StopLoss: Teaching Agents to Predict Failure and Stop Early

Anonymous authors
Paper under double-blind review

Abstract

Coding agents keep working on runs that will ultimately fail verification, and asking a frontier model to watch for this is expensive in its own right. We introduce StopLoss, a small external monitor that learns failure recognition from completed trajectories and their verifier outcomes. Final outcomes label earlier local windows, so a small fine-tuned language model can score the most recent tool steps in a single forward pass, and a threshold calibrated on successful runs turns that score into an abort decision without changing the executor. Trained on GPT-family SWE-rebench trajectories and evaluated on SWE-bench Verified and a SWE-bench Pro subset with GPT and Claude executors, StopLoss is compared against six general-purpose judges reading identical windows. Outcome supervision lifts failure prediction from near chance to a useful signal at the first check, at one to two orders of magnitude lower cost per check than any tested judge. This translates into savings: relative to completing every task, StopLoss lowers cost per retained solve by 4.5% on Verified while preserving 99% of successful runs, and by up to 31% on the costlier Pro subset, and in all four settings its cost per retained solve is below that of every judge policy. Ranking transfers less well to the unseen Claude executor, and a threshold calibrated on one benchmark does not carry its false-kill rate to another. Outcome-trained small monitors can thus make failure recognition cheap enough for early stopping to pay for itself.

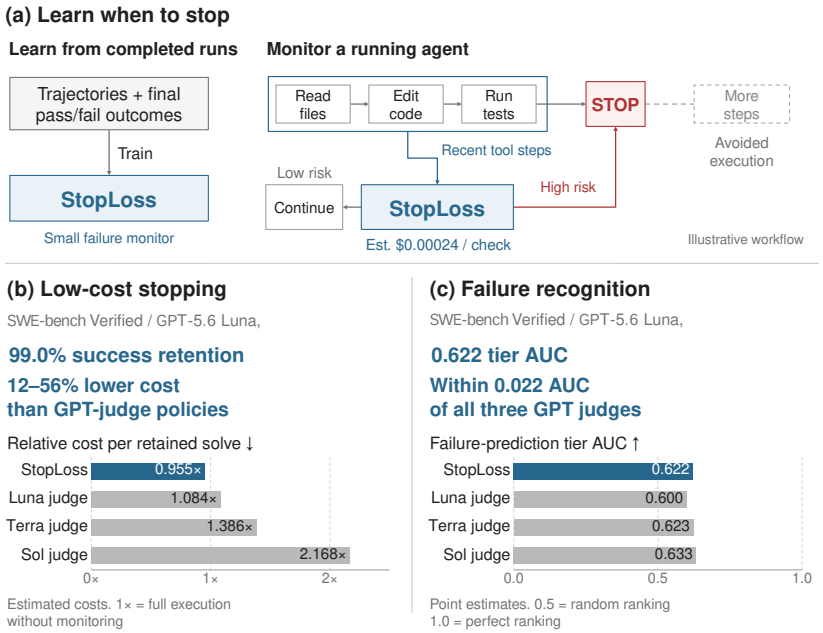

Figure 1: **StopLoss learns failure recognition for inexpensive early stopping.** (a) Final outcomes supervise a small monitor that scores recent tool steps and decides whether to continue or stop. (b) Cost and success retention on Verified/Luna runs; (c) tier AUC on Verified/Luna.

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.

1 INTRODUCTION

Coding agents can continue editing, testing, and invoking tools on runs that will ultimately fail verification. By the time the verdict arrives, the cost of those runs has already been paid. A monitor that recognizes failure risk from what is observable during execution could stop unproductive work before more computation is spent.

Two kinds of monitors are natural, and each has a problem. Budget-based approaches, from step or spend caps to the within-budget feasibility estimators of BAGEN (Lin et al., 2026), ask whether a run will finish within a budget, so a run that would eventually pass can still be cut off. A general-purpose model can instead read the trajectory and judge the agent's progress, but each judgment is itself an expensive call, and repeated judging can cost more than it saves. The underlying constraint is simple: early stopping lowers the cost per solved task only if the execution it avoids exceeds the monitoring overhead plus the successes it mistakenly kills (Section 5.3). A practical monitor must therefore be both discriminative and cheap.

Completed trajectories offer a way to obtain both. Verifier outcomes arrive only after execution, but in retrospect they label every earlier window of a run, providing supervision for a predictor of eventual failure from local records alone. Because the label is the outcome of continuing the observed execution, an expensive run that eventually passes remains a success: the monitor learns to recognize failure rather than to enforce a budget. A small model trained on such data can then assess new runs as they unfold in a single forward pass, reading only observable text rather than the executor's hidden states (Ruan et al., 2026), so it can monitor closed executors.

StopLoss implements this idea. We adapt 7B–8B models with LoRA (Hu et al., 2022) on GPT-family SWE-rebench trajectories. At scheduled checks, the monitor scores a short window of recent actions and observations and reads a failure probability directly from the output logits. A threshold calibrated on successful trajectories maps the score to an abort decision; the executor remains fixed. We evaluate on SWE-bench Verified and a three-repository SWE-bench Pro subset with `gpt-5.6-luna` (Luna) and `claude-sonnet-4-6` (Sonnet) executors, comparing against six general-purpose judges, three per executor family, that read identical windows, and charging every method for the checks it visits and the successes it loses.

The results connect learned recognition to inexpensive monitoring and early stopping. Figure 1 summarizes the workflow and the Verified/Luna results. Outcome supervision moves failure ranking from near chance to a usable signal: on Verified/Luna, training raises Qwen2.5's mean single-model tier AUC from 0.491 to 0.606, and the trained ensemble already separates outcomes at the first scheduled check, tool step 5 (Section 4.2). Each check costs an estimated \$0.00024, one to two orders of magnitude below every tested judge (Section 4.3). On calibration-disjoint Verified/Luna runs, the frozen policy retains 99.0% of otherwise successful runs and lowers cost per retained solve by 4.5% relative to completing every run, whereas every GPT-judge policy raises it; savings reach 31% on the costlier Pro subset, and across all four settings StopLoss has lower estimated cost per retained solve than every judge policy, with simultaneous intervals excluding zero in nine of twelve comparisons (Section 4.1). A judge can rank failures better and still lose money: on Verified/Sonnet, Opus 5 has the highest AUC in the study yet raises cost per solve by a third, because its monitoring bill exceeds the execution it saves.

Two limits shape the scope of these claims (Section 5). Ranking transfers less well to Sonnet, an executor family absent from training, where Opus 5 remains the stronger ranker. And a threshold calibrated on one benchmark does not carry its false-kill rate to another, even within the GPT family: the same threshold stops 1.0% of successes on Verified but 27.8% on Pro, so prediction quality and threshold transfer require separate validation.

Our contributions are:

- **Outcome-supervised failure monitoring.** We show that completed trajectories and their verifier outcomes suffice to train a small external monitor that scores local execution windows and triggers early stopping, without modifying the executor or accessing its internals.
- **Early recognition at low monitoring cost.** Outcome supervision supplies failure signal at the first planned check, at per-check costs one to two orders of magnitude below general-purpose judges.
- **Practical early-stopping evaluation.** Using replay that charges monitoring and lost successes, and a break-even analysis that relates the two, we show cost advantages over six judge policies across four executor–benchmark settings and characterize when prediction and threshold transfer support early stopping.

2 STOPLOSS

StopLoss learns failure signals from completed trajectories, scores local execution windows, and stops runs when a calibrated threshold is crossed. The executor remains fixed.

2.1 TRAJECTORY WINDOWS AND OUTCOME SUPERVISION

For task t, a completed trajectory has n_t tool steps $\mathcal{T}_t = ((a_{t,1}, o_{t,1}), \ldots, (a_{t,n_t}, o_{t,n_t}))$ and verifier outcome $y_t \in \{0, 1\}$, where $y_t = 1$ denotes success. At step k, the monitor reads the step count and the most recent three action–observation pairs as window $w_{t,k}$. We use a 1,600-token sequence limit; Appendix A specifies the encoding and truncation rules.

Final outcomes supervise earlier local windows: windows from failed runs receive ABORT; those at the last two tool steps of successful runs ($k \geq n_t - 1$) receive DELIVER, and earlier successful-run windows receive CONTINUE. DELIVER is an auxiliary training label; runtime stopping uses only the ABORT score. Outcomes supply offline labels, while runtime inputs contain only trajectory text observed by the current checkpoint.

2.2 LEARNING AND FAILURE SCORING

We adapt a pretrained 7B–8B causal language model with LoRA (Hu et al., 2022), using the outcome-derived labels and the standard causal language-model objective with loss computed only on target-answer tokens. Encoding and optimization details are in Appendix A.

For scoring, let $z_c(w_{t,k})$ be the next-token logit at the end of the templated prompt for the first token of candidate answer $c \in \mathcal{A} = \{\text{CONTINUE}, \text{DELIVER}, \text{ABORT}\}$. We read the failure score as

$$s_t(k) = \frac{\exp z_{\text{ABORT}}(w_{t,k})}{\sum_{c \in \mathcal{A}} \exp z_c(w_{t,k})}.$$

A check requires one forward pass per model, without generating an answer or reasoning trace. The small base, bounded input, and direct logit readout limit monitoring computation. An ensemble evaluates every member and averages their failure scores.

2.3 CALIBRATION AND EARLY STOPPING

Calibration and deployment share checkpoints $\mathcal{K}_t = \{5, 8, 11, \ldots\} \cap [1, n_t]$. For each successful calibration run reaching a checkpoint, we compute $z_t = \max_{k \in \mathcal{K}_t} s_t(k)$. This maximum captures the trajectory-level false-kill event: one threshold crossing aborts the entire run. We choose τ so that at most a target fraction ρ of these successes would be stopped (Appendix B.2). This is an empirical calibration constraint, not a guarantee under distribution shift.

During execution, a checkpoint score $s_t(k) \geq \tau$ triggers an abort; otherwise the executor continues. The stopping time is

$$K_t(\tau) = \min\{k \in \mathcal{K}_t : s_t(k) \geq \tau\}, \qquad \min \varnothing = \infty.$$

3 EXPERIMENTAL SETUP

3.1 DATA, EXECUTORS, AND EVALUATION COHORTS

Training trajectories come from SWE-rebench (Badertdinov et al., 2025), collected with GPT-family executors and Harbor's Terminus-2 scaffold (Terminal-Bench Team, 2025). We fit on 4,867 tasks (61,315 windows) and reserve 219 tasks from held-out repositories (1,858 windows) for model selection. External test tasks are disjoint from both sets.

Evaluation targets SWE-bench Verified's 500 tasks (Jimenez et al., 2024; OpenAI, 2024) and SWE-bench Pro's 266-task Python subset (Deng et al., 2025), spanning `ansible`, `openlibrary`, and `qutebrowser`. Luna and Sonnet 4.6 both use stock Terminus-2 and benchmark-provided environments; benchmark verifiers supply outcomes. Runs lacking a valid verdict are excluded, with collection details in Appendix B.

Threshold calibration. Methods evaluating Luna trajectories use 90 successful GPT runs from the 219 SWE-rebench model-selection validation tasks, not from the 500 Verified targets. Methods evaluating Sonnet trajectories use 76 successful early exploratory Sonnet runs: 51 from Verified and 25 from Pro. Each method obtains its own threshold under the nominal $\rho = 5\%$ false-kill target (Section 2.3), frozen for evaluation. GPT calibration reuses model-selection tasks, so it is not an independent certification set. Sonnet calibration adjusts thresholds without updating monitor parameters. Appendix B.2 gives checkpoint coverage and threshold computation.

Ranking and stopping cohorts. Table 1 distinguishes score ranking from evaluation of the calibrated stopping policy. Ranking uses all 1,523 valid runs on 763 tasks: 499 Verified runs per executor and 264/261 Pro runs for Luna/Sonnet. It includes the Sonnet calibration runs because AUC measures score ordering without applying a threshold. To evaluate stopping on tasks unseen during calibration or early exploration, we exclude all 122 exploratory tasks (74 Verified and 48 Pro) and require valid verdicts from both executors. This leaves the same 425 Verified and 213 Pro tasks for each executor, totaling 1,276 runs on 638 tasks, disjoint from both calibration pools. Thus, Verified's 500 targets become 499 valid runs and then 425 stopping runs after removing 74 exploratory tasks; the excluded tasks are not all calibration examples.

Cohorts used in tables and figures. Tables 2 and 3 report AUC on the full ranking cohort; cost and stopping outcomes in Table 2 use the independent stopping cohort. Figure 1 follows the same cohorts as Table 2. Figure 2 evaluates the original runs reaching each checkpoint before stopping is applied, with sample counts shown in the figure. The descriptive stopping-timing analysis in Section 5.1 uses the full Verified/Luna cohort.

Table 1: Evaluation cohorts. Ranking uses all valid runs; stopping uses the calibration-disjoint pool. Pro denotes the Python subset. Detailed coverage is in Appendix B.

Benchmark / executor	Ranking runs	Stopping runs
Verified / Luna	499	425
Verified / Sonnet	499	425
Pro / Luna	264	213
Pro / Sonnet	261	213

3.2 MONITORS AND COMPARISON METHODS

We train Qwen2.5-7B-Instruct, Qwen3-8B, and Qwen2.5-Coder-7B-Instruct (Qwen Team, 2025a;b) with a common LoRA recipe (Appendix A). We include all five trained Qwen2.5 seeds and all four seeds for each other base, without test-based selection. Each ensemble pays the sum of its members' inference costs. The primary five-member Qwen2.5 ensemble, used for StopLoss results unless otherwise specified, was frozen before formal judge scoring. Untuned bases use the same readout (Section 2.2); Qwen3 thinking is disabled.

Judging follows the same-family protocol in calibration and evaluation: Sol (`gpt-5.6-sol`), Terra (`gpt-5.6-terra`), and Luna read Luna trajectories; Opus 5 (`claude-opus-5`), Opus

4.6 (`claude-opus-4-6`), and Sonnet 4.6 read Sonnet trajectories. All small monitors read both executors. Methods receive identical local window bodies within each cell, with a probability-estimation suffix for judges. GPT judges use high reasoning effort; Claude judges use adaptive thinking with high effort. Prompts and configurations are fixed (Appendix A).

The step-count-only ranking baseline uses $s_t(k) = k$, without reading trajectory content.

3.3 Metrics and cost accounting

Stopping and cost. We evaluate the abort-only policy by replaying complete trajectories. Let $C_t(k)$ be execution cost through step k, with $C_t(\infty)$ including all execution calls, and $M_t(\tau)$ the cost of checks visited through stopping or completion. Cost per retained solve (CPS) is

$$\mathrm{CPS}(\tau) = \frac{\sum_t \{C_t(K_t(\tau)) + M_t(\tau)\}}{\sum_t y_t \, \mathbf{1}[K_t(\tau) = \infty]}. \tag{1}$$

The false-kill rate (FK) is the fraction of otherwise successful runs stopped; success retention is $1 - \mathrm{FK}$. Failure capture is the fraction of otherwise failed runs correctly stopped. Runs without checkpoints always survive, pay full execution cost, and incur no monitoring cost. We report $\Delta\mathrm{CPS} = 100(\mathrm{CPS}/\mathrm{CPS}_0 - 1)$, where CPS_0 is the cost per solve under full execution without monitoring; negative values indicate savings. Judge comparisons use paired differences between these changes on the same tasks, in percentage points (judge minus StopLoss); positive differences favor StopLoss. Comparisons include success retention because equal nominal targets do not ensure equal realized risk.

Prediction. Failure windows often occur later and at higher spend. Our primary ranking metric, *tier AUC*, divides each cell's checkpoint windows into quartiles of spent execution cost, computes failure AUC within each tier containing both outcomes, and averages valid tiers. This compares windows at similar accumulated costs, although longer runs still contribute more observations. Fixed-step diagnostics instead compute ordinary AUC with one observation per run reaching that step, before any stopping policy.

Cost accounting. We charge execution through stopping and all visited checks, including recorded judge attempts and every ensemble member. Table 2 reports cumulative monitoring cost $\sum_t M_t(\tau)$; Section 4.3 reports per-check costs. Under the frozen September 18 list-price schedule, cached input pays the full input rate, and output includes reasoning tokens. Small-monitor costs use batch-1 scoring-loop GPU time; fixed rules cost zero. Per-run costs exclude training, infrastructure, one-off calibration, and offline scoring of unvisited windows. Appendix C provides hardware, timing, prices, and call accounting; Section 5.3 and Appendix C.4 analyze proxy-cost sensitivity.

3.4 Statistical protocol and study scope

We use paired repository-cluster bootstrap draws shared across methods and executors within each benchmark (Efron & Tibshirani, 1994). The primary ensemble has 12 comparisons with same-family judges across four cells, with simultaneous 95% intervals within each metric family; other intervals are pointwise. Ranking superiority requires a simultaneous interval excluding zero and an absolute point difference of at least 0.03; equivalence requires the entire interval inside $[-0.03, +0.03]$. The Pro Python subset has only three repository clusters, so we also report task-cluster sensitivity (Appendix B). Bootstrap intervals describe test-set uncertainty conditional on the trained models. Full judge coverage was commissioned after inspecting initial judge results and full small-monitor scores; intervals do not adjust for that expansion decision (Appendix B).

4 Main results

4.1 Cost-effective early stopping

On the independent Verified/Luna pool, StopLoss combines low monitoring overhead with high success retention (see Table 2). Its estimated cumulative monitoring cost is \$0.44, and it retains 99.0% of otherwise successful runs, compared with 89.5–91.8% for the three GPT-judge policies.

Table 2: **StopLoss achieves lower estimated cost per retained solve than LLM-judge policies across all four settings.** ΔCPS reports the percentage change in cost per retained solve relative to full execution without monitoring, including monitoring overhead; negative values indicate savings. Monitoring is the estimated cumulative cost of monitor calls during execution. Tier AUC measures how well the monitor assigns higher failure scores to runs that eventually fail than to those that succeed (0.5 = chance; 1.0 = perfect ranking). Success retained is the percentage of otherwise successful runs preserved by the policy; failure capture is the percentage of otherwise failed runs correctly stopped. Stops reports the total number of stopped runs, followed by the breakdown into correct stops and false kills: total (correct/false).

Method	Cost		Prediction	Stopping outcomes		
	ΔCPS (%) ↓	Monitoring ($)	Tier AUC ↑	Success retained (%) ↑	Failure capture (%) ↑	Stops (correct/ false)
SWE-bench Verified / GPT-5.6 Luna	425 stopping runs; 306 successes; 119 failures					
StopLoss	-4.5	0.44	0.622	99.0	5.0	9 (6/3)
Sol	+116.8	33.20	0.633	91.8	20.2	49 (24/25)
Terra	+38.6	12.90	0.623	89.5	24.4	61 (29/32)
Luna	+8.4	4.18	0.600	90.8	14.3	45 (17/28)
SWE-bench Verified / Claude Sonnet 4.6	425 stopping runs; 326 successes; 99 failures					
StopLoss	+0.2	0.44	0.561	100.0	0.0	0 (0/0)
Opus 5	+34.7	127.13	0.747	95.7	32.3	46 (32/14)
Opus 4.6	+11.6	37.07	0.602	99.1	0.0	3 (0/3)
Sonnet 4.6	+10.3	37.97	0.602	97.5	3.0	11 (3/8)
SWE-bench Pro / GPT-5.6 Luna	213 stopping runs; 144 successes; 69 failures					
StopLoss	-24.2	0.20	0.614	72.2	43.5	70 (30/40)
Sol	+59.3	15.36	0.657	86.8	29.0	39 (20/19)
Terra	+24.9	6.44	0.651	79.2	27.5	49 (19/30)
Luna	+0.0	2.03	0.639	87.5	14.5	28 (10/18)
SWE-bench Pro / Claude Sonnet 4.6	213 stopping runs; 107 successes; 106 failures					
StopLoss	-30.6	0.30	0.579	91.6	19.8	30 (21/9)
Opus 5	-26.5	34.59	0.695	95.3	31.1	38 (33/5)
Opus 4.6	-17.2	23.98	0.614	97.2	15.1	19 (16/3)
Sonnet 4.6	-29.0	19.84	0.598	83.2	23.6	43 (25/18)

At the frozen operating points, its monitoring-inclusive cost per retained solve is approximately 12–56% lower than that of these policies. Compared with running all tasks to completion without monitoring, its estimated cost per retained solve is 4.5% lower.

The larger cost reductions on Pro highlight the economic value of inexpensive monitoring across evaluation benchmarks. Despite similar tier AUCs across Verified and Pro, StopLoss achieves estimated cost-per-solve reductions of 24.2% on Pro/Luna and 30.6% on Pro/Sonnet. These reductions accompany success retention of 72.2% and 91.6%, respectively, showing that the economic benefit depends on the stopping behavior and execution costs in each setting, beyond predictive discrimination alone.

4.2 Learning early failure recognition

Training provides most of the prediction gain, with smaller gains from ensembling. For Qwen2.5, training improves mean single-model tier AUC by $+0.115$ on Verified/Luna and $+0.148$ on Pro/Luna; ensembling adds approximately 0.014–0.016. All three trained ensembles achieve

higher AUC point estimates than their untuned bases across all four settings (Table 3). Appendix D reports the single-model results and paired training contrasts.

Table 3: **Trained ensembles achieve higher failure-prediction AUC than their untuned backbones in all four settings.** Parentheses give absolute AUC gains over the corresponding untuned base. Step count only uses the current tool-step index as the failure score, without reading trajectory content.

		Luna trajectories		Sonnet trajectories	
Backbone	**Configuration**	SWE-bench Verified	SWE-bench Pro	SWE-bench Verified	SWE-bench Pro
Qwen2.5	Untuned base	0.491	0.452	0.512	0.511
	Trained ensemble	0.622 (+0.131)	0.614 (+0.162)	0.561 (+0.049)	0.579 (+0.068)
Qwen3	Untuned base	0.504	0.507	0.481	0.516
	Trained ensemble	0.605 (+0.100)	0.621 (+0.114)	0.563 (+0.082)	0.608 (+0.092)
Coder	Untuned base	0.437	0.433	0.455	0.478
	Trained ensemble	0.612 (+0.175)	0.620 (+0.187)	0.561 (+0.106)	0.578 (+0.100)
Step count only		0.489	0.528	0.471	0.513

The trained monitors also outperform a simple step-count baseline. This baseline assigns higher failure scores to windows reached after more tool steps, without examining what happened during execution. Its tier AUC ranges from 0.471 to 0.528, below every trained ensemble in the corresponding setting (Table 3). The learned scores therefore rank failures more effectively than tool-step count alone.

The predictive signal is already present at the first scheduled check, tool step 5. The primary ensemble achieves ordinary AUCs of 0.642 on Verified/Luna and 0.645 on Pro/Luna, compared with 0.494 and 0.450 for its untuned base. These results show that training improves failure prediction early in execution. Figure 2 reports performance at later checkpoints; Section 5.1 examines when the stopping policy actually acts.

4.3 Prediction quality and monitoring cost

StopLoss provides failure predictions at low monitoring cost. The primary ensemble's estimated inference cost is $0.000237 per check under our GPU-time accounting. Under the tested configurations, GPT judges cost 9–74 times as much per check, pooling each judge's checks across its two benchmarks. On Luna trajectories, the primary ensemble's tier AUC differs from each judge's point estimate by at most 0.044. This combination of low monitoring cost and predictive discrimination helps explain the policy-level cost advantages in Section 4.1. On Sonnet trajectories, Opus 5 achieves higher AUC; Section 5.2 examines cross-executor transfer. Appendix C details the cost accounting.

5 Analysis and Discussion

5.1 When are failures recognized and stopped?

Figure 2 extends the first-check results in Section 4.2 across scheduled checkpoints. On Sonnet, the primary monitor also shows predictive signal at tool step 5, with ordinary AUCs of 0.604 on Verified and 0.645 on Pro. To examine stopping timing, we audit all Verified/Luna runs under the frozen GPT-calibrated threshold. Across the eight correctly stopped failures, the median fraction of execution cost remaining when the policy triggered was 64.5%. The three largest savings account for 65.8% of the total execution cost avoided by these correct stops. Thus, even a small number of correct stops can avoid substantial remaining execution, with savings concentrated in a few runs.

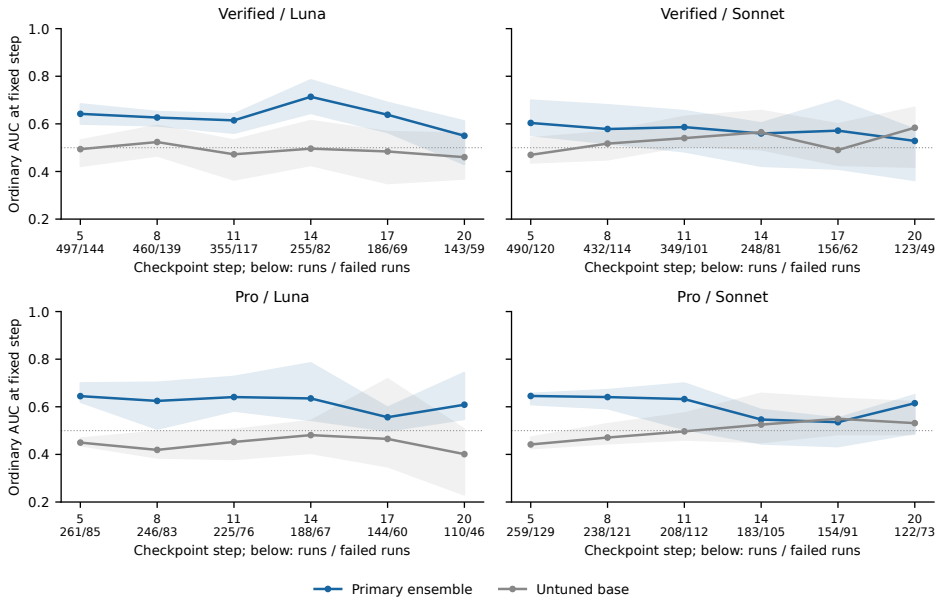

Figure 2: **StopLoss provides an early failure signal from the first scheduled check.** At tool step 5, the primary StopLoss ensemble (blue) achieves higher ordinary AUC than its untuned base (gray) in all four settings. Higher AUC indicates better separation of eventual failures from successes; the dotted line marks chance-level ranking (AUC = 0.5). This post-hoc evaluation uses one observation per original run reaching each checkpoint, without applying the stopping policy. Shaded bands show pointwise 95% repository-bootstrap intervals. Numbers below each checkpoint report total runs / failed runs. Later checkpoints contain different subsets of runs.

5.2 Transfer of predictions and stopping thresholds

StopLoss learns failure cues from local action–observation windows, and these cues vary with how an executor uses tools and responds to errors. A post-hoc audit shows that failed runs are longer than successful runs under both executors on both benchmarks, so the association between execution length and failure remains present on Sonnet. However, even at the same checkpoint, tool step 8 on Verified, the primary monitor achieves an ordinary AUC of 0.627 on Luna versus 0.578 on Sonnet, as shown in Figure 2. This fixed-checkpoint gap supports the interpretation that failure cues learned from GPT trajectories transfer only partially to Sonnet. Sonnet-specific calibration adjusts the stopping threshold but does not change the learned scores or their ranking.

Similar prediction quality can also produce different stopping behavior across benchmarks. On Luna, tier AUC is 0.622 on Verified and 0.614 on Pro, yet the frozen threshold yields false-kill rates of 1.0% and 27.8%, respectively, on the independent stopping cohorts in Table 2. StopLoss stops a run as soon as any checkpoint score crosses the threshold. The run-length audit shows that successful Luna runs have a median of 16 tool steps on Pro versus 13 on Verified. Longer trajectories allow more scheduled checks and may therefore increase the chance of a threshold crossing. This provides a plausible contributor to the higher false-kill rate on Pro, alongside possible shifts in the scores assigned to successful runs. Threshold transfer thus depends on the scores encountered throughout a successful trajectory, beyond the ranking quality summarized by AUC.

5.3 Execution cost and monitoring overhead

Let $C = \sum_t C_t(\infty)$, let g be the fraction of execution cost avoided by aborts, and let $m = \sum_t M_t/C$ be monitoring cost relative to C. Writing $r = \mathrm{FK}$, a policy retaining at least one success satisfies

$$\frac{\mathrm{CPS}}{\mathrm{CPS}_0} = \frac{1-g+m}{1-r}, \qquad \mathrm{CPS} < \mathrm{CPS}_0 \iff g > m + r. \tag{2}$$

Execution savings must cover both monitoring overhead and lost successes. A small monitor reduces m; whether its threshold yields enough savings g at an acceptable false-kill rate r remains empirical.

On Verified/Luna, the 425 complete runs cost \$28.44, while Sol's visited checks alone cost \$33.20 and StopLoss's cost \$0.44. This monitoring term explains why similar ranking points can produce very different costs per retained solve. On Verified/Sonnet, Opus 5 avoids 16.0% of execution cost but spends 44.9% of the original execution budget on monitoring, raising cost per solve by 34.7%. More expensive execution, as on Pro/Sonnet, leaves room for judges to recover their monitoring expense (see Table 2).

Sensitivity to the inference-cost proxy. We measure cost-per-solve changes relative to full execution while inflating only the ensemble's monitoring charge, holding scores, thresholds, visited checks, and false-kill rates fixed. On the independent Verified/Luna pool, doubling the charge moves the point estimate from -4.5% to -2.9%; at five times the charge it becomes $+1.8\%$. Both Pro cells retain savings at ten times the proxy, with unchanged stopping risks (see Appendix C.4).

6 Related Work

Trajectory evaluation and early stopping. Reward models and verifiers assess reasoning and agent trajectories to support verification, search, and corrective feedback (Lightman et al., 2024; Xia et al., 2025; Gandhi et al., 2025). Online failure monitoring turns trajectory assessment into early intervention. Ruan et al. (2026) use hidden-state probes with recall-controlled stopping, while AgentStop relies on inexpensive execution signals such as token log probabilities (Pham et al., 2026). PrefixGuard learns failure-warning monitors from observable traces (Huang et al., 2026). FailFast–RestartSmart trains a small text-based monitor using terminal outcomes and intermediate patch-test supervision, then combines stopping with edit-preserving restarts (Wang et al., 2026). StopLoss learns eventual failure from final verification outcomes and short action–observation windows, providing inexpensive stopping decisions without requiring executor internals or intermediate patch-test labels.

Cost-efficient agent execution. BAGEN studies remaining-resource estimation and completion within a budget (Lin et al., 2026), while EET uses execution experience to guide early termination during patch generation and selection (Guo et al., 2026). FrugalGPT and RouteLLM reduce inference costs through model cascades and routing (Chen et al., 2023; Ong et al., 2024). SWE-Router uses partial coding trajectories to decide whether to continue with a cheaper model or escalate to a stronger one (Son et al., 2026). StopLoss focuses on whether to continue the current run under a fixed executor, using predicted failure rather than budget exhaustion as its stopping signal. We evaluate this decision through cost per retained solve, accounting for monitoring charges and lost successes, and compare the trade-offs with prompted judges.

7 Limitations and Conclusion

Our models are trained exclusively on GPT-generated trajectories, which may limit their generalization to Claude executions. Budget constraints also restrict our training and evaluation to Python tasks, with our SWE-bench Pro evaluation covering only three repositories from its public Python subset. We believe StopLoss provides a general approach to early stopping that can be extended to other programming languages and benchmarks using suitable trajectories and outcome labels. We leave this broader evaluation to future work.

StopLoss trains a small external monitor from completed trajectories to recognize failure during execution and trigger early stopping, while leaving the executor fixed. Outcome supervision improves single-model predictions and supplies signal at the first planned check. With low estimated monitoring overhead, the primary ensemble achieves lower cost per retained solve than general-purpose judge policies at their frozen thresholds in nine of twelve paired comparisons, supported by simultaneous intervals. Improving cross-executor generalization and threshold transfer is the next step toward extending this inexpensive failure monitoring and its early-stopping benefits to more agent settings.

AI USE STATEMENT

We used generative AI tools to assist with language polishing and textual revisions of the manuscript, including improvements to clarity, grammar, and presentation. The authors take responsibility for the final content of the paper.

ETHICS STATEMENT

The experiments use public software repositories and benchmark tasks. We preserve their licensing requirements. An early-stop policy can silently reduce an agent's success rate; we report false kills and retained solves alongside resource savings to make this trade-off explicit.

REFERENCES

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*, 2025.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023. URL https://arxiv.org/abs/2305.05176.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall/CRC, 1994.

Shubham Gandhi, Jason Tsay, Jatin Ganhotra, Kiran Kate, and Yara Rizk. When agents go astray: Course-correcting SWE agents with PRMs. *arXiv preprint arXiv:2509.02360*, 2025. URL https://arxiv.org/abs/2509.02360.

Yaoqi Guo, Ying Xiao, Jie M. Zhang, Mark Harman, Yiling Lou, Yang Liu, and Zhenpeng Chen. EET: Experience-driven early termination for cost-efficient software engineering agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 33008–33024. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1652. URL https://aclanthology.org/2026.findings-acl.1652/.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Xinmiao Huang, Jinwei Hu, Rajarshi Roy, Changshun Wu, Yi Dong, and Xiaowei Huang. PrefixGuard: From LLM-agent traces to online failure-warning monitors. *arXiv preprint arXiv:2605.06455*, 2026. URL https://arxiv.org/abs/2605.06455.

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*, 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*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/aca97732e30bcf1303bc22ac3924fd16-Abstract-Conference.html.

Yuxiang Lin, Zihan Wang, Mengyang Liu, Yuxuan Shan, Longju Bai, Junyao Zhang, Xing Jin, Boshan Chen, Jinyan Su, Xingyao Wang, Jiaxin Pei, and Manling Li. BAGEN: Are LLM agents budget-aware? *arXiv preprint arXiv:2606.00198*, 2026. URL https://arxiv.org/abs/2606.00198.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs with preference data. *arXiv preprint arXiv:2406.18665*, 2024. URL https://arxiv.org/abs/2406.18665.

OpenAI. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/, 2024.

Dzung Pham, Kleomenis Katevas, Ali Shahin Shamsabadi, and Hamed Haddadi. AgentStop: Terminating local AI agents early to save energy in consumer devices. *arXiv preprint arXiv:2605.15206*, 2026. URL https://arxiv.org/abs/2605.15206.

Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2025a.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025b.

Kai Ruan, Zihe Huang, Ziqi Zhou, Qianshan Wei, Jinghao Lin, Xuan Wang, and Hao Sun. Doomed from the start: Early abort of LLM agent episodes via a recall-controlled probe cascade. *arXiv preprint arXiv:2607.06503*, 2026. URL https://arxiv.org/abs/2607.06503.

Seongho Son, Sangwoong Yoon, Jiahua Tang, Shuhan Wang, Lorenz Wolf, and Ilija Bogunovic. SWE-Router: Routing in multi-turn agentic software engineering tasks. *arXiv preprint arXiv:2607.00053*, 2026. URL https://arxiv.org/abs/2607.00053.

Terminal-Bench Team. Harbor and Terminal-Bench: a harness and benchmark for terminal agents (Terminus-2 reference agent). https://github.com/laude-institute/harbor, 2025.

Chenyu Wang, Yunbo Lyu, Junda He, Zhou Yang, Chenxing Zhong, Yaniv Harel, and David Lo. Fail-fast, restart-smart: Early failure prediction and restart for SWE agentic tasks. *arXiv preprint arXiv:2608.03222*, 2026. URL https://arxiv.org/abs/2608.03222.

Yu Xia, Jingru Fan, Weize Chen, Siyu Yan, Xin Cong, Zhong Zhang, Yaxi Lu, Yankai Lin, Zhiyuan Liu, and Maosong Sun. AgentRM: Enhancing agent generalization with reward modeling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19277–19290. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.945. URL https://aclanthology.org/2025.acl-long.945/.

A PROMPTS, LABELS, AND TRAINING DETAILS

The boxes reproduce the user-message template and judge suffix, with placeholders and line breaks normalized for typesetting. Each window contains up to three tool-step action/observation blocks in order. Construction first truncates each action (reasoning plus commands) to 700 characters and each observation to 900 characters. If the resulting monitor sequence exceeds 1,600 tokens, the builder removes text from the middle of the window body, retaining its beginning and end and marking the omission. It preserves the step-count header and monitor question, checking the full chat-templated prompt plus every candidate answer. The trainer and scorer then reject any overlong sequence; they do not silently truncate it again.

StopLoss monitor: user-message template

An autonomous agent has run ⟨k⟩ steps on a task. Its last steps:
`<action>`
⟨action text⟩
`</action>`
`<observation>`
⟨observation text⟩
`</observation>`
⟨repeat the block for the remaining recent steps⟩
Decide the agent's best move now: CONTINUE working, DELIVER (verify and declare done), or ABORT (hopeless). Answer with one word.

The score is the softmax over the first answer tokens, read as the probability of ABORT. For Qwen3 the chat template has thinking disabled. Weak outcome labels do not certify that a run could not recover under a different continuation.

Paid judges receive the same window body, replacing the monitor question with the following suffix. Their complete requests therefore differ in the question and provider formatting; the 1,600-token limit above is the small monitor's sequence budget, not a judge output or reasoning limit.

General-purpose judge: replacement question

You are an expert reviewer of autonomous coding-agent runs. The agent above is working on a software-engineering task in a repository; only its most recent steps are shown. Estimate the probability that this run will ULTIMATELY FAIL (its final patch will not pass the hidden tests) if the agent keeps going as it has been. Reply with at most three sentences of reasoning, then a final line exactly of the form:
`FAIL_PROBABILITY: <integer 0-100>`

The parsed probability is divided by 100. Score-acceptance checks are specified in Appendix B.3. Configurations and prompts remain frozen throughout the full-coverage evaluation.

Executor and judge configurations. Table 4 separates trajectory generation from judging, including when the same model serves both roles. Same-model self-judging follows the stated judging configuration, which need not match the executor's generation configuration. This distinction does not change the fixed input windows or model-family restriction. The trial configurations and benchmark task files for all 1,523 valid runs record 100 agent turns and separate 7,200-second agent and verifier timeouts. The collector also has a 16,000-second outer process timeout. These are execution limits; monitor checkpoints count tool steps.

Table 4: Recorded generation and judging configurations. Model identifiers are defined in the main text. Collection spend valves use the collection-time accounting schedule; reported costs are recomputed under the frozen full-price schedule.

Role	Model(s)	Reasoning setting	Limits and interface
Executor	Luna	Effort unset; upstream default not recorded	100 turns; 7,200 s agent timeout; $9 spend valve. Output-token caps stripped by the GPT gateway.
Executor	Sonnet	Thinking enabled, 8,000-token budget	100 turns; 7,200 s agent timeout; $26 spend valve. Gateway default output limit 16,000 tokens.
Judge	Sol, Terra, Luna	High reasoning effort	Native Responses API; output-token caps stripped, so no enforced per-call output cap.
Judge	Opus 5, Opus 4.6, Sonnet	Adaptive thinking, high effort	Native Messages API; output limits 4,000, 8,000, 16,000 on truncation retries.

The recorded scaffold is Terminus-2 version 2.0.0 in Harbor 0.17.0 (checkout `f2b2f7febe8e`, bound by the collection audit). The effective parser is JSON: the launcher passed `parser=xml`, but this constructor reads `parser_name`, whose default is JSON; sampled ATIF records from every collection source confirm JSON. No aggregate token budget was set. Neither executor explicitly sets a sampling temperature; both gateways drop sampling overrides. Luna's unset reasoning effort must not be read as high effort or disabled reasoning. Sonnet's 16,000-token value is the recorded gateway default, not a measured output length; no separate hard clamp was configured.

Encoding and training objective. The corpus and task split are specified in Appendix B.1. A training input is the chat-templated prompt followed by its complete target word, with no appended EOS token. The prompt and right-padding labels are masked with -100, and every answer token is supervised by the standard causal language-model loss over the vocabulary. The main recipe uses zero label smoothing and does not enable the alternative three-logit soft-target trainer. Each candidate word spans two tokens under the three base tokenizers. Scoring uses only the first-token readout in Section 2.2.

The shared encoder checks that the prompt/answer token boundary aligns. The label boundary remains $k \geq n_t - 1$ for successful runs, independently of the sampled window grid.

Optimization and selection. All three bases use one epoch, rank-16 LoRA with scale 32 and dropout 0.05 on q/k/v/o attention projections, AdamW with learning rate 10^{-4} and weight decay 0.05, cosine decay, and 3% warm-up. The effective batch size is 16: Qwen2.5 uses microbatch 2 with accumulation 8 on one device; Coder and Qwen3 use two devices with microbatch 2 and accumulation 4 per device. Validation occurs every 300 optimizer steps and at the end of training, selecting the checkpoint with best failure AUC on the fixed validation checkpoint windows. Scoring uses the audited BF16 autocast and FP32 readout softmax for both calibration and test. Earlier recipe and input-representation experiments remain exploratory.

B Data and evaluation protocol

B.1 Data construction and splits

The original SWE-rebench corpus contains 5,086 tasks and 63,173 windows. Corpus windows originate from the grid $\{2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 25, 30, 40, 50, 60, 75, 100\}$, with additional all-step windows in later collections; evaluation uses policy checkpoints $5, 8, 11, \ldots$. We reserve 219 tasks from held-out repositories, comprising 1,858 windows, for model selection. Before optimization, the trainer removes every window belonging to those tasks, leaving 4,867 fitting tasks and 61,315 windows. The external Verified and Pro tasks are disjoint from both partitions.

For external evaluation, we retain all collected runs with valid verdicts for ranking. We construct the independent stopping cohort by task identity: remove the 122 early exploratory tasks (74 Verified and 48 Pro), then retain only tasks with valid verdicts from both executors. The resulting shared task lists contain 425 Verified and 213 Pro tasks. They exclude all Sonnet calibration tasks; GPT calibration tasks belong to the separate SWE-rebench validation partition. Section 3.1 explains which reported results use each cohort.

Excluded exploratory feature comparisons. Unlike the monitors, LR/GBDT fitting included the validation tasks and therefore overlapped with GPT calibration data. External Verified/Pro tasks and Claude calibration remain disjoint from feature fitting, but the feature models still used a larger fitting set than the monitors. We exclude these comparisons without refitting the feature models; step and spend caps remain as simple controls.

B.2 Threshold calibration

Eligible runs and checkpoint windows. From the successful-run pools specified in Section 3.1, we retain runs reaching at least one policy checkpoint. The GPT all-step rendering yields 348 checkpoint windows from 90 successful runs. Of the 76 successful Sonnet runs, 74 reach a checkpoint and yield 378 windows; the remaining two do not enter threshold computation. All methods evaluating the same executor family use the same calibration runs and windows.

Threshold computation. For each method, take the maximum checkpoint score on each of the n eligible successful calibration runs, and sort these maxima as $z_{(1)} \geq \cdots \geq z_{(n)}$. With target rate ρ, set $b = \lfloor \rho n \rfloor$ and $\tau = z_{(b+1)} + 10^{-9}$. The offset places the threshold strictly above the selected maximum because stopping uses $s_t(k) \geq \tau$. At most b calibration successes are stopped; tied maxima are not split and can make the rule more conservative.

B.3 Evaluation coverage

Table 5 adds solved-run counts, runs without checkpoints, and checkpoint-window counts to the cohort definitions above. Infrastructure or verifier failures without a verdict are excluded, not labeled as model failures; executor solve-rate denominators remain the original 500 Verified and 266 Pro targets. Runs without checkpoints remain in cohort counts and stopping denominators but contribute no ranking windows or monitor calls. Full-cohort stopping replays, including the timing diagnostic in Section 5.1, are descriptive; the main policy comparisons use the independent stopping cohort.

Table 5: Detailed evaluation coverage. Solved runs, runs without checkpoints, and checkpoint windows refer to the full ranking cohort. Every checkpoint window is scored by all three same-family judges and all small monitors. The last column gives the calibration-disjoint stopping pool.

Benchmark / executor	Ranking runs	Solved	No checkpoint	Windows	Stopping runs
Verified / Luna	499	355	2	2307	425
Verified / Sonnet	499	377	9	2215	425
Pro / Luna	264	177	3	1569	213
Pro / Sonnet	261	132	2	1960	213

Coverage extension. The full-coverage experiment is a separately recorded extension decided after inspecting the initial judge results and full small-monitor scores; it was not fixed by the initial registration. Throughout the extension, we held the primary monitor, model configurations, window definition, calibration thresholds, prices, and comparison rules fixed. The extension collected 16,029 new valid scores and reused 8,124 accepted scores, including eligible Claude calibration windows, after checking inputs and configurations; accepted windows were not scored again. All 24,153 required judge–window scores are present, covering 3,876 windows per GPT judge and 4,175 per Claude judge. The manuscript reports this full-coverage analysis without reclassifying it as the original fixed-sample experiment.

Score acceptance and provenance checks. Task identity is resolved from the trial configuration. Before analysis, verifier labels, score records, prompt and configuration hashes, and call ledgers are checked for consistency. Accepted judge scores require successful parsing, normal completion, usage records, and an allowed backend model identity. Truncated answers are rejected even if they contain a parsable number. These checks bind measurements to their recorded inputs and calls.

C Cost accounting and sensitivity analysis

C.1 Token pricing and execution accounting

Table 6: Frozen September 18 full-price schedule in dollars per million tokens. Every input token is charged at the input rate, including cache reads and writes; output includes reasoning/thinking tokens.

Model identifier	Input	Output
`gpt-5.6-sol`	4.00	20.00
`gpt-5.6-terra`	2.00	12.00
`gpt-5.6-luna`	0.20	1.20
`claude-opus-5`	5.00	25.00
`claude-opus-4-6`	5.00	25.00
`claude-sonnet-4-6`	3.00	15.00

This is a standardized cost comparison under the frozen schedule, not an estimate of discounted or subscription charges. The source configuration records the provider pricing-page checks and rejects main-panel calls above its short-context guard rather than silently selecting a tariff. Tool-prefix spend is aligned to ledger calls; surviving runs also pay trailing non-tool calls. Abort decisions avoid future execution and future checks only.

C.2 Monitor inference cost

The score metadata records NVIDIA RTX A6000 GPUs, batch size 1, and BF16 autocast with FP32 readout softmax. The proxy divides elapsed scoring-loop wall time by completed windows at the last progress record (27,001 windows). For the five Qwen2.5 adapters, this is 0.211–0.215 seconds per window; summing members gives 1.065 GPU-seconds and $0.000237 at $0.80/GPU-hour. One adapter averages about $0.000047 per check. The loop includes tensor creation, forward computation, and probability transfer to the host, but begins after model loading and prompt tokenization and ends before the final consistency pass and output writing. This is an amortized scoring proxy, not measured online request latency or a latency percentile. It is reused across evaluation cells; concurrent ensemble execution, serving overhead, and utilization were not benchmarked. Training, fixed deployment, and idle-resource costs are outside this proxy.

Across all trained seeds, mean single-model / ensemble costs per check are $0.0000473 / $0.0002367 for Qwen2.5, $0.0000572 / $0.0002289 for Qwen3, and $0.0000473 / $0.0001893 for Coder. Single-model costs average the members; ensemble costs sum them, using the same GPU-time proxy.

Table 7: Mean per-prediction judge costs on the same full-cohort windows, using recorded attempts at full token prices. Ratios divide by the primary ensemble proxy of $0.000237 per check. This is independent of which checks a stopping policy visits.

Cell	Judge	Windows	$/check	Judge / ensemble
Verified / Luna	Sol	2307	0.01949	82.3
Verified / Luna	Terra	2307	0.00774	32.7
Verified / Luna	Luna	2307	0.00248	10.5
Verified / Sonnet	Opus 5	2215	0.07722	326.3
Verified / Sonnet	Opus 4.6	2215	0.02015	85.1
Verified / Sonnet	Sonnet 4.6	2215	0.02114	89.3
Pro / Luna	Sol	1569	0.01446	61.1
Pro / Luna	Terra	1569	0.00618	26.1
Pro / Luna	Luna	1569	0.00182	7.7
Pro / Sonnet	Opus 5	1960	0.02594	109.6
Pro / Sonnet	Opus 4.6	1960	0.01696	71.7
Pro / Sonnet	Sonnet 4.6	1960	0.01634	69.0

Across cells, judge costs range from $0.0018 to $0.0772 per prediction, approximately 8–326 times the primary ensemble proxy. Pooling each judge across its two benchmarks by window count gives ratios of 9–225; the GPT-only range reported in Section 4.3 is 9–74.

C.3 Collection accounting

Table 8 reports the experimental bill for newly issued scoring calls in the full-coverage extension. This differs from the monitoring charge in policy CPS: offline evaluation covers all required windows, whereas the stopping replay charges only checks visited before stopping or completion. One-off calibration is accounted for separately below.

Table 8: Collection accounting for the full-coverage extension. Covered windows include reused valid scores; new windows were scored once successfully, with any recorded failed attempts charged separately. Known dollars are the newly collected calls at list prices, excluding reused historical scores and calibration expenditure. Unknown-usage calls are additional failed attempts whose prices cannot be reconstructed; they are not priced at zero. The total is rounded after summing unrounded call costs.

Judge	Covered windows	New windows	Known new $	Unknown usage
Sol	3876	2751	48.64	3
Terra	3876	2751	19.71	2
Luna	3876	2751	6.12	7
Opus 5	4175	2592	146.36	47
Opus 4.6	4175	2592	49.07	0
Sonnet 4.6	4175	2592	49.68	1
Total	24153	16029	319.57	60

The extension adds 16,029 valid window scores with known recorded costs of $319.57 at full list prices. It also contains 60 failed attempts with unavailable usage; there are 64 such recorded attempts across all source packages, including calibration. Their cost is unknown, not zero. At most two ad-

ditional in-flight requests may have ended without ledger records. Valid scores cover every required window, but complete accepted scores do not imply that all failed requests have reconstructable charges. One-off calibration costs are \$5.67/\$2.34/\$0.72 for Sol/Terra/Luna and \$20.14/\$6.95/\$7.12 for Opus 5/Opus 4.6/Sonnet 4.6; these are separate from the new-call total and the policy replay bill.

Eight full-cohort Pro/Sonnet runs overlap an executor quota pause. They retain valid verifier outcomes and ledger/trajectory alignment and remain in the analysis. We use full input prices without cache discounts, so cache expiry does not create a change in the pricing convention. The pause's effect on executor behavior is not identified by these runs.

C.4 Sensitivity to monitoring cost

This post-hoc analysis multiplies the primary ensemble's monitoring charge by $\lambda \in \{1, 2, 5, 10\}$, holding predictions, thresholds, visited checkpoints, stopping positions, and retained solves fixed. Executor and judge prices also remain fixed. Let E be the execution cost incurred under the policy, M its original monitoring charge, and S the number of retained solves. Then

$$\mathrm{CPS}_\lambda = \frac{E + \lambda M}{S}.$$

At $\lambda = 1$, all four independent-pool cells reproduce the frozen analysis point estimates. Table 9 reports the adjusted cost-per-solve changes; no confidence intervals are recomputed.

Table 9: Independent-pool cost-per-solve change under inflated primary-ensemble monitoring charges. Scores, frozen nominal-5% thresholds, visited checks, and false-kill rates remain fixed. Negative values are savings.

Cell	$1\times$	$2\times$	$5\times$	$10\times$	FK%
Verified / Luna	-4.5	-2.9	+1.8	+9.7	1.0
Verified / Sonnet	+0.2	+0.3	+0.8	+1.5	0.0
Pro / Luna	-24.2	-23.0	-19.5	-13.6	27.8
Pro / Sonnet	-30.6	-30.5	-30.3	-29.9	8.4

Verified/Luna's cost-per-solve saving disappears when the monitoring charge reaches approximately $3.83\times$ its baseline value. Both Pro settings retain savings at $10\times$, with success retention unchanged. Verified/Sonnet stops no runs and therefore only adds monitoring cost.

D PREDICTION AND STOPPING DIAGNOSTICS

Table 10: **Complete training-seed results.**

Seed	Luna trajectories		Sonnet trajectories	
	Verified	Pro	Verified	Pro
Qwen2.5				
Seed 0	0.615	0.607	0.578	0.548
Seed 1	0.608	0.599	0.522	0.564
Seed 2	0.605	0.586	0.531	0.586
Seed 3	0.598	0.610	0.545	0.582
Seed 4	0.607	0.599	0.566	0.568
Mean ± SD	0.606 ± 0.006	0.600 ± 0.010	0.549 ± 0.023	0.569 ± 0.015
Qwen3				
Seed 0	0.602	0.621	0.549	0.610
Seed 1	0.604	0.608	0.540	0.594
Seed 2	0.602	0.614	0.566	0.601
Seed 3	0.604	0.616	0.576	0.598
Mean ± SD	0.603 ± 0.001	0.615 ± 0.005	0.558 ± 0.016	0.601 ± 0.007
Coder				
Seed 0	0.614	0.617	0.556	0.578
Seed 1	0.598	0.610	0.502	0.571
Seed 2	0.587	0.616	0.551	0.595
Seed 3	0.616	0.604	0.593	0.536
Mean ± SD	0.604 ± 0.013	0.612 ± 0.006	0.550 ± 0.037	0.570 ± 0.025

Illustrative stopping cases. For each outcome category in the full Verified/Luna cohort, we select the middle run ordered by complete execution cost, breaking ties by task identity. All three cases use the same threshold, 0.876, and illustrate behavior rather than error prevalence.

- **Correct stop: SymPy-12419.** At step 17, the window shows continued investigation of nested symbolic summation and Piecewise conditions. The failure score is 0.894, and the logged continuation fails verification. Stopping avoids $0.225 of the run's $0.306 execution cost.
- **Missed failure: SymPy-20428.** The maximum checkpoint score is 0.550 at step 11. The window reports passing local tests and readiness to finish, but the completed run fails verification.
- **False kill: scikit-learn-25931.** At step 20, repeated terminal-recovery attempts and echoed keystrokes receive a failure score of 0.955, although the completed run passes verification.