains unexplored. This challenge aligns with a well-studied problem in reinforcement learning where an agent needs to explore an unknown environment efficiently. We apply this perspective to LLM-based test generation. We view iterative test generation as the exploration of an unknown program, where each executed test reveals part of the program's behavior through the branches it reaches. Rather than treating coverage solely as an evaluation metric, we feed back a structured coverage map to the LLM and use it as the basis for information-gain-driven action selection.

For decades, researchers have been exploring how to transform curiosity into a computational feature that can be integrated into modern AI systems (Schmidhuber, 2010). In exploration theory, Schmidhuber (1991b; 2010) formalized curiosity as the drive to maximize learning progress by seeking experiences that improve the agent's world model. Sun et al. (2011) proved that, in the Bayesian setting, the optimal strategy for maximizing cumulative information gain requires planning ahead, not greedy selection, because information gain is additive only in expectation. They demonstrated this with the corridor problem, where a deterministic passage connects two information-rich regions. Greedy exploration ignores the corridor (zero immediate information gain); only planning-aware exploration traverses it. We build on two ideas from this framework: (1) the use of an explicit exploration state, and (2) a curiosity value that scores actions by their expected reveal. Our setting uses coverage as a proxy for information gain.

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.

With the expansion of LLMs, we have also seen an increase in their usage to generate code. The increasing lines of code also requires more and better code tests. LLM-based test generation is one such setting, where the LLM must interact with the program iteratively to discover its behavior. Tools like CodaMosa (Lemieux et al., 2023), CoverUp (Altmayer Pizzorno & Berger, 2025), and TestForge (Jain & Goues, 2025) generate tests that improve code coverage. However, they are all greedy: they select each test by its immediate coverage gain, without estimating what the test makes reachable later. When reaching deep branches requires a sequence of setup steps that individually contribute zero coverage (e.g., import chains, class initialization, passing validation gates), greedy strategies plateau before reaching the deeper code.

In this work, we approach LLM-based test generation as Bayesian exploration of an unknown environment (Sun et al., 2011). In this environment, the program's branch structure is the unknown parameter, the coverage map and a per-function reachability posterior summarize the agent's exploration state, and the LLM with its interaction history serves as the world model. Our method, *CovQValue*, generates candidate test plans, each targeted at a different region of unexplored code, and predicts which functions each plan will execute and which it will make reachable. The posterior turns these predictions into a curiosity Q-value that balances immediate discovery against the value of the resulting state. It then executes the highest-scoring plan, updates the posterior, and loops again. We conduct experiments on TestGenEval Lite, an existing benchmark built on SWE-Bench; and on *RepoExploreBench*, a benchmark we introduce for iterative test generation, built on 93 modules from 9 popular Python packages. We run the experiments on three LLMs (Gemini 3.8 Flash, Gemma 4 31B and GLM 5.3). CovQValue achieves 129–157% more branch coverage than greedy selection on RepoExploreBench and 103–150% more on TestGenEval Lite, winning on 83–97% of targets.

Our contributions can be summarized as follows:

1. **We frame LLM test generation as Bayesian Exploration of an unknown environment**, maintaining an explicit exploration state (a coverage map and per-function reachability estimates updated from execution) that tells the agent what remains unexplored and where to look next.
2. **We release RepoExploreBench**, a benchmark of 93 modules from 9 popular Python packages designed to evaluate iterative, exploration-based test generation. Existing benchmarks evaluate one-shot generation; RepoExploreBench targets code with corridor structure where multi-round feedback is essential.
3. **We introduce CovQValue, a method for iterative code test generation** that uses the exploration state to target candidate test plans at unexplored code, and selects among them with a Q-value derived from the LLM's predictions and the posterior. Evaluated on RepoExploreBench and the established TestGenEval Lite benchmark with three different LLMs, CovQValue achieves 103–157% more branch coverage than greedy selection across both benchmarks, winning on 83–97% of targets.

Beyond test generation, posterior-guided exploration can apply to other agents that interact sequentially with an unknown environment (e.g. API discovery, scientific experimentation). Our results suggest that the gap between greedy and exploratory strategies widens as exploration proceeds, indicating that agents benefit most from an explicit estimate of what remains unexplored. As LLM agents are deployed in increasingly open-ended setting, providing them with principled exploration strategies will be essential for navigating environments.

2 Related Work

Artificial Curiosity and Optimal Exploration Curiosity is commonly defined as the motivation that drives exploration and problem-solving in artificial agents (Singh et al., 2010). Schmidhuber (1991b) describes intrinsic motivation as the drive to improve one's predictive world model. Later, Schmidhuber (1991a) expands this theory into a general framework that formalizes curiosity as an optimization problem in reinforcement learning, where the goal is to maximize the improvement of a predictive world model. Storck et al. (1995) instantiated this principle by using KL divergence between successive probability estimates as the curiosity reward in non-deterministic environments. Building on this, Sun et al. (2011) showed how an agent can optimally plan action sequences based on previous experiences such that the cumulative expected information gain is maximized.

LLM-based Test Generation Traditionally, automated test generation has relied on search-based software testing, where tools like EvoSuite (Fraser & Arcuri, 2011) or Pynguin (Lukasczyk & Fraser, 2022) mutate test inputs and maximize coverage. These tools operated at the input level without generating semantic test scripts. The use of LLMs for writing code has increased tremendously due to their good results and the verifiable nature of code (Chen et al., 2021). The rise of automatically generated code has increased the need for automated testing as a self-verification mechanism. ChatUniTest (Chen et al., 2024) and ChatTester (Yuan et al., 2024) are examples of this. LLMs can generate tests, but coverage plateaus on complex code. Jain et al. (2024) introduces TestGenEval to measure this. It is a file-level test generation benchmark on real-world projects built on SWE-Bench (Jimenez et al., 2023). Existing approaches for LLM-based test generation are single-step or reactive. CodaMosa (Lemieux et al., 2023) presents a hybrid method that detects coverage stalls and calls the LLM reactively. CoverUp (Altmayer Pizzorno & Berger, 2025), TELPA (Yang et al., 2024), and Xu et al. (2026) treat coverage as a sequence of independent single-step optimization problems, where the test at step n is not chosen for what it enables at step $n + 1$, and therefore maximize immediate coverage gain. They rely on methods that generate tests, measure coverage, feed the uncovered branches and generate more tests. TestForge (Jain & Goues, 2025) takes an agentic approach, iteratively refining tests based on execution feedback, but still selects actions greedily. None of these approaches reason about how covering intermediate branches opens access to deeper ones.

Information Gain to guide LLM Actions A recent line of work tries to apply information-gain maximization to guide LLM behavior at inference time. Uncertainty of Thoughts (Hu et al., 2024) uses tree search with information-gain rewards to select questions that reduce the LLM's uncertainty. BED-LLM (Choudhury et al., 2025) formalizes this more rigorously as sequential Bayesian Experimental Design (BED), creating estimators to maximize the expected information gain derived from the LLM's predictive distribution. CuriosiTree (Cooper et al., 2025) takes a similar heuristic approach to zero-shot information acquisition. These methods operate in static environments where the questions do not change the state of the world. Test generation introduces a different challenge where, although the program's branch structure is static, the agent's knowledge of how to reach deep branches evolves with each execution. Reaching certain branches requires discovering specific setup sequences through prior tests, creating epistemic corridors analogous to those in Sun et al. (2011)'s MDP experiments, where only multi-step planning succeeds.

3 METHOD

Test Generation inspired by Bayesian Exploration We frame iterative test generation as a heuristic approximation of Bayesian exploration in an unknown environment (Sun et al., 2011). The agent interacts with a program whose internal branch reachability is initially unknown. At each step, the agent generates a test (action), executes it, and observes which branches are reached (observation). The goal is to choose tests such that cumulative branch coverage (our measure of knowledge about the program) grows as quickly as possible.

This is grounded in the theory of Sun et al. (2011), which presents a study of optimal exploration in dynamic environments. An agent interacts with an environment in discrete time steps, performing actions and receiving observations. The environment is characterized by an unknown parameter Θ, and the agent maintains a posterior $p(\Theta|h)$ over Θ given the interaction history $h = (a_1, o_1), \ldots, (a_t, o_t)$.

The expected information gain of performing action, a, given history, h, is:

Table 1: Correspondence between the Bayesian exploration framework of Sun et al. (2011) and our test-generation method.

Definitions	**Symbol**	**Test Generation**
Unknown environment	Θ	Program's branch reachability structure
Prior	$p(\Theta)$	No branches observed
History	$h = (a_1, o_1) \cdots (a_t, o_t)$	Sequence of (test, coverage result) pairs
Exploration state	$p(\Theta\|h)$	Coverage map and per-function reachability p_f
Action	a	Test plan (sequence of S scripts)
Observation	o	Branches hit by executing the plan
Expected info gain	$\tilde{g}(a\|h) = I(O; \Theta \mid h, a)$	Proxy: posterior-weighted size of unexecuted functions a executes
Curiosity Q-value	$q(h, a) = \tilde{g}(a\|h) + \gamma \cdot \mathbb{E}_{o\|h,a}[v(hao)]$	Immediate gain + value of the posterior after a
Corridor		Validation gates, setup sequences

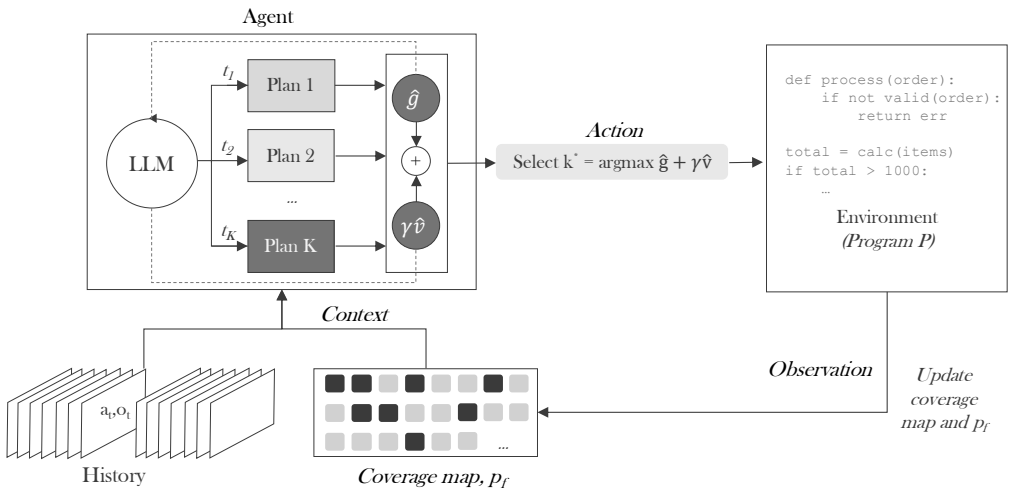


Figure 1: **Overview of CovQValue**. In each round, the LLM writes K plans, each targeting a different group of untested functions. It predicts which functions each plan runs, and the probabilities p_f turn these predictions into the immediate gain $\hat{g}$ and the future value $\hat{v}$ (Equation 3). The plan with the highest $\hat{g} + \gamma\hat{v}$ is executed, and its coverage updates the coverage map and probabilities p_f.

$$\tilde{g}(a \mid h) = I(O; \Theta \mid h, a) \tag{1}$$

$I(\cdot;\cdot)$ is the mutual information between the unknown parameter, Θ and the observation, O, conditioned on history, and action. Because information gain is additive only in expectation (Sun et al., 2011), it cannot be directly substituted as a reward in standard reinforcement learning. Instead, the optimal exploration policy is defined through a recursive curiosity Q-value:

$$q(h, a) = \tilde{g}(a \mid h) + \gamma \cdot \mathbb{E}_{o|h,a}[v(hao)] \tag{2}$$

where $v(h) = \max_a q(h, a)$ is the curiosity value of history h, and γ is a discount factor. The Q-value captures not just the immediate expected information gain of action a, but also the expected value of the state the agent ends up in after taking a and observing o. Table 1 shows how we instantiate this framework for test generation, and Figure 1 gives an overview of the method.

Exploration State We found proxy signals such as sampling entropy to be poorly calibrated for uncertainty about program behavior, so we build the exploration state from execution instead. The *coverage map* records the branches executed so far and is shown to the LLM as text (Appendix K.1). In addition, for each function f that no test has executed yet, we keep a $\mathrm{Beta}(\alpha_f, \beta_f)$ posterior over whether our tests can reach it, starting from $\mathrm{Beta}(1, 1)$. Its mean, $p_f = \alpha_f/(\alpha_f + \beta_f)$, is our estimate that f can be reached.

CovQValue In each round, CovQValue proposes a few candidate plans, each a short sequence of test scripts aimed at a different part of the untested code, and runs the plan with the highest estimated Q-value (Figure 1). For each plan a, the LLM predicts which untested functions the plan will run, and which ones it will make easier to reach later, for example by creating an object they need. Let $r_f(a) = 1$ if the LLM predicts that plan a runs function f, and 0 otherwise. We estimate the immediate gain and the change in future value in Equation 2 by counting untested lines of code, weighting each function by its length $|f|$ and its probability p_f of being reached:

$$\hat{g}(a) = \sum_{f \in U} r_f(a)\, p_f\, |f|, \qquad \hat{v}(a) = \lambda \sum_{f \in N(a)} (1 - p_f)\, |f|, \tag{3}$$

where U is the set of untested functions and $N(a) \subseteq U$ the functions the LLM predicts plan a makes easier to reach without running them. The immediate gain $\hat{g}(a)$ is the untested code the plan

is expected to run. The future value $\hat{v}(a)$ is the expected increase in reachable untested code after the plan: for each $f \in N(a)$, the probability of reaching it rises from p_f to $p_f + \lambda(1 - p_f)$, with $\lambda = 0.09$ measured from logged runs. We run the plan with the highest $\hat{g}(a) + \gamma\,\hat{v}(a)$. Afterwards, each predicted function updates its posterior: α_f increases by one if the function is now covered, and β_f otherwise.

Coverage Corridors Some code can only be reached after setup steps that add no coverage themselves, such as creating an object or passing a validation check. We call these *corridors*. Greedy strategies skip them, since setup looks useless on its own. CovQValue still values such a plan, because the functions it makes reachable enter $N(a)$ and raise $\hat{v}(a)$ in Equation 3.

4 EXPERIMENTS

Benchmarks We evaluate whether coverage-map-guided planning improves branch coverage over standard baselines across diverse real-world code. To validate our methods, we use two benchmarks including 233 real-world Python source files across 19 repositories:

- **TestGenEval Lite** (Jain et al., 2024) is an existing benchmark of 160 file-level test generation targets from SWE-bench (Jimenez et al., 2023). It is built over 11 Python repositories including `django`, `sympy`, `pytest`, and `matplotlib`. We evaluate on 140 files, excluding scikit-learn which requires per-container C extension compilation. Each target runs in a repository-specific SWE-bench Docker testbed with the correct Python environment and version-pinned dependencies.

Since TestGenEval Lite and other benchmarks evaluate one-shot generation rather than iterative exploration, we introduce RepoExploreBench for the iterative setting. We report results on both.

- **RepoExploreBench** (Ours) comprises 93 modules from 9 popular open-source Python packages: `click`, `requests`, `flask`, `rich`, `jinja2`, `httpx`, `pydantic`, `werkzeug`, and `starlette`. Packages were selected from the top 150 most-downloaded PyPI packages using three criteria: (1) Python source with $\geq$5K lines, (2) no C extensions or build tools, and (3) corridor structure (e.g., import chains, class hierarchies, and configuration requirements). From each package, we select modules with $\geq$200 lines, yielding 93 targets totaling $\sim$77K lines of code. All modules execute inside a Docker container with packages pre-installed. We provide more details about the benchmark in Appendix A.

Strategies We compare four strategies with the same execution budget, plus two external tools:

- **Random**: generates K=3 test scripts from a standard prompt containing the module source code and recent test history. One script is selected uniformly at random.
- **Greedy**: same generation as Random, but the LLM selects which candidate is "most likely to cover new code paths." This isolates LLM self-selection without coverage feedback; tools such as CoverUp (Altmayer Pizzorno & Berger, 2025) add coverage feedback, which we evaluate through CovGreedy and by running CoverUp directly.
- **CovGreedy**: augments the generation prompt with the coverage map. The coverage map is a structured text block reporting branches discovered, recent discovery rate, stagnation warnings, and the most informative prior tests. Its full format and evolution are shown in Appendix K.1. The LLM is instructed to target undiscovered branches. Unlike Greedy, it selects one of the K=3 candidates uniformly at random.
- **CovQValue** (ours): generates K=3 plans, each consisting of S=3 sequential test scripts and targeted at a different chunk of still-unexecuted functions (up to 6 functions per chunk). Plans are scored by Equation 3 with γ=1, and the highest-scoring plan is executed (Section 3, Algorithm 1).
- **External baselines** (RepoExploreBench only): *Pynguin* (Lukasczyk & Fraser, 2022), a search-based generator without LLM, with 60 seconds of search per module, and *CoverUp* (Altmayer Pizzorno & Berger, 2025), an LLM-based tool that iteratively prompts for tests targeting uncovered code. Both use their native budgets. Due to library version incompatibilities, we only run them on RepoExploreBench.

Table 2: **Mean per-module Branch Coverage** ($\pm$ standard error). Gemma 4 31B on RepoExploreBench is averaged over 3 seeds per module; all other cells use one seed. CovQValue vs. Greedy comparisons: $p < 0.0001$ (paired t-test). d: Cohen's d effect size. Win rate: fraction of targets on which CovQValue covers more branches than Greedy.

	RepoExploreBench			**TestGenEval Lite**		
	Gemini 3.8 Flash	Gemma 4 31B	GLM 5.3	Gemini 3.8 Flash	Gemma 4 31B	GLM 5.3
Random	37.9 $\pm$4.0	33.5 $\pm$3.7	41.3 $\pm$4.1	56.3 $\pm$5.4	57.4 $\pm$6.0	62.8 $\pm$5.4
Greedy	40.2 $\pm$4.2	33.4 $\pm$3.8	44.3 $\pm$4.4	57.7 $\pm$5.7	56.4 $\pm$6.2	66.4 $\pm$6.0
CovGreedy	93.4 $\pm$7.8	35.1 $\pm$4.2	64.5 $\pm$6.3	138.9 $\pm$11.4	68.6 $\pm$6.1	94.2 $\pm$7.8
CovQValue	**103.5** $\pm$7.7	**85.5** $\pm$7.1	**101.6** $\pm$7.6	**144.0** $\pm$10.6	**114.6** $\pm$8.5	**138.7** $\pm$10.7
Δ vs Greedy	+63.3	+52.1	+57.3	+86.3	+58.2	+72.3
Cohen's d	1.23	1.09	1.10	0.91	0.89	0.77
Win rate	97%	95%	94%	88%	86%	83%

Models We evaluate with three LLMs to test generalization across model families: Gemini 3.8 Flash (Google), Gemma 4 31B-it (Google, open weights), and GLM 5.3 (Z.ai). The same model is used for all components within a run: test generation, Q-value scoring, and greedy selection. Generation uses temperature 0.9; scoring and selection use temperature 0.3. Gemini 3.8 Flash and GLM 5.3 are reasoning models. We run them at low reasoning effort and add a 4,096-token thinking allowance to every output limit so that answers are not truncated by reasoning.

Metrics To measure the effect of our methods, we use *branch coverage*, which is the cumulative number of unique branches executed across all test scripts. Branch coverage directly measures how much of the program's behavior has been discovered, making it the natural metric for an exploration framework. We also report *line coverage* to confirm that results hold under the metric used by TestGenEval (Jain et al., 2024), and *pass rate* (fraction of generated scripts that execute without error) to differentiate whether coverage gains come from writing more valid tests or from targeting more informative code paths. Both are reported in Appendix C.

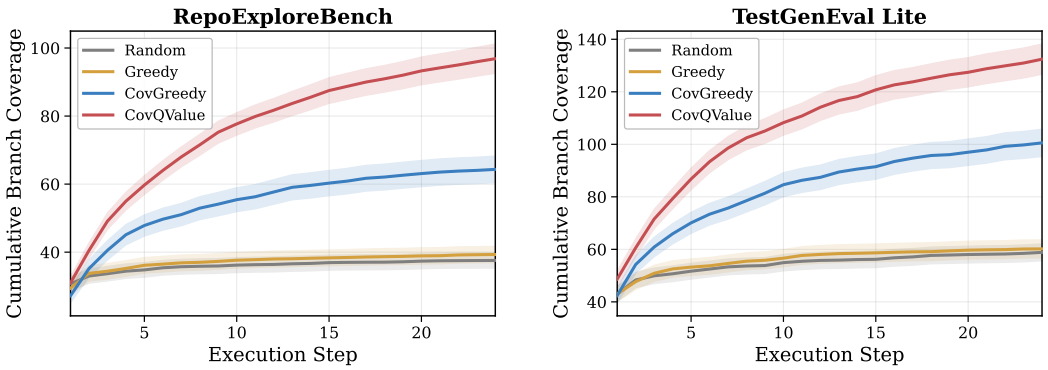
Execution Budget All strategies receive exactly the same number of test script executions (N=24). Random, Greedy, and CovGreedy each execute 24 individual scripts. CovQValue executes 8 plan rounds of 3 scripts each ($8 \times 3 = 24$), and the LLM sees the updated coverage map once per round (8 times in total). CovQValue additionally requires K=3 Q-value scoring calls per round. Thus, while total executions are equalized, the strategies differ in feedback frequency and total LLM calls (Appendix F compares their inference cost). Despite receiving fewer feedback updates, CovQValue outperforms baselines. Grouping executions into plans also reduces the number of LLM calls; Appendix G analyzes the trade-off between plan length, coverage, and cost.

5 RESULTS

CovQValue outperforms baselines in all models and benchmarks Table 2 shows branch coverage in all settings. CovQValue achieves the highest coverage for every model and benchmark, with improvements over Greedy ranging from +52.1 to +86.3 branches. The method wins on 83–97% of individual targets, which shows that it generalizes across the different model families evaluated. CovGreedy, which adds coverage map feedback but no planning, improves over Greedy, but not consistently, as its gains depend strongly on the model.

Table 3: Branch Coverage comparison with published tools on RepoExploreBench ($\pm$ SE). Pynguin (no LLM, 60 s search) and CoverUp (run to completion) use their native budgets.

	RepoExploreBench		
	Gemini 3.8	Gemma 4	GLM 5.3
Pynguin (no LLM)	.....	34.7 $\pm$4.3	.....
CoverUp	98.0 $\pm$8.4	45.2 $\pm$5.4	59.8 $\pm$6.4
CovQValue	**103.5** $\pm$7.7	**85.5** $\pm$7.1	**101.6** $\pm$7.6
vs. Pynguin	+68.8	+50.8	+66.9
vs. CoverUp	+5.5	+40.3	+41.8



(a) RepoExploreBench (b) TestGenEval Lite

Figure 2: Cumulative branch coverage over execution steps, averaged across Gemini 3.8 Flash, Gemma 4 31B, and GLM 5.3. CovQValue (ours) continues discovering branches while Random and Greedy stall earlier. Shaded region indicates ± 1 SE.

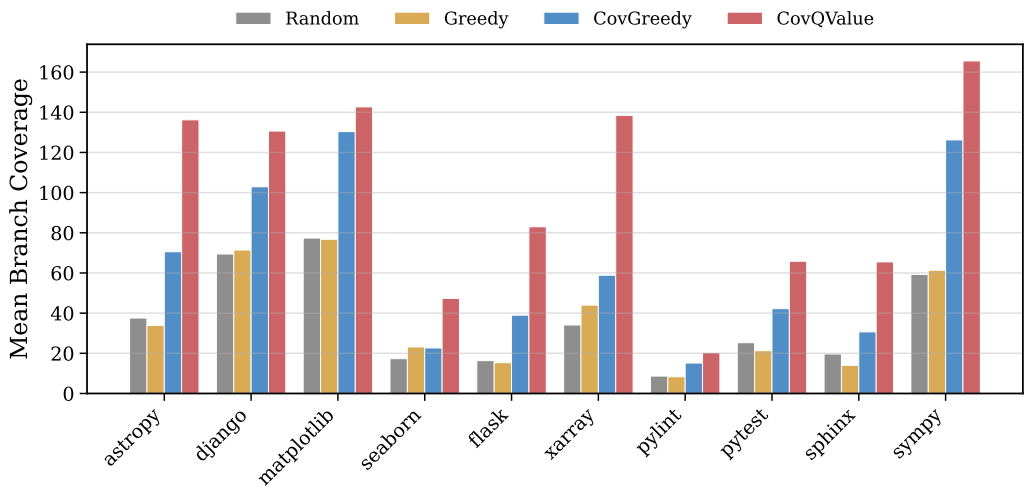
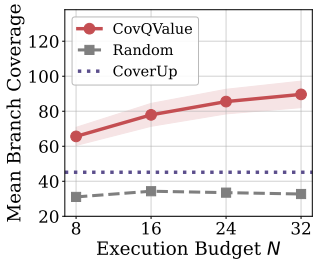


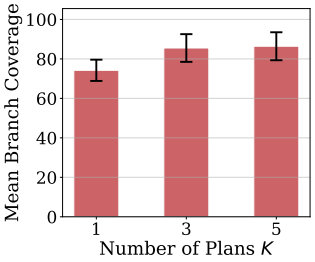
Figure 3: Mean branch coverage by repository on TestGenEval Lite, averaged across Gemini 3.8 Flash, Gemma 4 31B, and GLM 5.3. CovQValue outperforms other methods on every repository, including the three largest: django (+83% over Greedy), sympy (+170%), and pytest (+209%).

CovQValue also reaches higher coverage than the published tools Pynguin and CoverUp with every model (Table 3), with the smallest margin over CoverUp on Gemini 3.8 Flash. Finally, Greedy does not perform better than Random, which suggests that LLM-based selection without coverage feedback cannot predict which test will discover new branches from code inspection alone.

Coverage grows with exploration budget Figure 2 shows cumulative branch coverage over execution steps, averaged across models (per-model curves are in Appendix I). Random and Greedy plateau after about five steps, as they keep generating variations of the same tests. CovGreedy keeps climbing as the coverage map points to untested code, and CovQValue climbs fastest. The gap between the two widens over the run, and in the second half of the budget CovQValue discovers about twice as many new branches as CovGreedy on RepoExploreBench and about 40% more on TestGenEval Lite. Targeting keeps each round aimed at code that is still untested, so CovQValue does not keep revisiting what it has already covered.



(a) Execution budget N.



(b) Number of plans K.

Setting	Coverage
$\gamma = 0$	88.6
$\gamma = 1/3$	86.8
$\gamma = \mathbf{1}$	**85.5**
$\hat{v}$ only (no $\hat{g}$)	82.6
Cov. map	35.4 ±4.2
+ Div. hints	43.1 ±4.8
+ Targeting	80.0 ±6.0
+ Function scorer†	86.5 ±7.1
+ Bayesian value† (CovQValue)	85.5 ±7.1

(c) Hyperparams. & components.

Figure 5: Ablation studies on RepoExploreBench (93 targets, Gemma 4 31B). (a) CovQValue scales steadily with budget. (b) More candidate plans help up to K=3. (c) Top: discount factor sensitivity (default in bold). Bottom: component contributions; †averaged over 3 seeds.

Consistent results across repositories Figure 3 shows coverage by repository on TestGenEval Lite, averaged across models. CovQValue performs best on every repository, including the three that make up most of the benchmark (django, sympy, and pytest). Even on repositories where all strategies reach low coverage, CovQValue matches or exceeds the other methods. RepoExploreBench shows the same pattern (Appendix H).

Coverage gains come from deeper exploration, not safer tests Figure 4 plots test pass rate against branch coverage. Random and Greedy reach high pass rates but low coverage: they write safe tests that revisit the same code. CovGreedy and CovQValue have lower pass rates, as the coverage map pushes the LLM toward more ambitious tests that target unexplored code. These tests fail more often, but they cover more branches. CovQValue offers the best trade-off, reaching the highest coverage with a higher average pass rate than CovGreedy. Line coverage and pass rates in Appendix C confirm these findings.

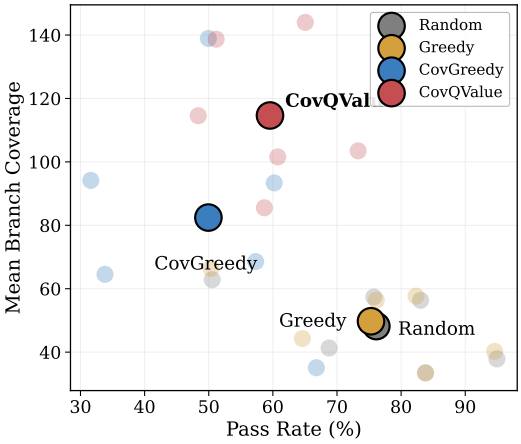


Figure 4: Trade-off between test pass rate and branch coverage. Large dots represent the mean across models and benchmarks, and transparent dots show each model on each benchmark. Coverage-map strategies discover more branches with more ambitious tests, at a cost in pass rate.

5.1 Ablation Studies

We run ablation studies for the key components of CovQValue on RepoExploreBench with Gemma 4 31B to understand what is driving the improvement. Results are summarized in Figure 5.

- **Execution budget** Figure 5a shows coverage as a function of the execution budget N. CovQValue keeps discovering new branches as the budget grows, while Random stays flat at every budget. Already at N=8, CovQValue reaches about twice the coverage that Random reaches at N=32.
- **Number of plans** Figure 5b varies the number of candidate plans K generated per round. Going from a single plan to three gives a clear improvement, since the scorer can then choose among plans aimed at different parts of the program, while five plans perform about the same as three. We also vary the plan length S in Appendix G: at a fixed number of executions, shorter plans reach slightly higher coverage because they receive feedback more often, but they need more LLM calls.

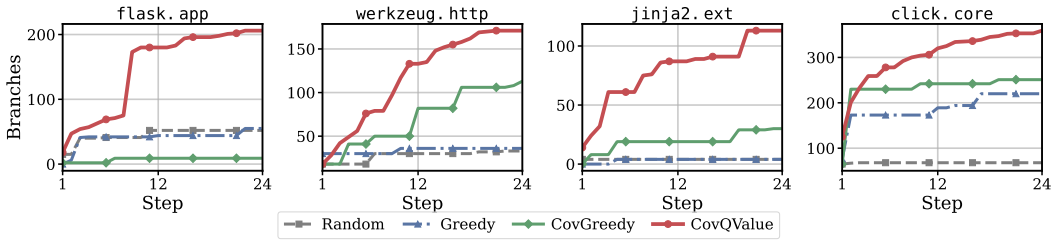


Figure 6: Corridor navigation on four RepoExploreBench modules (GLM 5.3). CovQValue keeps discovering branches after the baselines plateau, including through setup corridors.

- **Discount factor** (γ) The discount factor controls how much the Q-value weighs the future value relative to the immediate gain. Coverage changes little for γ between 0 and 1, and drops slightly when plans are ranked by the future value alone. On RepoExploreBench, the future-value term therefore has no measurable effect on the final coverage, and most of the benefit comes from which plans are proposed and how their immediate gain is estimated. Designing better future-value estimators is an area for future work.

- **Component contributions** Figure 5c (bottom) adds the components of CovQValue one at a time, starting from the coverage-map baseline. Diversity hints give a small improvement. Targeting plans at the functions the coverage map reports as unexecuted gives by far the largest gain, nearly doubling coverage. Scoring plans with the function-grounded estimate, instead of picking one at random, adds a further improvement. The LLM's predictions of which functions a plan will execute, on which this score relies, are correct 80% of the time (Appendix E). Adding the Bayesian valuation does not change coverage noticeably (86.5 vs. 85.5, within one standard error).

5.2 Case Study

To illustrate how CovQValue navigates corridor structure, Figure 6 shows the step-by-step branch coverage for four modules from RepoExploreBench with GLM 5.3. In `flask.app`, Random and Greedy reach about 40 branches within three steps and then stall, and CovGreedy stays below 10. CovQValue climbs steadily, then jumps from 75 to 173 branches in a single step (step 9) and ends at 206. In `werkzeug.http`, Random and Greedy stay near 30 branches, while CovGreedy makes partial progress in occasional jumps (113 branches); CovQValue climbs steadily from the first steps and reaches 171. In `jinja2.ext`, Random and Greedy never pass 4 branches, and CovGreedy reaches 30; CovQValue passes through the extension-registration corridor early (from 32 to 61 branches at step 4) and ends at 113. Finally, `click.core` shows that the gap persists when all baselines make progress: Greedy and CovGreedy jump to 173 and 230 branches at step 2 and then plateau, while CovQValue keeps discovering new branches throughout and reaches 359. Appendix J walks through the generated test scripts for one module in detail.

6 Conclusions

In this work, we framed LLM-based test generation as exploration of an unknown program, using a coverage map and a per-function reachability posterior as the exploration state, targeting candidate test plans at unexplored code, and selecting among them with a posterior-valued Q-value. To evaluate our method, we introduced *RepoExploreBench*, a benchmark designed for evaluating iterative exploration-based test generation. We evaluated our method, *CovQValue*, on RepoExploreBench and the public benchmark TestGenEval Lite. Our results show that CovQValue achieves higher branch coverage than greedy generation with consistent improvements across multiple LLMs. Our analysis shows that the coverage map alone is not enough: the major gains come from using the exploration state to decide where to explore next. Without coverage feedback, LLM-based selection is no better than random. We believe that the framework can also be applied to other settings where LLM agents must discover structure through interaction with an unknown environment and can become a general-purpose exploration module for agentic systems operating under partial observability.

Ethics Statement

Automated test generation can contribute to software reliability by increasing the coverage and precision of code evaluation. As more AI-generated code is deployed in the real world, the need for automatically generated tests grows. In this paper, we show methods that generate tests automatically without human review in the loop, and practitioners should treat generated tests with the same caution as other automatically generated tests.

Our framework gives the LLM greater autonomy in deciding what to explore, guided by its own metrics rather than explicit human instructions. In our experimental setting, exploration is naturally bounded by the program under test and the coverage map. In more open-ended real-world deployments, where an LLM agent interacts with live systems, APIs, or external services, such natural boundaries may not exist, and additional guardrails would be necessary to prevent unintended or harmful exploratory actions.

We acknowledge that further scaling these agents to other domains demands careful consideration of what constitutes safe exploratory actions. Future work should investigate guardrails that constrain the space of exploration, as well as the domains to which such methods could extend.

Reproducibility Statement

We will release the code, benchmark definitions, experiment configurations, and results; the link will be provided. RepoExploreBench is built on open-source repositories, and its targets and the Docker image for coverage measurement are included in the repository. TestGenEval Lite uses publicly available SWE-bench Docker images.

Gemini 3.8 Flash and GLM 5.3 are accessible through public APIs. Gemma 4 31B is an open-weight model available on Hugging Face, and we serve it locally with vLLM. Hyperparameters are fixed in the main experiments and reported in Section 4.

AI use statement

In this work, we used generative AI tools to help formulate the mathematical claims, as an assistant for designing or providing feedback on experiments and data analysis, and as a coding assistant for implementing the methods and for collecting and cleaning the data for RepoExploreBench. Additionally, we have used AI tools as an assistant to generate images and plots in this paper, create or edit software, and draft parts of the paper.

We have not used generative AI tools to generate the core idea of this paper or the fundamental method presented here. The rest of the required disclosure tasks are not applicable to this work. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

References

Juan Altmayer Pizzorno and Emery D Berger. Coverup: Effective high coverage test generation for python. *Proceedings of the ACM on Software Engineering*, 2(FSE):2897–2919, 2025.

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.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 572–576, 2024.

Deepro Choudhury, Sinead Williamson, Adam Goliński, Ning Miao, Freddie Bickford Smith, Michael Kirchhof, Yizhe Zhang, and Tom Rainforth. Bed-llm: Intelligent information gathering with llms and bayesian experimental design. *arXiv preprint arXiv:2508.21184*, 2025.

Michael Cooper, Rohan Wadhawan, John Michael Giorgi, Chenhao Tan, and Davis Liang. The curious language model: Strategic test-time information acquisition. *arXiv preprint arXiv:2506.09173*, 2025.

Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 416–419, 2011.

Zhiyuan Hu, Chumin Liu, Xidong Feng, Yilun Zhao, See-Kiong Ng, Anh T Luu, Junxian He, Pang W Koh, and Bryan Hooi. Uncertainty of thoughts: Uncertainty-aware planning enhances information seeking in llms. *Advances in Neural Information Processing Systems*, 37:24181–24215, 2024.

Kush Jain and Claire Le Goues. Testforge: Feedback-driven, agentic test suite generation. *arXiv preprint arXiv:2503.14713*, 2025.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark. *arXiv preprint arXiv:2410.00752*, 2024.

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? *arXiv preprint arXiv:2310.06770*, 2023.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 919–931. IEEE, 2023.

Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 168–172, 2022.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2023.

Jürgen Schmidhuber. Curious model-building control systems. In *Proc. international joint conference on neural networks*, pp. 1458–1463, 1991a.

Jürgen Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proc. of the international conference on simulation of adaptive behavior: From animals to animats*, pp. 222–227, 1991b.

Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE transactions on autonomous mental development*, 2(3):230–247, 2010.

Satinder Singh, Richard L Lewis, Andrew G Barto, and Jonathan Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*, 2(2):70–82, 2010.

Jan Storck, Sepp Hochreiter, Jürgen Schmidhuber, et al. Reinforcement driven information acquisition in non-deterministic environments. In *Proceedings of the international conference on artificial neural networks, Paris*, volume 2, pp. 159–164, 1995.

Yi Sun, Faustino Gomez, and Jürgen Schmidhuber. Planning to be surprised: Optimal bayesian exploration in dynamic environments. In *International conference on artificial general intelligence*, pp. 41–51. Springer, 2011.

WeiZhe Xu, Mengyu Liu, and Fanxin Kong. Enhancing llm-based test generation by eliminating covered code. *arXiv preprint arXiv:2602.21997*, 2026.

Chen Yang, Junjie Chen, Bin Lin, Ziqi Wang, and Jianyi Zhou. Advancing code coverage: Incorporating program analysis with large language models. *ACM Transactions on Software Engineering and Methodology*, 2024.

Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726, 2024.

A REPOEXPLOREBENCH BENCHMARK CREATION

RepoExploreBench is our benchmark designed to evaluate exploration-based test generation on real-world Python code available online with corridor structure.

Repository selection. We selected 9 packages from the top 150 most-downloaded (as of March 2026) PyPI packages (30-day rolling window) using three criteria: (1) pure Python source with $\geq$5K lines, (2) no C extensions, build tools, or type stubs, and (3) corridor structure such as deep import chains, class hierarchies, and configuration requirements.

Module selection. From each package, we selected approximately 10 modules using the following criteria:

1. Minimum 200 source lines (enough branches for meaningful exploration)
2. Larger modules preferred (more paths to discover)
3. Functional diversity within each repo (avoid selecting similar utilities)
4. Each module must expose callable classes or functions, not just constants
5. Exclude test-only modules (except testing utilities such as `click.testing`)
6. Exclude private modules (except httpx, which uses _ prefix by convention)
7. Exclude vendored code and legacy compatibility layers

Module candidates were identified by enumerating all submodules in each package and verifying their line counts. Each package contains many more modules meeting the minimum size criterion; we sample approximately 10 per package for a manageable evaluation set, but the benchmark can be scaled without additional infrastructure. This yields 93 targets totaling 77,242 lines of code. Table 4 summarizes the benchmark composition.

Table 4: RepoExploreBench composition. Packages selected from top 150 PyPI downloads (as of March 2026). Corridor structure describes the sequential dependencies that gate access to deep branches.

Package	Domain	Mod.	Lines	Corridor Structure
click	CLI framework	10	8,293	Command setup → option parsing → type coercion
flask	Web framework	10	5,369	App creation → config → routing → request context
httpx	Async HTTP client	10	6,925	Client config → auth → transport → URL parsing
jinja2	Template engine	12	12,751	Env setup → lexer → parser → AST → compiler
pydantic	Data validation	10	11,727	Model definition → schema → validation → serialization
requests	HTTP client	6	4,375	Session setup → auth → adapters → connection pool
rich	Terminal rendering	12	12,792	Console setup → style parsing → segment → rendering
starlette	ASGI framework	11	4,625	App → routing → middleware → request/response
werkzeug	WSGI utilities	12	10,385	Routing setup → request parsing → response building
Total		**93**	**77,242**	

Execution environment. All modules execute inside a single Docker image with all 9 packages pre-installed. Branch coverage is measured using Python's standard branch coverage tooling.

B ALGORITHM

Our method, which we call **CovQValue**, operates as follows. Given a target source file and an execution budget of N test executions:

Each plan consists of S test scripts, designed as a coherent sequence. For example, the first script may set up fixtures, the second may exercise a specific code path, and the third may probe edge cases exposed by the previous steps. Because the chunks are contiguous in source order, a chunk usually covers the methods of one class, so the plan must discover that class's setup to reach them. If fewer than K chunks remain, the remaining plans receive a generic exploration directive (e.g., "focus on error handling") instead of a target.

Algorithm 1 CovQValue: Curiosity-Driven Test Plan Selection

Require: Source file P with functions F, execution budget N, number of plans K, discount γ, steps per plan S, unlock increment λ
Ensure: Coverage map $\mathcal{C}$

$\mathcal{C} \leftarrow \emptyset$; $h \leftarrow \emptyset$ {Coverage map, test history}
$(\alpha_f, \beta_f) \leftarrow (1, 1)$ for all $f \in F$ {Reachability posterior}
while executions remaining > 0 **do**
  $U \leftarrow \{f \in F :$ no line of f executed in $\mathcal{C}\}$
  $(c_1, \ldots, c_K) \leftarrow \textsc{Chunk}(U, K)$ {Contiguous, in source order}
  // Generate K targeted plans in parallel
  for $k = 1$ **to** K **do**
    $\pi_k \leftarrow \text{LLM.}\textsc{GeneratePlan}(P, \mathcal{C}, h, c_k)$ {Each $\pi_k = (t_1^k, \ldots, t_S^k)$}
  end for
  // Perceive, then value with the posterior
  for $k = 1$ **to** K **do**
    $(E_k, N_k) \leftarrow \text{LLM.}\textsc{ExecutesEnables}(\pi_k, U)$
    $Q_k \leftarrow \sum_{f \in E_k} p_f |f| + \gamma\lambda \sum_{f \in N_k} (1 - p_f)|f|$ {Eq. 3, $\lambda = 0.09$}
  end for
  // Select and execute best plan
  $k^* \leftarrow \arg\max_k Q_k$ {Ties broken uniformly at random}
  for $s = 1$ **to** S **do**
    $o_s \leftarrow \textsc{Execute}(P, t_s^{k^*})$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \textsc{Branches}(o_s)$; $h \leftarrow h \cdot (t_s^{k^*}, o_s)$
  end for
  // Update reachability estimates
  for $f \in E_{k^*} \cup N_{k^*}$ **do**
    if f now covered in $\mathcal{C}$ **then**
      $\alpha_f \leftarrow \alpha_f + 1$
    else
      $\beta_f \leftarrow \beta_f + 1$
    end if
  end for
end while
return $\mathcal{C}$

C Line Coverage and Pass Rates

Table 5 reports **line coverage** and **test pass rates** for all strategies and models on both benchmarks. The trends mirror the branch coverage results. CovQValue achieves the highest line coverage in every model and benchmark, while Random and Greedy plateau at lower values.

Table 5: Line coverage and pass rate. Results are consistent with branch coverage.

	Gemini 3.8 Flash		Gemma 4 31B		GLM 5.3	
Strategy	Lines	Pass%	Lines	Pass%	Lines	Pass%
RepoExploreBench						
Random	137	95%	121	84%	137	69%
Greedy	142	95%	121	84%	143	65%
CovGreedy	246	60%	118	67%	184	34%
CovQValue	**267**	73%	**236**	59%	**262**	61%
TestGenEval Lite						
Random	146	83%	130	76%	153	51%
Greedy	148	82%	131	76%	154	50%
CovGreedy	246	50%	138	57%	183	32%
CovQValue	**252**	65%	**204**	48%	**241**	51%

D NORMALIZED COVERAGE

Absolute branch counts give more weight to large modules. To check that our conclusions do not depend on a few large targets, Table 6 reports coverage normalized per target: for each target, we divide the coverage of a run by the highest coverage reached on that target by any strategy, model, or seed in our experiments, and then average these percentages over targets. Each target therefore contributes equally, regardless of its size. This normalization measures coverage relative to the best observed run rather than the fraction of all branches in the file. CovQValue achieves the highest normalized coverage for every model and benchmark, so the ranking of Table 2 is not driven by large modules.

Table 6: Mean per-target coverage as a percentage of the best coverage observed on that target ($\pm$ standard error). Targets on which no run covered any branch are excluded.

	RepoExploreBench			**TestGenEval Lite**		
	Gemini 3.8 Flash	Gemma 4 31B	GLM 5.3	Gemini 3.8 Flash	Gemma 4 31B	GLM 5.3
Random	37.5% ±2.7	33.2% ±2.5	41.8% ±2.7	36.8% ±2.3	35.6% ±2.2	42.5% ±2.3
Greedy	38.8% ±2.6	32.9% ±2.5	43.5% ±2.7	37.9% ±2.3	35.3% ±2.3	43.4% ±2.3
CovGreedy	84.7% ±1.9	33.8% ±2.7	60.3% ±2.9	82.0% ±1.9	45.4% ±2.4	60.9% ±2.4
CovQValue	**94.5%** ±1.1	**78.1%** ±1.7	**92.5%** ±1.0	**88.2%** ±1.6	**73.1%** ±1.9	**84.4%** ±1.8

E ACCURACY OF THE LLM'S PREDICTIONS

CovQValue's Q-value is computed from the LLM's predictions of which untested functions a plan will execute. To measure how accurate these predictions are, we record, at every round, the untested functions and the LLM's prediction for the plan that is executed, and compare it with the functions the plan actually executes (those that are no longer untested after the round). Each untested function is one yes/no prediction. Table 7 reports the resulting accuracy, precision, recall, and F1 score with Gemma 4 31B.

On both benchmarks, the LLM's predictions are correct 80% of the time. On RepoExploreBench, it identifies 81% of the functions a plan executes and correctly excludes 80% of those it does not; its main error is over-prediction (precision 49%), naming functions that the plan does not reach. On TestGenEval Lite, precision and recall are lower. Its files come from large projects that need more setup (e.g., Django), and the generated tests fail more often, so a plan frequently stops before reaching the functions it targets; this is a likely cause of the lower precision. The posterior partly compensates for this, since functions whose predicted execution repeatedly fails receive a lower reachability estimate p_f and therefore count less in the Q-value. Predictions of which functions a plan makes reachable for later tests carry a weaker signal: such functions are covered in the next round 17.8% of the time, compared with 15.2% for the other untested functions, consistent with the small effect of the future-value term in our ablations.

Table 7: Accuracy of the LLM's predictions of which untested functions a plan will execute (Gemma 4 31B). Each untested function at each round is one prediction.

Metric	**RepoExploreBench**	**TestGenEval Lite**
Accuracy	0.80	0.80
Precision	0.49	0.27
Recall	0.81	0.42
F1	0.61	0.32
Predictions	12,178	34,538

F Inference Cost

All strategies receive the same number of test executions, but they differ in how many LLM calls they make. Random and CovGreedy generate $K=3$ candidates at each of the 24 steps (72 calls), and Greedy adds one selection call per step (96 calls). CovQValue generates and scores $K=3$ plans in each of its 8 rounds (48 calls). Table 8 reports the measured calls and tokens per module on the 10 `click` modules of RepoExploreBench with Gemma 4 31B. CovQValue makes the fewest calls and uses about as many tokens as Random, so its higher coverage does not come from a larger inference budget.

Table 8: Inference cost per module on the 10 `click` modules of RepoExploreBench (Gemma 4 31B, 24 test executions).

Strategy	Branches	LLM calls	Input tokens	Output tokens
Random	30.4	72	62.6K	10.2K
Greedy	18.9	96	79.3K	9.5K
CovGreedy	23.7	72	75.1K	16.3K
CovQValue	**74.5**	**48**	57.1K	16.1K

G Plan Length

Table 9 varies the plan length S while keeping the budget fixed at 24 test executions (Gemma 4 31B, RepoExploreBench). Shorter plans reach slightly higher coverage, because the coverage map and the posterior are updated after every plan and short plans receive feedback more often. They are also more expensive: each round generates and scores $K=3$ plans, so $S=1$ needs three times as many LLM calls as $S=3$. We use $S=3$ as a compromise between coverage and cost.

Table 9: Plan length S at a fixed budget of 24 test executions (RepoExploreBench, Gemma 4 31B). LLM calls are per module.

	$S=1$	$S=3$	$S=5$
Rounds	24	8	5
LLM calls	144	48	30
Branch coverage	89.1 $\pm$7.2	85.5 $\pm$7.1	79.9 $\pm$6.5

H Per-repo Data

Here, we provide per-repository breakdowns of branch coverage for both benchmarks. Figure 7 visualizes the mean coverage on RepoExploreBench averaged for all models. Tables 10 and 11 give the full numbers for TestGenEval Lite and RepoExploreBench in every model. Averaged across models, CovQValue achieves the highest coverage on every repository in both benchmarks. Per model, it is best in all but three cells, all with Gemini 3.8 Flash, where CovGreedy is ahead: `matplotlib` and `sympy` on TestGenEval Lite and `click` on RepoExploreBench. The gap between CovQValue and CovGreedy is largest with Gemma 4 31B, where CovGreedy improves only slightly over Random.

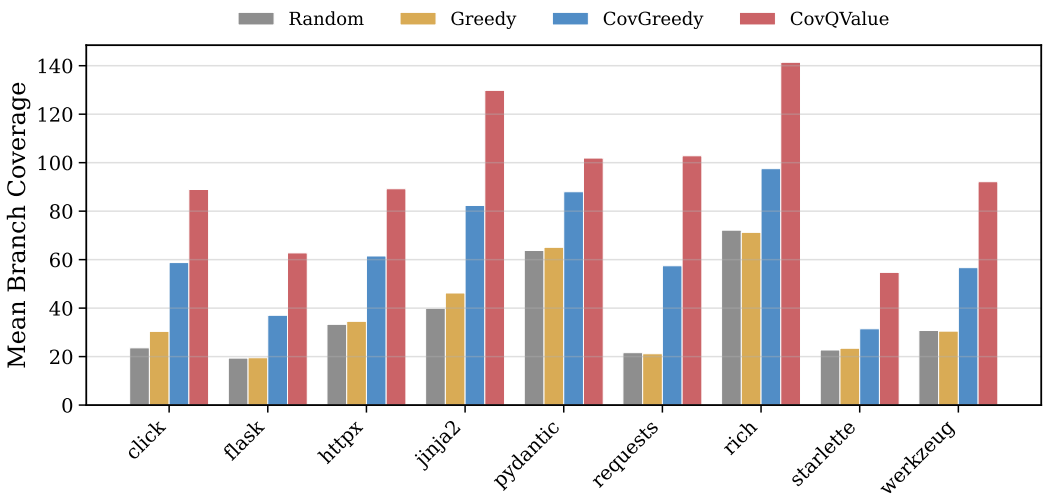


Figure 7: Mean branch coverage by repository on RepoExploreBench, averaged across Gemini 3.8 Flash, Gemma 4 31B, and GLM 5.3. CovQValue outperforms all baselines on every repository.

Table 10: Per-repository branch coverage on TestGenEval Lite by model. Each cell averages across modules in that repo. Mean row shows the per-module average. Best per group in bold.

	Gemini 3.8 Flash				Gemma 4 31B				GLM 5.3			
Repo	Rnd	Gdy	CG	CQ	Rnd	Gdy	CG	CQ	Rnd	Gdy	CG	CQ
astropy	29	24	116	**153**	44	43	47	**112**	39	35	49	**144**
django	72	77	138	**142**	59	61	73	**113**	77	76	98	**137**
matplotlib	74	74	**176**	157	72	68	91	**117**	86	88	125	**154**
seaborn	28	37	44	**61**	9	9	10	**40**	15	24	14	**40**
flask	10	14	70	**78**	17	14	22	**87**	22	18	25	**84**
xarray	36	37	74	**141**	29	32	47	**133**	37	63	56	**142**
pylint	10	8	20	**28**	7	7	7	**8**	10	10	18	**26**
pytest	27	20	56	**75**	23	23	34	**61**	26	21	37	**62**
sphinx	16	17	29	**80**	14	14	28	**35**	28	11	36	**82**
sympy	47	45	**182**	178	72	68	80	**144**	59	71	117	**175**
Mean	56.3	57.7	138.9	**144.0**	57.4	56.4	68.6	**114.6**	62.8	66.4	94.2	**138.7**

I Cumulative Branch Coverage per Model

Figures 8 and 9 show the cumulative branch coverage for each model separately, complementing the model average in Figure 2.

J Case Study: Generated Test Scripts

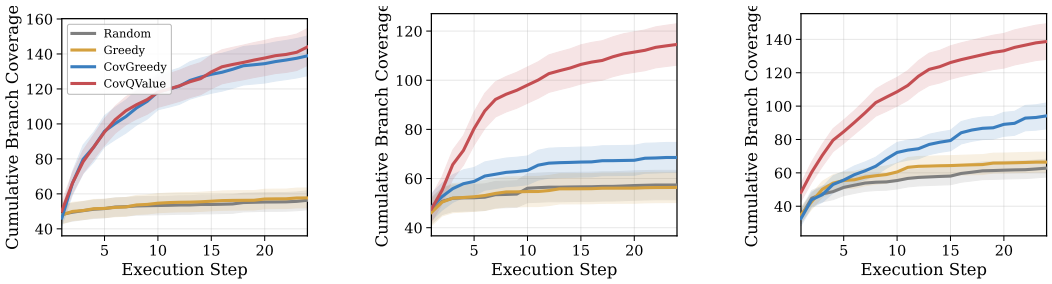
We examine the test scripts generated for `werkzeug.http` with GLM 5.3, the run shown in Figure 6 (second panel). Greedy ends at 36 branches and CovQValue at 171.

Module structure. `werkzeug.http` implements over 40 functions for parsing and generating HTTP headers: dates and ETags, content-type options, `Accept` headers, byte ranges, cookies, cache control, and conditional requests.

Greedy (36 branches). Greedy's first script discovers 30 branches by calling the date and ETag helpers. In the remaining 23 steps it finds new branches only once (+6 at step 10); every other script is a variation of the first, for example step 2:

Table 11: Per-repository branch coverage on RepoExploreBench by model. Each cell averages across modules in that repo; Gemma 4 31B is also averaged over 3 seeds. Mean row shows the per-module average. Best per group in bold.

	Gemini 3.8 Flash				Gemma 4 31B				GLM 5.3			
Repo	Rnd	Gdy	CG	CQ	Rnd	Gdy	CG	CQ	Rnd	Gdy	CG	CQ
click	25	23	**99**	96	22	27	24	**76**	24	41	53	**94**
flask	15	16	55	**62**	17	17	22	**52**	26	26	34	**75**
httpx	31	35	82	**91**	34	32	32	**83**	35	37	70	**94**
jinja2	42	52	131	**138**	38	37	31	**115**	39	50	84	**137**
pydantic	63	67	98	**105**	55	56	78	**98**	73	72	88	**102**
requests	20	21	83	**112**	17	17	26	**89**	27	25	64	**108**
rich	77	78	148	**154**	62	60	55	**128**	77	76	90	**143**
starlette	21	25	45	**62**	20	19	15	**45**	27	26	35	**58**
werkzeug	33	32	84	**103**	25	26	29	**77**	34	34	57	**96**
Mean	37.9	40.2	93.4	**103.5**	33.5	33.4	35.1	**85.5**	41.3	44.3	64.5	**101.6**



(a) Gemini 3.8 Flash (b) Gemma 4 31B (c) GLM 5.3

Figure 8: Cumulative branch coverage over execution steps on TestGenEval Lite plotted individually for each model.

```
from werkzeug.http import (parse_date, http_date,
    parse_etags, generate_etag)
from datetime import datetime, timezone

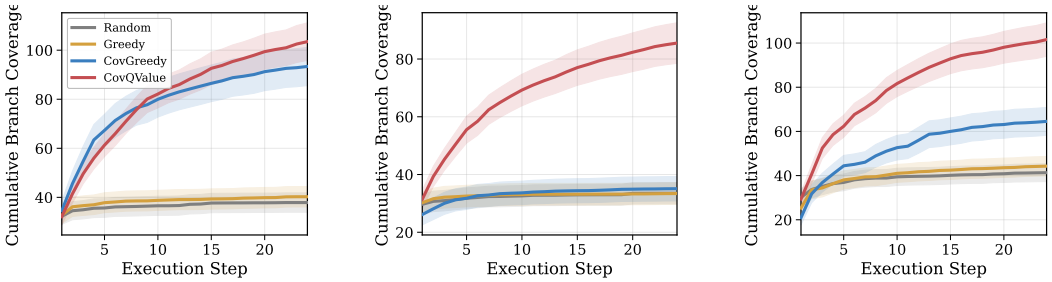
d = parse_date("Mon, 15 Jan 2024 12:00:00 GMT")
print(http_date(d))
print(d == datetime(2024, 1, 15, 12, 0, tzinfo=timezone.utc))
print(parse_etags('W/"abc", "def"'))
print(generate_etag(b"hello world"))
```

Without coverage feedback, Greedy has no signal that the rest of the module is untested. The LLM picks the candidate it considers "most likely to cover new code paths," and keeps choosing the same familiar functions.

CovQValue (171 branches). CovQValue's plans move through different groups of the module's functions from round to round, and 19 of its 24 scripts discover new branches. Step 6 (+20 branches) exercises the content-type and `Accept` parsers with quoted and escaped values (comments omitted):

```
from werkzeug.http import (parse_options_header,
    parse_accept_header)

print(parse_options_header('text/html; charset=utf-8; q=0.9'))
print(parse_options_header('"application/x-foo\\"bar"; name="a\\\\b"'))
print(parse_options_header(''))
acc = parse_accept_header('text/html;q=0.5, */*;q=0.1, application/json')
print(list(acc))
```



(a) Gemini 3.8 Flash (b) Gemma 4 31B (c) GLM 5.3

Figure 9: Cumulative branch coverage over execution steps on RepoExploreBench plotted individually for each model.

```
print(parse_accept_header(None))
```

Step 9 (+18 branches) moves on to byte ranges and cookie edge cases, which no earlier test had touched:

```
from werkzeug.http import (is_byte_range_valid,
    parse_range_header, parse_cookie)

print(is_byte_range_valid(0, 499, 1000))
print(is_byte_range_valid(500, 400, 1000))
print(is_byte_range_valid(500, 600, 500))
print(parse_range_header("bytes=0-499").ranges)
print(parse_range_header("bytes=500-999, -200").ranges)
print(parse_cookie("quoted=\"a;b=1\"; bare", cls=dict))
print(parse_cookie("=", cls=dict))
print(parse_cookie("empty="))
```

The following rounds continue with age and cookie serialization (step 10, +20 branches) and conditional requests (step 11, +16 branches). Each plan is aimed at functions the coverage map still reports as unexecuted, so the rounds cover different parts of the module instead of revisiting the same functions, as Greedy does.

K Prompts and Coverage Map

In this appendix, we present the prompts used in CovQValue and show how the coverage map evolves during exploration. The plan generation prompt receives the module source code (truncated to 2500 characters), the coverage map, the three most recent tests, and the plan's target. The scoring prompt receives only the list of still-unexecuted functions, the functions already executed, and the plan.

K.1 Coverage Map

The coverage map is fed to the LLM as structured text. Below we show its evolution for `werkzeug.http` during the GLM 5.3 CovQValue run from Figure 6, at three of the points where a new round of plans is generated. Each informative test is identified by the first 80 characters of its script, which can span two lines.

Coverage Map — Round 1 (initial state)

```
COVERAGE MAP (Bayesian posterior -- what you know about the program):
  Branches discovered: 0
```

Coverage Map — Round 2, after Step 3 (42 branches)

```
COVERAGE MAP (Bayesian posterior -- what you know about the program):
  Branches discovered: 42
  Last step discovered: 14 new branches
  Average per step: 14.0 branches
  Most informative tests:
    from werkzeug.http import parse_cache_control_header
      from werkzeug.datastructure -> 18 branches
    from werkzeug.http import parse_set_header, parse_if_range_header
      from werkzeug. -> 14 branches
    from werkzeug.http import parse_csp_header
      from werkzeug.datastructures import C -> 10 branches
```

Coverage Map — Round 5, after Step 12 (133 branches)

```
COVERAGE MAP (Bayesian posterior -- what you know about the program):
  Branches discovered: 133
  Last step discovered: 0 new branches
  Average per step: 11.1 branches
  Most informative tests:
    from werkzeug.http import parse_options_header, parse_accept_header
      # token hea -> 20 branches
    from datetime import timedelta
      from werkzeug.http import dump_age, dump_cookie,  -> 20 branches
    from werkzeug.http import parse_cache_control_header
      from werkzeug.datastructure -> 18 branches
```

When the most recent steps discover no new branches, the map also adds a stagnation warning ("Coverage has stagnated for n steps. You may need a fundamentally different approach"). In this run, the warning never appeared at a round boundary.

K.2 PLAN GENERATION PROMPT

Each of the K=3 plans receives a different target: a contiguous chunk of up to 6 still-unexecuted functions, listed with their sizes, followed by their source (up to 1500 characters, since the module source above is truncated). When fewer than K chunks remain, the target line is replaced by a generic directive: "*Focus on the MAIN functionality – constructors, primary methods*", "*Focus on ERROR HANDLING – invalid inputs, edge cases, exceptions*", or "*Focus on INTERACTIONS – create objects, pass them to each other, chain calls*".

Plan Generation Prompt (with target)

```
Module: {module_name}
{source code (truncated to 2500 chars)}

{coverage map}

Previous tests:
  {test_1} -> {output} (new branches: 31)
  {test_2} -> {output} (new branches: 4)
  {test_3} -> {output} (new branches: 19)

PLAN a sequence of 3 test scripts that TOGETHER will
reach UNCOVERED code paths. TARGET these functions,
which still have UNEXECUTED code: {f1} ({n1} lines
unexecuted), {f2} ({n2} lines unexecuted), ...
Plan the setup and inputs needed to reach their
unexecuted parts and call them.
```python
{source of the targeted functions}
```

Think about what setup is needed:
Step 1: What basic setup/import is needed to reach
        deeper code?
Step 2: Building on step 1's result, what exercises
        the next layer?
Step 3: Now target the deepest uncovered branches.

For each step, write a separate test script (5-15
lines each). Import from the module and print results.

Format your response as:
TEST 1
```python
[code]
```
TEST 2
```python
[code]
```
TEST 3
```python
[code]
```
```

K.3 Q-VALUE SCORING PROMPT

After generating K plans, each is scored by the following prompt. The list of unexecuted functions is sorted by size (at most 80 shown), and the list of executed functions is truncated to 60 names. From the response we keep only names that appear in the unexecuted list. Names under `EXECUTES` are the functions the plan is predicted to run. Names under `ENABLES` are the functions it makes easier to reach; we keep the first 10 that come with a reason ("<−") and are not already under `EXECUTES`.

Q-Value Scoring Prompt

```
Module: {module_name}

FUNCTIONS WITH CODE NO TEST HAS EXECUTED YET:
  {f1} ({n1} lines unexecuted)
  {f2} ({n2} lines unexecuted)
  ...

TEST PLAN (scripts run in order):

Step 1:
```python
{script_1}
```
Step 2:
```python
{script_2}
```
Step 3:
```python
{script_3}
```

FUNCTIONS ALREADY EXECUTED BY EARLIER TESTS (their
setup is already known):
  {g1}, {g2}, ...

Answer in exactly two sections, using ONLY names from
the NOT-YET-EXECUTED list:
EXECUTES:
<functions this plan will actually execute, directly
or indirectly; one per line>
ENABLES:
<at most 10 functions this plan does NOT execute,
which a test could NOT easily call without first doing
something this plan does for the first time (building
a specific object, reaching a configured state,
finding valid inputs). Setup that earlier tests
already did does not count. One per line, formatted
as:
function_name <- what it needs from this plan>
Write NONE for an empty section.
```

The two sets are then valued with the posterior using Equation 3, and the plan with the highest $\hat{q}$ is selected for execution.