# LATENTSIFT: POLICY-STATE FILTERING FOR TOKEN-EFFICIENT VERIFICATION OF SOFTWARE ENGINEERING AGENTS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Test-time scaling improves software engineering agents by generating multiple candidate trajectories and selecting the best one. Verifying and selecting among these long interactions can consume as many tokens as generation itself. Existing hybrid workflows first apply an LLM-based execution-free (EF) verifier to filter candidates before running tests, which adds another model pass over every trajectory. We introduce LatentSift, a token-free and execution-free filter that replaces this first stage with hidden states the policy already produces while generating the candidates. It represents each candidate through its reasoning, observation, and function-call states, compares them with positive and negative banks of such states collected from successful and unsuccessful trajectories during policy training, and fuses the resulting distance scores with a learned linear score to retain promising candidates for the execution-based stages. On SWE-bench Verified, across three agents and two policy sizes, LatentSift cuts EF-verifier tokens by 66.6–81.0% and total verification tokens, which include test generation, by 49.1–62.1% at $K = 16$, while hybrid Best@16 matches or improves on each agent's reference workflow, rising from 59.26% to 60.06% on DeepSWE-Preview.

## 1 INTRODUCTION

Efforts to improve LLM reasoning in academia and industry follow two broad directions: training more capable models to solve harder problems, and allocating more computation to each problem at inference time. Training can strengthen the model's reasoning ability before deployment (Guo et al., 2025). The latter approach, known as test-time scaling, seeks better answers by allowing longer reasoning, generating multiple candidates, or searching with verifier feedback (Muennighoff et al., 2025; Wang et al., 2023; Aggarwal et al., 2023; Snell et al., 2025; Wu et al., 2025). Rather than relying on a single attempt, these methods spend additional inference compute exploring and selecting solutions of the problem.

For software engineering (SWE) tasks, recent open agents use a hybrid verification workflow to select among generated trajectories (Luo et al., 2025; Jain et al., 2025). Figure 1 shows its four stages: an execution-free (EF) verifier scores all candidates and retains half; instance-supplied tests provide regression signals; LLM-generated tests provide additional execution signals; and the EF scores derived from Stage 1 determine the final choice among survivors. The Stage 1 EF verifier is typically a separately fine-tuned LLM that reads the trajectories produced by the policy (Shum et al., 2026). Execution-free does not mean token-free: each candidate is processed by the verifier model again, adding verifier input and output tokens. This cost is particularly high on SWE tasks, where a trajectory commonly contains tens of thousands of tokens from reasoning, repository exploration, tool feedback, and code edits. Reprocessing such trajectories adds substantial token consumption to already expensive long-context inference (Kim et al., 2026; Kwon et al., 2023; Xie et al., 2025). For a recent open policy such as DeepSWE-Preview with $K = 16$ candidates, Stage 1 alone consumes approximately 850K tokens per task instance and the full verification workflow approximately 1,050K, exceeding the approximately 850K tokens used to generate the candidates. Candidate verification can thus rival generation as a source of inference-time token cost.

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

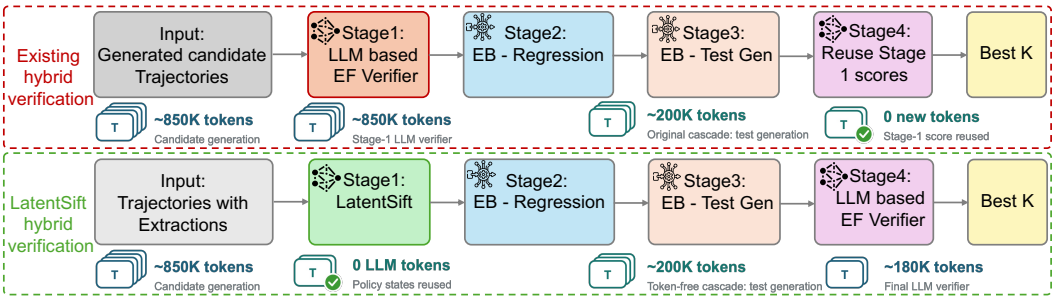


Figure 1: **Existing hybrid verification and the workflow with LatentSift.** The existing pipeline follows the hybrid verification design used by R2E-Gym and DeepSWE-Preview (Jain et al., 2025; Luo et al., 2025). Both pipelines select a patch from the same pool of $K$ generated candidates. The red and green dashed boxes mark inference-time verification for the existing pipeline and ours, respectively. The existing pipeline scores all candidates with an LLM-based execution-free (EF) verifier in Stage 1 and reuses these scores in Stage 4. LatentSift replaces Stage 1 with a filter that reuses cached policy states without additional LLM tokens. Both pipelines retain half the candidates before regression testing and LLM-generated testing; ours applies the final EF verifier only to the remaining candidates. Token counts show average costs per task instance on DeepSWE-Preview at $K = 16$, with candidate generation reported separately.

We introduce a token-efficient verification method called **LatentSift**. It filters candidate trajectories using hidden states collected during the generating process, introducing no additional LLM token cost and requiring no test execution. We represent each trajectory through three channels: reasoning, environment observations, and function calls. For each channel, we compare candidate states with positive and negative banks collected during policy training, aggregate the step-level distance margins, and rank candidates for the task instance. We take the minimum of the three channel ranks as the distance score, so a candidate scores highly only if it ranks highly in every channel. We also train a linear scorer on policy hidden states to assign each candidate a learned trajectory score. We rank these scores within each task instance, normalize the ranks, and average them with the candidates' distance-based scores to obtain the final filtering scores. This learned signal forms part of the filter. We use LatentSift to replace the highly token-costing Stage 1 of the hybrid workflow, retaining half the candidates so execution-based stages and the final LLM verifier operate on a smaller pool, significantly reducing the token cost of the verification workflow.

**Contributions.** (i) We introduce LatentSift, a token-free, execution-free filtering mechanism for long-horizon software engineering agents, and integrate it into an inference-time hybrid verification workflow. (ii) We design a three-channel trajectory representation that separately pools reasoning, observation, and function-call states already computed by the generating policy. We then develop a trajectory scoring method, which combines contrastive distances to successful and unsuccessful experience across the three channels through minimum-rank fusion, then integrates a learned trajectory score to rank candidates within each task instance. (iii) We evaluate LatentSift across multiple software engineering agents and policies on SWE-bench Verified, demonstrating substantial verification-token savings while maintaining competitive selection quality. On DeepSWE-Preview, it reduces total verification tokens by 62.1% while improving hybrid Best@16.

## 2 Related Work

**Test-time scaling and agent search.** Test-time compute can be spent on repeated sampling, adaptive stopping, parallel probing, search, and verifier-guided selection (Wang et al., 2023; Aggarwal et al., 2023; Li et al., 2024; Zheng et al., 2026a; Snell et al., 2025; Wu et al., 2025); on learned allocation and multi-agent pipelines (Wang et al., 2025; Zheng et al., 2026b; Motwani et al., 2025; Zhu et al., 2025); or on interaction, revision, and search, as in ReAct, Reflexion, Tree of Thoughts, and language-agent tree search (Yao et al., 2023b; Shinn et al., 2023; Yao et al., 2023a; Zhou et al., 2024). For SWE-bench, CodeMonkeys separates candidate coverage from selection and pays for selection with generated tests and a model-based selector (Ehrlich et al., 2025). We study the marginal cost of that selection when the policy already produces long trajectories.

**Software agents and patch verification.** SWE-bench evaluates generated patches against repository tests (Jimenez et al., 2024), and existing systems combine tool-using agents, search, execution

signals, and learned execution-free verifiers (Yang et al., 2024; Xia et al., 2025; Antoniades et al., 2025; Jain et al., 2025; Shum et al., 2026). SWE-Gym trains agents and verifiers on executable environments and reports Best@$K$ under verifier-guided scaling (Pan et al., 2025), the recipe inherited by the hybrid cascades we build on, whereas SWE-Search spends verifier tokens on LLM value estimates at every search node (Antoniades et al., 2025). Process supervision, code-agent reward models, and execution-free patch reasoning further strengthen candidate assessment (Han et al., 2026; Dihan & Khan, 2026; Xu et al., 2025); we target the token cost of applying such assessment at inference time.

**Verifiers and process rewards.** Outcome verifiers and process supervision were developed for mathematical reasoning (Cobbe et al., 2021; Lightman et al., 2024; Uesato et al., 2022; Wang et al., 2024; Ma et al., 2023) and extended to Q-value ranking, practical process-reward design, and stepwise promise or progress for agents (Li & Li, 2025; Choudhury, 2025; Zhang et al., 2025b; Xi et al., 2026). Preference optimization offers a related route to comparative judgments (Rafailov et al., 2023), and RewardBench, reward-overoptimization scaling laws, and LLM-as-a-judge studies document the robustness limits of learned evaluators (Lambert et al., 2025; Gao et al., 2023; Zheng et al., 2023). Our verifier pairs a non-parametric distance judgment with a lightweight learned score, both computed from cached policy states.

**Signals in internal representations.** Activations encode truth, confidence, hallucination, and reasoning correctness (Azaria & Mitchell, 2023; Burns et al., 2023; Kadavath et al., 2022; Orgad et al., 2025; Li et al., 2023; Marks & Tegmark, 2024; Kossen et al., 2024; Zhang et al., 2025a; Ghasemabadi & Niu, 2025), and distance to expert representations can steer visuomotor policies toward familiar behavior (Sun & Song, 2025). We turn such evidence into candidate selection for long-horizon language agents: success and failure banks built during policy training supply outcome-sensitive experience, the three-channel decomposition preserves the structure of agent interaction, and online state collection avoids replaying a trajectory through another language model.

# 3 METHOD

## 3.1 PROBLEM SETUP

For each task instance $x_m$, a fixed policy generates a pool of $K$ candidate trajectories, $\mathcal{T}_m = \{\tau_{m,i}\}_{i=1}^{K}$. Each trajectory ends in a patch, with $y_{m,i} \in \{0, 1\}$ indicating whether it resolves the task under the evaluation protocol. A hybrid selection procedure $\mathcal{S}$ returns a candidate index $i^\star_m = \mathcal{S}(x_m, \mathcal{T}_m)$. As illustrated in Figure 1, this procedure combines LLM-based execution-free (EF) verification with execution-based (EB) verification using regression and LLM-generated tests. The stages filter candidates using verifier scores and test results, then select a final patch. The selection procedure has no access to the outcome labels or the held-out tests used for final evaluation. Across $M$ task instances, we measure the selected resolution rate $\frac{1}{M}\sum_m y_{m,i^\star_m}$ and compare it with the oracle rate $\frac{1}{M}\sum_m \max_i y_{m,i}$ of the same candidate pools. Our focus is to reduce the additional language-model tokens consumed by hybrid verification while maintaining the selected resolution rate. We denote this cost by $V(\mathcal{S}) = N_{\text{in}} + N_{\text{out}}$, counting input and output tokens used by LLM-based EF verification and test generation, and excluding candidate generation.

## 3.2 OVERVIEW

Figure 2 illustrates LatentSift, which serves as the first-stage candidate filter in the hybrid verification workflow shown in Figure 1. It replaces the initial LLM-based EF verifier and retains half the candidates for the subsequent regression tests, LLM-generated tests, and final LLM-based EF verification. During policy training, we collect reasoning, observation, and function-call states from trajectories with known outcomes and store them together in positive and negative banks. During inference, we collect candidate states as the policy generates each trajectory. The distance branch compares these states with successful and unsuccessful experience in the banks, aggregates the evidence over steps, and combines the three channel ranks into a distance score. A trained linear scorer provides a learned trajectory score for each candidate. We rank the learned scores within the task instance and average the normalized ranks with the distance scores to obtain the final filtering scores. The filter retains half the candidates without additional LLM tokens or test execution. The

subsequent EB stages and final LLM verifier operate on the retained pool. The following sections describe state extraction, bank construction, distance scoring, and fusion with the learned signal.

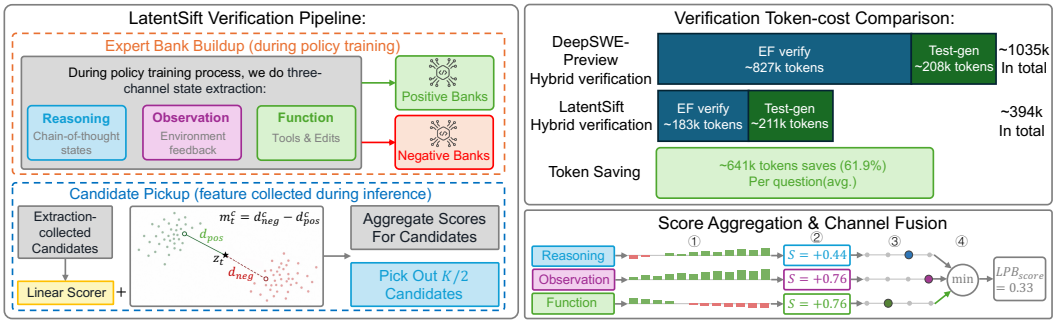


Figure 2: **LatentSift verification design. Left:** Reasoning, observation, and function states from outcome-labeled training trajectories form positive and negative banks. Inference reuses candidate states from ordinary policy forward passes, and Stage 1 averages the distance score with the linear scorer's within-instance rank to retain the top half of candidates. **Lower right:** Distance scoring: (1) retrieve the nearest positive and negative states and compute $m_t^c = d_{\text{neg}}^c - d_{\text{pos}}^c$. (2) aggregate by step-weighted mean; (3) rank within each task instance and channel; (4) take the minimum channel rank. **Upper right:** Average verification tokens per task instance for the original and token-free cascades at $K = 16$ (Section 4.2.1).

### 3.3 Three-channel representations and bank construction

We represent each trajectory through three types of spans: **reasoning**, the policy's analysis outside function calls; **observation**, environment feedback such as retrieved code, command output, and test results; and **function**, tool names and arguments, including commands and code edits. We identify these spans using message roles and function-call boundaries. For trajectory $\tau_i$, let $h_{i,p}^{(L)} \in \mathbb{R}^d$ denote the final-layer state at token position $p$. At step $t$, we mean-pool the states assigned to channel $c$:

$$z_{i,t}^c = \frac{1}{|\mathcal{S}_{i,t}^c|} \sum_{p \in \mathcal{S}_{i,t}^c} h_{i,p}^{(L)}, \qquad c \in \{\text{cot}, \text{obs}, \text{fn}\}, \tag{1}$$

where $\mathcal{S}_{i,t}^c$ contains the token positions of that channel. The task description and structural delimiters are excluded from pooling.

During policy training, we collect these representations from trajectories with executable outcome labels. Trajectories that resolve their tasks form the positive bank $\mathcal{B}_{\text{pos}}$, and unsuccessful trajectories form the negative bank $\mathcal{B}_{\text{neg}}$. Each bank entry stores the reasoning, observation, and function state sequences of one trajectory together. The banks are built from training repositories that do not overlap with the evaluation repositories.

During inference, we collect the same three types of representations from candidate trajectories as they are generated. Training representations come from the policy being trained, whereas candidate representations come from the policy used for inference, and both use final-layer states. These candidate states are then compared with the corresponding channel states in the positive and negative banks, as described in the next section.

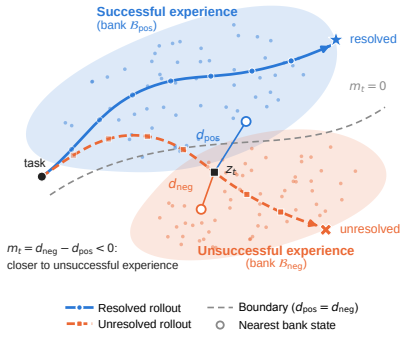


Figure 3: **Contrastive distance** of a candidate state to the successful and unsuccessful banks, with margin $m_t = d_{\text{neg}} - d_{\text{pos}}$.

### 3.4 Distance scoring and latent filtering

For a candidate state $z_{i,t}^c$ we retrieve its nearest positive and negative bank states in the same channel (Figure 3), writing $\mathcal{Z}^c(\mathcal{B})$ for the channel-$c$ states of bank $\mathcal{B}$: $d_{\text{pos}}^c(z) = \min_{u \in \mathcal{Z}^c(\mathcal{B}_{\text{pos}})} \|z - u\|_2$ and $d_{\text{neg}}^c(z) = \min_{u \in \mathcal{Z}^c(\mathcal{B}_{\text{neg}})} \|z - u\|_2$. The signed margin is aggregated over steps with position

weights,

$$m_{i,t}^c = d_{\text{neg}}^c(z_{i,t}^c) - d_{\text{pos}}^c(z_{i,t}^c), \qquad q_i^c = \frac{\sum_{t=0}^{T_i-1} t\, m_{i,t}^c}{\sum_{t=0}^{T_i-1} t}, \tag{2}$$

so that later steps, taken after environment feedback, count more. The positive value indicates proximity to successful rather than unsuccessful experience. Within each task instance and channel, we rank candidates by $q_i^c$ in ascending order, average the ranks of tied scores, and scale the zero-based ranks to $[0, 1]$. Higher scores receive higher ranks $r_i^c$ (Appendix A.1). We take $s_{\text{dist}}(\tau_i) = \min_c r_i^c$, so a candidate scores highly only if every channel ranks it highly. Appendix A.1 gives the rationale and cost accounting, and Section 4.3 ablates channel subsets, one-sided distances, and naive filtering rules including uniform step weights and trajectory length.

### 3.5 Linear scorer and signal fusion

A lightweight linear scorer inspired by SWIFT (Guo et al., 2026) complements the distance judgment. Trained on the same outcome-labeled trajectories, it reads pooled reasoning and function states from all transformer layers, excluding observations. A shared linear head maps each span to a gate logit and a local reward, and their gated average is the trajectory score, trained with binary cross-entropy against the outcome (Appendix A.3). In inference, it reads only cached candidate states. We rank the learned trajectory scores within each task instance using the normalization in Appendix A.1, and denote the resulting score for candidate $\tau_i$ by $s_{\text{lin}}(\tau_i) \in [0, 1]$. We combine the two branch scores with equal weights,

$$s(\tau_i) = 0.5\, s_{\text{dist}}(\tau_i) + 0.5\, s_{\text{lin}}(\tau_i), \tag{3}$$

and the $\max(3, \lfloor K/2 \rfloor)$ highest-scoring candidates, the top half at $K = 16$, proceed to the execution-based stages and the final EF verifier.

## 4 Experiments

### 4.1 Experimental setup

**Task.** We evaluate on the 500 human-validated GitHub issues in SWE-bench Verified (Jimenez et al., 2024). For each task instance, an agent generates multiple trajectories, each ending in a candidate repository patch. The verification workflow evaluates these candidates and selects one patch. We determine whether the selected patch resolves the instance using the official evaluation. All verification methods use the same candidate pool within each agent-policy configuration, so differences in performance reflect candidate verification rather than generation.

**Agents, policies, and candidate pools.** We evaluate LatentSift on multiple agents, including DeepSWE Agent, R2EGym Agent (Jain et al., 2025), and CWM Agent (FAIR CodeGen Team et al., 2025). DeepSWE Agent denotes the official DeepSWE-Preview (Luo et al., 2025) agentic scaffold, comprising its prompt template, tool protocol, and agent configuration, and R2EGym Agent denotes the official R2E-Gym scaffold.[1] Both are implemented in the R2E-Gym framework and share its editing interface, but they are distinct agent configurations. The verification workflows of DeepSWE and R2EGym are similar, both composed of the four stages shown in Figure 1. For CWM, since its test-time scaling verifier is not open-sourced, we reuse the verification workflow of DeepSWE Agent.

We also conduct evaluation on various policy models. DeepSWE-Preview is RL-trained from Qwen3-32B (Luo et al., 2025), R2EGym-32B is fine-tuned from Qwen2.5-Coder-32B-Instruct (Jain et al., 2025), and CWM is a 32B open-weights model (FAIR CodeGen Team et al., 2025). We compare the performance on the same agent across different policy models by evaluating on R2EGym Agent with R2EGym-32B and R2EGym-14B (Jain et al., 2025) models. Detailed checkpoint identities and implementation choices are provided in Appendix A.2.

For each policy–agent configuration, we generate 16 trajectories for each of the 500 task instances, yielding 8,000 trajectories per configuration. Verifier comparisons use the same pool. Changing

[1] Official DeepSWE reproduction guide.

the agent scaffold requires generating new trajectories and running the tests on their patches again. Following prior hybrid workflows (Luo et al., 2025; Jain et al., 2025), the R2E-TestgenAgent (R2E-Gym, 2025a) was employed to generate one set of tests per task instance. For each agent, we build positive and negative banks from trajectories and their three-channel hidden states collected during generator model training, containing 1,383 successful and 1,383 unsuccessful trajectories from repositories disjoint from the evaluation repositories. For each $K$, we report mean Best@$K$ over 200 draws of $K$ candidates from each fixed pool, using stored test results.

**Verifier baselines.** We compare our LatentSift with multiple EF verifiers, including DeepSWE-Verifier (Agentica, 2025), R2EGym-Verifier (R2E-Gym, 2025b), DEV matching-pairs (R2E-Edits, 2025) and AgentPRM (Xi et al., 2026). Each baseline uses its own verifier as the EF verifier in the agent's verification workflow (i.e., Stages 1 and 4 in Figure 1). Ours uses LatentSift in Stage 1 and the policy-associated EF verifier in Stage 4. Each method retains $\max(3, \lfloor K/2 \rfloor)$ candidates, the top half at $K = 16$.

**Metrics.** Our primary quality metric is Best@$K$, the percentage of task instances solved by the candidate selected from a pool of size $K$; hybrid Best@$K$ is this rate after the complete four-stage cascade. Oracle Pass@$K$ reports the corresponding candidate-generation ceiling by assuming that the best candidate will always be selected. We measure verification token cost by reporting EF-verifier tokens and total verification tokens including test generation. EF-verifier token counts use the calls made by each workflow, reusing Stage 1 scores in Stage 4 when available. We charge the logged test-generation cost to a workflow only if it reaches Stage 3.

### 4.2 Main results

We evaluate hybrid verification across agent configurations and policy sizes. The cross-agent experiment compares four baseline EF verifiers with LatentSift, each baseline serving in both EF stages and ours paired with the policy-associated final verifier. The cross-policy experiment compares each policy's own verifier with LatentSift under a fixed scaffold.

#### 4.2.1 Cross-agent evaluation

We compare DeepSWE Agent (DeepSWE-Preview), R2EGym Agent (R2EGym-32B), and CWM Agent (CWM). CWM ships no verifier, so we retrained four verifiers on Llama-3.1-8B-Instruct with the baselines' original recipes.

Table 1: Cross-agent hybrid results on SWE-bench Verified at $K = 16$. Best@16 is the resolution rate (%). Token cost gives EF-verifier (EF) and total verification tokens per task instance, in thousands.

| Stage 1 verifier | DeepSWE Agent | | | R2EGym Agent | | | CWM Agent | | |
|---|---|---|---|---|---|---|---|---|---|
| | Best@16 | Token cost (K) | | Best@16 | Token cost (K) | | Best@16 | Token cost (K) | |
| | | EF | Total | | EF | Total | | EF | Total |
| DeepSWE-Verifier | 59.26 | 826.7 | 1035.1 | 49.70 | 367.4 | 599.4 | 54.21 | 692.6 | 930.0 |
| R2EGym-Verifier | 57.95 | 774.1 | 982.0 | 46.56 | 366.2 | 599.6 | 53.51 | 707.1 | 944.6 |
| DEV matching-pairs | 58.75 | 826.7 | 1037.6 | 48.38 | 367.4 | 599.4 | 53.11 | 692.6 | 925.6 |
| AgentPRM | 59.76 | 810.9 | 1022.8 | 46.56 | 358.3 | 586.8 | 53.31 | 677.3 | 909.8 |
| **Ours** | **60.06** | **181.0** | **391.9** | **47.73** | **76.0** | **304.9** | **54.21** | **231.3** | **466.8** |

Table 1 reports hybrid Best@16 and verification cost for the five Stage-1 verifiers under the protocol of Section 4.1. For DeepSWE, no baseline verifier exceeds LatentSift (60.06%), and on CWM Agent LatentSift matches the strongest baseline (54.21%) with a third of its EF tokens (231.3K against 692.6K). For the R2EGym-32B pool, LatentSift raises Best@16 by 1.17 points over the original R2EGym-Verifier cascade while cutting EF tokens by 4.8× and total verification cost by half. Additional Best@$K$ results for $K = 4, 8, 12$ appear in Appendix A.5.

Figure 4 (top) shows that LatentSift tracks the Best@$K$ of the existing hybrid at every candidate budget on all three agents while its verification cost grows more slowly, so the token saving widens with $K$. At $K = 16$ it improves Best@16 over the existing hybrid on DeepSWE-Preview and R2EGym-32B and matches it on CWM Agent, while reducing total verification tokens by 62.1%, 49.1%, and 49.8%, respectively. The bottom row plots the same runs as Best@$K$ against verification

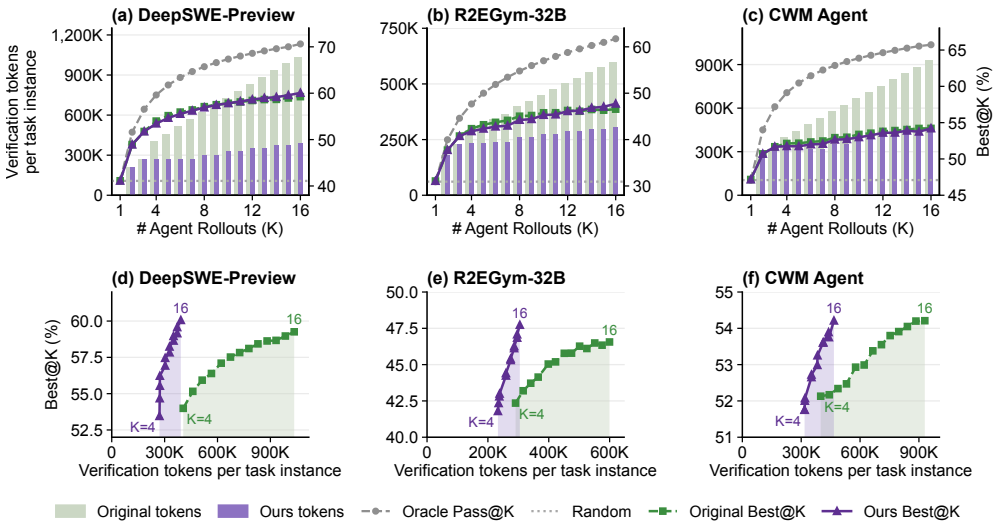


Figure 4: **Verification cost and resolution rate on the three agents.** Top: verification tokens per task instance, including test generation (bars, left axis), and Best@$K$ with Oracle Pass@$K$ and random selection (curves, right axis) as $K$ grows. Bottom: the same runs as Best@$K$ against verification tokens for $K$ from 4 to 16. The baseline hybrid uses the policy-associated verifier in Stages 1 and 4, and ours replaces it in Stage 1 with LatentSift.

tokens: at a matched verification cost, LatentSift reaches a higher Best@$K$ than the original cascade on all three agents.

### 4.2.2 Cross-policy evaluation

We hold R2EGym Agent fixed and compare R2EGym-32B with R2EGym-14B to conduct evaluation across policy sizes. Each policy supplies its own candidate pool, bank, and linear scorer. Both use R2EGym-Verifier in the original cascade's Stages 1 and 4 and in our Stage 4. Figure 5 also includes DeepSWE-Preview with DeepSWE Agent as a reference from the cross-agent comparison in Section 4.2.1.

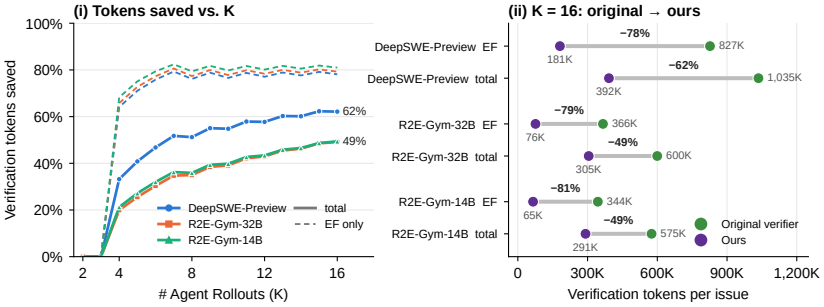


Figure 5: **Verification tokens saved across policy–agent configurations.** The cross-policy comparison holds R2EGym Agent fixed and varies the policy between R2EGym-32B and R2EGym-14B. (i) Rate of verification tokens saved as $K$ grows (solid: total including test generation; dashed: EF-verifier tokens only). (ii) Tokens per task instance at $K = 16$ with the policy-associated verifier and with LatentSift in Stage 1.

Figure 5 shows that LatentSift substantially reduces verification tokens on both R2EGym policies while maintaining hybrid resolution rates. At $K = 16$, EF-verifier tokens fall from 366K to 76K per task instance on R2EGym-32B and from 344K to 65K on R2EGym-14B. Including test generation, total verification cost falls by about 49% on each policy, from 600K to 305K and from 575K to 291K, respectively (Appendix A.4). Hybrid Best@16 rises from 46.56% to 47.73% on 32B and remains within 0.33 percentage points of the original cascade on 14B (Table 7, Appendix A.5). The EF-token saving reaches about 80% by $K \approx 6$ on both policies and persists as the candidate budget grows.

### 4.2.3 SCORE DISTRIBUTIONS

To show where the discrimination of LatentSift comes from, we randomly sample 100 of the 500 task instances and plot the scores of their DeepSWE-Preview candidates in Figure 6. In every channel, resolved candidates concentrate at positive distance margins (Equation 2), the successful side of the decision boundary in Figure 3, whereas unresolved candidates shift to negative margins with a long tail. The final score $s$ (Equation 3, panel d) separates the two groups further, so keeping the top half of each pool discards mainly unresolved candidates.

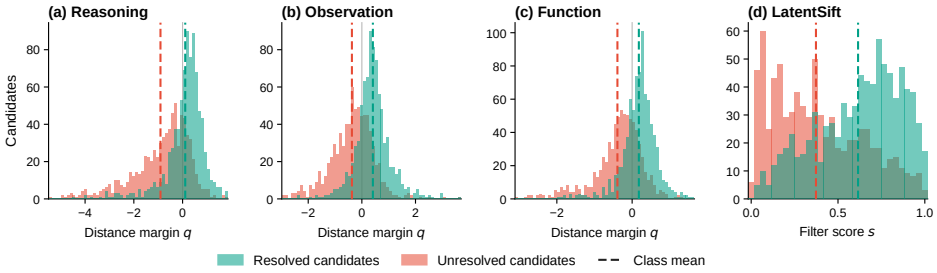


Figure 6: **Score distributions of resolved and unresolved DeepSWE-Preview candidates** on 100 randomly sampled task instances. (a)–(c) Step-weighted distance margin $q^c$ in the reasoning, observation, and function channels. (d) Final LatentSift score $s$. Dashed lines mark class means.

## 4.3 ABLATION STUDY

We first isolate the Stage 1 filter through its oracle retention (Section 4.3.1), then ablate three design choices of the distance branch at $K = 16$, with the policy, scaffold, banks, execution signals, linear scorer, and Stage 4 verifier fixed within each configuration: channel selection, distance construction, and naive filtering rules. Channel subsets take the minimum rank over the selected channels and are fused with the linear scorer as in the full method.

### 4.3.1 STAGE-1 RETENTION

Best@16 credits the Stage 1 filter together with three downstream stages. To isolate the filter we measure its *oracle retention*: among task instances whose $K$ candidates contain a correct patch, the fraction whose retained $\max(3, \lfloor K/2 \rfloor)$ candidates still contain one. Table 2 reports it on DeepSWE-Preview at even $K$ for the LLM EF verifier, LatentSift, and random filtering. At $K = 16$ LatentSift retains a correct candidate on 97.72% of solvable task instances, against 97.44% for the LLM verifier and 93.19% for random filtering. For $K \leq 8$ the LLM verifier leads by less than one point. The difference narrows as $K$ grows, falls to 0.07 point at $K = 12$, and favors LatentSift from $K = 14$ on. These results show that LatentSift retains correct candidates at a rate comparable to the LLM EF verifier, without additional LLM tokens.

Table 2: Oracle retention (%) after Stage 1 on DeepSWE-Preview at even $K$, retaining $\max(3, K/2)$ candidates, over the same 200 draws as Figure 4.

| Stage-1 filter | $K$=4 | $K$=6 | $K$=8 | $K$=10 | $K$=12 | $K$=14 | $K$=16 |
|---|---|---|---|---|---|---|---|
| LLM EF verifier | 97.67 | 94.67 | 95.79 | 96.37 | 96.81 | 97.13 | 97.44 |
| **LatentSift (ours)** | 97.07 | 93.77 | 94.95 | 95.97 | 96.74 | 97.28 | 97.72 |
| Random filtering | 92.31 | 85.91 | 88.27 | 90.03 | 91.49 | 92.50 | 93.19 |

### 4.3.2 CHANNEL SELECTION

Table 3: Channel selection ablation of LatentSift at $K = 16$: hybrid Best@16 (%) with the distance branch restricted to subsets of the reasoning (R), observation (O), and function (F) channels.

| Distance channels | **R+O+F** | R+O | R+F | O+F | R | O | F |
|---|---|---|---|---|---|---|---|
| DeepSWE Agent | **60.06** | 59.26 | 59.66 | 59.86 | 59.46 | 59.26 | 59.46 |
| R2EGym Agent | **47.73** | 46.60 | 46.90 | 47.10 | 46.71 | 46.46 | 47.25 |

Restricting the distance branch to single channels or pairs, with the learned signal and downstream stages fixed, lowers Best@16 in every case (Table 3). On DeepSWE Agent the best pair, observation + function, comes closest (59.86% against 60.06%) and single channels lose more; on R2EGym

Agent the function channel alone is the strongest subset (47.25% against 47.73%), ahead of every pair, so the relative value of the channels differs by agent. The three channels therefore carry complementary information.

#### 4.3.3 DISTANCE CONSTRUCTION

We evaluate the benefit of using distances to both successful and unsuccessful experience. Our two-sided distance score, $m^c_{i,t} = d^c_{\text{neg}} - d^c_{\text{pos}}$ (Equation 2), combines proximity to successful trajectories with separation from unsuccessful trajectories. We compare it with positive-only scoring using $d^c_{\text{pos}}$ (smaller is better) and negative-only scoring using $d^c_{\text{neg}}$ (larger is better), keeping all other components fixed.

Table 4: Distance construction ablation at $K = 16$ (%): hybrid Best@16 and oracle retention (Ret.).

| | DeepSWE Agent | | R2EGym Agent | |
|---|---|---|---|---|
| Distance score | Best@16 | Ret. | Best@16 | Ret. |
| $d^c_{\text{neg}} - d^c_{\text{pos}}$ | **60.06** | **97.72** | **47.73** | **92.13** |
| $d^c_{\text{pos}}$ | 58.05 | 97.30 | 44.16 | 89.31 |
| $d^c_{\text{neg}}$ | 59.86 | 97.44 | 47.30 | 91.48 |

Two-sided distance scoring achieves the highest hybrid Best@16 and oracle retention on both agents (Table 4). Compared with the one-sided alternatives, it improves Best@16 by 0.20–2.01 percentage points on DeepSWE Agent and 0.43–3.57 points on R2EGym Agent. These results support using both successful and unsuccessful experience for candidate filtering. We also observe an interesting pattern: negative-only scoring outperforms positive-only scoring, suggesting that unsuccessful experience may provide more informative evidence for candidate filtering.

#### 4.3.4 NAIVE FILTERING RULES

Finally, we test whether step weighting is necessary and how naive rules that bypass the three-channel representation perform. The uniform step mean $q^c_{i,\text{uniform}} = \frac{1}{T_i}\sum_t m^c_{i,t}$ drops only the step weighting and stays fused with the linear scorer. Two length rules, preferring shorter trajectories in tokens or agent steps, serve alone as the Stage-1 filter. Stages 2–4 are unchanged.

Table 5: Naive Stage-1 filtering rules against LatentSift at $K = 16$ (%): hybrid Best@16 and oracle retention (Ret.).

| | DeepSWE Agent | | R2EGym Agent | |
|---|---|---|---|---|
| Stage-1 rule | Best@16 | Ret. | Best@16 | Ret. |
| **LatentSift (step-weighted mean)** | **60.06** | **97.72** | **47.73** | **92.13** |
| Uniform step mean | 59.66 | 97.44 | 46.36 | 90.16 |
| Trajectory length, tokens | 57.24 | 94.87 | 45.65 | 91.15 |
| Trajectory length, steps | 58.39 | 95.29 | 47.60 | 92.40 |

LatentSift achieves the highest Best@16 on both agents (Table 5), outperforming the three naive rules by 0.40–2.82 points on DeepSWE Agent and by 0.13–2.08 points on R2EGym Agent. The relative performance of these naive rules varies across agents, whereas LatentSift consistently achieves the highest performance on both agents.

## 5 CONCLUSION AND FUTURE WORK

We presented LatentSift, a token-free and execution-free filter that scores candidate trajectories from the policy's own reasoning, observation, and function-call states instead of replaying them through a second language model. Across three agents at $K = 16$, LatentSift cuts EF-verifier tokens by 66.6–79.2% and total verification cost by 49.1–62.1% while raising or matching hybrid Best@16, and the savings carry over to a smaller policy and grow with the candidate budget. Policy-state filtering is therefore a drop-in replacement for the most token-consuming stage of hybrid verification.

LatentSift applies to any policy whose hidden states are accessible, including the open-weight agents studied here. Our evaluation covers candidate budgets up to $K = 16$ on the Python repositories of SWE-bench Verified. Extending it to larger budgets, other programming languages and benchmarks, and larger experience banks is a natural next step and may improve performance-efficiency tradeoffs further.

## AI USE STATEMENT

Generative AI tools were used only to edit, rephrase, and reorganize author-written text for clarity and readability, and to assist with LaTeX formatting and manuscript presentation. They were not used for research ideation, theoretical development, mathematical derivation or verification, hypothesis formulation, methodology or experimental design, implementation, code generation or modification, data generation or processing, literature analysis, or interpretation of experimental results.

All AI-assisted text was reviewed and verified by the authors. The authors take full responsibility for the accuracy, claims, results, and final content of the manuscript.

## REPRODUCIBILITY STATEMENT

Reproducibility materials will include bank construction procedures, split identifiers, label provenance, extraction code, scorer configurations, and evaluation scripts. Appendix A lists the corresponding reporting details.

## REFERENCES


Agentica. DeepSWE-Verifier. Hugging Face model repository, 2025. URL https://huggingface.co/agentica-org/DeepSWE-Verifier.

Pranjal Aggarwal, Aman Madaan, Yiming Yang, et al. Let's sample step by step: Adaptive-consistency for efficient reasoning and coding with LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12375–12396, 2023.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-Search: Enhancing software agents with Monte Carlo tree search and iterative refinement. In *International Conference on Learning Representations*, volume 2025, pp. 64485–64515, 2025.

Amos Azaria and Tom Mitchell. The internal state of an LLM knows when it's lying. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 967–976, 2023.

Collin Burns, Haotian Ye, Dan Klein, and Jacob Steinhardt. Discovering latent knowledge in language models without supervision. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ETKGuby0hcs.

Sanjiban Choudhury. Process reward models for LLM agents: Practical framework and directions. *arXiv preprint arXiv:2502.10325*, 2025.

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

Mahir Labib Dihan and Md Ashrafur Rahman Khan. SWE-Shepherd: Advancing PRMs for reinforcing code agents. *arXiv preprint arXiv:2604.10493*, 2026.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. CodeMonkeys: Scaling test-time compute for software engineering. *arXiv preprint arXiv:2501.14723*, 2025.

FAIR CodeGen Team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, et al. CWM: An open-weights LLM for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10835–10866. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/gao23h.html.

Amirhosein Ghasemabadi and Di Niu. Can LLMs predict their own failures? self-awareness via internal circuits. *arXiv preprint arXiv:2512.20578*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

Jizhou Guo, Zhaomin Wu, Hanchen Yang, and Philip S. Yu. Mining intrinsic rewards from LLM hidden states for efficient best-of-N sampling. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1*, pp. 371–382. ACM, 2026. doi: 10.1145/3770854.3780302.

Hao Han, Jin Xie, Xuehao Ma, Weiquan Zhu, Ziyao Zhang, ZhiLiang Long, Hongkai Chen, and Qingwen Ye. SWE-TRACE: Optimizing long-horizon SWE agents through rubric process reward models and heuristic test-time scaling. *arXiv preprint arXiv:2604.14820*, 2026.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Second Conference on Language Modeling*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, et al. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221*, 2022.

Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. The cost of dynamic reasoning: Demystifying AI agents and test-time scaling from an AI infrastructure perspective. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–16. IEEE, 2026.

Jannik Kossen, Jiatong Han, Muhammed Razzak, Lisa Schut, Shreshth Malik, and Yarin Gal. Semantic entropy probes: Robust and cheap hallucination detection in LLMs. *arXiv preprint arXiv:2406.15927*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, et al. RewardBench: Evaluating reward models for language modeling. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1755–1797, 2025.

Kenneth Li, Oam Patel, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. Inference-time intervention: Eliciting truthful answers from a language model. *Advances in Neural Information Processing Systems*, 36:41451–41530, 2023.

Wendi Li and Yixuan Li. Process reward model with Q-value rankings. In *International Conference on Learning Representations*, volume 2025, pp. 14708–14726, 2025.

Yiwei Li, Peiwen Yuan, Shaoxiong Feng, Boyuan Pan, Xinglin Wang, Bin Sun, Heda Wang, and Kan Li. Escape sky-high cost: Early-stopping self-consistency for multi-step reasoning. In *International Conference on Learning Representations*, volume 2024, pp. 14751–14768, 2024.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. DeepSWE: Training a fully open-sourced, state-of-the-art coding agent by scaling RL. Notion blog post, 2025. URL https://pretty-radio-b75.notion.site/DeepSWE-Training-a-Fully-Open-sourced-State-of-the-Art-Coding-Agent-by-Scaling-RL-22281902c1468193aabbe9a8c59bbe33.

Qianli Ma, Haotian Zhou, Tingkai Liu, Jianbo Yuan, Pengfei Liu, Yang You, and Hongxia Yang. Let's reward step by step: Step-level reward model as the navigators for reasoning. *arXiv preprint arXiv:2310.10080*, 2023.

Samuel Marks and Max Tegmark. The geometry of truth: Emergent linear structure in large language model representations of True/False datasets. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=aajyHYjjsk.

Sumeet Ramesh Motwani, Chandler Smith, Rocktim Jyoti Das, Rafael Rafailov, Philip Torr, Ivan Laptev, Fabio Pizzati, Ronald Clark, and Christian Schroeder de Witt. MALT: Improving reasoning with multi-agent LLM training. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=jXP9bgFack.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori B Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20286–20332, 2025.

Hadas Orgad, Michael Toker, Zorik Gekhman, Roi Reichart, Idan Szpektor, Hadas Kotek, and Yonatan Belinkov. LLMs know more than they show: On the intrinsic representation of LLM hallucinations. In *International Conference on Learning Representations*, volume 2025, pp. 66880–66913, 2025.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

R2E-Edits. DeepSWE Verifier (matching-pairs-v1). Hugging Face model repository, 2025. URL https://huggingface.co/r2e-edits/deepswe-swebv-eval-n16-verifier-qwen3-lora64-matching-pairs-v1.

R2E-Gym. R2E-TestgenAgent. Hugging Face model repository, 2025a. URL https://huggingface.co/R2E-Gym/R2E-TestgenAgent.

R2E-Gym. R2EGym-Verifier. Hugging Face model repository, 2025b. URL https://huggingface.co/R2E-Gym/R2EGym-Verifier.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, X. W., Jiaxi Yang, Yuzhen Huang, Junyang Lin, and Junxian He. SWE-RM: Execution-free feedback for software engineering agents. In *International Conference on Learning Representations*, volume 2026, pp. 143426–143446, 2026.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations*, volume 2025, pp. 10131–10165, 2025.

Zhanyi Sun and Shuran Song. Latent policy barrier: Learning robust visuomotor policies by staying in-distribution. *Advances in Neural Information Processing Systems*, 38:174280–174305, 2025.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

Fali Wang, Hui Liu, Zhenwei Dai, Jingying Zeng, Zhiwei Zhang, Zongyu Wu, Chen Luo, Zhen Li, Xianfeng Tang, Qi He, et al. AgentTTS: Large language model agent for test-time compute-optimal scaling strategy in complex tasks. *Advances in Neural Information Processing Systems*, 38:98396–98433, 2025.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for LLM problem-solving. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 55824–55845, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/8c3caae2f725c8e2a55ecd600563d172-Paper-Conference.pdf.

Zhiheng Xi, Chenyang Liao, Guanyu Li, Zhihao Zhang, Wenxiang Chen, Binghai Wang, Senjie Jin, Yuhao Zhou, Jian Guan, Wei Wu, et al. AgentPRM: Process reward models for LLM agents via step-wise promise and progress. In *Proceedings of the ACM Web Conference 2026*, pp. 4184–4195, 2026.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025.

Wenya Xie, Shaochen Zhong, Hoang Anh Duy Le, Zhaozhuo Xu, Jianwen Xie, and Zirui Liu. Word salad chopper: Reasoning models waste a ton of decoding budget on useless repetitions, self-knowingly. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33576–33586, 2025.

Junjielong Xu, Boyin Tan, Xiaoyuan Liu, Chao Peng, Pengfei Gao, and Pinjia He. Scalable supervision for software agents via patch reasoning. *arXiv preprint arXiv:2510.22775*, 2025.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36:11809–11822, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Anqi Zhang, Yulin Chen, Jane Pan, Chen Zhao, Aurojit Panda, Jinyang Li, and He He. Reasoning models know when they're right: Probing hidden states for self-verification. *arXiv preprint arXiv:2504.05419*, 2025a.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. The lessons of developing process reward models in mathematical reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 10495–10516, 2025b.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

Tong Zheng, Chengsong Huang, Runpeng Dai, Yun He, Rui Liu, Xin Ni, Huiwen Bao, Kaishen Wang, Hongtu Zhu, Jiaxin Huang, et al. Parallel-Probe: Towards efficient parallel thinking via 2D probing. *arXiv preprint arXiv:2602.03845*, 2026a.

Tong Zheng, Haolin Liu, Chengsong Huang, Huiwen Bao, Sheng Zhang, Rui Liu, Runpeng Dai, Ruibo Chen, Chenxi Liu, Tianyi Xiong, et al. LLMs improving LLMs: Agentic discovery for test-time scaling. *arXiv preprint arXiv:2605.08083*, 2026b.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/zhou24r.html.

King Zhu, Hanhao Li, Siwei Wu, Tianshun Xing, Dehua Ma, Xiangru Tang, Minghao Liu, Jian Yang, Jiaheng Liu, Yuchen Eleanor Jiang, et al. Scaling test-time compute for LLM agents. *arXiv preprint arXiv:2506.12928*, 2025.

# A Implementation and Reporting Details

## A.1 Distance aggregation, selection, and cost accounting

**Banks and retrieval.** Each bank entry stores the three channel sequences of one trajectory, so channel-$c$ retrieval searches the channel-$c$ states of all bank trajectories and no separate per-channel bank is built. The linear scorer is trained on the same trajectories (Appendix A.3). Neither signal uses evaluation outcomes.

**Contrastive distance.** The step margin $m^c_{i,t}$ of Equation 2 is positive when the nearest successful bank state is closer to the candidate state than the nearest unsuccessful one. A one-sided distance measures proximity to a single outcome, while the margin weighs the two outcomes against each other. Section 4.3.3 compares the three constructions.

**Temporal and channel aggregation.** The weight of step $t$ grows linearly with $t$, so evidence from later steps, taken after the agent has received environment feedback and revised its solution, counts more. Section 4.3.4 evaluates the uniform step mean. The three channel scores differ in scale and distribution, so summing them directly would let the channel with the largest magnitude dominate. Within-instance ranks place the channels on a common scale without requiring calibrated success probabilities. Taking the minimum rank sets the distance score to the rank of the weakest channel, so ranks of $0.67$, $1.00$, and $0.33$ give $s_{\mathrm{dist}} = 0.33$. No separate rejection threshold is needed, and the learned score enters afterwards through Equation 3.

**Within-instance rank normalization.** For each task instance and channel $c$, let $n$ be the number of candidates being ranked. We sort their scores $q^c_i$ in ascending order and assign zero-based ranks $0, \ldots, n-1$, giving tied candidates the mean of the ranks they occupy. Writing this rank as $\rho^c_i$, we compute

$$r^c_i = \frac{\rho^c_i}{\max(n-1, 1)}, \qquad s_{\mathrm{dist}}(\tau_i) = \min_c r^c_i. \tag{4}$$

Ranks are never pooled across task instances. Without ties and for $n > 1$, the lowest and highest scores map to 0 and 1. If all $n > 1$ scores are equal, every candidate receives rank $0.5$, and a single candidate receives rank 0. For example, scores $(1, 1, 2, 2)$ yield normalized ranks $(1/6, 1/6, 5/6, 5/6)$. Averaging tied ranks makes the result independent of the order of equal-scoring candidates.

**Candidate selection.** In the hybrid cascade, LatentSift retains the $\max(3, \lfloor K/2 \rfloor)$ candidates with the highest fused scores $s(\tau_i)$. The execution-based stages prune this set further, and the final EF verifier scores only the candidates that survive them (Appendix A.4).

**Cost accounting.** Token-free means that the filter adds no LLM input or output tokens, not that it requires no computation. At inference, the filter mean-pools states that the policy's forward passes already produce, retrieves nearest neighbors from the two banks in each channel, and applies one linear head. Bank construction and scorer training are performed offline. A text verifier that rereads $K$ trajectories processes roughly $K$ times the average trajectory length in input tokens before prompt overhead, so the verification tokens of the original cascade grow linearly with $K$ (Figure 4). We count EF-verifier calls and test-generation calls separately, include only calls that are actually made, and exclude candidate generation. In the original cascade, Stage 4 reuses the Stage-1 EF scores and makes no new verifier calls (Appendix A.2).

## A.2 EF verifier configurations

DeepSWE-Verifier and R2EGym-Verifier are the official verifiers of DeepSWE-Preview and R2E-Gym. They are LoRA adapters on Qwen3-14B and Qwen2.5-Coder-14B-Instruct, respectively. DEV matching-pairs is another Qwen3-14B LoRA adapter, trained on matched pairs of one successful and one unsuccessful rollout of the same task. All three adapters use rank 64 and $\alpha = 128$ (Agentica, 2025; R2E-Gym, 2025b; R2E-Edits, 2025). The DeepSWE-Verifier and DEV matching-pairs model cards report a learning rate of $10^{-5}$, two training epochs, and cosine learning-rate decay with

a warmup ratio of 0.05. AgentPRM (Xi et al., 2026) was proposed for web agents such as web shopping and browser navigation. We port its official implementation to coding-agent trajectories and train it as a LoRA adapter with the same base model and configuration as DeepSWE-Verifier. CWM releases no verifier, so for CWM Agent we retrain the four baseline verifiers on Llama-3.1-8B-Instruct with their original recipes.

In the official hybrid workflow, the final stage ranks the candidates that pass the regression and generated tests by their stored Stage-1 EF scores and calls no separate verifier (Luo et al., 2025; Jain et al., 2025). Each EF baseline therefore uses its own checkpoint in Stages 1 and 4. LatentSift replaces Stage 1 and calls the policy-associated verifier only in Stage 4, namely DeepSWE-Verifier for DeepSWE-Preview, R2EGym-Verifier for both R2EGym policies, and the retrained DeepSWE-Verifier for CWM, whose workflow follows DeepSWE Agent.

### A.3 Learned scorer architecture and training

The learned scorer reads cached, mean-pooled hidden states rather than trajectory text. At every step it keeps the model-authored reasoning and function-call spans and excludes environment observations, following the assistant-only scoring scope of SWIFT (Guo et al., 2026). The spans are ordered as they occur in the trajectory. Let $u_{i,j}^{(\ell)} \in \mathbb{R}^d$ denote the mean-pooled state of span $j$ at layer $\ell$. We concatenate the states of all $L$ transformer layers without normalization:

$$\tilde{u}_{i,j} = [u_{i,j}^{(1)}; u_{i,j}^{(2)}; \ldots; u_{i,j}^{(L)}] \in \mathbb{R}^{Ld}. \tag{5}$$

A shared linear head maps each span to a gate logit and a local reward,

$$(\tilde{g}_{i,j}, v_{i,j}) = W\tilde{u}_{i,j} + b, \qquad W \in \mathbb{R}^{2\times Ld}, \quad b \in \mathbb{R}^2, \tag{6}$$

and the trajectory logit is their gated average:

$$q_{i,\mathrm{lin}} = \frac{\sum_j \sigma(\tilde{g}_{i,j}) v_{i,j}}{\max\left(\sum_j \sigma(\tilde{g}_{i,j}), 10^{-8}\right)}. \tag{7}$$

The head has $2Ld + 2$ trainable parameters. We apply $W$ as one bias-free $d$-to-2 projection per layer, sum the outputs, and add the shared bias, which computes Equation 6 without forming the concatenated vector.

The training target of bank trajectory $i$ is its outcome label $y_i = \mathbf{1}[R_i \geq 0.5]$, where $R_i$ is its rollout reward, and we minimize binary cross-entropy with $q_{i,\mathrm{lin}}$ as the logit. The bank is split into 80% training and 20% validation trajectories with random seed 0. We optimize with AdamW using a learning rate of $3 \times 10^{-6}$, weight decay of $10^{-5}$, and gradient accumulation over 16 trajectories for at most 40 epochs, and keep the checkpoint with the lowest validation loss. Each candidate trajectory is scored once with this checkpoint. Its logit is then converted to a normalized rank within the candidate set of its task instance (Appendix A.1), which gives the learned score $s_{\mathrm{lin}}(\tau_i)$ of Equation 3.

### A.4 Token accounting details

EF-verifier tokens sum the input and output tokens of the verifier calls actually made rather than estimating them from a global mean length. An average EF call uses 22.9K tokens over the full R2EGym-32B candidate pool and 18.9K tokens over the candidates that reach the final verifier. The corresponding DeepSWE-Preview averages are 51.7K and 46.2K tokens. The original cascade scores all 16 candidates, whereas LatentSift calls the verifier only for the candidates that survive the execution stages, about four per task instance ($76.0\mathrm{K}/18.9\mathrm{K} \approx 4.0$ on R2EGym-32B and $181.0\mathrm{K}/46.2\mathrm{K} \approx 3.9$ on DeepSWE-Preview, Table 1). Most of the saving therefore comes from making fewer EF calls, and a smaller part comes from the shorter average length of the surviving trajectories.

We generate Stage 3 tests once for each of the 500 task instances with R2E-TestgenAgent (R2E-Gym, 2025a), the open-source test generator of the DeepSWE-Preview recipe, and run the same tests on the candidate patches of every configuration. A workflow is charged the test-generation cost only when it reaches Stage 3, at an average of 244.3K tokens per invocation according to the

generation logs. On R2EGym-32B, Stage 3 is reached for 95.5% of task instances in the original cascade and for 93.7% in ours. The totals are therefore $366.2\text{K} + 0.955 \times 244.3\text{K} \approx 600\text{K}$ and $76.0\text{K} + 0.937 \times 244.3\text{K} \approx 305\text{K}$ tokens per task instance, the 49.1% saving in Table 1 and Figure 5. The other configurations follow the same calculation with their own Stage 3 rates, for example 575.3K and 291.2K tokens on R2EGym-14B (Table 7).

### A.5 Additional results

**Smaller candidate budgets.** Table 6 extends the cross-agent comparison of Table 1 to $K = 4$, 8, and 12 under the same protocol. The original cascade of each agent uses the policy-associated verifier in both EF stages. At every budget, LatentSift stays within 0.81 points of this cascade, and at $K = 16$ it matches or exceeds it on all three agents, while its token saving widens with $K$ (Figure 4).

Table 6: Hybrid Best@$K$ (%) of the cross-agent comparison at $K = 4$, 8, 12, and 16. The $K = 16$ column repeats Table 1.

| | DeepSWE Agent | | | | R2EGym Agent | | | | CWM Agent | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Stage 1 verifier | $K$=4 | $K$=8 | $K$=12 | $K$=16 | $K$=4 | $K$=8 | $K$=12 | $K$=16 | $K$=4 | $K$=8 | $K$=12 | $K$=16 |
| DeepSWE-Verifier | 53.99 | 57.10 | 58.43 | 59.26 | 42.65 | 46.11 | 48.23 | 49.70 | 52.13 | 52.93 | 53.80 | 54.21 |
| R2EGym-Verifier | 53.67 | 56.38 | 57.59 | 57.95 | 42.35 | 45.04 | 46.27 | 46.56 | 52.20 | 53.28 | 53.77 | 53.51 |
| DEV matching-pairs | 53.71 | 56.70 | 57.82 | 58.75 | 42.49 | 45.58 | 47.38 | 48.38 | 52.13 | 52.66 | 53.07 | 53.11 |
| AgentPRM | 53.55 | 56.70 | 58.40 | 59.76 | 41.47 | 44.22 | 45.74 | 46.56 | 51.96 | 52.58 | 52.85 | 53.31 |
| **LatentSift (ours)** | 53.45 | 56.92 | 58.64 | 60.06 | 41.79 | 44.23 | 46.12 | 47.73 | 51.75 | 52.65 | 53.60 | 54.21 |

**Cross-policy comparison.** Table 7 lists the resolution rates and token costs behind Figure 5. LatentSift reduces EF-verifier tokens by 79.2% on R2EGym-32B and by 81.0% on R2EGym-14B, and total verification tokens by 49.1% and 49.4%, respectively. Hybrid Best@16 rises by 1.17 points on R2EGym-32B and stays within 0.33 points on R2EGym-14B.

Table 7: Cross-policy results on R2EGym Agent at $K = 16$. Best@16 is the resolution rate (%). Token cost gives EF-verifier (EF) and total verification tokens per task instance, in thousands. Both cascades use R2EGym-Verifier in Stage 4.

| | R2EGym-32B | | | R2EGym-14B | | |
|---|---|---|---|---|---|---|
| Stage 1 verifier | Best@16 | Token cost (K) | | Best@16 | Token cost (K) | |
| | | EF | Total | | EF | Total |
| R2EGym-Verifier | 46.56 | 366.2 | 599.6 | 43.28 | 344.2 | 575.3 |
| **LatentSift (ours)** | **47.73** | **76.0** | **304.9** | **42.95** | **65.5** | **291.2** |