

Bayesian Control for Coding Agents

Theodore Papamarkou^{1,2} $\diamond$ Vladislav Smirnov³ $\diamond$ Viktor Mazanov⁴ $\diamond$ Artem Vazhentsev³

Preslav Nakov³ Timothy Baldwin³ Artem Shelmanov³

¹PolyShape ²National Technical University of Athens ³MBZUAI ⁴Innopolis University

$\diamond$ Equal contribution

Abstract

Modern coding agents pair LLM generators with various tools, including cheap diagnostics and expensive verifiers. The tool-use decisions are typically governed by orchestrators that often use fixed rules and ignore uncertainty. We formulate orchestration as cost-sensitive sequential hypothesis testing: a Bayesian controller maintains a belief over candidate correctness and dynamically decides whether to gather more evidence, refine the candidate, verify it, or stop. Across six generators and nine coding benchmarks, Bayesian control proves to be most valuable when verification is costly and critics are informative but imperfect. Beyond control, the belief state yields an interpretable correctness score that outperforms token-probability and raw tool-success baselines for uncertainty quantification.¹

1 Introduction

Large Language Models (LLMs) are increasingly capable of generating sophisticated code, but their practical impact is most fully realized through coding agents that orchestrate generation, diagnosis, refinement, and verification. A modern coding-agent harness typically wraps an LLM-based generator with auxiliary tools, such as syntax checkers, public tests, LLM-based reviewers, refinement prompts, etc. (Yang et al., 2024; Wang et al., 2025). Because these tools vary in cost and reliability, the central engineering problem is to produce correct code while efficiently allocating limited resources by strategically deciding when to refine, discard, or verify a candidate solution.

Many existing orchestration policies address this problem using fixed rules, such as always verifying, best-of- N sampling, single-critic gates, or predefined generate–critique–regenerate–verify loops (Inala et al., 2022; Li et al., 2022; Chen et al.,

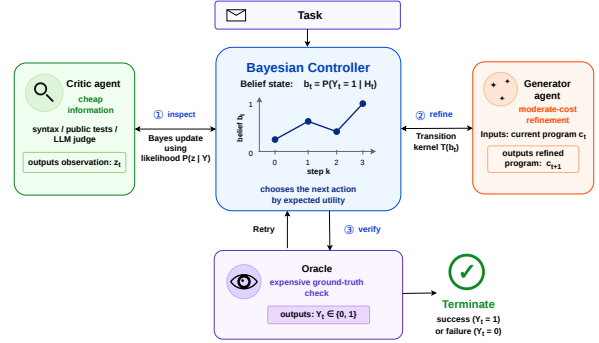


Figure 1: Illustration of the Bayesian controller.

2023; Huang et al., 2023). Although such policies can be competitive, they do not maintain a posterior over candidate correctness or explicitly weigh the value of critic calls against their cost. As a result, they are not designed to adapt stopping decisions to prior task difficulty, critic reliability, generator repair probability, or verifier cost.

In this work, following Papamarkou et al. (2026), we study code-generation control as a Bayesian decision problem. The latent state is candidate correctness, $Y \in \{0, 1\}$. The controller maintains a belief $b = P(Y = 1 | \text{evidence})$ over whether the current candidate will pass the verifier. Critic calls are treated as noisy observations, generator calls as stochastic transitions that may fix or break a candidate, and verifier calls as costly terminal actions. The objective is expected utility, that is, reward for a correct solution minus the accumulated costs of generation, criticism, refinement, and verification. This yields a sequential hypothesis-testing problem that can be encoded as a Partially Observable Markov Decision Process (POMDP) (Kaelbling et al., 1998), whose Bellman equation expresses the value of information. Figure 1 illustrates our Bayesian control loop.

We implement this idea with two controllers derived from the Bellman equation. The first is a one-step Bayesian greedy controller that uses

¹https://github.com/rvz16/bayesian_control_for_coding_agents

critic likelihoods to decide whether to call another critic, verify, regenerate, or stop. The second is a finite-horizon Bayesian dynamic-programming (DP) controller that performs backward induction and uses a measured generator transition kernel, $\hat{P}(\text{fix} \mid \text{broken})$ and $\hat{P}(\text{break} \mid \text{correct})$, estimated from iterative refinement trajectories. Both controllers operate on top of frozen LLMs: the base generators are not trained, and the optimization problem is entirely at the control layer.

We ask a series of empirical questions: when does a Bayesian belief-state controller improve cost-adjusted code generation; when does a simple heuristic suffice; and when is verification so cheap or the prior probability of success so high that no orchestration is needed?

We evaluate six generators across nine coding benchmarks and compare policies in terms of expected utility under varying cost-reward trade-offs. The results show that Bayesian control is most useful in low-prior regimes with informative but imperfect critics, while simple public test gating or always-verifying policies are preferable when public tests are highly predictive of hidden-test success, candidate patches are already likely to be correct, or verification is cheap. Dynamic programming adds value only when measured repair transitions make multi-step refinement worthwhile. Our empirical study supports these conclusions through out-of-sample validation, comparisons with stronger refinement baselines, and negative results that identify when naive Bayesian variants fail.

Contributions. Our contributions are as follows. (1) We formulate code-generation control as a cost-sensitive sequential hypothesis-testing problem over candidate correctness. (2) We derive two Bayesian belief-state controllers from the Bellman equation, namely a one-step greedy controller and a finite-horizon dynamic-programming controller. (3) Experiments across diverse generators and benchmarks reveal three regimes governing which orchestration strategy dominates and clarify when DP planning beats the greedy policy. (4) We further show that the posterior beliefs induced by our hypothesis-testing framework serve as effective uncertainty scores for the coding agent that could be used to detect low-quality agent outputs.

2 Related Work

Bayesian decision theory. Our approach uses sequential analysis and the value of information.

Wald (1947) studied sequential tests that trade error against sampling cost, while Howard (1966) formalized the economic value of evidence. POMDPs generalize this logic to control under latent state uncertainty (Kaelbling et al., 1998). The recent work of Papamarkou et al. (2026) argues that the control layer of agentic AI should maintain beliefs over task-relevant latent quantities and act by utility. Our paper implements this idea for code generation. The latent variable is candidate correctness, the observations are syntax checks, public tests, and LLM judges, the transitions are refinement steps, and the terminal action is an expensive verifier.

Bayesian program synthesis and adaptive computation. Bayesian program-induction methods place priors over program structures and infer posteriors from examples (Liang et al., 2010). Our framework instead maintains a posterior over the correctness of an LLM-generated candidate and uses it to select actions. Adaptive computation methods vary inference-time compute according to prediction difficulty. For language generation, CALM uses confidence-based early exits to allocate different computation across tokens (Schuster et al., 2022). Our controller operates at a higher level, allocating calls among generators, critics, and verifiers rather than layers within a model.

POMDP methods for LLM agents. Recent work has begun to use Bayesian information acquisition for LLM agents. Choudhury et al. (2026) use sequential Bayesian experimental design to choose questions that maximize expected information gain in multi-turn information gathering. Suri et al. (2026) model tool-argument clarification as structured uncertainty and use a POMDP to choose clarifying questions. These papers are close in spirit because they treat interaction as costly information acquisition. They differ from our setting in the latent state, action space, and empirical target. We study code-generation control, where the agent must decide whether to buy more diagnostic evidence, refine a candidate, or pay for verification, and we evaluate utility across coding benchmarks rather than clarification success or information-gathering games.

Coding agents. A large line of work is devoted to coding agents. Generated tests and reranking have been used to select better candidates (Chen et al., 2023; Inala et al., 2022; Zhang et al., 2023). Iterative self-feedback and verbal memory methods refine model outputs over multiple turns (Madaan

et al., 2023; Shinn et al., 2023). Multi-agent and tree-search systems such as AgentCoder and LATS introduce richer generate-test-refine or planning loops (Huang et al., 2023; Zhou et al., 2024). Repository-level agents such as SWE-agent and OpenHands focus on interfaces, sandboxed execution, and autonomous tool use for realistic software engineering tasks (Yang et al., 2024; Wang et al., 2025). Our empirical setting follows established code-generation and program-repair benchmarks, including LiveCodeBench, SWE-Bench, EvalPlus, HumanEvalFix, and CodeContests (Jain et al., 2025; Jimenez et al., 2024; Liu et al., 2023; Muennighoff et al., 2024; Li et al., 2022).

These methods and benchmarks demonstrate the value of external feedback, iterative control, and agentic stopping policies, and provide the experimental basis for our evaluation. Our approach differs in quantifying the agent’s uncertainty as a posterior belief over candidate correctness, which serves both as a decision state for cost-aware control and as an uncertainty score. Existing agentic harnesses can thus be augmented with posterior beliefs rather than replaced, and our cost-and-reliability regime analysis identifies when such beliefs improve over fixed workflows and when simpler policies suffice.

3 Task and Environment

We consider the problem of automated program synthesis under resource constraints. This section formalizes the task environment, the abstraction of computational agents, and the decision-theoretic objective that drives control.

Let $\mathcal{X}$ denote the space of natural language specifications and $\mathcal{C}$ the space of executable programs. A task instance consists of a specification $x \in \mathcal{X}$ and an oracle verification function $\mathcal{O} : \mathcal{C} \times \mathcal{X} \rightarrow \{0, 1\}$, which returns 1 if and only if program c satisfies the requirements of x . Access to $\mathcal{O}$ is mediated by a *verifier agent* a_{ver} , hereafter called the *oracle*, whose invocation incurs a significant computational or monetary cost C_{ver} . For any candidate c_t , let $Y_t = \mathcal{O}(c_t, x) \in \{0, 1\}$ be its latent correctness, indicating whether the candidate passes oracle verification ($Y_t = 1$) or not ($Y_t = 0$).

To actively improve the candidate program, we can invoke an LLM G_θ with fixed parameters θ . We denote this *generator agent* by a_{gen} . At iteration t , given a specification x and a current candidate c_t , the generator produces a new candidate $c_{t+1} \sim$

$G_\theta(\cdot \mid x, c_t)$ at cost C_{gen} . The new candidate patch c_{t+1} may improve the program, but it may also leave it unchanged or make it worse.

We assume access to k critic agents $\{a_{\text{crit}}^1, \dots, a_{\text{crit}}^k\}$ that inspect c_t without modifying it. Critic i returns a cost-effective noisy diagnostic observation $z_t^{(i)} \in \{\text{pass}, \text{fail}\}$ with likelihood $P_i(z \mid Y_t)$ and cost C_{crit}^i .

A control policy π orchestrates the program generation process with the goal of producing a program that satisfies x . It interacts with a set of agents $\mathcal{A} = \{a_{\text{gen}}, a_{\text{ver}}, a_{\text{crit}}^1, \dots, a_{\text{crit}}^k\}$ over discrete time steps $t = 1, \dots, T$. At each step, the policy selects an agent $a_t \in \mathcal{A}$ based on the current history $\mathcal{H}_t = (x, c_1, a_1, o_1, \dots, a_{t-1}, o_{t-1}, c_t)$, where o_t denotes the observation returned by the selected agent. The quality of a policy is measured by its expected utility, defined as the value of the final outcome minus the accumulated agent costs. Formally, if an episode terminates at a step T with candidate c_T , the expected utility is

$$U(\pi) = \mathbb{E}_\pi \left[R \cdot Y_T - \sum_{t=1}^T C(a_t) \right], \quad (1)$$

where R represents the reward for a correct solution ($Y_T = 1$), and $C(a_t)$ is the cost of the selected agent: $C(a_{\text{gen}}) = C_{\text{gen}}$, $C(a_{\text{crit}}^i) = C_{\text{crit}}^i$, and $C(a_{\text{ver}}) = C_{\text{ver}}$.

Our goal is to derive a policy that maximizes this utility by estimating the *action value* (or *Q-value*) of each agent, thus quantifying the trade-off between the cost of an action and its expected contribution to solving the task.

4 Bayesian Decision-Theoretic Control

General design of Bayesian control. We adopt Bayesian decision theory through a POMDP abstraction. At time step t , the controller holds a candidate program c_t and a belief state $b_t = P(Y_t = 1 \mid \mathcal{H}_t)$ over its correctness, and selects actions by trading off the cost of information, improvement and verification. The belief update process constitutes a sequential hypothesis test, iteratively accumulating evidence to decide between the null hypothesis $Y_t = 0$ and the alternative $Y_t = 1$.

If $a_t = a_{\text{gen}}$, the belief is pushed through a transition kernel $\mathcal{T}(b_t)$, which accounts for the probability that the generator introduces a bug into a correct program or fixes an incorrect one,

$$\mathcal{T} = \begin{pmatrix} P(Y_{t+1} = 0 \mid Y_t = 0) & P(Y_{t+1} = 1 \mid Y_t = 0) \\ P(Y_{t+1} = 0 \mid Y_t = 1) & P(Y_{t+1} = 1 \mid Y_t = 1) \end{pmatrix},$$

parameterized by the empirical fix probability $p_{01} = P(Y_{t+1} = 1 \mid Y_t = 0)$ and break probability $p_{10} = P(Y_{t+1} = 0 \mid Y_t = 1)$. Thus, after invoking the generator, the controller propagates its belief as $b_{t+1} = b_t(1 - p_{10}) + (1 - b_t)p_{01}$.

If $a_t = a_{\text{crit}}^i$ and $z_t^{(i)} = z$, the belief is updated via Bayes' rule:

$$\begin{aligned} b_{t+1} &= P(Y_t = 1 \mid \mathcal{H}_t, z_t^{(i)} = z) \\ &= \frac{P_i(z \mid Y_t = 1)b_t}{P_i(z \mid Y_t = 1)b_t + P_i(z \mid Y_t = 0)(1 - b_t)}. \end{aligned}$$

Critic i reduces uncertainty about Y_t . This interaction constitutes an information purchase; the controller pays a low cost C_{crit}^i to acquire evidence before committing to refinement or verification.

The controller can submit the candidate to the oracle to resolve the episode. In this case, the controller receives the definitive outcome $Y_t = \mathcal{O}(c_t, x)$. While this interaction provides perfect information and determines the reward R , it incurs the highest cost C_{ver} .

Control via value of information. The controller implements a policy π that maps the current belief state to an agent selection, $\pi(b_t) \in \mathcal{A}$, with the goal of maximizing expected future utility. We write the value function as $V(b_t)$; in finite-horizon implementations, the state also includes the remaining budget and which critics have already been queried. The Bellman optimality equation is $V(b_t) = \max \{Q_{\text{gen}}(b_t), \max_i Q_{\text{crit}}^i(b_t), Q_{\text{ver}}(b_t)\}$, where the action values for the generator, critic, and verifier are defined, respectively, as

$$\begin{aligned} Q_{\text{gen}}(b_t) &= \mathbb{E}[V(b_{t+1}) \mid b_t, a_{\text{gen}}] - C_{\text{gen}}, \\ Q_{\text{crit}}^i(b_t) &= \mathbb{E}_z[V(b_{t+1}) \mid b_t, a_{\text{crit}}^i] - C_{\text{crit}}^i, \\ Q_{\text{ver}}(b_t) &= b_t R - C_{\text{ver}}. \end{aligned}$$

Q_{gen} represents the expected long-term value of improving the candidate program via the generator agent. Q_{crit}^i captures the expected value of acquiring information from critic i . Q_{ver} represents the immediate expected utility of stopping through oracle verification. To compute the optimal policy, we solve this Bellman recursion by finite-horizon dynamic programming over the belief state.

Optimal control vs heuristics. Orchestration heuristics, such as thresholding a confidence score, collapse the decision problem into a scalar boundary. This can be insufficient because the optimal

policy depends on the posterior probability of correctness, action-specific likelihoods, costs, remaining budget, and which information sources have been consumed. For example, a low-confidence candidate before any critic has been queried may have high information value, making a critic call preferable to refinement. A low-confidence candidate after a failed public test may have little remaining diagnostic value, making refinement preferable. A static threshold can treat these cases identically, whereas the Bellman update separates them through the Q -values of the generator, critic, and verifier actions.

Estimating critic likelihoods. For each critic i and oracle label $y \in \{0, 1\}$, we estimate $P_i(z=\text{pass} \mid Y=y)$ from the 75% calibration split using Beta(1, 1) smoothing. Because critic outputs are binary, the FAIL likelihood is the complement of the PASS likelihood. We keep these estimates fixed on the disjoint 25% test split. Estimator and count definitions are given in § 5.4. Sensitivity to perturbations of $\hat{P}(z \mid Y)$ is analyzed in § 6.4.

Estimating transition probabilities. We estimate the transition probabilities p_{01} and p_{10} in the transition matrix $\mathcal{T}$ from refinement trajectories with oracle labels at consecutive steps, aggregating (Y_t, Y_{t+1}) pairs and applying beta-binomial smoothing. These counts come from a held-out iterative-refinement corpus, as detailed in § 5.

5 Experimental Setup

We investigate the conditions under which the Bayesian controller outperforms heuristic baselines and its failure modes. Additional setup details are provided in Appendix A.

5.1 Benchmarks and Generators

Benchmarks. We experiment with nine benchmarks spanning three task families (Table 3). The *function-level synthesis benchmarks* are LCB at three difficulty tiers, covering LeetCode problems from releases v1 through v6, MBPP+ (Liu et al., 2023), and HumanEval+ (Liu et al., 2023). The *repository-level patch-generation benchmarks* are SWE-Bench Lite and SWE-Bench Verified (Jimenez et al., 2024). The *bug-fixing benchmarks* are HumanEvalFix (Muennighoff et al., 2024) and CodeContests (Li et al., 2022).

Generators. We test six LLMs across two access modes (Table 2), with 7–200B effective parameters. For each generator-benchmark cell, the generator's

single-shot pass rate defines the initial prior pass probability $b_0 = P(Y_0=1)$, the probability that a fresh candidate passes the oracle verifier before any critic evidence is observed. Across cells, b_0 ranges from 0.05 to 0.96 (Table 5).

5.2 Critics and Oracle Verifier

The oracle verifier represents an expensive-to-run test suite that deterministically checks the correctness of the solution. For LCB, it is a full hidden test suite; for SWE-Bench, it is a Podman-based harness; and for MBPP+/HumanEval+, it is represented by assert-based unit tests.

Critics provide lightweight but noisy evidence. The syntax critic Cr_{syn} runs `ast.parse` on each modified file and is essentially free in wall time. The test critic Cr_{test} runs public tests, using LiveCodeBench public examples (and the analogous public tests for MBPP+, HumanEval+, HumanEval-Fix, and CodeContests). SWE-Bench has no public-test split distinct from the oracle verifier, so Cr_{test} is undefined for SWE-Bench. The LLM critic Cr_{llm} uses a separate small model, Claude Haiku-4.5 by default, prompted with the output of the `diff` command and the problem statement to return a one-token PASS/FAIL judgment. In Appendix A.3 (Table 6, Fig. 6), we report the per-critic informativeness measured as $\gamma_i = P_i(z=\text{pass} \mid Y=1) - P_i(z=\text{pass} \mid Y=0)$ across all evaluated (benchmark, generator) pairs: $\gamma_i \approx 0$ means the critic carries no useful signal, while $|\gamma_i| \rightarrow 1$ means it is highly informative; values near 1 indicate near-oracle alignment with correctness, and values near -1 indicate an inverted signal.

Critic and oracle verifier cost vectors. These are deployment-specific hyperparameters. To obtain the cost vectors used in our experiments, we collect telemetry from a representative deployment. We measure generation cost, critic costs, and oracle verifier cost. For each action, we log wall-clock latency, token usage, and API cost. We use wall-clock latency as the primary proxy cost metric because it best captures the operational bottlenecks that matter in practice, including local process overhead, CI waiting time, container execution, and external API latency. Details on measurements and the exact vectors are given in Table 4.

We report results under two cost regimes that span the typical oracle/critic ratio space. The “*slow oracle, fast critics*” regime sets $C_{\text{ver}}=90$, $C_{\text{crit}}^{\text{syn}}=1$ for the syntax critic, $C_{\text{crit}}^{\text{test}}=2$ for the public-test

critic, $C_{\text{crit}}^{\text{llm}}=5$ for the LLM critic, $C_{\text{gen}}=10$ (e.g., a CI pipeline with frontier LLM judges). The “*fast oracle, balanced critics*” regime sets $C_{\text{ver}}=5$, $C_{\text{crit}}^{\text{syn}}=C_{\text{crit}}^{\text{test}}=C_{\text{crit}}^{\text{llm}}=1$, $C_{\text{gen}}=10$ (e.g., a unit-test loop where verification is cheap). The fast-oracle regime represents bug-fixing settings such as HumanEvalFix and CodeContests, where verification is relatively cheap and generation is the dominant costly action. The slow-oracle regime represents SWE-Bench-like deployments, where full verification is a major bottleneck and critic gating has higher value. We set $R=100$ as a utility normalization rather than a measured latency. One verified correct solution is worth 100 utility units, which makes C_{ver}/R the posterior success threshold at which verification becomes cost-neutral, and we report $R \cdot C_{\text{ver}}$ sweeps to check that the qualitative regimes are not tied to this single scale choice.

5.3 Baseline Policies

We compare the Bayesian controllers against eight baseline policies grouped into two families. The stateless baselines operate over the same action space $\{Cr_{\text{syn}}, Cr_{\text{test}}, Cr_{\text{llm}}, \text{verify}, \text{regen}, \text{stop}\}$, while the iterative-refinement baselines use the same verifier and refinement budget but follow their published update rules.

Stateless per-patch baselines. These policies take action on each patch in isolation, with no belief state carried across actions. `always_verify` skips every critic and calls the oracle verifier on every patch. `best_of_N` generates $N=3$ independent patches, runs the oracle on each, and returns the best, paying $3C_{\text{gen}} + 3C_{\text{ver}}$ per instance. It isolates pure generator-diversity gains without critic assessment. The `gate_Cri`, $i \in \{\text{syn}, \text{test}, \text{llm}\}$ baselines run critic Cr_i once and call the oracle only if it returns PASS; otherwise, request regeneration. `fixed_pipeline` runs all critics in the order $Cr_{\text{syn}} \rightarrow Cr_{\text{test}} \rightarrow Cr_{\text{llm}}$ and verifies only if every critic returns PASS, testing the simple strategy of stacking all critics together with an AND gate.

Iterative-refinement baselines. We compare against the *Self-Refine* (Madaan et al., 2023) and *Reflexion* (Shinn et al., 2023) coding-agent baselines. Both are run with up to four refinement steps per instance and the same verifier budget as the Bayesian controllers, and neither uses policy replays on our calibration trajectories. After each failed verification, Self-Refine prompts the generator to critique its own patch and regenerate, stop-

ping when the verifier returns PASS or the refinement budget is exhausted. Reflexion extends Self-Refine with a verbal-memory buffer of past failures that is fed back into the regeneration prompt.

5.4 Details of Bayesian Belief-State Policies

Bayesian controllers. Our framework derives two controllers from the Bellman equation of the POMDP (see also Section 4). Both maintain a posterior $b = P(Y=1 \mid \text{evidence})$ and select actions to maximize expected utility.

(1) *bayesian_greedy*. This controller performs one-step lookahead. At each decision point, with current candidate c_t and belief b_t , it evaluates each action’s immediate Q -value under the assumption that the next move is `verify`, and selects the action with largest value. When computing the value of a generator call, the controller treats the resulting patch as a fresh sample and resets its belief to the prior pass rate $b_0 = P(Y_0=1)$, rather than propagating the current belief through the measured fix/break probabilities. As a result, it cannot value multi-step refinement chains.

(2) *bayesian_DP*. This controller performs finite-horizon backward induction over states (b, h) , where b is the posterior probability of correctness and $h \in \{0, \dots, H\}$ is the remaining planning depth. We discretize the belief interval to a uniform 51-point grid and solve the Bellman recursion backward over horizon $H=3$, using the empirically estimated transition kernel $\hat{P}(\text{fix} \mid \text{broken})$ and $\hat{P}(\text{break} \mid \text{correct})$ for the regeneration branch. The kernel propagates beliefs across regenerations, allowing DP to value multi-step critic-refine-verify chains that the greedy controller cannot see.

Estimating the parameters of Bayesian control policies. We estimate the quantities required by the Bayesian controllers from held-out calibration data (split 75% for training, 25% for testing), namely the initial belief $b_0 = P(Y_0=1)$, also called the prior pass probability (Table 5), the critic likelihoods, and the refinement-transition probabilities $\hat{P}(\text{fix} \mid \text{broken})$ and $\hat{P}(\text{break} \mid \text{correct})$.

Single-shot calibration draws $m=3$ independent patches per instance and runs the full critic stack, followed by the oracle verifier. From the resulting per-patch records, we estimate the prior pass probability $b_0 = P(Y_0=1)$, which is the probability that a fresh patch is correct before observing critic outputs, and the critic likelihoods $P_i(z=\text{pass} \mid Y_0=y)$.

For each critic i and oracle label $y \in \{0, 1\}$, we

estimate the critic likelihood $\hat{P}_i(z = \text{pass} \mid Y_t=y)$ as the empirical fraction of patches with label y on which critic i returns PASS, with one PASS and one FAIL pseudocount for smoothing:

$$\hat{P}_i(z = \text{pass} \mid Y_t=y) = \frac{n_{y,i}^{\text{pass}} + 1}{n_{y,i} + 2}.$$

Here, $n_{y,i}^{\text{pass}}$ is the number of calibration patches with the oracle label y on which critic i returns PASS, and $n_{y,i}$ is the total number of patches with the oracle label y assessed by critic i . We use the same critic likelihood model at each decision step.

Transition probabilities are estimated from consecutive parent-child generation pairs in the iterative-refinement corpus, using smoothing:

$$\hat{p}_{01} = \frac{n_{0 \rightarrow 1} + 1}{n_{\text{from } 0} + 2}, \quad \hat{p}_{10} = \frac{n_{1 \rightarrow 0} + 1}{n_{\text{from } 1} + 2}.$$

5.5 Evaluation Metric

For each generator/benchmark/cost-vector triple, we evaluate the ten policies above on the same instance cohort and patches, yielding a vector of mean utilities $\bar{U}_\pi$. The main metric is $\Delta_\pi = \bar{U}_\pi - \bar{U}_{\text{always_verify}}$, the per-instance utility gap against the reference policy. Thus, $\Delta_\pi > 0$ beats the “oracle on every patch” default, $\Delta_\pi = 0$ ties, and $\Delta_\pi < 0$ underperforms. Paired 95% bootstrap CIs on Δ_π are used to identify winners.

6 Results

We organize results around three questions: *where* in the $(P(Y=1), C_{\text{ver}}/R)$ plane each policy dominates (§ 6.1); *which* policy wins on held-out splits against baselines, including Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) agents (§ 6.2); and *what* signals guide the controller’s decisions (§ 6.3). We then test cost, critic, and verifier sensitivity (§ 6.4), compare Bayes-DP to LLM-driven agents (§ 6.5), quantify its overhead (§ 6.6), and discuss how our framework improves uncertainty quantification for agents (§ 6.7).

6.1 Regimes of Control Policy Prevalence

Figure 2 aggregates 7,020 sample points pooled across all (benchmark, generator) pairs and the full (R, C_{ver}) sweep grid into a single empirical decision map (grid resolution in Appendix A.4). The plot indicates *which policy wins* at each coordinate in the $(P(Y=1), C_{\text{ver}}/R)$ plane. Three regimes emerge, each driven by a distinct mechanism that

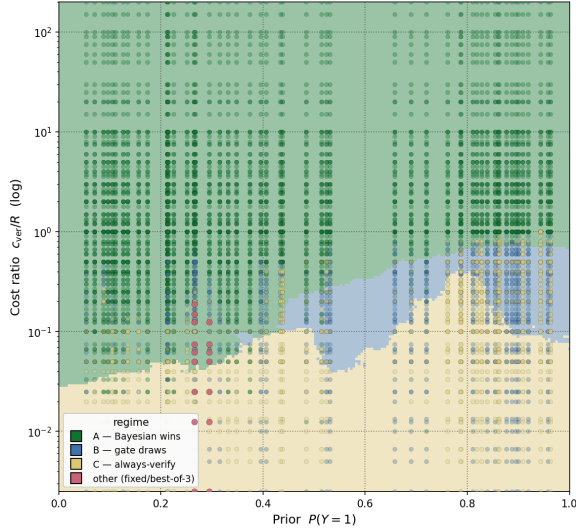


Figure 2: Empirical decision regions over the $(P(Y=1), C_{\text{ver}}/R)$ plane: k -NN ($k=300$) over 7,020 sweep points. Colors: **A** (green) Bayesian; **B** (blue) gate(Cr_{test}); **C** (yellow) always_verify; *other* (coral). See §6.1 for the full regime description.

ties back to the per-cell prior (Table 5) and per-critic informativeness (Table 6).

Regime A: Bayesian wins (green, $C_{\text{ver}}/R \gtrsim 1$). When verification is at least as expensive as the reward, paying C_{ver} on every patch is structurally unprofitable. The Bayesian policies maintain a posterior $P(Y=1 \mid z^{(1)}, \dots, z^{(k)})$ over the critics and verify only when the expected value of information exceeds C_{ver} . bayesian_DP also pulls measurably ahead of the one-step-lookahead bayesian_greedy: when each round of verification is unaffordable, planning multiple steps ahead over the transition kernel lets the controller defer verification across refinement rounds until the posterior is sharp enough to commit.

Regime B: gate(Cr_{test}) draws (blue wedge, $10^{-1} \lesssim C_{\text{ver}}/R \lesssim 1$). At moderate verification cost, the optimal Bayes decision often is a gate on the public-test critic. On cells where Cr_{test} is highly informative ($\gamma_{\text{test}} \gtrsim 0.7$ on LCB and on most HumanEvalFix and CodeContests cells; $\gamma_{\text{test}} \gtrsim 0.9$ on LCB-easy specifically, Table 6; SWE-Bench has no public-test critic distinct from the oracle and is excluded), a single PASS observation moves the posterior past the verification threshold by itself, so adding Cr_{syn} or Cr_{llm} to the update rarely changes the decision Cr_{test} already implies. The Bayesian controllers and gate(Cr_{test}) make the same decision in most instances, but the gate baseline ties or marginally wins from the absence of inference over-

head. The wedge narrows on cells where the public test is a weaker signal (MBPP+, HumanEval+: $\gamma_{\text{test}} \in [0.07, 0.75]$).

Regime C: always_verify wins (yellow, $C_{\text{ver}}/R \lesssim 10^{-1}$). When verification is essentially free, the cost of running any critic – let alone of a Bayesian update – approaches or exceeds C_{ver} itself, so blind verification dominates. Two pockets appear. The bottom-right (high prior, low cost) covers cells where the step-0 prior is already above the indifference threshold – almost every patch passes and always_verify simply collects the reward without paying for information it does not need. The bottom-left (low prior, low cost) covers cells where most patches fail; the few passes are cheap to harvest by verifying everything, and any critic-based skip risks dropping a true pass for a saving that does not justify the loss.

6.2 Policy Comparison

Figure 3 presents a representative experiment comparing all policies on SWE-Bench Lite with claude-haiku-4.5. SWE-Bench Lite uses the slow oracle ($C_{\text{ver}}=90$), so its cells fall into regime A of Figure 2, where the framework predicts that the Bayesian variants win. The two Bayesian variants are effectively tied for the best performance: bayesian_greedy and bayesian_DP both substantially improve over always_verify. Among the remaining baselines, the syntax gate gate(Cr_{syn}) is the strongest but still well below the Bayesian controllers; the LLM-judge gate gate(Cr_{llm}) and fixed_pipeline improve over always_verify by smaller margins. The public-test gate gate(Cr_{test}) is omitted on SWE-Bench because there is no public-test critic distinct from the oracle verifier (see §5.2). The refinement baselines SR and Reflexion, together with best_of_3, underperform always_verify in this setting. Detailed results for each dataset and generator are presented in Table 7 in Appendix B.

6.3 Critic Informativeness

Bayesian control requires informative critic likelihoods (Appendix A.3) and is most useful when evidence is distributed across complementary imperfect critics rather than concentrated in one near-oracle signal. The Bayesian update weights each critic by its likelihood gap γ_i , so combining several moderately informative signals yields a sharper posterior than any single critic: low-gap critics, such as the syntax critic (Cr_{syn}) in many function-level

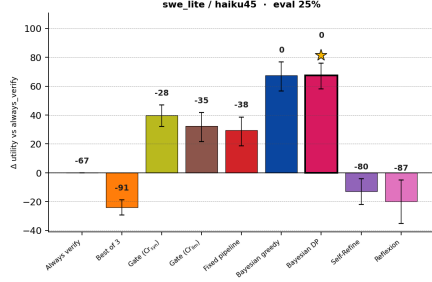


Figure 3: Per-policy Δ utility against `always_verify` on SWE-Bench Lite / claude-haiku-4.5. Gold stars mark cell winners; numbers above bars are absolute mean utility; error bars are paired-bootstrap 95% CIs ($B=1000$). Discussion in §6.2; evaluation-set bars across all (benchmark, generator) pairs in Appendix E.1; train-vs-evaluation comparison in Appendix E.2.

synthesis cells, barely affect the update, whereas informative and complementary critics, such as the public-test critic (C_{test}), receive greater weight and improve the belief state. This weighting and aggregation of imperfect signals explains the gains from Bayesian policies in Regime A.

6.4 Critic and Verifier Sensitivity

Figure 4 sweeps $C_{\text{ver}} \in [1, 100]$ on MBPP+ / claude-haiku-4.5 at fixed $R=100$, with other costs at their defaults. The sweep traverses all three regimes within a single cell, illustrating the $C \rightarrow B \rightarrow A$ handoff. At low C_{ver} , `always_verify` wins because the prior is high enough that blind verification has a positive expected value. At moderate C_{ver} , `gate( $C_{\text{test}}$ )` exploits the public-test critic. At high C_{ver} , the Bayesian variants take over because the gate policy is forced to refuse too many uncertain patches. The per-benchmark/per-generator C_{ver} sweeps covering all 54 (benchmark, model) pairs are in Appendix E.3.

We also test sensitivity to the estimated critic likelihoods in Appendix C.4, using perturbations of up to ± 0.20 . Bayes-DP maintains a positive utility gain over `always_verify` when verification is costly and continues to select `always_verify` when verification is cheap.

Finally, Appendix C.1 models the terminal verifier as a noisy observation with false-positive rate α and false-negative rate β , allowing the controller to update its belief and continue. Realized utility is evaluated against the clean label. On CodeContests with gpt-5-mini, symmetric error rates of up to 20% keep realized utility within one point of the clean value. False positives are more damaging:

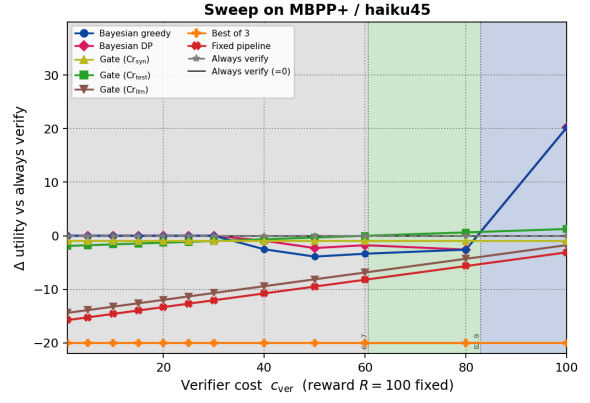


Figure 4: Per-policy Δ utility relative to `always_verify` on MBPP+ / claude-haiku-4.5 as a function of verifier cost C_{ver} ($C_{\text{crit}}^{\text{syn}}=1$, $C_{\text{crit}}^{\text{test}}=2$, $C_{\text{crit}}^{\text{llm}}=5$, $C_{\text{gen}}=10$, $R=100$ fixed). Shaded regions mark the dominant policy in each verifier-cost sub-range; gray: `always_verify` / near-tied region at low verifier cost; green: `gate( $C_{\text{test}}$ )`; blue: Bayesian. Per-(benchmark, generator) plots are in Appendix E.3.

at $\alpha=0.33$, realized utility drops by 7.9. Overall, Bayes-DP is robust to moderate symmetric noise but more sensitive to false positives.

6.5 Comparison with LLM-Driven Agents

We compare Bayes-DP controller with five LLM-driven agents under a unified experimental setup: Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), LangChain-ReAct (Yao et al., 2023), AutoGen (Wu et al., 2024), and smolagents (Roucher et al., 2025). All agents use Claude Haiku 4.5 and are evaluated on five benchmarks: HumanEval+, HumanEvalFix, MBPP+, LCB-medium, and LCB-hard. Each agent is limited to eight LLM calls per task and has access to the same four tools: syntax checking, public testing, patch generation, and oracle verification.

Table 11 in Appendix C.2 reports averages across datasets for pass@1, cost per task, wall-clock time, the number of verifier calls, and the cost-adjusted utility $U(\pi)$ defined in Eq. (1). Table 12 presents the per-dataset results.

Bayes-DP achieves the highest cost-adjusted utility: +39.4 on average versus +20.3 for the best baseline (Reflexion), and -13.3 on LCB-hard versus -55.0 for the second-best method. Although four of the five baselines reach a higher pass@1, they do so at 3–25 $\times$ the cost per task and substantially longer wall-clock times, so the additional accuracy does not pay for itself under the evaluated cost-reward trade-off. Figure 5 compares the

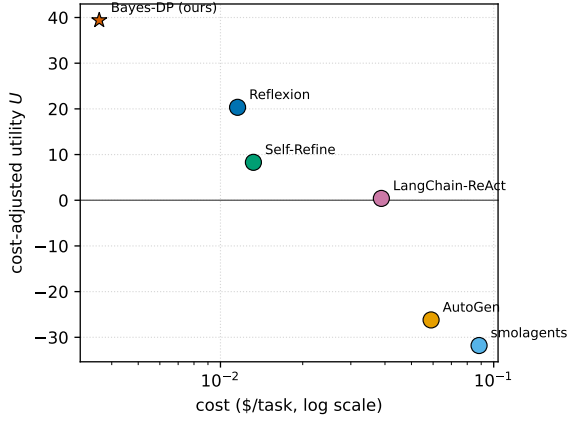


Figure 5: Mean per-task cost (log scale) versus cost-adjusted utility U for Bayes-DP and five LLM-driven agents using Claude Haiku 4.5.

approaches in the cost-utility plane. Bayes-DP attains the highest utility at the lowest cost, and thus Pareto-dominates all five baselines in this setup.

Table 13 in Appendix C.2 reports cost-vector sensitivity, and Table 14 in Appendix C.3 reports cost-aware prompting results. Thus, Bayes-DP’s performance depends not only on cost-aware information but also on how it makes decisions.

6.6 Efficiency Analysis

The greedy policy scores a fixed actions at each step and contributes less than 0.0001% of the runtime. The Bayes-DP policy is solved offline on a 51-point grid ($H = 3$), after which each decision requires an $O(1)$ lookup. Its overhead remains below 0.4%. Both policies follow a single candidate trajectory without additional LLM generations. Thus, their overhead is negligible relative to the cost of LLM generation and test verification. Moreover, the Bayesian controller is substantially cheaper than LLM-driven agents, which add an LLM generation at each decision.

6.7 Uncertainty Quantification for Agents

Beyond using Bayesian controllers to optimize expected utility, we investigate whether the belief state can quantify uncertainty for an LLM-based controller (see Appendix D for details). We use a SAGE agent (Suri et al., 2026) with gpt-oss-20b. The Bayesian model is used only post hoc to aggregate critic feedback into a belief about whether the code is correct before the oracle verification outcome is observed. Table 1 reports results on LCB-Medium, LCB-Hard, and CodeContests. On average, the Bayesian belief state achieves the best

Method	LCB-Medium	LCB-Hard	CodeContests	Avg
Perplexity	.043 [-.145, .241]	.067 [-.188, .327]	.311 [-.109, .497]	.140 [-.015, .266]
Seq. Prob.	.296 [.085, .505]	.174 [-.090, .438]	.666 [-.483, .814]	.379 [-.256, .504]
Verb. Conf.	.211 [-.019, .414]	.347 [-.060, .568]	.350 [-.143, .535]	.303 [-.165, .418]
Tool Success Rate	.177 [-.055, .369]	.324 [-.058, .575]	.781 [-.633, .894]	.428 [-.299, .537]
Bayes State	.080 [-.076, .362]	.398 [-.111, .648]	.849 [-.718, .945]	.442 [-.336, .579]
Bayes State + UQ	.245 [-.023, .437]	.436 [-.159, .667]	.854 [-.728, .950]	.512 [-.385, .616]

Table 1: PRR@0.5 $\uparrow$ for uncertainty quantification methods evaluated before verification using the SAGE (Suri et al., 2026) with gpt-oss-20b. Brackets show 95% bootstrap CIs, and Avg is the mean across datasets.

ranking performance among the evaluated methods, outperforming the token-probability, verbalized-confidence, and tool success rate baselines. We additionally incorporate sequence probability as an observation in the belief update, as detailed in Appendix D. The resulting Bayes State + UQ method further improves the average PRR by 0.07. This suggests that a belief state obtained through Bayesian aggregation of critic observations can provide a useful uncertainty score even for coding agents controlled by an LLM.

7 Conclusion

We formulated LLM-based code-generation orchestration as a cost-sensitive sequential hypothesis testing problem, where the controller maintains a posterior belief over candidate correctness and uses it to choose between criticism, refinement, verification, and stopping. Our experiments reveal three regimes. (A) Bayesian controllers work best when verification is expensive and critics are informative but imperfect, because they can aggregate multiple noisy signals and downweight weak critics. (B) A simple public-test gate is sufficient when the public-test critic is already near-oracle, leaving little benefit for more complex Bayesian aggregation. (C) Unconditional verification dominates when verification is cheap or the generator’s prior probability of correctness is high, since gathering additional information is then not worth the cost. This helps identify when Bayesian control is useful and when simpler orchestration policies are sufficient.

Beyond control, our results suggest that the belief can be used as an interpretable uncertainty score to identify whether the generated code is correct. This makes the proposed Bayesian framework complementary to existing coding agents: it can be used either to control the agentic loop directly or to provide uncertainty scores for trajectories produced by external controllers. Investigating this research direction more systematically is an important direction for future work.

Limitations

The benchmarks used in our evaluation primarily focus on Python, since most widely used datasets and benchmarks are Python-based. However, because modern LLMs have broad reasoning capabilities across code, we are confident that our findings generalize well to other programming languages. These benchmarks also contain a limited number of instances. Still, our experiments cover six models and nine datasets, which supports the robustness of our findings. Finally, different prompts or judge models may affect the results of the LLM-based critic. However, we use a standard prompt and a modern default model, which helps support the reliability of our findings. We focus only on coding-agents and code generation; however, the proposed framework could be utilized in any agentic workflow with orchestration.

Ethical Considerations

Our experiments use open-weight and closed-API LLMs and established code-generation benchmarks and do not involve human subjects or target harmful content. Nevertheless, generated code may contain defects or security vulnerabilities, and uncertainty estimates may be miscalibrated. In particular, it may assign high uncertainty to correct code or low uncertainty to incorrect code, limiting its suitability for high-stakes deployment. Thus, as with other uncertainty quantification methods, it has limited applicability.

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful comments and suggestions, which have greatly strengthened this paper.

We used AI assistants when working on this paper, in order to improve grammatical accuracy. All resulting edits were reviewed by the authors.

References

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

Deepro Choudhury, Sinead Williamson, Adam Goliński, Ning Miao, Freddie Bickford Smith, Michael Kirchhof, Yizhe Zhang, and Tom Rainforth. 2026. BED-LLM: Intelligent information gathering with LLMs

and Bayesian experimental design. In *International Conference on Learning Representations (ICLR)*.

Marina Fomicheva, Shuo Sun, Lisa Yankovskaya, Frédéric Blain, Francisco Guzmán, Mark Fishel, Nikolaos Aletras, Vishrav Chaudhary, and Lucia Specia. 2020. Unsupervised quality estimation for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:539–555.

Ronald A. Howard. 1966. Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, 2(1):22–26.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agent-Coder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Coda, Mark Encarnación, Shuvendu K. Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Live-CodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

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

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustín Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Percy Liang, Michael I. Jordan, and Dan Klein. 2010. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 639–646.

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

- Conference on Neural Information Processing Systems (NeurIPS)*.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Andrey Malinin and Mark Gales. 2021. Uncertainty estimation in autoregressive structured prediction. In *International Conference on Learning Representations (ICLR)*.
- Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2024. OctoPack: Instruction tuning code large language models. In *International Conference on Learning Representations (ICLR)*.
- Theodore Papamarkou, Pierre Alquier, Matthias Bauer, Wray Buntine, Andrew Davison, Gintare Karolina Dziugaite, Maurizio Filippone, Andrew Y. K. Foong, Vincent Fortuin, Dimitris Fouskakis, Jes Frellsen, Eyke Hüllermeier, Theofanis Karaletsos, Mohammad Emtiyaz Khan, Nikita Kotelevskii, Salem Lahlou, Yingzhen Li, Fang Liu, Clare Lyle, and 11 others. 2026. Position: Agentic AI orchestration should be Bayes-consistent. In *International Conference on Machine Learning (ICML)*.
- Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 2025. ‘smolagents’: a smol library to build great agentic systems. <https://github.com/huggingface/smolagents>.
- Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Q. Tran, Yi Tay, and Donald Metzler. 2022. Confident adaptive language modeling. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R. Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Manan Suri, Puneet Mathur, Nedim Lipka, Franck Dernoncourt, Ryan A. Rossi, and Dinesh Manocha. 2026. Structured uncertainty guided clarification for LLM agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 40811–40838, San Diego, California, United States. Association for Computational Linguistics.
- Katherine Tian, Eric Mitchell, Allan Zhou, Archit Sharma, Rafael Rafailov, Huaxiu Yao, Chelsea Finn, and Christopher Manning. 2023. Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5433–5442, Singapore. Association for Computational Linguistics.
- Roman Vashurin, Ekaterina Fadeeva, Artem Vazhentsev, Lyudmila Rvanova, Daniil Vasilev, Akim Tsvigun, Sergey Petrakov, Rui Xing, Abdelrahman Sadallah, Kirill Grishchenkov, Alexander Panchenko, Timothy Baldwin, Preslav Nakov, Maxim Panov, and Artem Shelmanov. 2025. Benchmarking uncertainty quantification methods for large language models with LM-Polygraph. *Transactions of the Association for Computational Linguistics*, 13:220–248.
- Abraham Wald. 1947. *Sequential analysis*. John Wiley & Sons.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.
- John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.
- Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida I. Wang. 2023. Coder reviewer reranking for code generation. In *International Conference on Machine Learning (ICML)*.
- Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 62138–62160. PMLR.

A Additional Details on Experimental Setup

A.1 Models

Table 2 lists the closed-API and open-weight generator models.

Access	Model
Closed-API	gpt-5-mini
	qwen3-coder
	claude-haiku-4.5
	claude-sonnet-4.5
Open-weight	Qwen2.5-Coder-32B-Instruct
	gpt-oss-20b

Table 2: Generator models used in the experiments. Closed-API models are served via OpenRouter; open-weight models are served locally via vLLM.

A.2 Dataset Statistics

Table 3 presents the benchmarks and datasets used in the experiments, their evaluation-set sizes, and verifier types. Table 4 reports the measured per-action wall-clock latency across all benchmarks.

Benchmark	N	Verifier
<i>Function-level synthesis</i>		
LCB-hard	102	hidden tests
LCB-medium	207	hidden tests
LCB-easy	135	hidden tests
MBPP+	378	assert tests
HumanEval+	164	assert tests
<i>Repository-level patch generation</i>		
SWE-Bench Lite	300	SWE harness
SWE-Bench Verified	500	SWE harness
<i>Bug-fixing</i>		
HumanEvalFix	164	assert tests
CodeContests	165	hidden tests

Table 3: Evaluation benchmarks grouped by task, with dataset size and verifier type.

A.3 Critic Informativeness

The controller’s posterior $P(Y=1 \mid z^{(1)}, \dots, z^{(k)})$ is only as useful as the underlying likelihoods $P_i(z \mid Y)$. We measure the per-critic informativeness $\gamma_i = P_i(z=\text{pass} \mid Y=1) - P_i(z=\text{pass} \mid Y=0)$ across all evaluated (benchmark, generator) pairs: $\gamma_i \approx 0$ means the critic provides little information about correctness, $\gamma_i \rightarrow 1$ means it is near-oracle (Table 6, Fig. 6).

First, Cr_{syn} is the least informative critic on average. On MBPP+, HumanEval+, and LCB-easy

every generator has $|\gamma_{\text{syn}}| \leq 0.10$, and on LCB-medium four of six generators have $|\gamma_{\text{syn}}| \leq 0.005$ – the syntax check passes on both $Y=1$ and $Y=0$ candidates and tells the controller nothing, because modern instruction-tuned generators produce parseable code regardless of whether it is semantically correct.

Second, Cr_{test} ’s informativeness tracks the relationship between public and hidden tests on each benchmark, not the generator. It is near-oracle ($\gamma_{\text{test}} \gtrsim 0.7$) on LCB at all three difficulty tiers, on both SWE-Bench variants, and on CodeContests – benchmarks where the public test suite either *is* the hidden suite or is strongly predictive of it. On MBPP+ and HumanEval+, γ_{test} drops to 0.07–0.75 because the public-test critic catches surface failures, but the hidden test suite catches behavioral failures the public asserts pass on these benchmarks; the public-test critic mainly detects failures covered by the visible assertions on those pairs. On average, Cr_{test} is the most informative of the three critics. This explains why single-critic gating on Cr_{test} is the dominant policy over an entire sub-range of verifier costs on some benchmark-generator pairs – e.g. the green band in Fig. 4 on MBPP+ / claude-haiku-4.5, where $\text{gate}(\text{Cr}_{\text{test}})$ beats both *always_verify* and the Bayesian variants for moderate C_{ver} .

Third, Cr_{llm} is consistently weaker than Cr_{test} on benchmark-generator pairs where the latter is near-oracle ($\gamma_{\text{llm}} < 0.5$ throughout LCB and SWE-Bench, often below 0.3 on the LCB rows), but becomes the dominant signal on HumanEvalFix, where $\gamma_{\text{llm}} \geq 0.7$ for all six generators and the problem statement gives the judge enough context to read the diff semantically. This is what gives *bayesian_greedy* and *bayesian_DP* their regime-A wins: when no single critic is near-oracle, the controller updates the belief through Bayes’ rule on each critic’s per-setup likelihood $P_i(z \mid Y)$, so the contribution of critic i scales with its informativeness γ_i – a near-zero-gap critic barely moves the belief, while two moderately-informative critics compose into a posterior sharper than either alone produces.

A.4 Regime-Map Sweep Grid

The empirical decision map in Figure 2 pools 7,020 sample points. These are the Cartesian product of all 54 (benchmark, generator) pairs with a (C_{ver}, R) sweep grid of 13 verifier costs $C_{\text{ver}} \in \{1, 5, 10, 20, 25, 30, 50, 60, 75, 90, 100, 150, 200\}$

	a_{gen} (s)		a_{ver} (s)		L_0	L_2	L_3	Gen \$
Benchmark	med	p90	med	p90	med	med	med	(10^{-3})
<i>Function-level synthesis</i>								
LCB-easy	3.4	5.2	0.20	0.33	<0.001	0.054	1.46	2.4
LCB-medium	6.8	8.6	0.25	0.41	<0.001	0.059	1.74	5.5
LCB-hard	7.6	9.8	0.30	0.50	0.001	0.062	1.74	6.3
MBPP+	1.3	2.8	0.05	0.08	<0.001	0.063	1.24	0.5
HumanEval+	2.2	3.1	0.02	0.03	<0.001	0.037	1.14	1.6
<i>Bug-fixing</i>								
HumanEvalFix	2.1	3.4	0.043	0.050	0.021 [‡]	0.021 [‡]	1.84	3.0
CodeContests	4.0	9.0	0.068	0.879	0.021 [‡]	0.021 [‡]	1.84	5.0
<i>Repository-level patches</i>								
SWE-Bench	10.0	31.4	1.9	682	0.362	2.02	1.91	6.0

Table 4: Measured per-action wall-clock latency across all benchmarks. Claude-haiku-4.5 is used throughout, except the SWE-Bench row, which pools 5 models on 4 held-out instances. Synthesis rows (LCB-easy/medium/hard, MBPP+, HumanEval+) are calibration runs with $n=30$ instances per benchmark ($n=29$ for LCB-hard). Bug-fixing rows (HumanEvalFix, CodeContests) use the instrumented haiku-4.5 pilot (15 tasks $\times$ 2 variants); their L_0 and L_2 entries report the local-process critic median pooled over L_0 – L_2 ($n=17$). SWE-Bench critics are Docker pytest runs ($n=183$ pooled L_0 – L_3). All timing is in seconds (median / p90 where reported). L_0 =regex syntax check, L_2 =local public-test execution (Docker on SWE-Bench), and L_3 =LLM judge instrumented separately on 60 HEFix patches with haiku-4.5; SWE-Bench L_3 reports Docker critic_mid. The last column is per-generate API cost in milli-dollars at OpenRouter haiku-4.5 rates; local critics and the local verifier incur no API charge. ^{||}SWE-Bench Docker verifier is bimodal: heavy-suite psf/requests median ≈ 672 s ($n=35$) versus fast-fail sub-second ($n=25$); pooled median 1.9 s, pooled p90 682 s. [‡]Local critic median pooled across L_0 – L_2 .

and 10 rewards $R \in \{1, 10, 20, 25, 50, 75, 100, 150, 200, 400\}$, giving $54 \times 13 \times 10 = 7,020$ points. At each point, we record the utility-maximizing policy; the background fill in Figure 2 is a k -NN ($k=300$) smoothing of that per-point winner over the $(P(Y=1), C_{\text{ver}}/R)$ plane. The remaining costs are held at the canonical values used throughout the paper ($c_{\text{gen}}=10$, $c_{C_{\text{syn}}}=1$, $c_{C_{\text{test}}}=2$, $c_{C_{\text{llm}}}=5$).

A.5 Per-benchmark Generator Locations

Figure 7 plots where each benchmark–generator pair falls in the $(P(Y=1), C_{\text{test}}\text{-gap})$ plane that drives the regime structure of Figure 2.

A.6 Policy Specifications

This section specifies each policy used in Table 7 and Figures 2, 3, and 4. Each entry describes the decision rule, data dependence, and the empirical regime in which the policy is competitive. Throughout b denotes the controller’s posterior belief, $P(Y=1 \mid \text{evidence})$, $z^{(i)}$ the observation returned by critic Cr_i , and R, c_a – the reward and per-action cost parameters from §4.

Policies fall into two families: (i) **unsupervised baselines** with no learnable parameters, and (ii) **fitted Bayesian controllers** (bayesian_greedy, bayesian_DP). Pre-collected calibration trajec-

ries are used to estimate the parameters. During evaluation, each policy executes its decision rule online, selecting every action from its current state; no action sequence is fixed in advance. Self-Refine and Reflexion are evaluated both as implementations of the published methods and under the same online, cached-artifact protocol.

A.6.1 Unsupervised Baselines

always_verify. Reference policy: ignore every critic, call the verifier on every patch, return Y . Per-instance utility $R \cdot Y - c_{\text{ver}}$ on the first patch verified; if $Y=0$, the controller consumes the next pre-sampled patch from the 3-patch pool. Has no free parameters; serves as the $\Delta_\pi=0$ baseline against which every Δ_π in the paper is computed. Dominates regime C (high prior and/or cheap verification).

best_of_3. Generate $N=3$ patches, verify each, and return the patch with the highest Y . Pays $3c_{\text{gen}} + 3c_{\text{ver}}$ per instance and never under-spends on verification. Isolates the generator-diversity contribution: any advantage of best_of_3 over always_verify is attributable to taking the maximum over 3 i.i.d. draws rather than to any orchestration logic. Across the evaluated (benchmark, generator) pairs, it is rarely the per-cell winner.

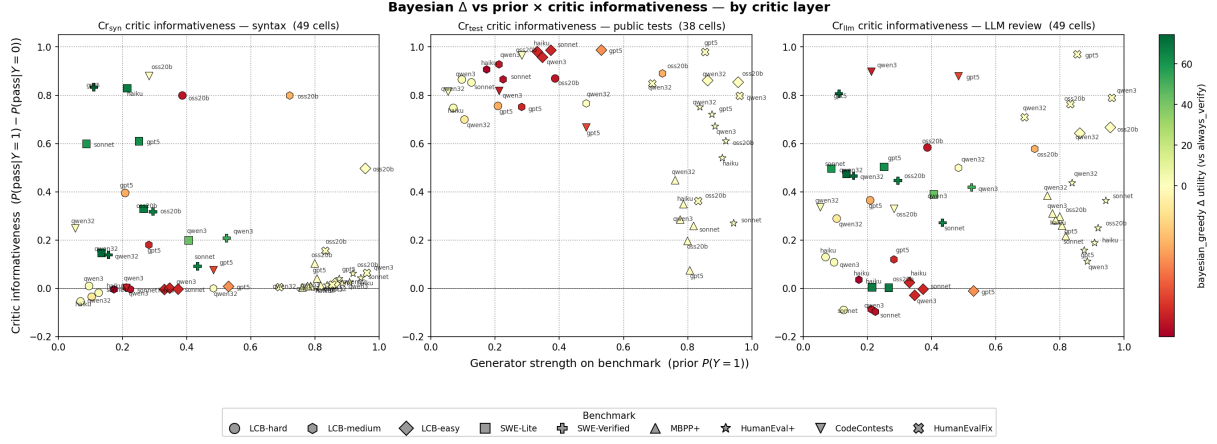


Figure 6: Critic informativeness $\gamma_i = P_i(z=\text{pass} \mid Y=1) - P_i(z=\text{pass} \mid Y=0)$ vs prior $P(Y=1)$ across the available benchmark–generator pairs, one subplot per critic layer. Point color indicates the Bayesian Δ for each benchmark–generator pair. SWE-Bench cells do not appear in the Cr_{test} (public tests) subplot: SWE-Bench has no public-test split distinct from the oracle verifier (the test_patch is the verifier), so there is no independent public-test critic to estimate a gap for on those cells.

gate(Cr_i) for $i \in \{\text{syn}, \text{test}, \text{llm}\}$. Single-critic gate: query critic Cr_i on the current patch; verify iff $z^{(i)}=\text{pass}$, else regenerate (consume the next patch in the 3-pool). Three variants span the cost/informativeness spectrum: $\text{gate}(Cr_{\text{syn}})$ uses the (essentially-free, weak) `ast.parse` check; $\text{gate}(Cr_{\text{test}})$ uses the mid-cost, near-oracle public-test critic; $\text{gate}(Cr_{\text{llm}})$ uses the mid-cost LLM judge. The decision rule is hand-coded: PASS $\rightarrow$ verify; FAIL $\rightarrow$ regen. Across the evaluated pairs, $\text{gate}(Cr_{\text{test}})$ dominates regime B by exploiting Cr_{test} ’s near-oracle informativeness at low cost; the other two variants are rarely competitive.

fixed_pipeline. Run critics in fixed order $Cr_{\text{syn}} \rightarrow Cr_{\text{test}} \rightarrow Cr_{\text{llm}}$ and verify iff every critic returns PASS (logical AND-gate). Tests the naive “stack everything” intuition. Two structural issues hurt its performance: (i) the AND-gate multiplies false-negative rates across critics, missing $Y=1$ patches whenever any single critic fires incorrectly; (ii) every patch pays the full critic stack even when Cr_{syn} or Cr_{test} would have already settled the decision. Rarely the per-cell winner across the evaluated pairs.

Self-Refine (Madaan et al., 2023). After each verify-fail, prompt the generator to critique its own patch and produce a refined patch. The agent loops up to $K=4$ refinement steps; at each step, it pays c_{gen} for the refinement and c_{ver} for the verification. Self-Refine has no fitted parameters: the generator adapts via in-context feedback rather than

via learned weights. We report it both as an executable implementation (§5.3) and under the same online, cached-artifact protocol over the iterative-refinement corpus. On 22 of 24 LCB cells, the Bayesian framework dominates Self-Refine; the two exceptions (LCB-easy gpt-5-mini and haiku-4.5) are high- P_{fix} cells where mechanical retry has positive expected value.

Reflexion (Shinn et al., 2023). Extends Self-Refine with an explicit verbal-memory buffer of past failures, fed back into the regeneration prompt. Same cost structure and same lack of fitted parameters as Self-Refine. The buffer is local to the current instance; nothing is fit from historical data. Reflexion outperforms Self-Refine on the two high- P_{fix} LCB-easy cells; otherwise, the two heuristics track each other closely.

A.6.2 Bayesian Controllers

Both Bayesian controllers share the same Bellman backbone (§4) and the same fitted Beta(1,1)-Laplace estimates: the prior $\hat{P}(Y=1)$, the critic likelihoods $\hat{P}_i(z \mid Y=y)$, and (for DP) the transition kernel $\hat{P}(\text{fix} \mid \text{broken})$, $\hat{P}(\text{break} \mid \text{correct})$. They differ in the lookahead horizon and in how the regenerate branch is valued.

bayesian_greedy (BG). One-step Bellman lookahead. At every state (b, t) the controller computes Q -values for each available action and picks

the argmax:

$$\begin{aligned}
Q_{\text{verify}}(b) &= R \cdot b - c_{\text{ver}}, \\
Q_{\text{stop}} &= 0, \\
Q_{\text{gen}}(b) &= -c_{\text{gen}} + (\hat{P}(Y=1) \cdot R - c_{\text{ver}}), \\
Q_{\text{crit}}^i(b) &= -c_{\text{crit}}^i + \mathbb{E}_{z \sim \hat{P}_i(\cdot|b)}[Q_{\text{verify}}(b_{t+1})],
\end{aligned}$$

where b_{t+1} is the Bayes posterior from Section 4. The defining property: the regenerate branch substitutes the prior $\hat{P}(Y=1)$ for the post-regen belief, so BG cannot value multi-step refinement chains. Dominates regime A on cells where the critic stack is informative, but no single critic is near-oracle.

bayesian_DP (BDP). Full backward induction on the (b, t) state space. We discretize b to a uniform 51-point grid on $[0, 1]$ and solve the Bellman recursion backwards over horizon $H=3$. The defining differences from BG are: (i) the regenerate branch uses the *measured* transition kernel, so the post-regen belief is $b(1 - \hat{P}_{\text{break}}) + (1 - b)\hat{P}_{\text{fix}}$ rather than the prior; (ii) critic Q-values recurse on V_t rather than truncating to Q_{verify} , so BDP can plan multi-step critic-refine-verify chains. At inference, the precomputed policy $\pi_t(b)$ is an $O(1)$ lookup. BDP strictly dominates BG only on cells where the measured $\hat{P}_{\text{fix}} \gtrsim 0.15$ makes multi-step chains positive-EV; on the remaining pairs, BDP and BG converge (§5.4).

Replay convention. Both BG and BDP are evaluated by replaying their decision rule over the pre-collected calibration corpus: the patches and their critic verdicts are fixed (3 patches per instance, seed=42); the controller selects which observations to consult and when to verify or regenerate. The “regenerate” action consumes the next patch from the pool; the controller never generates new patches in replay mode.

Online action selection via cached artifacts.

Both BG and BDP select actions online. At each step, the controller updates its belief from the observations obtained so far and selects the next action. To avoid repeated LLM and critic calls, evaluation reuses three pre-generated patches per instance (seed=42) and the corresponding cached critic outputs. A “regenerate” action advances to the next cached patch, while a critic action retrieves the cached output for the current patch. These caches avoid recomputation, yet they neither determine the controller’s decisions nor fix its action sequence, and they do not change the evaluation outcome. This protocol is different from the uncertainty-

Benchmark	Generator	$\hat{P}(Y=1)$	95% CI
<i>Function-level synthesis</i>			
LCB-hard	gpt-5-mini	0.209	[0.16, 0.26]
	qwen3-coder	0.096	[0.06, 0.14]
	claude-haiku-4.5	0.070	[0.04, 0.11]
	claude-sonnet-4.5	0.126	[0.09, 0.17]
	Qwen2.5-Coder-32B	0.104	[0.07, 0.15]
LCB-medium	gpt-oss-20b	0.387	[0.33, 0.45]
	gpt-5-mini	0.283	[0.24, 0.32]
	qwen3-coder	0.212	[0.18, 0.25]
	claude-haiku-4.5	0.173	[0.14, 0.21]
	claude-sonnet-4.5	0.225	[0.19, 0.26]
LCB-easy	Qwen2.5-Coder-32B	0.484	[0.44, 0.53]
	gpt-oss-20b	0.722	[0.68, 0.76]
	gpt-5-mini	0.531	[0.48, 0.59]
	qwen3-coder	0.348	[0.30, 0.40]
	claude-haiku-4.5	0.331	[0.28, 0.38]
MBPP+	claude-sonnet-4.5	0.374	[0.32, 0.43]
	Qwen2.5-Coder-32B	0.862	[0.82, 0.90]
	gpt-oss-20b	0.957	[0.93, 0.98]
	gpt-5-mini	0.807	[0.78, 0.83]
	qwen3-coder	0.777	[0.75, 0.80]
HumanEval+	claude-haiku-4.5	0.787	[0.76, 0.81]
	claude-sonnet-4.5	0.819	[0.79, 0.84]
	Qwen2.5-Coder-32B	0.761	[0.73, 0.79]
	gpt-oss-20b	0.800	[0.77, 0.83]
	gpt-5-mini	0.876	[0.84, 0.91]
	qwen3-coder	0.886	[0.85, 0.92]
	claude-haiku-4.5	0.908	[0.88, 0.94]
	claude-sonnet-4.5	0.943	[0.92, 0.96]
	Qwen2.5-Coder-32B	0.838	[0.80, 0.87]
	gpt-oss-20b	0.919	[0.89, 0.94]
<i>Repository-level patch generation</i>			
SWE-Bench Lite	gpt-5-mini	0.251	[0.22, 0.28]
	qwen3-coder	0.406	[0.37, 0.44]
	claude-haiku-4.5	0.214	[0.18, 0.25]
	claude-sonnet-4.5	0.087	[0.07, 0.11]
	Qwen2.5-Coder-32B	0.134	[0.11, 0.16]
SWE-Bench Verified	gpt-oss-20b	0.266	[0.23, 0.30]
	gpt-5-mini	0.111	[0.08, 0.14]
	qwen3-coder	0.524	[0.48, 0.57]
	claude-haiku-4.5	0.438	[0.39, 0.48]
	claude-sonnet-4.5	0.434	[0.39, 0.48]
	Qwen2.5-Coder-32B	0.156	[0.14, 0.18]
	gpt-oss-20b	0.295	[0.27, 0.32]
<i>Bug-fixing</i>			
HumanEvalFix	gpt-5-mini	0.854	[0.82, 0.89]
	qwen3-coder	0.962	[0.94, 0.98]
	claude-haiku-4.5	0.895	[0.86, 0.92]
	claude-sonnet-4.5	0.887	[0.85, 0.92]
	Qwen2.5-Coder-32B	0.690	[0.64, 0.74]
CodeContests	gpt-oss-20b	0.833	[0.79, 0.87]
	gpt-5-mini	0.484	[0.42, 0.55]
	qwen3-coder	0.213	[0.16, 0.26]
	claude-haiku-4.5	0.213	[0.16, 0.26]
	claude-sonnet-4.5	0.265	[0.21, 0.32]
	Qwen2.5-Coder-32B	0.053	[0.03, 0.08]
	gpt-oss-20b	0.283	[0.24, 0.33]

Table 5: Per-cell single-shot prior pass rate $\hat{P}(Y=1)$ of the LLM-generated step-0 code, with Wilson 95% confidence intervals. The prior anchors the regime structure of Figure 2: cells with high prior favour always_verify (regime C), low priors favour the Bayesian variants (regime A) when verification is expensive. Priors span more than an order of magnitude across cells.

quantification experiment in Appendix D, which evaluates uncertainty scores post hoc on previously collected external-agent trajectories.

Benchmark	Generator	γ_{syn}	γ_{test}	γ_{llm}
<i>Function-level synthesis</i>				
LCB-hard	gpt-5-mini	0.395	0.756	0.365
	qwen3-coder	0.009	0.865	0.108
	claude-haiku-4.5	-0.054	0.747	0.129
	claude-sonnet-4.5	-0.018	0.853	-0.090
	Qwen2.5-Coder-32B	-0.035	0.699	0.289
	gpt-oss-20b	0.799	0.869	0.583
LCB-medium	gpt-5-mini	0.180	0.751	0.120
	qwen3-coder	0.004	0.928	-0.087
	claude-haiku-4.5	-0.004	0.907	0.035
	claude-sonnet-4.5	-0.004	0.865	-0.097
	Qwen2.5-Coder-32B	-0.000	0.766	0.499
	gpt-oss-20b	0.799	0.890	0.578
LCB-easy	gpt-5-mini	0.008	0.987	-0.011
	qwen3-coder	0.001	0.957	-0.029
	claude-haiku-4.5	-0.005	0.980	0.024
	claude-sonnet-4.5	-0.003	0.986	-0.004
	Qwen2.5-Coder-32B	0.019	0.861	0.642
	gpt-oss-20b	0.497	0.854	0.667
MBPP+	gpt-5-mini	0.041	0.074	0.259
	qwen3-coder	0.009	0.285	0.310
	claude-haiku-4.5	0.009	0.349	0.283
	claude-sonnet-4.5	0.011	0.258	0.217
	Qwen2.5-Coder-32B	0.003	0.448	0.384
	gpt-oss-20b	0.103	0.196	0.297
HumanEval+	gpt-5-mini	0.039	0.720	0.157
	qwen3-coder	0.020	0.671	0.112
	claude-haiku-4.5	0.026	0.540	0.188
	claude-sonnet-4.5	0.043	0.270	0.363
	Qwen2.5-Coder-32B	0.013	0.751	0.436
	gpt-oss-20b	0.062	0.610	0.250
<i>Repository-level patch generation</i>				
SWE-Bench Lite	gpt-5-mini	0.610	-	0.504
	qwen3-coder	0.200	-	0.390
	claude-haiku-4.5	0.830	-	0.005
	claude-sonnet-4.5	0.599	-	0.497
	Qwen2.5-Coder-32B	0.148	-	0.475
	gpt-oss-20b	0.330	-	0.004
SWE-Bench Verified	gpt-5-mini	0.834	-	0.807
	qwen3-coder	0.209	-	0.419
	claude-haiku-4.5	0.370	-	0.381
	claude-sonnet-4.5	0.092	-	0.273
	Qwen2.5-Coder-32B	0.139	-	0.466
	gpt-oss-20b	0.319	-	0.447
<i>Bug-fixing</i>				
HumanEvalFix	gpt-5-mini	0.015	0.979	0.969
	qwen3-coder	0.064	0.797	0.789
	claude-haiku-4.5	0.222	0.947	0.938
	claude-sonnet-4.5	0.276	0.974	0.965
	Qwen2.5-Coder-32B	0.005	0.848	0.708
	gpt-oss-20b	0.156	0.362	0.763
CodeContests	gpt-5-mini	0.075	0.666	0.878
	qwen3-coder	0.002	0.818	0.897
	claude-haiku-4.5	0.037	0.882	0.947
	claude-sonnet-4.5	0.013	0.863	0.926
	Qwen2.5-Coder-32B	0.248	0.815	0.336
	gpt-oss-20b	0.879	0.965	0.329

Table 6: Per-cell critic informativeness gap $\gamma_\ell = P(z_\ell=\text{pass} \mid Y=1) - P(z_\ell=\text{pass} \mid Y=0)$ estimated with Beta(1,1) smoothing on the calibration cohort. $\gamma_\ell \rightarrow 1$ corresponds to a near-oracle critic; $\gamma_\ell \rightarrow 0$ corresponds to a critic that carries no Bayes signal. Cr_lint is omitted – it is a no-op pass-through in this codebase so its gap is structurally zero on every cell. Cr_test is shown as “-” on all SWE-Bench cells: SWE-Bench has no public-test split distinct from the oracle verifier (the test_patch is the verifier), so there is no independent public-test critic to estimate a gap for. This structure explains the regime B/A boundary in Figure 2: near-oracle γ_{test} favours gating; moderate gaps across critics favour Bayesian composition.

B Detailed Results

Table 7 reports the per-cell utility difference Δ_π against `always_verify` for every (generator, benchmark, policy) triple in our experiments, under the “slow oracle, fast critics” cost vector; bug-fixing rows use the fast-oracle variant noted in the caption. Bold marks the per-cell winner among the reported methods. Columns **BG** and **BDP** denote `bayesian_greedy` and `bayesian_DP`.

Benchmark	Generator	BoN	$t_{C_{\text{syn}}}$	$t_{C_{\text{test}}}$	$t_{C_{\text{llm}}}$	FP	SR	Rfx	BG	BDP
<i>Function-level synthesis</i>										
LCB-hard	gpt-5-mini	-20.0	+3.1	+4.3	-6.0	-7.5	-33.2	-4.4	-7.6	+0.3
	qwen3-coder	-20.0	-1.4	-12.8	-10.0	-12.1	-25.6	-15.4	+2.4	+2.4
	claude-haiku-4.5	-20.0	-1.0	-13.8	-14.3	-16.4	-10.8	-29.3	+2.8	+2.1
	claude-sonnet-4.5	-23.8	-1.0	-8.7	-16.1	-18.3	-5.4	-5.2	+2.5	+2.5
	Qwen2.5-Coder-32B	-31.5	-1.0	-9.7	-26.2	-28.8	-17.7	-15.3	-2.6	-2.6
LCB-medium	gpt-oss-20b	-24.0	+0.0	+5.1	-1.1	-3.0	-34.4	+25.8	-17.0	+4.6
	gpt-5-mini	-11.3	-1.2	+0.7	+1.7	+0.3	-6.5	-14.1	-15.7	+1.9
	qwen3-coder	-18.1	-1.0	-11.8	-6.5	-8.3	-15.0	-23.2	-16.9	+2.2
	claude-haiku-4.5	-18.1	-1.0	-13.1	-3.3	-4.8	-10.4	-17.9	-18.8	+2.6
	claude-sonnet-4.5	-18.1	-1.3	-10.0	-10.2	-11.8	+6.0	+10.6	-17.3	+2.3
LCB-easy	Qwen2.5-Coder-32B	-21.9	-1.0	-4.3	-12.1	-13.9	-16.0	-16.4	+0.0	+0.0
	gpt-oss-20b	-23.8	+2.6	+2.4	-7.5	-9.2	-35.0	+6.1	-7.2	+2.3
	gpt-5-mini	-25.9	-1.0	+10.8	-5.3	-6.4	-11.2	+4.3	-9.1	+0.6
	qwen3-coder	-20.0	-1.0	-11.0	-1.3	-2.4	-10.6	-20.7	-15.7	+1.4
	claude-haiku-4.5	-20.0	-1.0	-11.0	-2.6	-3.8	-10.9	-0.1	-15.7	+1.9
MBPP+	claude-sonnet-4.5	-17.1	-1.0	-8.2	-2.5	-3.6	-11.5	+0.5	-15.7	+1.9
	Qwen2.5-Coder-32B	-20.0	-1.0	-1.0	-5.6	-6.8	-17.6	-1.0	+0.0	-1.0
	gpt-oss-20b	-20.0	-1.0	+4.2	+1.0	-0.1	-22.9	-3.6	+0.0	+0.0
	gpt-5-mini	-21.1	-0.0	+1.5	-8.6	-9.9	-21.3	-46.0	+0.0	+0.0
	qwen3-coder	-30.7	-1.0	-3.7	-16.1	-17.5	-28.8	-7.8	+0.0	+0.0
HumanEval+	claude-haiku-4.5	-20.0	-1.0	-1.8	-8.5	-9.8	-9.0	+2.2	+0.0	+0.0
	claude-sonnet-4.5	-21.1	-1.0	-1.4	-13.6	-14.9	-16.1	-3.0	+0.0	+0.0
	Qwen2.5-Coder-32B	-22.1	-1.0	-3.3	-16.9	-18.4	-35.7	-125.9	+0.0	+0.0
	gpt-oss-20b	-18.9	-1.0	-1.0	-6.3	-7.7	-34.3	-125.5	+0.0	+0.0
	gpt-5-mini	-17.6	-1.0	+3.3	-21.0	-22.3	-32.0	-80.3	+0.0	+0.0
	qwen3-coder	-17.4	-1.0	+3.0	-20.1	-21.5	-9.2	-12.5	+0.0	+0.0
	claude-haiku-4.5	-22.4	-1.0	-1.0	-20.8	-22.1	-11.2	-4.5	+0.0	+0.0
	claude-sonnet-4.5	-22.4	-1.0	-1.0	-21.0	-22.3	-12.0	-1.0	+0.0	+0.0
	Qwen2.5-Coder-32B	-17.6	-1.0	+3.6	-16.1	-17.7	-26.1	-119.8	+0.0	+0.0
	gpt-oss-20b	-17.6	-1.0	+3.3	-10.6	-11.9	-48.8	-82.5	+0.0	+0.0
<i>Repository-level patch generation</i>										
SWE-Bench Lite	gpt-5-mini	-17.3	+21.7	-	+13.7	+11.6	-24.4	-102.7	+62.0	+62.0
	qwen3-coder	-20.0	+1.7	-	+9.0	+7.1	-57.7	-136.0	+43.3	+43.3
	claude-haiku-4.5	-24.0	+39.7	-	+32.3	+29.3	-12.9	-20.0	+67.3	+67.6
	claude-sonnet-4.5	-50.7	+28.9	-	+22.3	+19.9	-24.8	-93.3	+59.3	+59.3
	Qwen2.5-Coder-32B	-17.3	+6.1	-	+27.9	+25.5	-16.0	-268.0	+74.0	+74.0
SWE-Bench Verified	gpt-oss-20b	-12.0	+8.3	-	+32.3	+29.3	-52.0	-256.0	+67.3	+67.3
	gpt-5-mini	-24.0	+41.9	-	+34.7	+32.3	-55.0	-226.0	+74.0	+74.0
	qwen3-coder	-16.0	+2.8	-	+5.6	+3.8	-42.6	-116.0	+52.0	+52.0
	claude-haiku-4.5	-18.0	+13.9	-	+7.2	+5.0	-14.6	-166.0	+56.0	+56.0
	claude-sonnet-4.5	-20.0	+1.7	-	+2.4	+0.8	-27.4	-122.0	+66.0	+66.0
	Qwen2.5-Coder-32B	-19.2	+8.6	-	+27.4	+25.2	-14.6	-266.4	+74.8	+74.8
	gpt-oss-20b	-8.8	+12.5	-	+14.5	+12.7	-47.0	-180.0	+68.4	+68.4
<i>Bug-fixing</i>										
HumanEvalFix	gpt-5-mini	-22.4	-1.0	+0.3	+0.3	-0.8	-32.7	+5.2	+0.0	+0.1
	qwen3-coder	-20.0	-1.0	-1.4	-1.8	-2.9	-37.6	+1.2	+0.0	+0.0
	claude-haiku-4.5	-21.8	-0.6	+3.8	+3.0	+1.9	-35.1	+1.0	-13.3	-3.8
	claude-sonnet-4.5	-16.3	+1.3	+5.9	+5.2	+3.9	-12.9	+4.2	-17.6	-2.4
	Qwen2.5-Coder-32B	-22.4	+0.9	-2.7	-30.5	-32.4	-30.7	-102.8	+0.0	+0.0
CodeContests	gpt-oss-20b	-22.5	-1.0	-1.9	-5.3	-6.8	-27.5	-103.9	+0.0	+0.0
	gpt-5-mini	-15.5	-1.0	-7.7	-4.9	-6.9	-49.1	-7.9	-13.5	-4.3
	qwen3-coder	-41.4	-1.0	-9.9	-11.3	-13.5	-40.0	-15.1	-16.2	+2.0
	claude-haiku-4.5	-25.3	+1.1	-7.8	-9.6	-12.1	-12.9	-11.5	-19.4	+2.7
	claude-sonnet-4.5	-25.3	-1.0	-12.4	-11.9	-14.4	-22.4	-30.6	-17.4	+2.3
	Qwen2.5-Coder-32B	-22.4	-5.7	-7.8	-21.8	-24.6	-18.0	-47.2	+3.3	+3.3
	gpt-oss-20b	-13.8	+7.7	+12.5	-28.2	-30.2	-19.4	+8.2	+0.0	+11.7

Table 7: Per-cell utility difference $\Delta_\pi = \bar{U}_\pi - \bar{U}_{\text{always_verify}}$ across all generators, benchmarks, and nine non-reference methods. FP denotes fixed_pipeline. All values computed on the held-out 25% eval split with the anchored cal $\cap$ SR $\cap$ Rfx pool (STAT_ANCHOR). The reference column always_verify is omitted (all entries $\Delta_\pi = 0$ by definition). Bold marks the per-cell winner across the nine method columns. “-” indicates a cell with no data available yet. $t_{C_{\text{test}}}$ is omitted on SWE-Bench cells (Repository-level patch generation): SWE-Bench has no public-test split distinct from the oracle verifier, so the public-test critic is undefined there.

C Additional Ablations

C.1 Imperfect Oracle

The main analysis treats the terminal verifier as ground truth. We relax this assumption by modeling its verdict $\tilde{Y}$ as a noisy observation of the latent correctness label Y^* , with false-positive rate α and false-negative rate β . The verdict therefore updates the belief through the verifier likelihood rather than setting the posterior to zero or one. The controller may continue after verification when stopping is not the highest-utility action. In this ablation, the same verdicts also provide the calibration labels, so verifier errors affect both the fitted prior and the critic likelihoods.

We evaluate this setting on CodeContests/gpt-5-mini. For each noise setting, we apply the corresponding noise channel, re-estimate the prior and critic likelihoods from the corrupted calibration labels, re-solve the controller, and evaluate 400 Monte Carlo draws with $C_{\text{ver}}=30$ and $R=100$. We report utility scored against the noisy verdict $\tilde{Y}$ (reported utility) and against the clean hidden-test label Y^* (realized utility).

At symmetric error rates of 10% and 20%, realized utility remains close to the clean value, while reported utility is lower. At 30%, realized utility also decreases. The asymmetric settings show a larger effect from false positives in this experiment. With $\alpha=0.33$ and $\beta=0$, reported utility rises from 19.5 to 26.3, whereas realized utility falls to 11.6. With $\alpha=0$ and $\beta=0.20$, reported utility falls to 6.2, while realized utility remains at 18.2.

Table 9 presents results obtained using either the public tests ($L2$) or the LLM reviewer ($L3$) as the terminal oracle $\tilde{Y}$, with the hidden test suite Y used only as ground truth. The selected oracle is removed from the possible action set. Therefore, the difference between reported and realised $E[U]$ comes from actual disagreements between the verifier and the hidden test suite, not from artificially injected errors. The results show that Bayes-DP remains effective under naturally imperfect oracles, indicating that the robustness observed in Table 8 is not merely an artifact of synthetic noise.

Table 10 compares three actions after an imperfect oracle verdict: stop, regenerate, and re-verify. A false acceptance from weak tests is fixed for a given patch and can only be avoided through regeneration, whereas a flaky rejection can be corrected through re-verification. The results show that continuing can recover correct solutions,

Noise setting	α	β	Reported $E[U]$	Realized $E[U]$
Clean (no noise)	0.00	0.00	19.5	19.5
Symmetric 10%	0.10	0.10	14.1	18.6
Symmetric 20%	0.20	0.20	12.8	19.2
Symmetric 30%	0.30	0.30	13.8	16.3
FP only	0.33	0.00	26.3	11.6
FN only	0.00	0.20	6.2	18.2

Table 8: Utility under terminal-verifier noise. Reported utility is scored against the noisy verdict $\tilde{Y}$, whereas realized utility is scored against the clean label Y^* . The asymmetric settings separate the effects of false positives and false negatives.

Oracle $\tilde{Y}$	FP	FN	Reported	Realised
<i>codecontests / gpt5-mini</i> ($P(Y=1)=0.49$)				
perfect (Y)	0.0%	0.0%	19.5	19.5
weak tests ($L2$)	32.6%	0.0%	24.0	7.9
LLM judge ($L3$)	11.0%	0.0%	28.0	20.9
<i>LCB-medium / sonnet45</i> ($P(Y=1)=0.29$)				
perfect (Y)	0.0%	0.0%	9.4	9.4
weak tests ($L2$)	7.5%	0.0%	4.3	-1.0
LLM judge ($L3$)	64.6%	26.7%	37.2	-1.0

Table 9: Evaluation with real imperfect oracles without injected noise. Public tests ($L2$) or the LLM reviewer ($L3$) serve as the terminal oracle $\tilde{Y}$, while the hidden test suite Y provides the ground truth. Reported $E[U]$ uses the oracle outcome, whereas realised $E[U]$ is computed against Y .

c_{ver} ($R=100$)	stop	regen	re-verify
2 (cheap tests)	41.3	44.9	45.0
8 (break-even)	35.3	35.3	31.4
15	28.3	24.0	15.4
30 (SWE harness)	13.3	-0.2	-18.7

Table 10: Realised $E[U]$ across verifier costs for three cost-aware actions after an imperfect oracle verdict: stop, regen, and re-verify. The best result in each row is shown in bold.

but the additional verification is worthwhile only when verification is cheap, with a break-even point around $c_{\text{ver}} \approx 8$. This means that trusting an imperfect oracle is itself a cost-dependent decision, so an imperfect oracle is better treated as a cost-aware critic than as ground truth.

C.2 Comparison with Baseline Coding Agents

Table 11 reports averages across datasets for pass@1, cost per task, wall-clock time, the number of verifier calls, and the cost-adjusted utility $U(\pi)$ defined in Eq. (1), and Table 12 reports the per-dataset utility.

Agent	pass@1	\$/task	wall (s)	verifier calls	U (avg)
Bayes-DP (ours)	.573	0.0036	6.5	1.09	+39.4
Reflexion	.640	0.0116	16.1	2.15	+20.3
Self-Refine	.573	0.0132	16.4	0.00	+8.3
LangChain-ReAct	.787	0.0388	30.1	1.57	+0.4
AutoGen	.740	0.0590	38.7	1.51	-26.2
smolagents	.767	0.0884	41.0	1.71	-31.8

Table 11: Comparison of Bayes-DP with five LLM-based agent approaches using Claude Haiku 4.5 on HumanEval+, HumanEvalFix, MBPP+, LCB-Medium, and LCB-Hard, with $n=30$ tasks per dataset.

This experiment evaluates agentic approaches along two distinct dimensions: task correctness and resource use. Several baselines achieve higher pass@1, but at higher cost and latency (Figure 5). With the utility defined in Eq. (1), Bayes-DP ranks first because it gains more utility from using fewer actions than it loses from its lower pass@1.

Additionally, in Table 13, we present an experiment in which we recomputed utility for the main experiments using alternative weights for the verifier and critics. Bayes-DP again achieves the highest utility in every setting, further demonstrating its effectiveness for cost-aware tasks.

C.3 Cost-Quality: Cost-Aware Prompting and Budget Constraints

Table 14 presents results for Reflexion and Self-Refine with cost-aware prompts. We provide both methods with the same cost and reward information available to Bayes-DP by encoding the POMDP formulation in their system prompts (§ 6.5). We evaluate them on HumanEval+, HumanEvalFix, and MBPP+, using $n=30$ tasks per dataset.

Including the cost model in the prompt does not close the gap. Reflexion remains largely unchanged, while Self-Refine achieves lower utility, suggesting that it responds to the budget information by iterating more rather than less and over-editing working code. Thus, the ranking cannot be explained solely by whether the agent is given the cost-aware formulation.

Finally, Table 15 shows the highest achievable pass@1 under per-task cost and wall-clock budgets. Bayes-DP performs best under tight budgets, while methods with higher pass@1 become feasible only when the budgets are substantially increased.

C.4 Sensitivity to the Estimated Critic Likelihoods

Table 16 presents the results for a sensitivity analysis of the critic likelihoods estimated on the 75%

calibration split. For each estimate $\hat{P}(z | Y)$, we shift all pass probabilities uniformly by ± 0.10 and ± 0.20 . We also apply two adversarial perturbations that decrease $P(z=\text{pass} | Y=1)$ and increase $P(z=\text{pass} | Y=0)$ by 0.10 or 0.20, making the critic less informative. We clip all probabilities to $[0.01, 0.99]$ and set each failure probability to one minus the corresponding pass probability.

We then recompute each policy using the perturbed likelihoods and evaluate it on the held-out 25% of the data from three benchmarks: LCB-Hard, LCB-Medium, and HumanEvalFix. Utilities are reported as Δ relative to `always_verify`, with paired-bootstrap 95% confidence intervals ($B=1000$). This analysis investigate whether the policy comparison depends on errors in the estimated critic likelihoods.

First, when verification is relatively costly ($C_{\text{ver}}/R \gtrsim 0.1$; § 6.1), e.g., on LCB-hard, Bayes-DP consistently outperforms `always_verify`. Specifically, Δ remains positive under every perturbation and decreases by at most 2.7 utility units under the worst adversarial shift. Second, when verification is cheap ($C_{\text{ver}}/R \lesssim 0.1$) or the prior pass probability is already high, e.g., on HumanEvalFix, Bayes-DP reduces to `always_verify`, with Δ slightly higher than zero, and its actions remain unchanged under likelihood perturbations. Finally, some perturbations do not change the utility, producing the same Δ as the original Bayes-DP estimate (Δ clean). Thus, small likelihood changes do not move the posterior belief across a decision boundary. Overall, these results demonstrate that Bayes-DP remains stable under perturbations of the estimated critic likelihoods.

D Bayesian Belief States for Uncertainty Quantification of Coding Agents

In this experiment, we explore whether the Bayesian belief states can quantify uncertainty for an externally controlled agentic coding trajectory before the final oracle verification outcome is observed. We use SAGE agent (Suri et al., 2026) using `gpt-oss-20b` as the controller in this experiment. We run the agent on LCB-Medium, LCB-Hard, and CodeContests, use the training split to estimate the initial prior and critic likelihoods, and evaluate uncertainty scores for the final candidate on the held-out test split.

We experiment with six uncertainty quantification methods. First, we use three standard methods:

Agent	HE+	HEfix	MBPP+	LCB-m	LCB-h	avg
Bayes-DP (ours)	+67.9	+63.2	+68.3	+10.8	−13.3	+39.4
Reflexion	+70.8	+64.2	+71.7	−31.7	−73.3	+20.3
Self-Refine	+60.3	+12.0	+56.0	−32.0	−55.0	+8.3
LangChain-ReAct	+37.5	+35.0	+39.0	−26.4	−82.9	+0.4
AutoGen	+25.4	+21.8	+19.7	−61.2	−136.6	−26.2
smolagents	+3.1	+8.5	+4.8	−62.9	−112.6	−31.8

Table 12: Per-dataset cost-adjusted utility for Bayes-DP and LLM-based agent approaches using Claude Haiku 4.5 on HumanEval+ (HE+), HumanEvalFix (HEfix), MBPP+, LCB-Medium, and LCB-Hard, with $n=30$ tasks per dataset.

Agent	U vs. c_{ver} ($R=100$)			U vs. R	
	1	15	30	$R=50$	$R=200$
Bayes-DP (ours)	43.7	28.5	12.2	10.7	96.7
Reflexion	28.9	−1.1	−33.3	−11.7	84.3
Self-Refine	8.3	8.3	8.3	−20.4	65.6
LangChain-ReAct	6.7	−15.2	−38.7	−38.9	79.1
AutoGen	−20.1	−41.3	−64.0	−63.2	47.8
smolagents	−25.0	−48.9	−74.5	−70.2	44.8

Table 13: Sensitivity to the utility coefficients, obtained by re-pricing the same runs with $c_{\text{gen}}=10$ and $c_{\text{crit}}=1$. Bayes-DP has the highest utility in every reported column; the best value in each column is bold.

Agent	pass@1	\$/task	wall	calls	$E[U]$
Bayes-DP (ours)	0.573	0.0036	6.5	1.32	+39.4
Reflexion	0.640	0.0116	16.1	3.29	+20.3
+ cost prompt	0.647	0.0125	18.7	3.31	+20.9
Self-Refine	0.573	0.0132	16.4	4.91	+8.3
+ cost prompt	0.533	0.0165	20.3	6.27	−9.4

Table 14: Cost-aware prompting ablation for the Self-Refine and Reflexion baselines on HumanEval+, HumanEvalFix, and MBPP+, using a subsample of $n=30$ tasks from each dataset. “Calls” denotes generations per task.

Constraint	Best feasible agent	pass@1
Cost $\leq$ \$0.01/task	Bayes-DP (ours)	0.573
Cost $\leq$ \$0.02/task	Reflexion	0.720
Cost $\leq$ \$0.04/task	LangChain-ReAct	0.787
Latency $\leq$ 10 s	Bayes-DP (ours)	0.573
Latency $\leq$ 20 s	Reflexion	0.720
Latency $\leq$ 45 s	LangChain-ReAct	0.787

Table 15: Highest achievable pass@1 under a per-task cost or wall-clock budget on HumanEval+, HumanEvalFix, and MBPP+, using a subsample of $n=30$ tasks from each dataset.

perplexity and sequence probability (Fomicheva et al., 2020), both computed from the token log-

probabilities of the final generated code, and *verbalized confidence* (Tian et al., 2023), obtained by asking the target model to assess its confidence in the solution. Second, the *Tool Success Rate* is the empirical success rate of non-final critic calls observed before any verifier call. The terminal verifier decision itself is excluded to avoid label leakage. When no such tool evidence is available, we set the score to the train-set prior. Third, the original *Bayes Belief State* corresponds to a posterior probability of correctness, initialized from the train-set prior and updated using fitted critic likelihoods and generation transition dynamics. As with the tool success rate, the belief state is measured before the terminal correctness label is revealed.

Finally, we supplement the Bayes Belief State with an additional uncertainty quantification signal. Specifically, *Bayes State* + *UQ* augments the final Bayesian belief state by treating the sequence probability of the final generated code as an additional observation. Let u_{sp} denote the log sequence probability and let Y indicate whether the solution is correct. On the training split, we select the threshold τ that maximizes the positive likelihood ratio:

$$\frac{P(u_{\text{sp}} \geq \tau \mid Y = 1)}{P(u_{\text{sp}} \geq \tau \mid Y = 0)}. \quad (2)$$

The corresponding likelihoods are estimated using Laplace smoothing. At test time, all parameters are frozen, and a single additional Bayesian update is applied according to whether $u_{\text{sp}} \geq \tau$.

Table 1 shows the Prediction Rejection Ratio (PRR) (Malinin and Gales, 2021; Vashurin et al., 2025), which measures the area under the rejection curve. This curve plots the average quality of the remaining generations as we progressively abstain from an increasing fraction of the most uncertain predictions. Following standard practice (Vashurin et al., 2025), we compute PRR up to a rejection

Benchmark	Generator	b_0	n	Δ clean	Δ worst-uniform	Δ worst-alt
LCB-hard	Haiku 4.5	.079	102	+23.1 [18.2, 27.1]	+20.5 [16.9, 23.6]	+20.5 [16.9, 23.6]
LCB-hard	Sonnet 4.5	.173	102	+14.6 [11.3, 17.3]	+13.7 [10.6, 16.0]	+15.2 [12.0, 17.6]
LCB-medium	Haiku 4.5	.196	207	+16.6 [15.0, 18.2]	+15.6 [14.2, 16.9]	+17.1 [15.6, 18.5]
LCB-medium	Sonnet 4.5	.292	207	+13.7 [12.0, 15.6]	+13.1 [11.2, 15.0]	+14.4 [12.9, 16.1]
HumanEvalFix	Haiku 4.5	.889	164	+ 4.1 [0.8, 7.7]	+ 1.4 [-1.9, 5.2]	+ 4.1 [0.8, 7.7]
HumanEvalFix	Sonnet 4.5	.868	164	+ 7.2 [3.3, 11.1]	+ 5.1 [1.2, 9.0]	+ 7.2 [3.3, 11.1]

Table 16: Sensitivity of Bayes-DP to perturbations of the estimated critic likelihoods. b_0 is the prior pass rate, n is the number of tasks, and Δ is the mean utility gain over `always_verify` with paired-bootstrap 95% CIs. “Worst-uniform” and “worst-alt” denote the worst result across the four uniform and two adversarial alternating perturbations, respectively.

rate of 50%.

The Bayes State + UQ yields the best average results outperforming the second-best method by 7.4% on average and is particularly strong on Code-Contests. This suggests that aggregating multiple tool observations using a Bayesian model with uncertainty is more informative than treating all tool outcomes uniformly. The tool success rate performs better on average than other uncertainty quantification methods. Sequence probability is also competitive, indicating that the model’s own generation probability contains substantial information about final correctness. In contrast, its length-normalized variant, perplexity, is less reliable, especially on LCB-Medium, where normalization discards much of the length-sensitive signal captured by sequence probability.

Overall, this result demonstrates that aggregating multiple observations from critics and baseline uncertainty quantification methods using a Bayesian belief state provides a highly effective uncertainty measure for arbitrary external agents, showing strong performance in identifying erroneous generations.

E Additional Results

This appendix collects the per-benchmark, per-strategy, per-critic, and per-(benchmark, generator) slices of the regime map. All figures are reproduced from the same single-shot calibration + iterative-refinement corpus described in §5, covering the 9-benchmark $\times$ 6-generator grid. Unless otherwise noted, figures use the per-benchmark cost vectors: *fast-oracle* ($C_{\text{ver}}=5$, $C_{\text{crit}}^{\text{syn}}=C_{\text{crit}}^{\text{test}}=C_{\text{crit}}^{\text{llm}}=1$, $C_{\text{gen}}=10$, $R=100$) on the function-level synthesis and bug-fixing benchmarks; *slow-oracle* ($C_{\text{ver}}=90$, $C_{\text{crit}}^{\text{syn}}=1$, $C_{\text{crit}}^{\text{test}}=2$, $C_{\text{crit}}^{\text{llm}}=5$, $C_{\text{gen}}=10$, $R=100$) on the SWE-Bench rows.

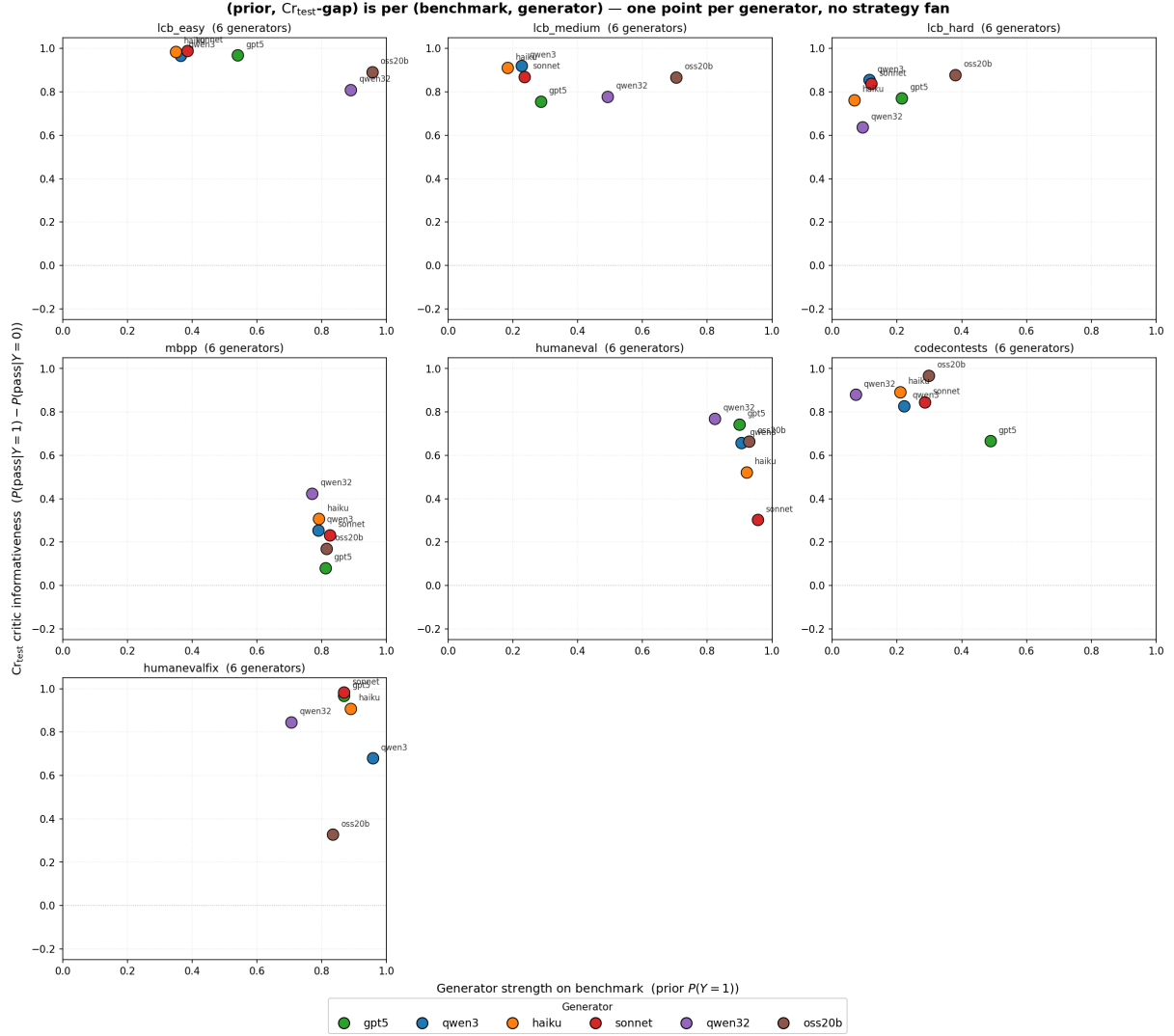


Figure 7: Per-benchmark location of each generator in $(P(Y=1), Cr_{\text{test}} \text{ critic informativeness } \gamma_{\text{test}} = P_{\text{test}}(z=\text{pass} | Y=1) - P_{\text{test}}(z=\text{pass} | Y=0))$. One subplot per benchmark, each dot one generator. The position of each cell in this plane determines which regime (A/B/C in Figure 2) it falls into. SWE-Bench Lite and SWE-Bench Verified are omitted: their oracle verifier is the project’s own test suite, so there is no public-test critic Cr_{test} distinct from the verifier to plot.

E.1 Held-out Policy Comparison Across Benchmark–Generator Pairs

Figures 8–16 expand the headline Figure 3 to every (benchmark, generator) pair on the held-out 25% eval split. One bar chart per cell; the gold-starred bar marks the winner. The Bayesian variants win or tie on the regime-A cells; $\text{gate}(\text{Cr}_{\text{test}})$ wins on regime-B; `always_verify` wins on the saturated cells. Self-Refine and Reflexion are evaluated as actual agents (not policy replays) and consistently underperform on cells with weak refine signals.

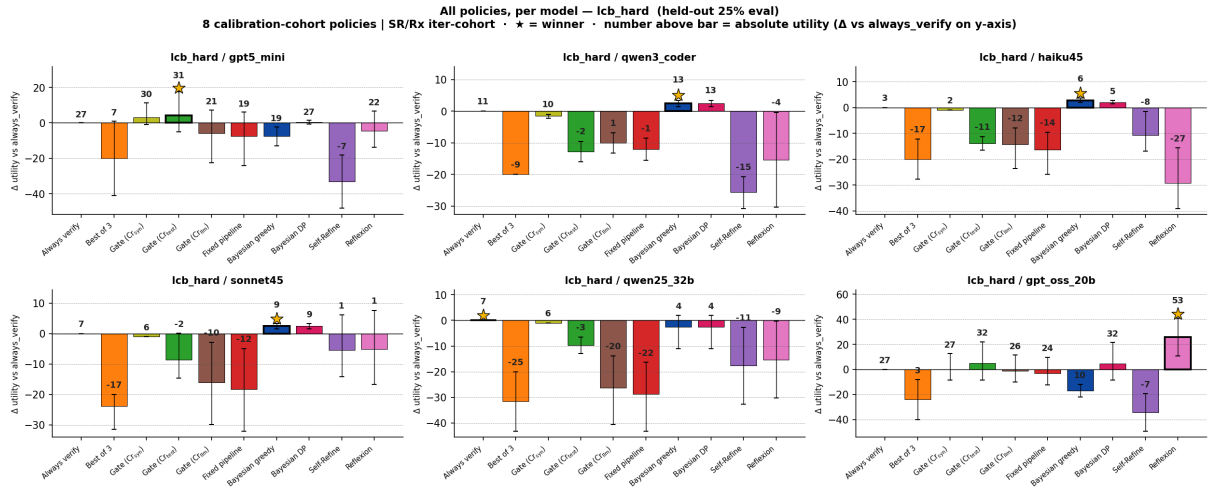


Figure 8: Held-out eval policy comparison on LCB-hard, all generators in one plot. Bayesian variants dominate this regime-A cell across every generator. Companion to Figure 3 (SWE-Bench Lite / claude-haiku-4.5 cell shown in the main text).

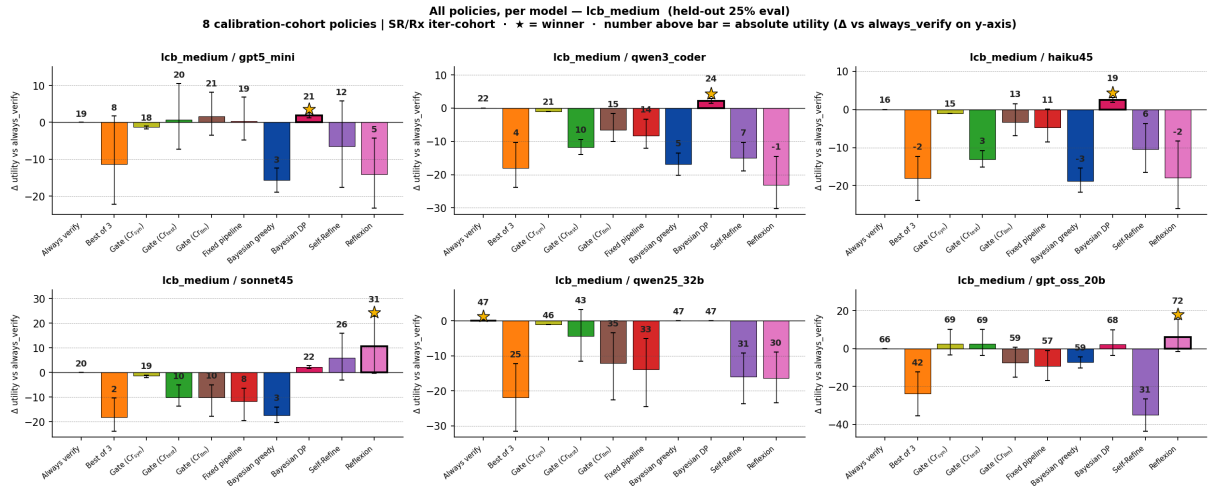


Figure 9: Held-out eval policy comparison on LCB-medium, all generators in one plot.

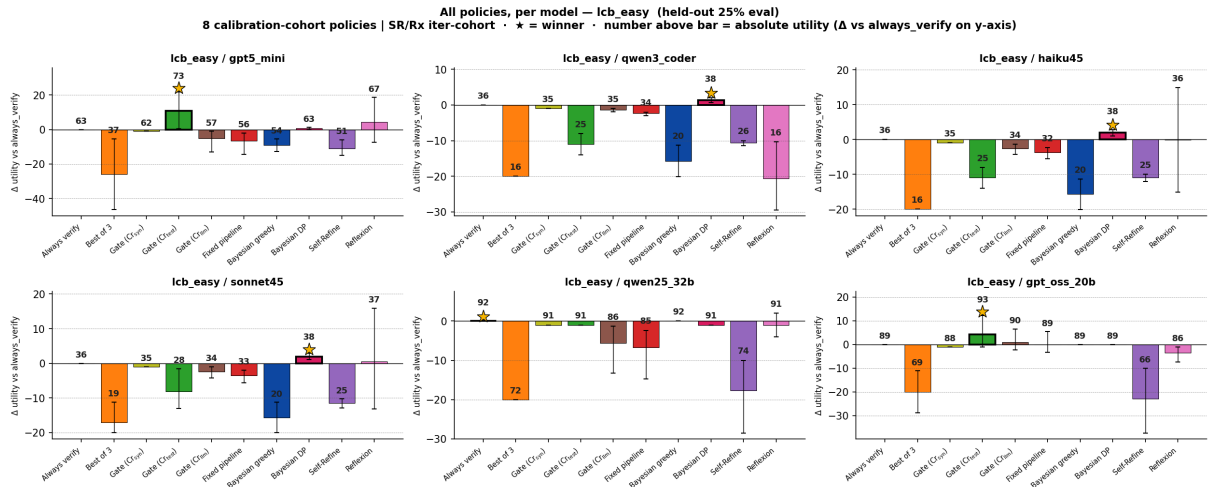


Figure 10: Held-out eval policy comparison on LCB-easy, all generators in one plot. gate(C_{test}) wins on most cells here – LCB-easy’s near-oracle public-test critic places these cells in regime B.

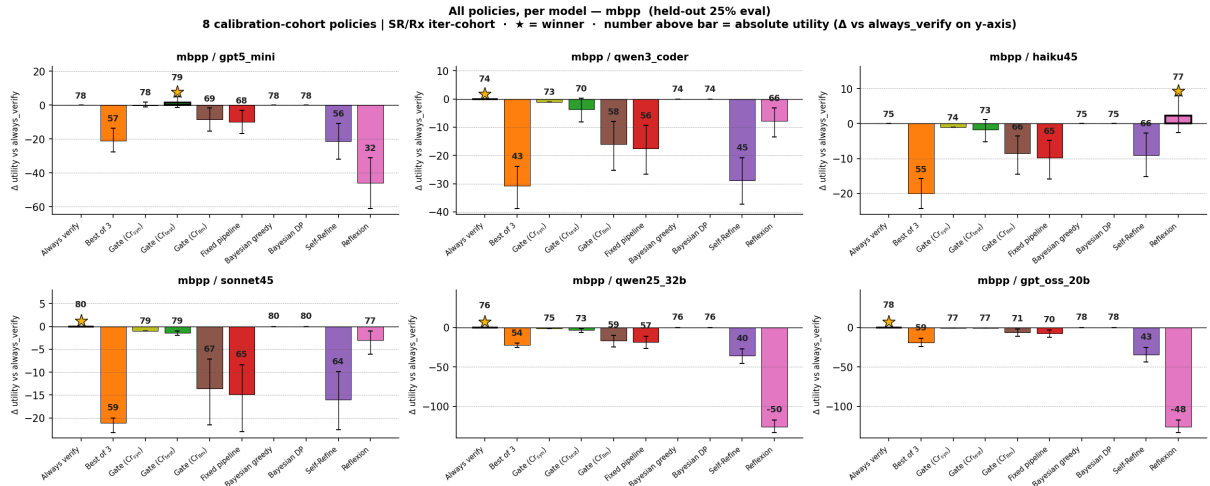


Figure 11: Held-out eval policy comparison on MBPP+, all generators in one plot.

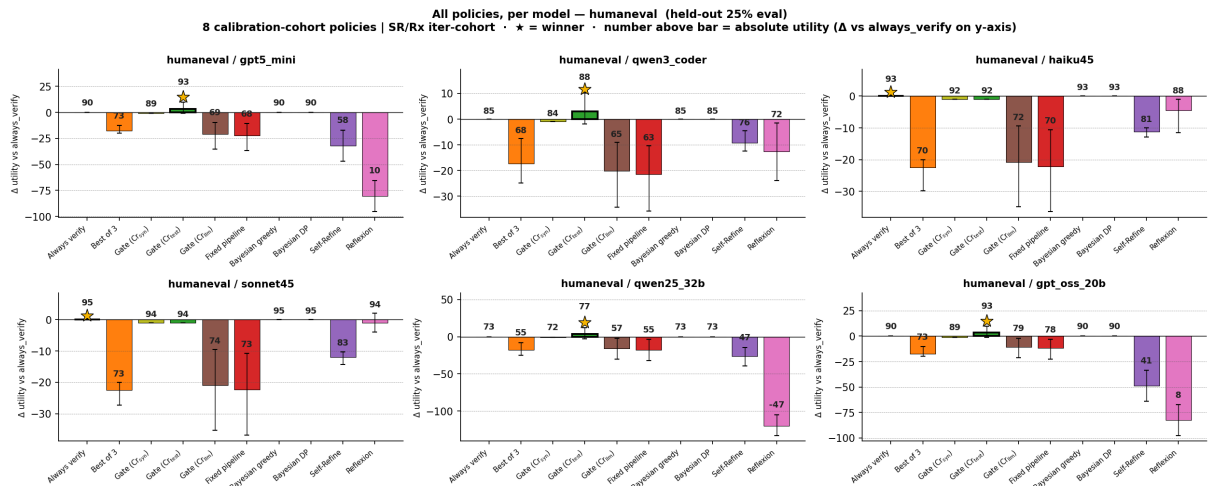


Figure 12: Held-out eval policy comparison on HumanEval+, all generators in one plot.

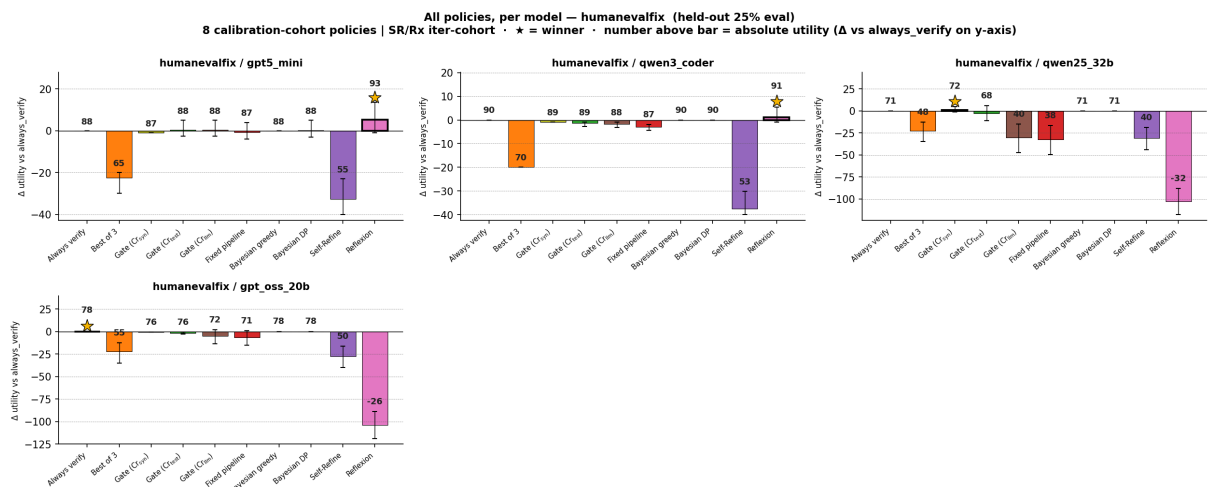


Figure 13: Held-out eval policy comparison on HumanEvalFix.

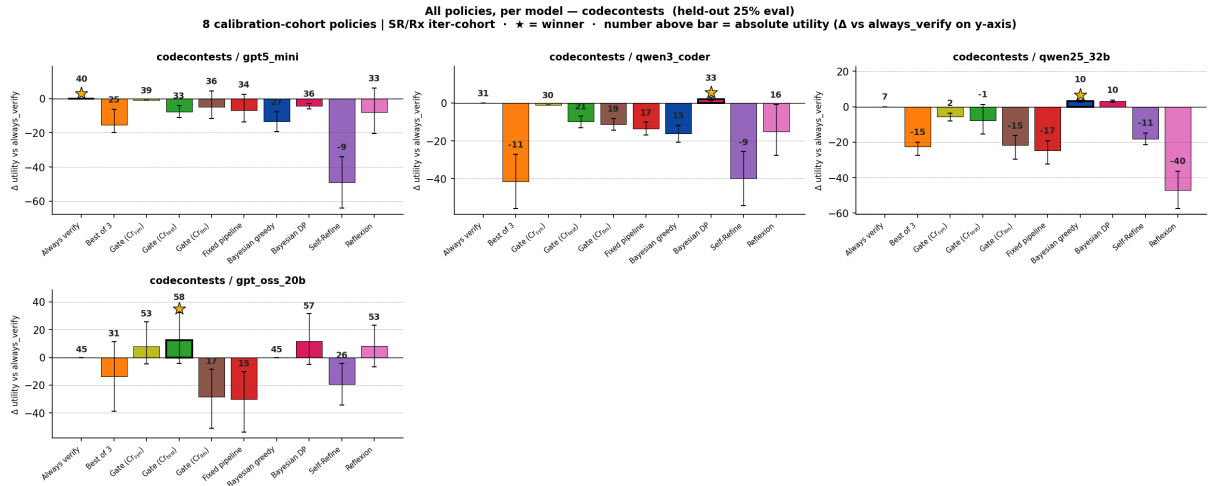


Figure 14: Held-out eval policy comparison on CodeContests.

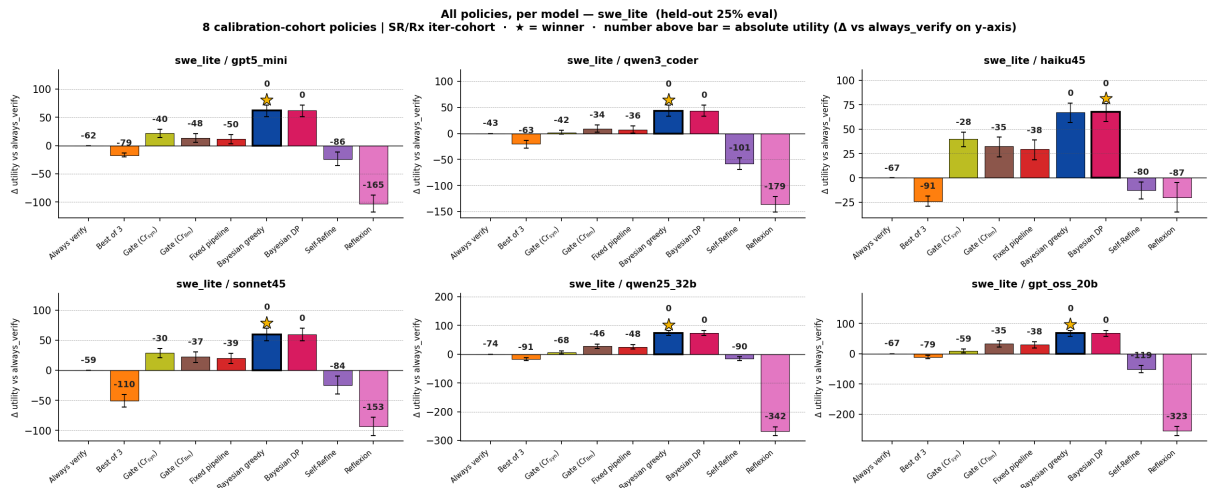


Figure 15: Held-out eval policy comparison on SWE-Bench Lite, all generators in one plot (the claude-haiku-4.5 cell of this figure is the headline shown as Figure 3 in the main text).

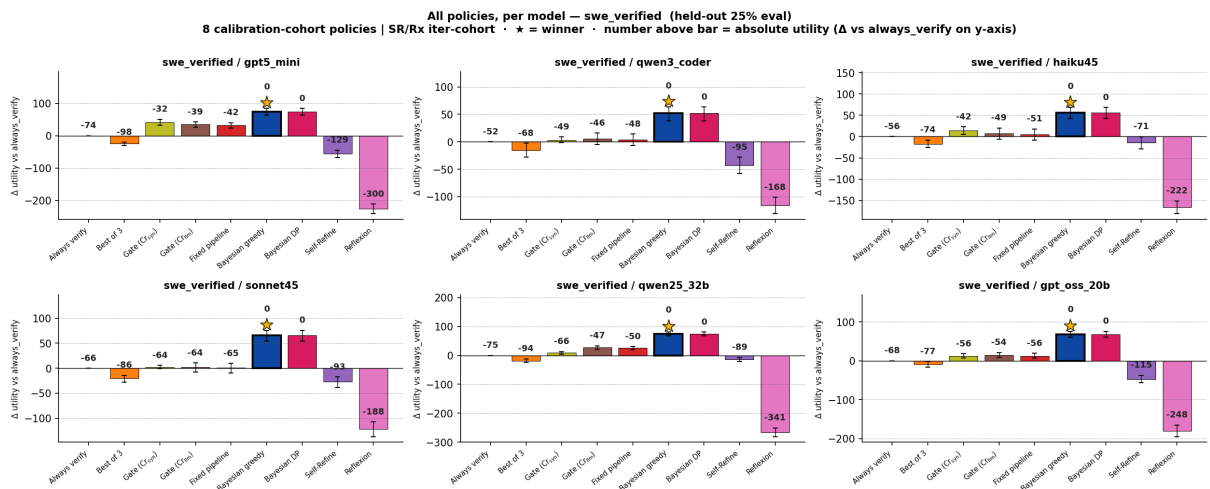


Figure 16: Held-out eval policy comparison on SWE-Bench Verified, all generators in one plot.

E.2 Train-vs-Evaluation Policy Comparison Across Benchmark–Generator Pairs

Figures 17–25 extend the headline train-vs-eval comparison from Figure 3 (SWE-Bench Lite / claude-haiku-4.5) to every benchmark. Each figure shows one benchmark across all six generators; the left column is the 75% train cohort that the controllers were fit on, the right column is the held-out 25% eval cohort. The Bayesian variants maintain positive Δ on *both* cohorts in every regime-A cell, ruling out overfitting of the likelihoods, prior, or kernel to the calibration instances.

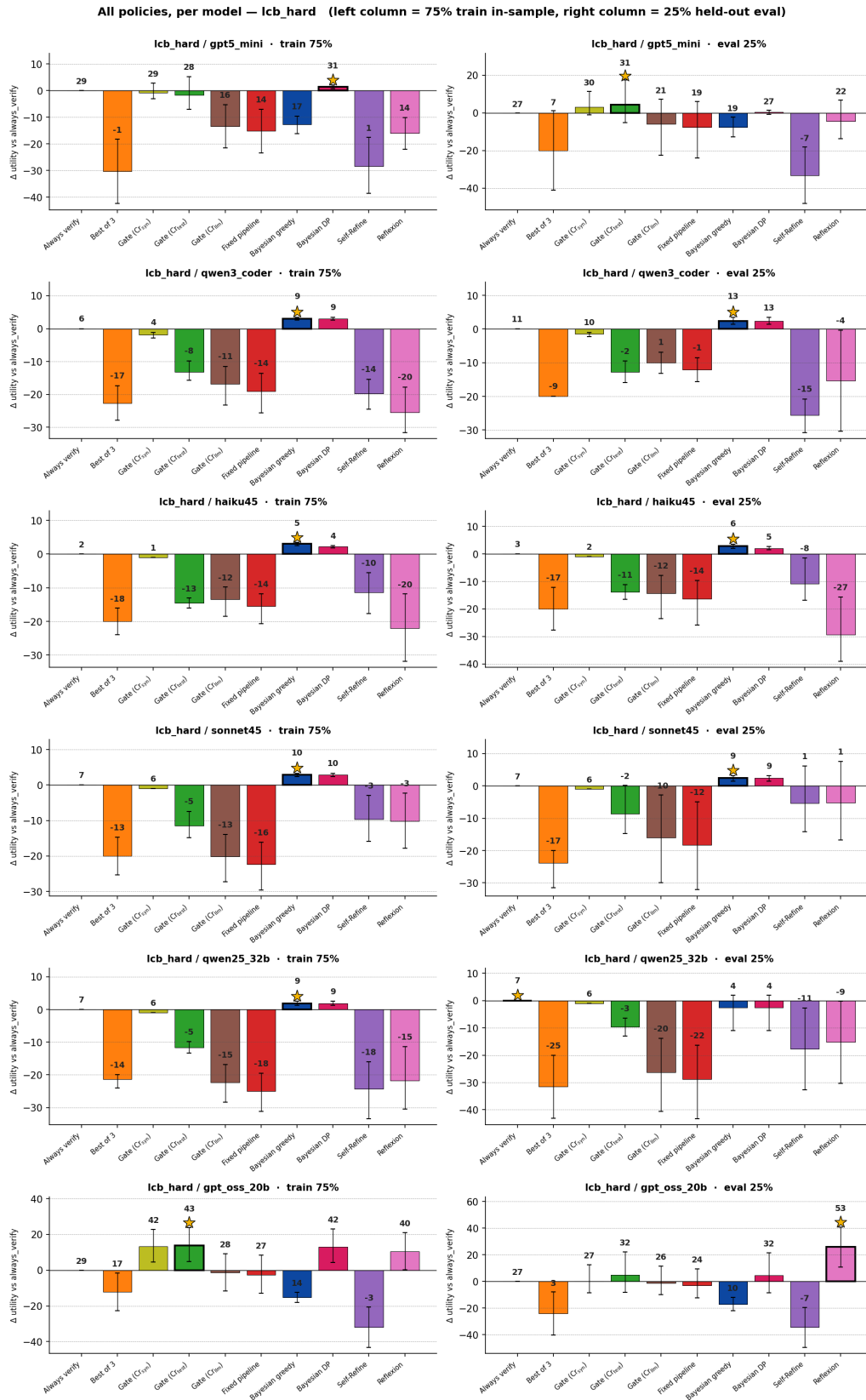


Figure 17: Train-vs-eval policy comparison on LCB-hard. Left column: 75% train (in-sample fit). Right column: 25% held-out eval. One row per generator.

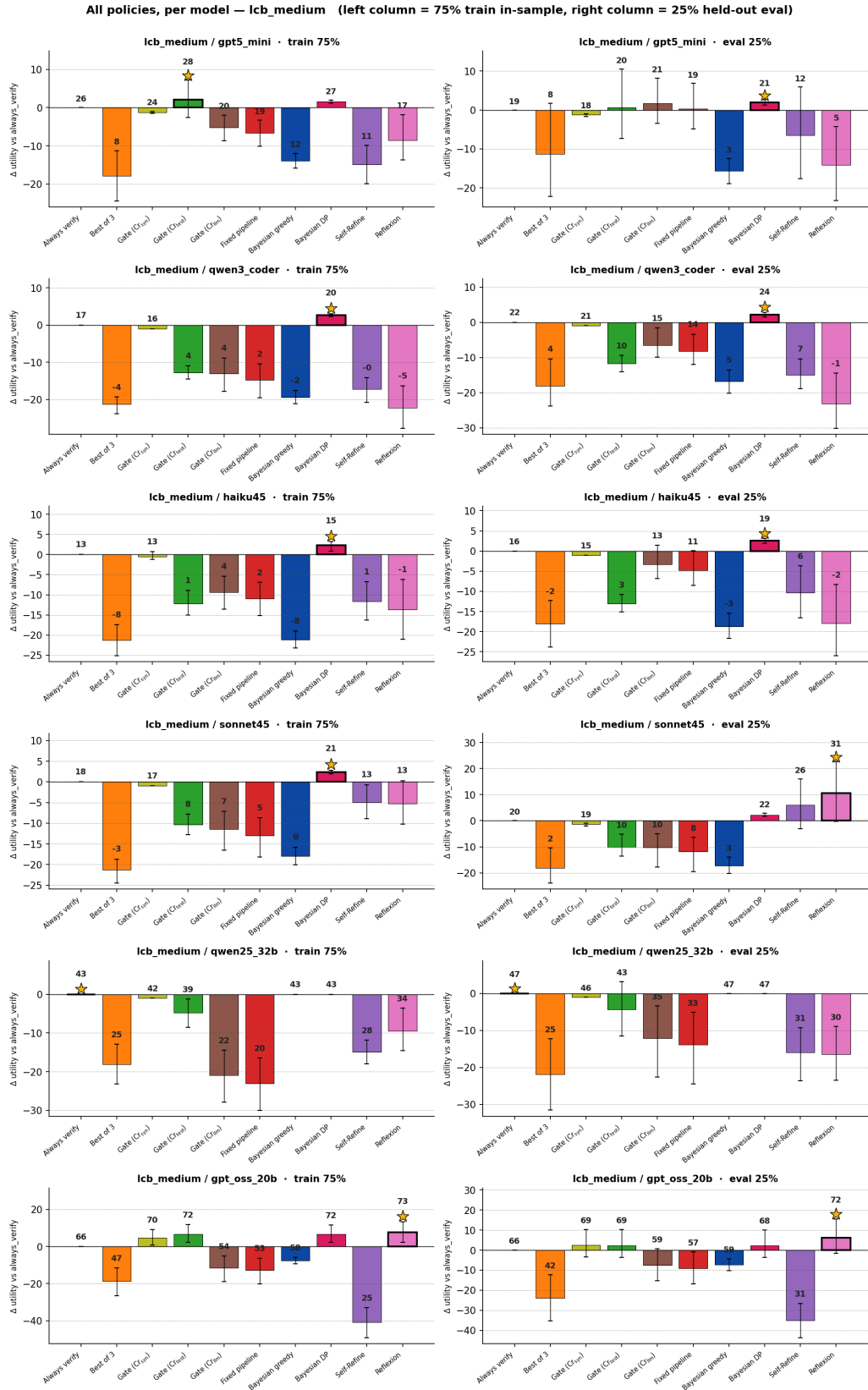


Figure 18: Train-vs-eval policy comparison on LCB-medium.

All policies, per model — lcb_easy (left column = 75% train in-sample, right column = 25% held-out eval)

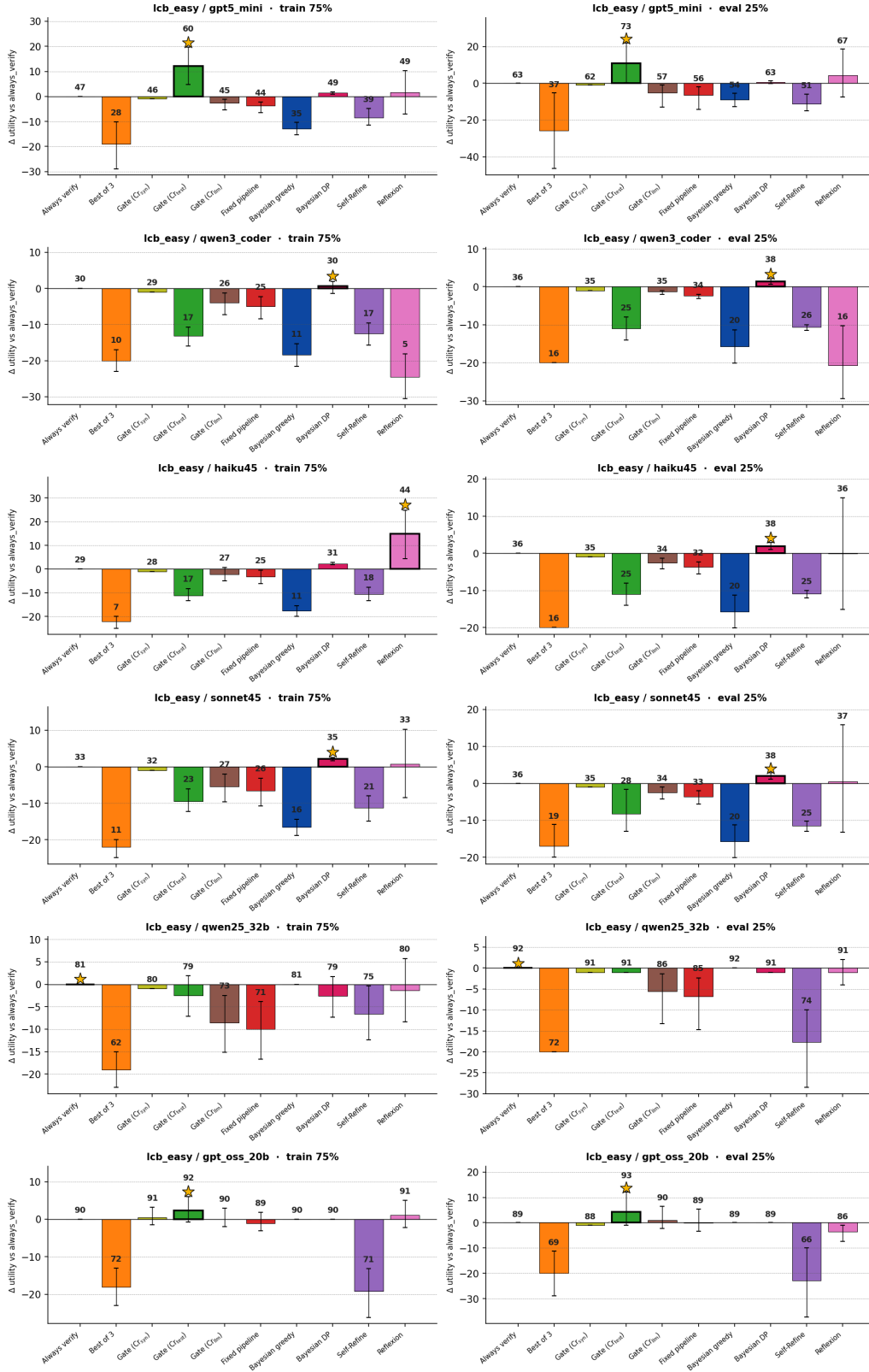


Figure 19: Train-vs-eval policy comparison on LCB-easy.

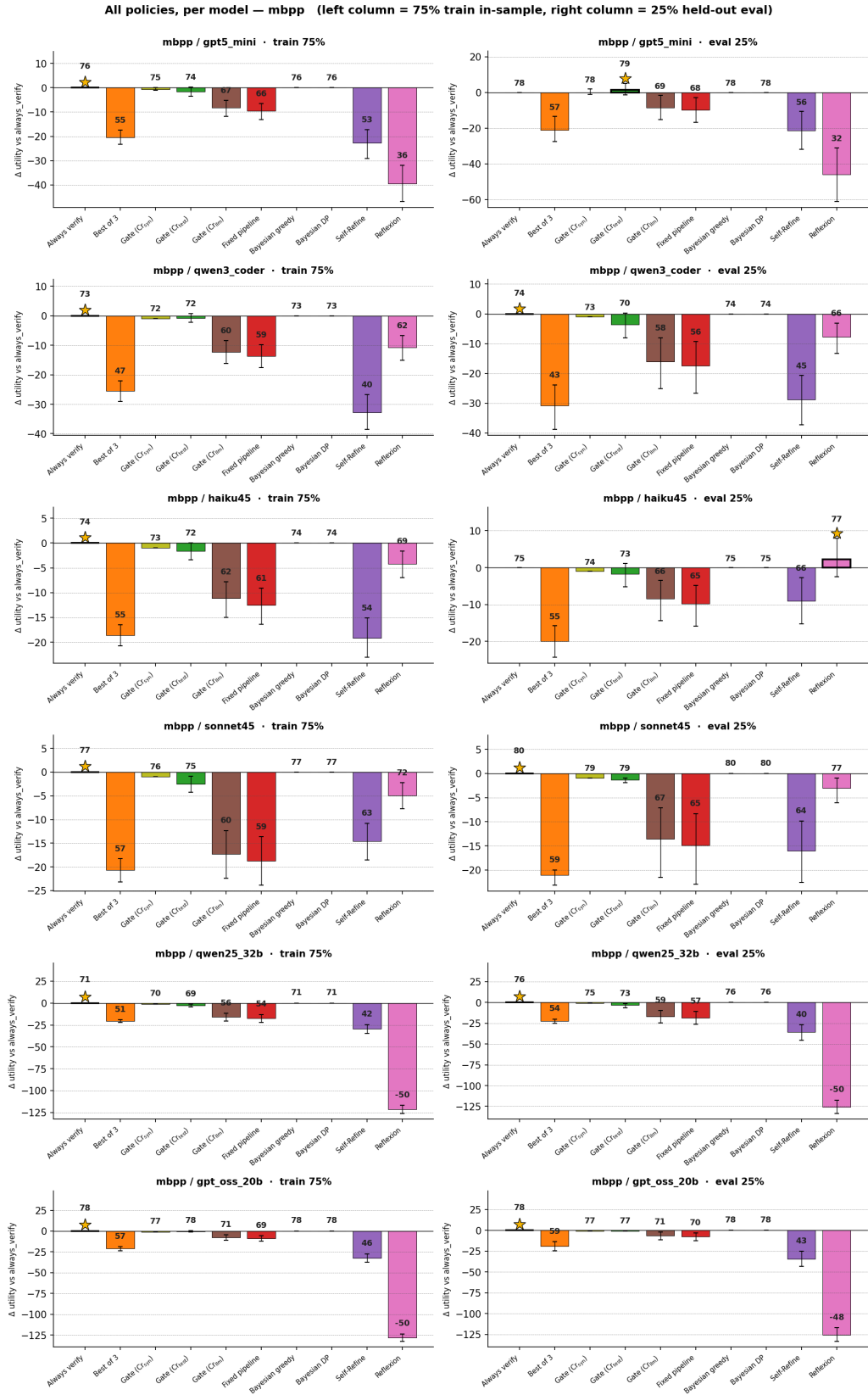


Figure 20: Train-vs-eval policy comparison on MBPP+.

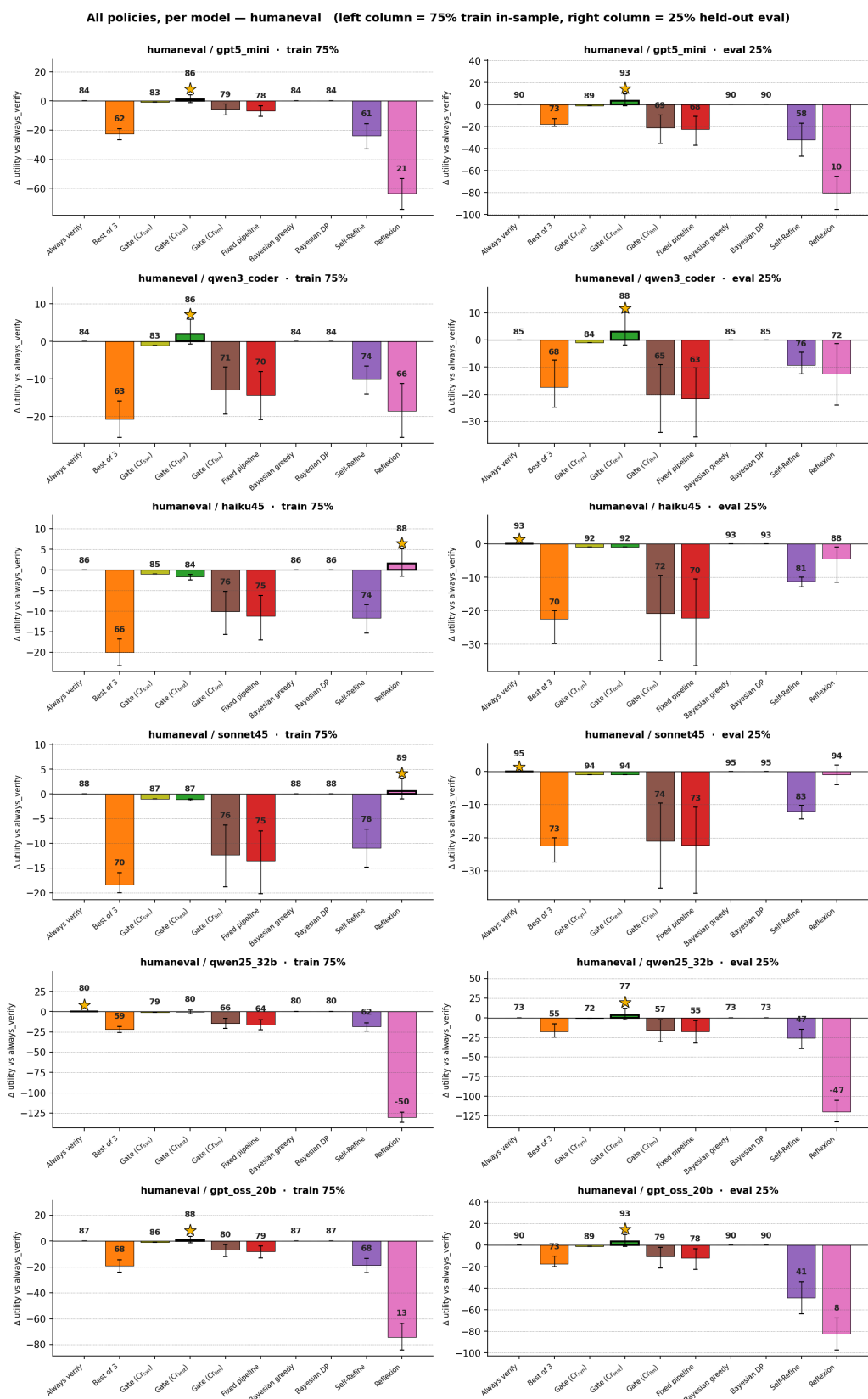


Figure 21: Train-vs-eval policy comparison on HumanEval+.

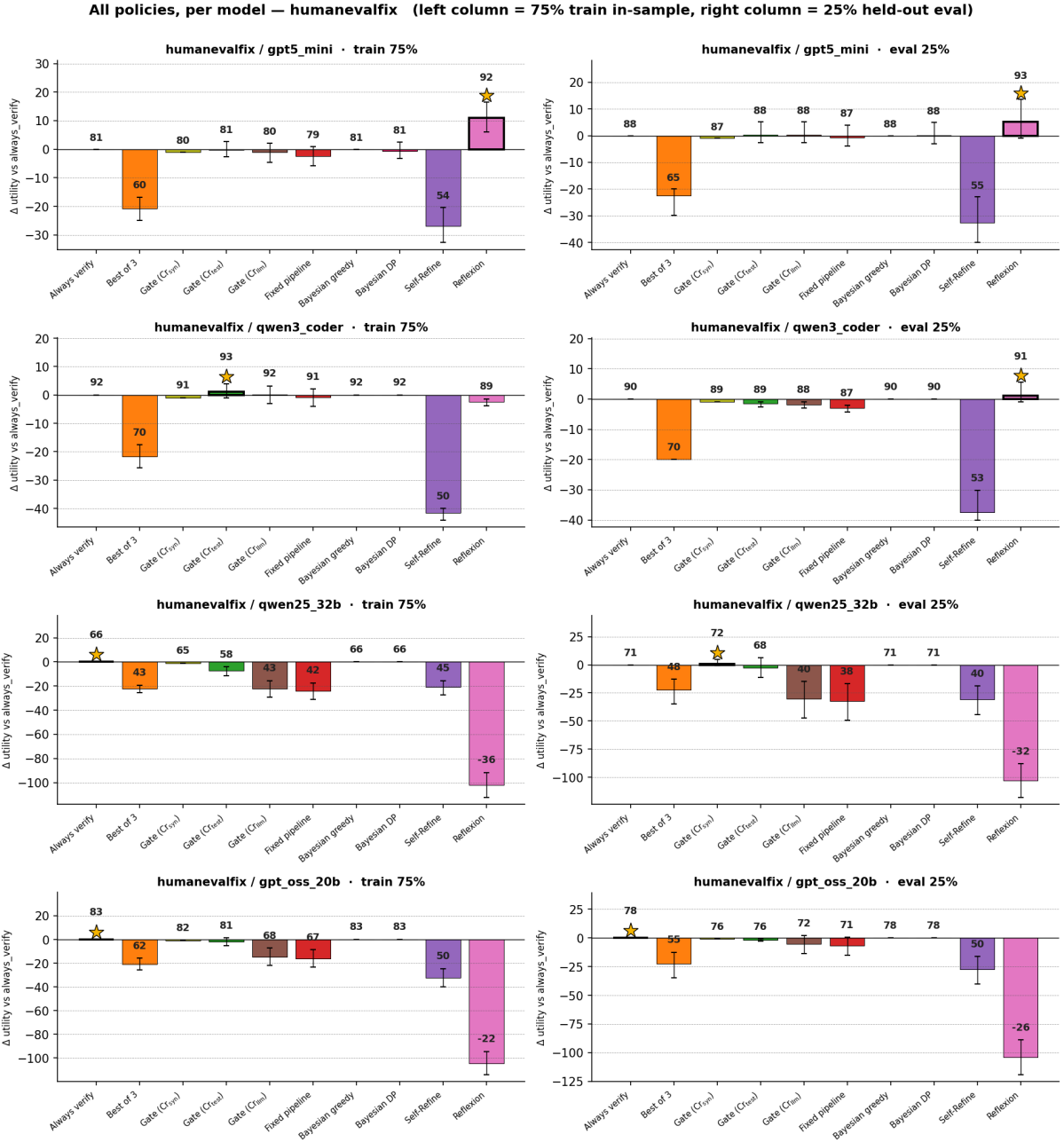


Figure 22: Train-vs-eval policy comparison on HumanEvalFix.

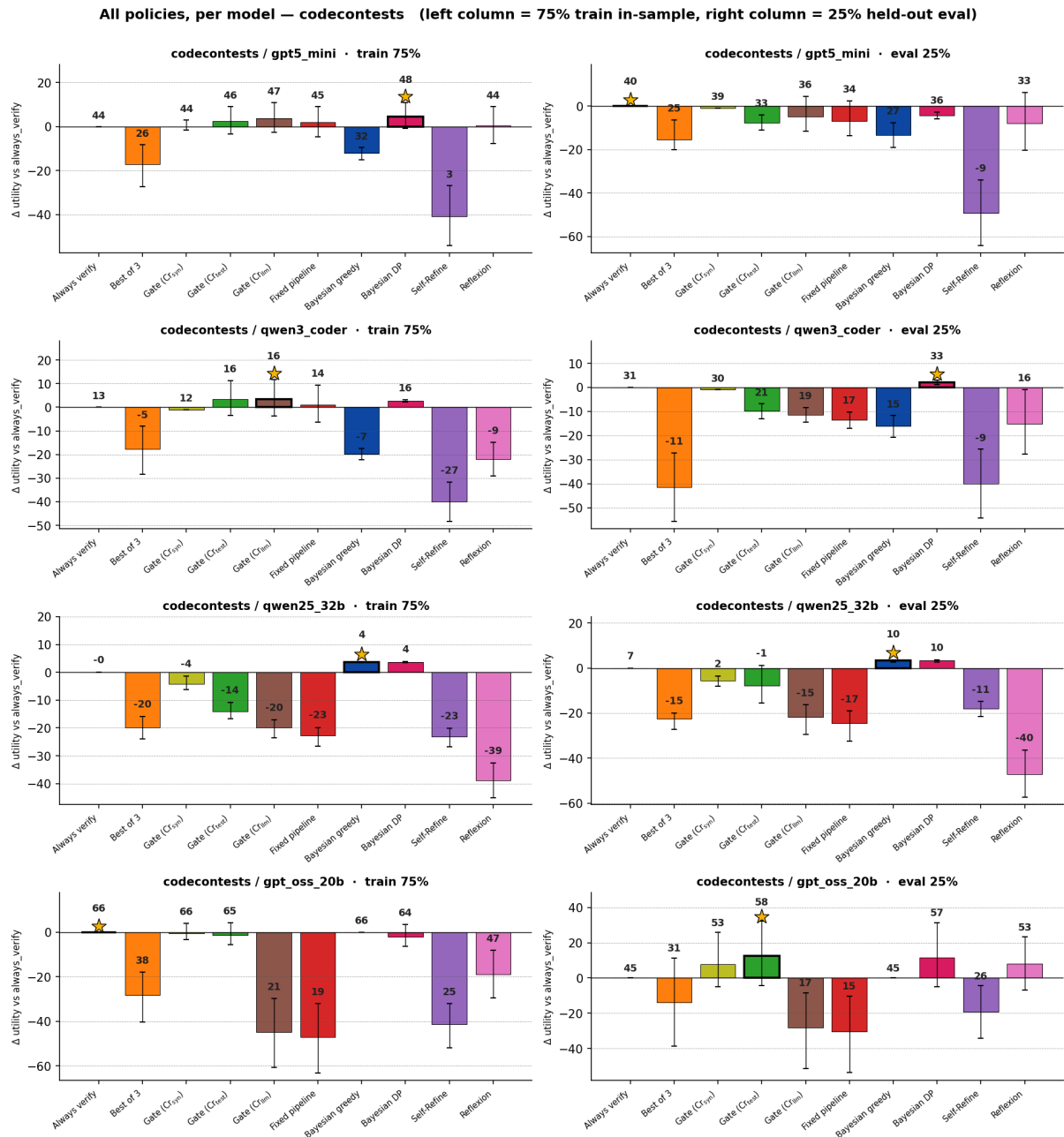


Figure 23: Train-vs-eval policy comparison on CodeContests.

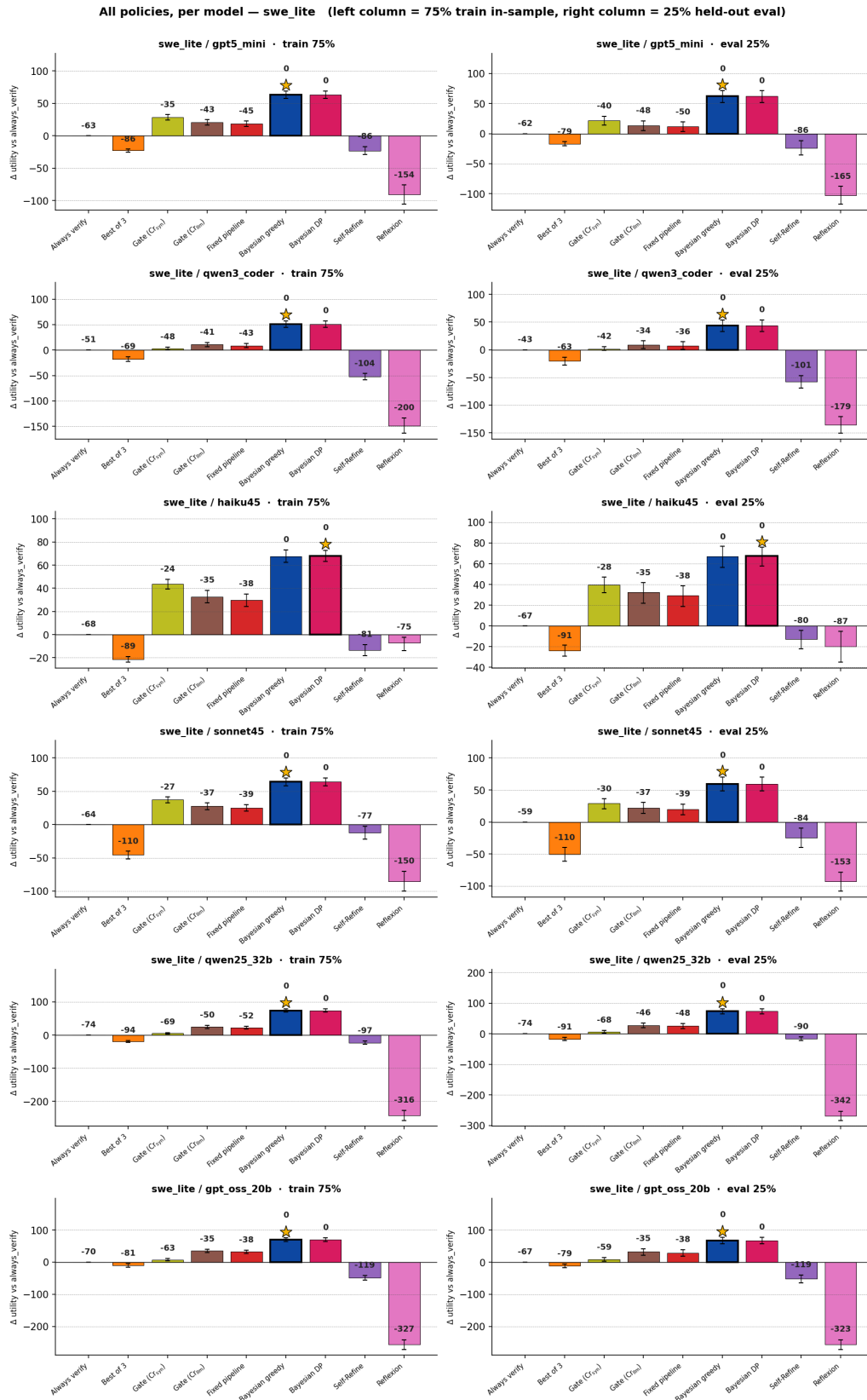


Figure 24: Train-vs-eval policy comparison on SWE-Bench Lite. The claude-haiku-4.5 row is the headline shown as Figure 3 in the main text.

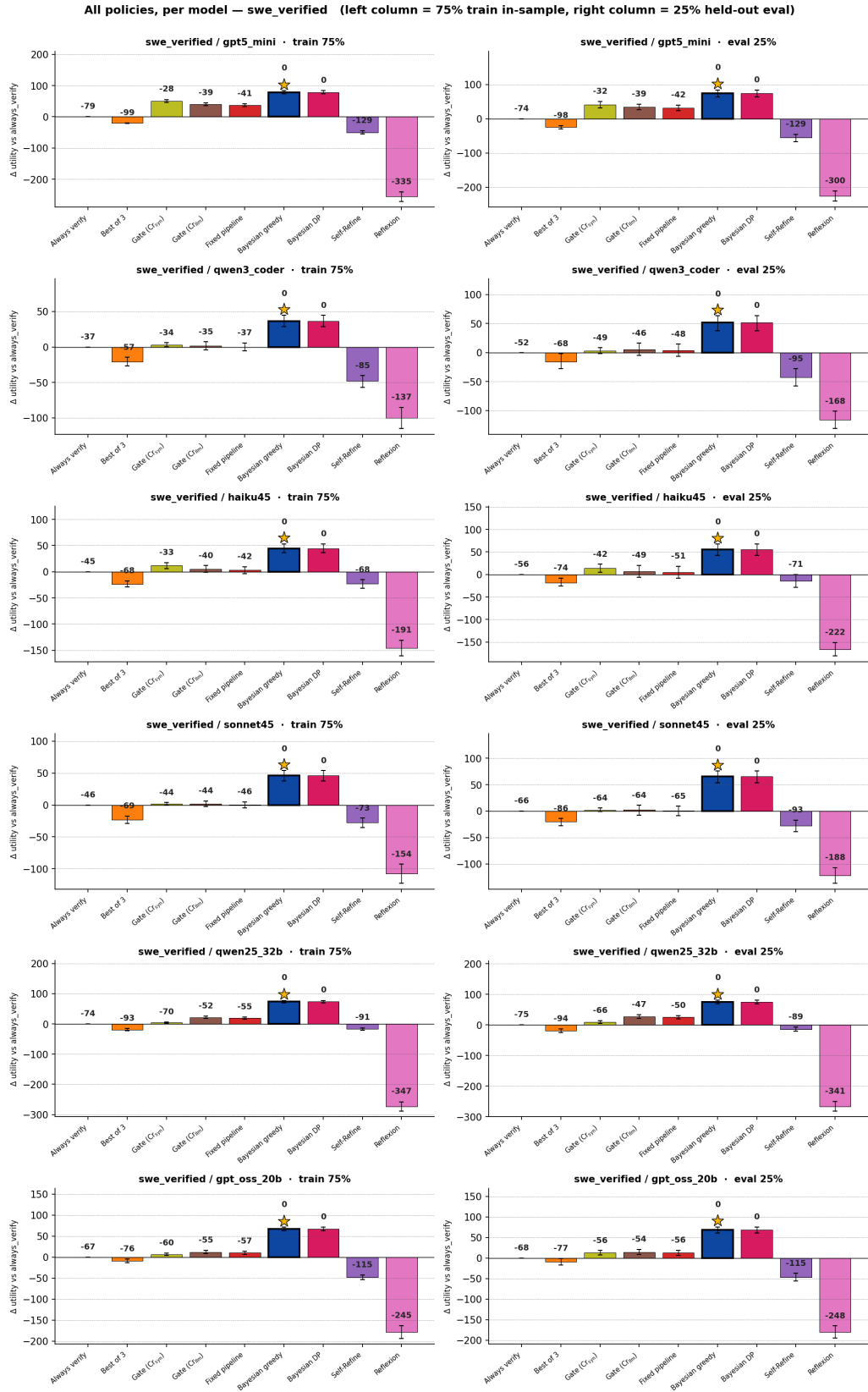


Figure 25: Train-vs-eval policy comparison on SWE-Bench Verified.

E.3 Per-Benchmark Verification-Cost Sweeps

Figures 26–34 extend the headline C_{ver} sweep from Figure 4 (MBPP+ / claude-haiku-4.5) to every evaluated benchmark. Each figure shows one benchmark with all six generators on a single composite grid; same axes and policy palette as the headline. Per-generator analytic crossover C_{ver}^*/R tracks each cell’s prior $P(Y=1)$ and is visible as the empirical leader-change boundary in each sub-plot.

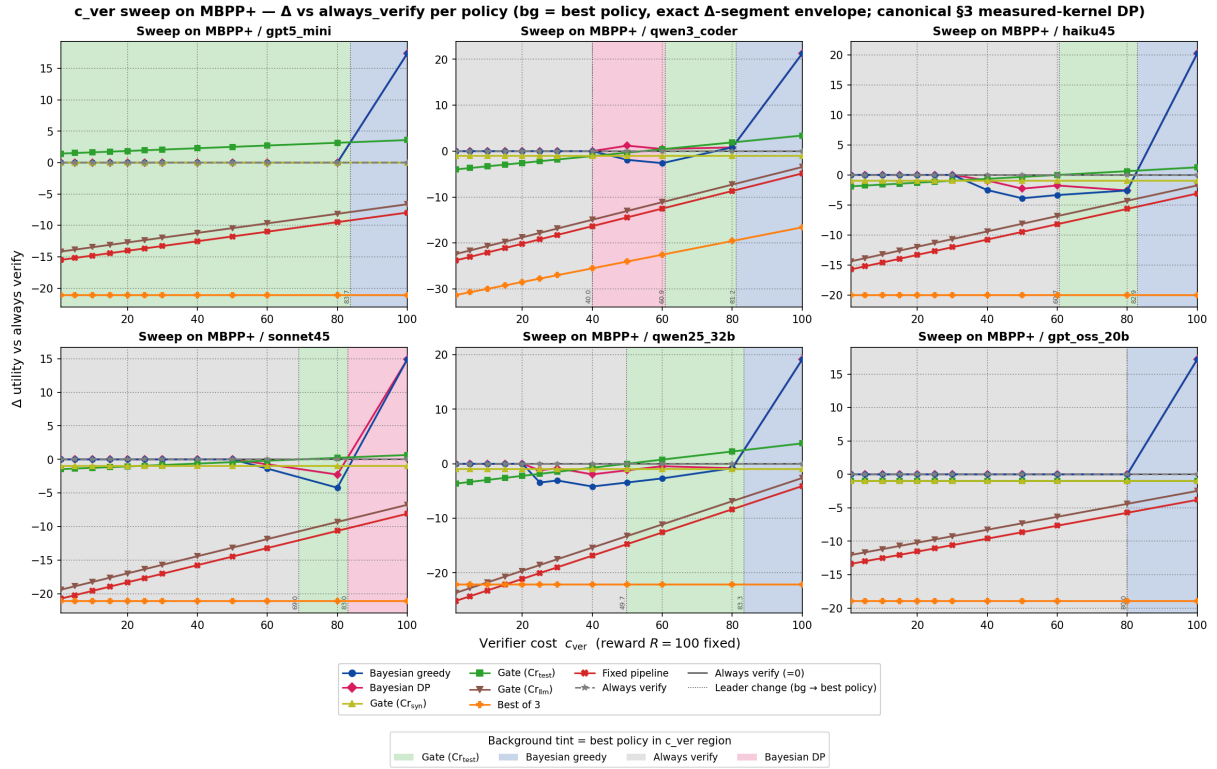


Figure 26: C_{ver} sweep on MBPP+, one subplot per generator (companion to Figure 4). The $C \rightarrow B$ handoff happens at slightly different C_{ver} values per generator, tracking each cell's prior $P(Y=1)$.

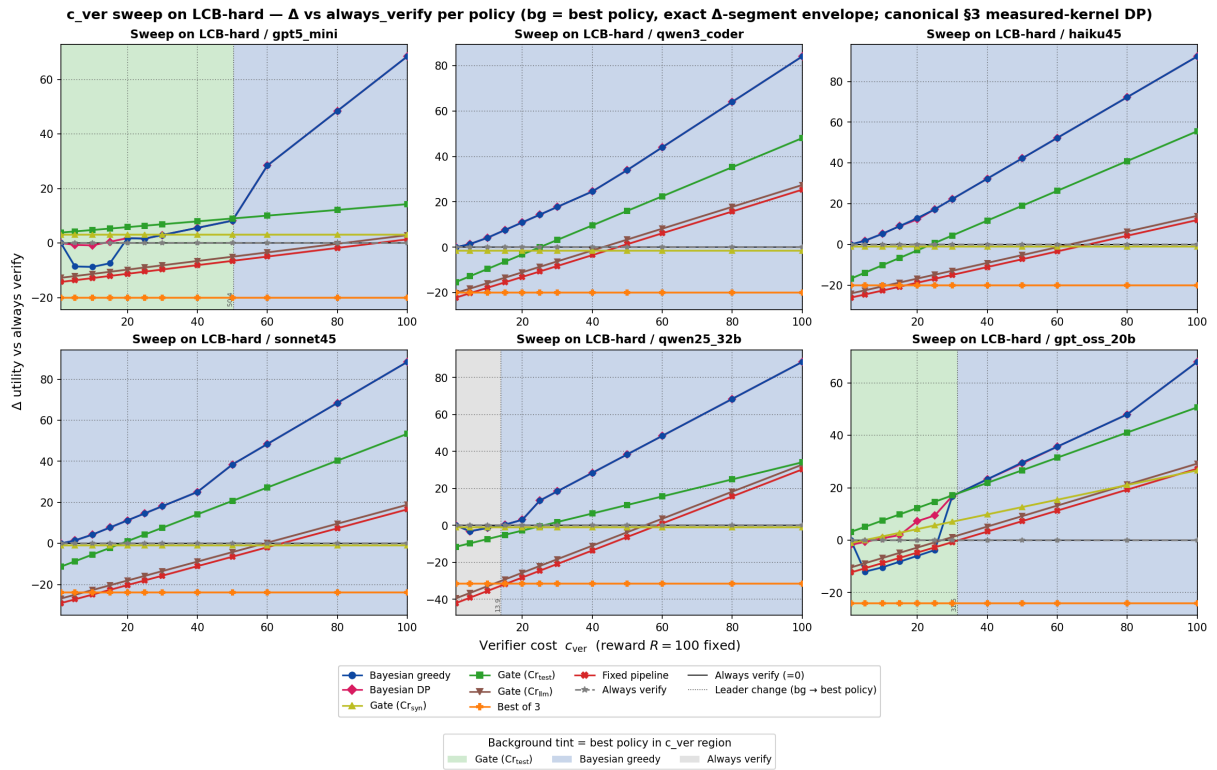


Figure 27: C_{ver} sweep on LCB-hard, one subplot per generator. LCB-hard's low prior ($P(Y=1) \approx 0.17$) places the analytic crossover at $C_{\text{ver}}^* \approx 17$, so the cell sits in regime A for nearly the entire sweep – the Bayesian variants lead above $C_{\text{ver}} \approx 20$. The two Bayesian curves overlap because the measured $P(\text{fix} \mid \text{broken}) \approx 0.07$ makes multi-step refinement negative-EV, so bayesian_DP collapses to the same one-step policy as bayesian_greedy.

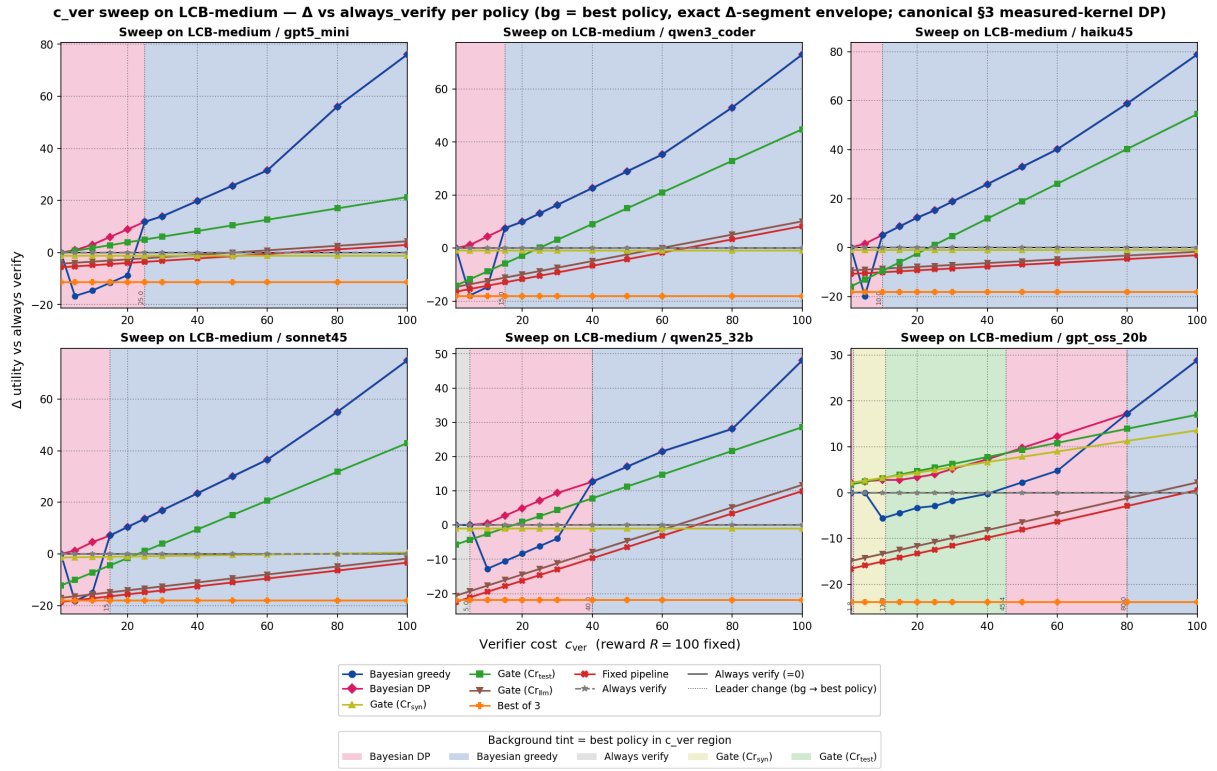


Figure 28: C_{ver} sweep on LCB-medium, one subplot per generator.

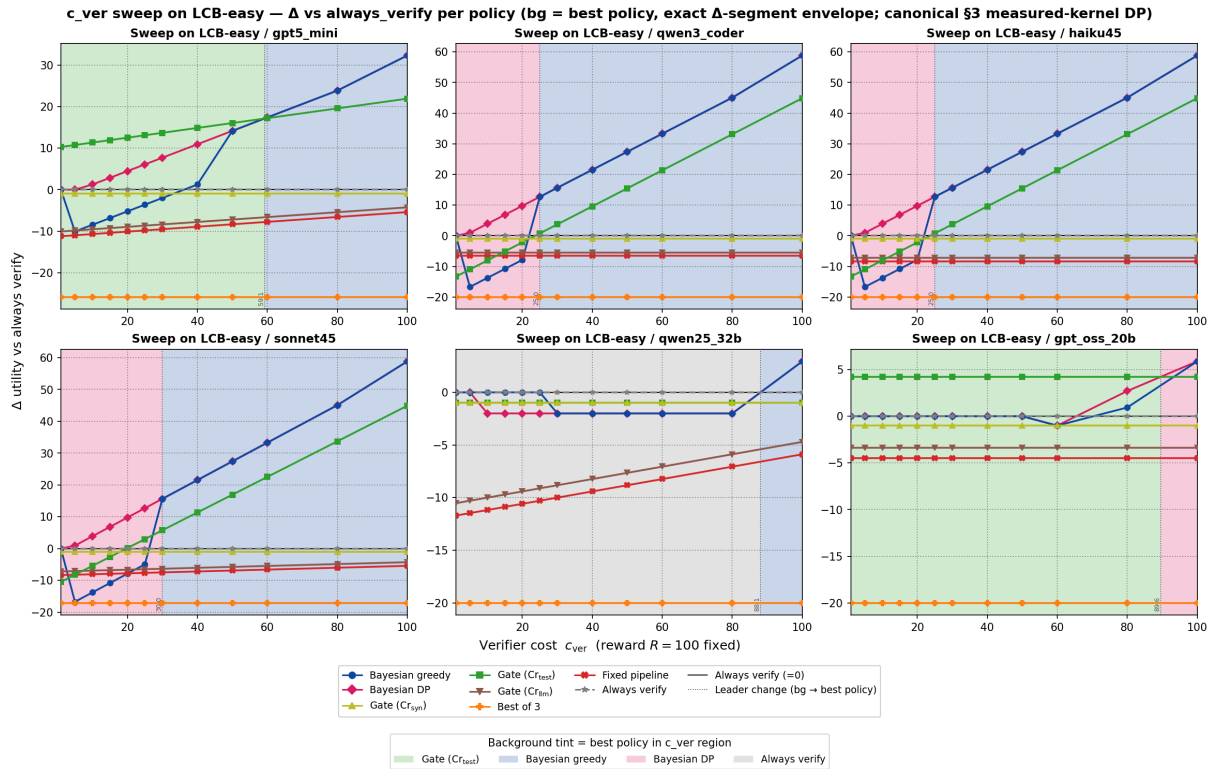


Figure 29: C_{ver} sweep on LCB-easy, one subplot per generator.

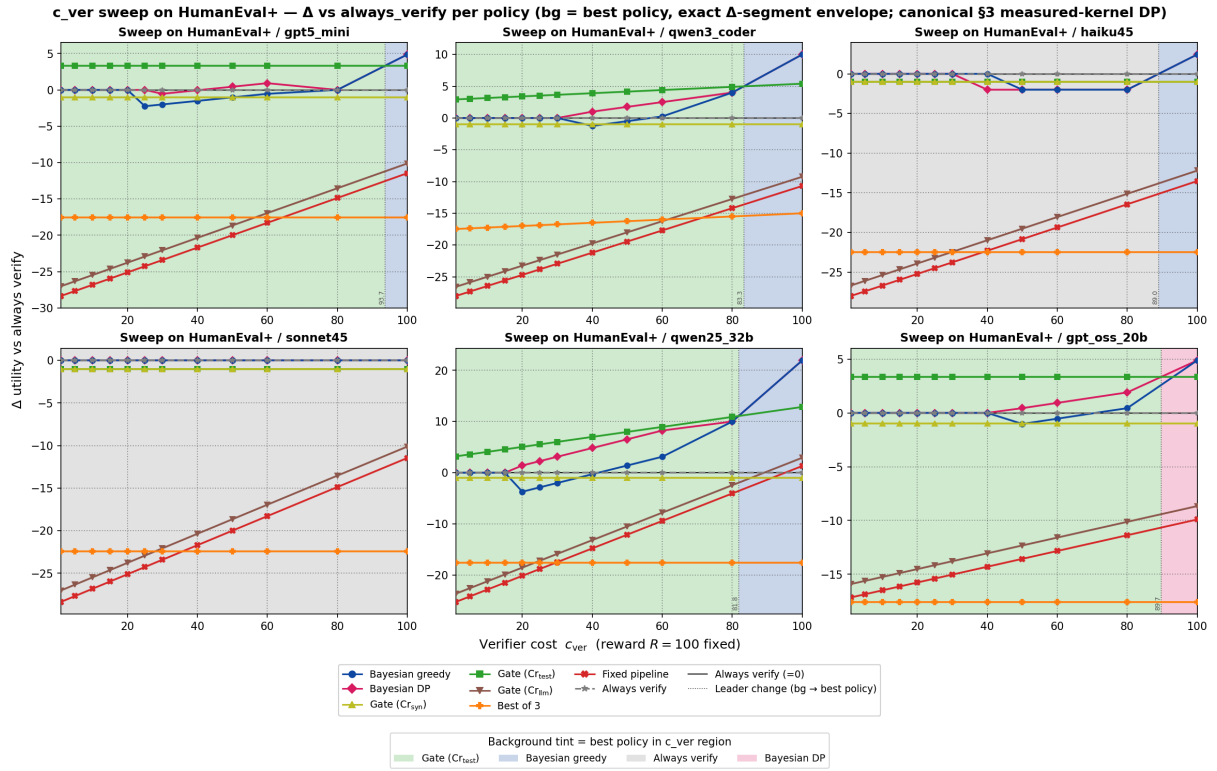


Figure 30: C_{ver} sweep on HumanEval+, one subplot per generator.

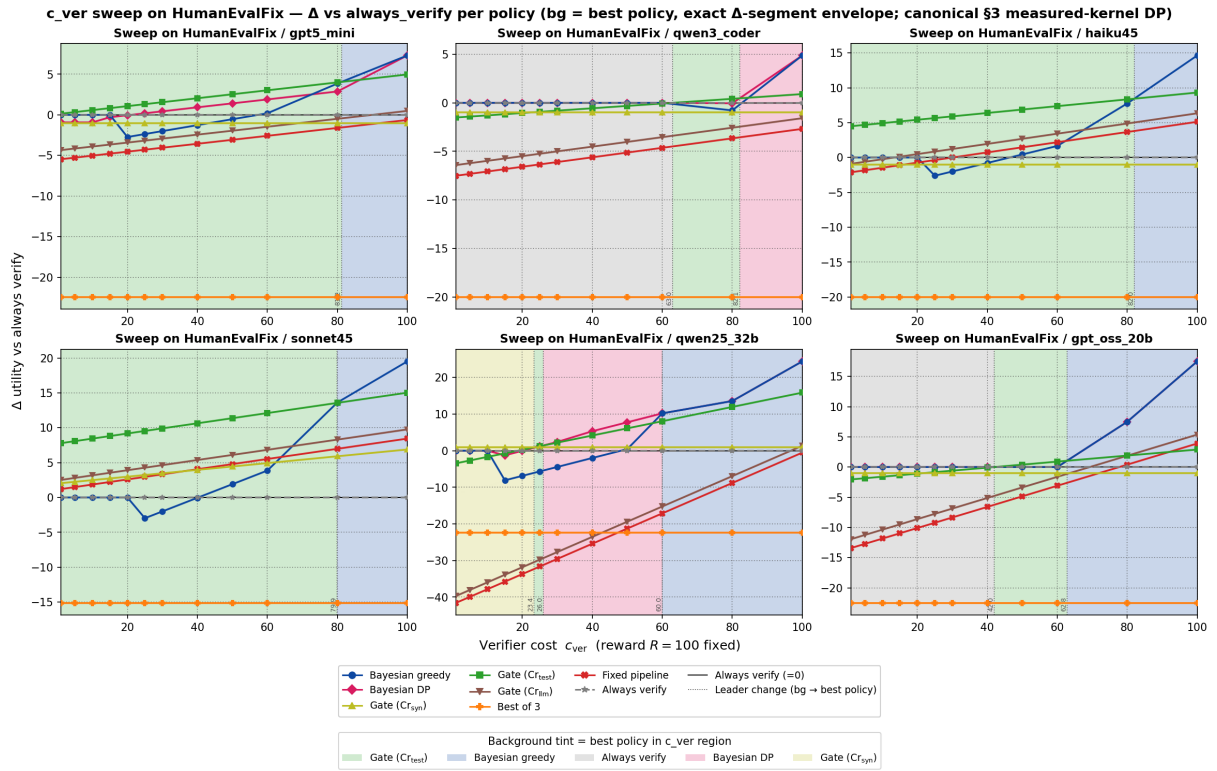


Figure 31: C_{ver} sweep on HumanEvalFix, one subplot per generator.

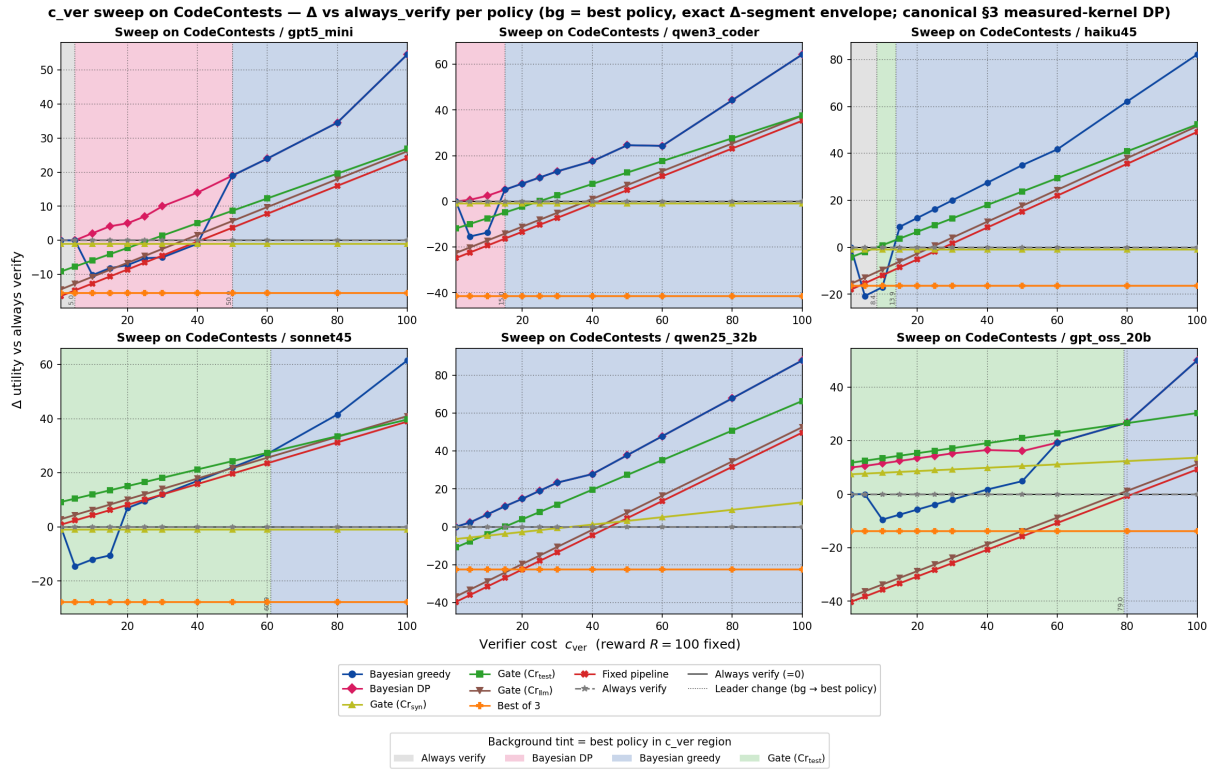


Figure 32: C_{ver} sweep on CodeContests, one subplot per generator.

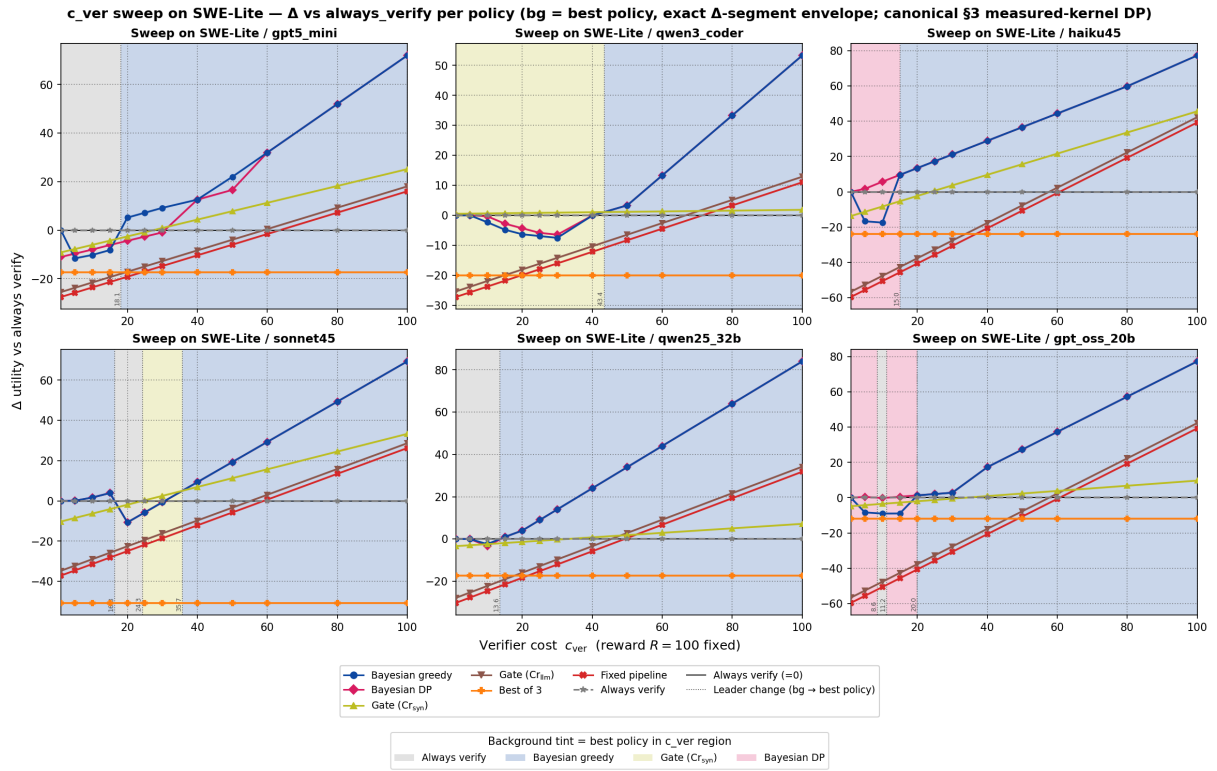


Figure 33: C_{ver} sweep on SWE-Bench Lite, one subplot per generator.

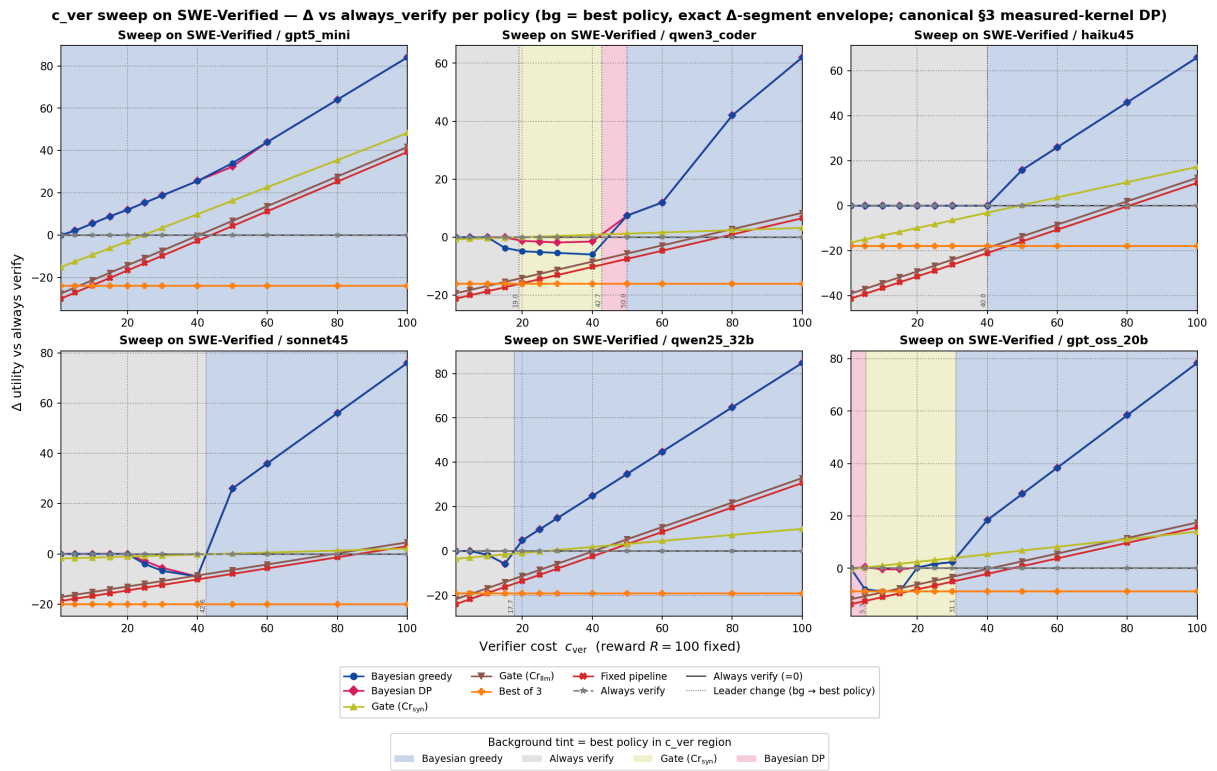


Figure 34: C_{ver} sweep on SWE-Bench Verified, one subplot per generator.