

Scaling Test-Time Compute for Agentic Coding

Joongwon (Daniel) Kim^{†§}, Winnie Yang^{†‡}, Kelvin Niu[†], Hongming Zhang[†],
 Yun Zhu[†], Eryk Helenowski[†], Ruan Silva[†], Zhengxing Chen[†],
 Srini Iyer[†], Manzil Zaheer[¶], Daniel Fried^{†◇}, Hannaneh Hajishirzi[§],
 Sanjeev Arora^{*}, Ruslan Salakhutdinov[◇], Gabriel Synnaeve[†], Anirudh Goyal[†]
[†]Meta Superintelligence Labs [§]University of Washington [‡]New York University
[¶]Google DeepMind ^{*}Princeton University [◇]Carnegie Mellon University

Abstract

Test-time scaling has emerged as a powerful way to improve LLMs. However, existing methods are best suited to short, bounded outputs that can be directly compared, ranked or refined. Long-horizon coding agents violate this premise: each attempt produces an extended trajectory of actions, observations, errors, and partial progress taken by the agent. In this setting, the main challenge is no longer generating more attempts, but representing prior experience in a form that can be effectively selected from and reused. We propose a test-time scaling framework for agentic coding based on compact representations of rollout trajectories. Our framework converts each rollout into a structured summary that preserves its salient hypotheses, progress, and failure modes while discarding low-signal trace details. These representations enable two complementary forms of scaling. For parallel scaling, we introduce *Recursive Tournament Voting (RTV)*, which recursively narrows a population of rollout summaries through small-group comparisons. For sequential scaling, we adapt *Parallel-Distill-Refine (PDR)* to condition new rollouts on summaries of prior attempts. Our method consistently improves the performance of frontier coding agents across SWE-Bench Verified and Terminal-Bench v2.0. For example, by using our method Claude-4.5-Opus improves from 70.9% $\rightarrow$ 77.6% on SWE-Bench Verified (mini-SWE-agent) and 46.9% $\rightarrow$ 59.1% on Terminal-Bench v2.0 (Terminus 1). Our results suggest that test-time scaling for long-horizon agents is fundamentally a problem of representation, selection, and reuse.

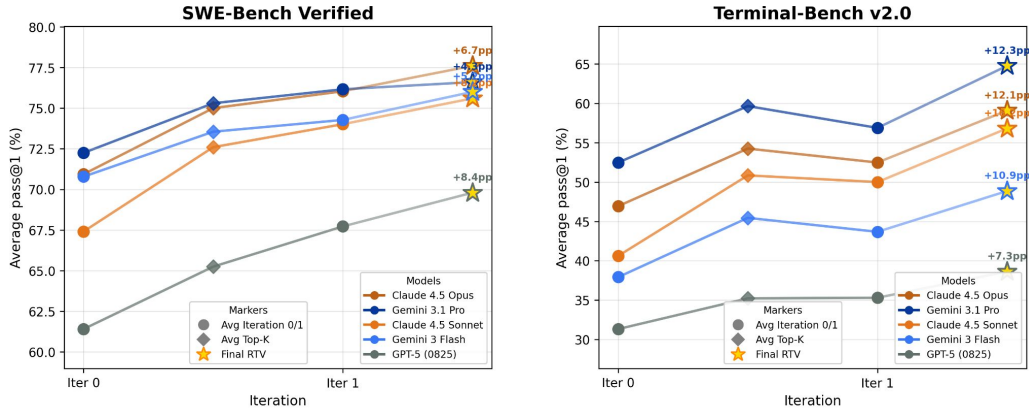


Figure 1: Main results of our agentic **PDR+RTV** test-time scaling method. We improve Claude-4.5-Opus from 70.9% $\rightarrow$ 77.6% on SWE-Bench Verified (mini-SWE-agent) and from 47.0% $\rightarrow$ 59.1% on Terminal-Bench v2.0 (Terminus 1), and Gemini-3.1-Pro from 72.3% $\rightarrow$ 76.6% on SWE-Bench Verified and from 52.5% $\rightarrow$ 64.8% on Terminal-Bench v2.0.

1 Introduction

Test-time scaling has become a major driver of progress in language models (OpenAI, 2024; Anthropic, 2025; Google, 2026; Feng et al., 2026). Rather than treating a model as a one-shot predictor, one can allocate additional inference-time compute to sample multiple candidates, aggregate or compare them, refine later attempts using information from earlier ones (Wang et al., 2023; Madaan et al., 2023; Snell et al., 2024; Madaan et al., 2025). This paradigm has proven effective in domains such as mathematical reasoning and single-turn code generation, where model outputs are compact enough to be manipulated directly in context (Zhang et al., 2025; Li et al., 2025; Zhuang et al., 2026).

Agentic coding introduces a distinct regime from such existing domains. Instead of producing a single bounded completion, the model interacts with an external environment over many steps: reading files, editing code, inspecting logs, executing commands, and responding to intermediate failures. Each attempt therefore produces a long trajectory rather than a short answer. These trajectories contain useful signal, but they also contain noisy, verbose details, and are difficult to compare or reuse directly (Xiao et al., 2025; Li et al., 2026; Ehrlich et al., 2025; Antoniadou et al., 2024; Ahmed et al., 2025). As a result, standard test-time scaling methods do not transfer cleanly to long-horizon agents.

In this work, we argue that the central bottleneck in scaling agentic coding is *representation*. To scale effectively, inference should operate not on raw trajectories, but on their compact representations. We therefore summarize each rollout into a structured artifact that captures its key hypotheses, decisions, progress, and failure modes while omitting low-value trace detail (Lindenbauer et al., 2025; Lu et al., 2025). These summaries become the interface through which prior attempts can be selected from and reused.

We apply these artifacts along two orthogonal dimensions of test-time scaling. First, we explore the *parallel* dimension (Kang et al., 2025; Zhu et al., 2025; Zhao et al., 2025) and introduce **Recursive Tournament Voting (RTV)** to select the strongest attempt from a population of rollouts without access to ground-truth outcomes (Section 2.3). Second, we explore the *sequential* dimension (Qu et al., 2024; Ding & Zhang, 2026; Zhuang et al., 2026) and adapt **Parallel-Distill-Refine (PDR)** (Madaan et al. (2025)) to agentic coding by executing new rollouts conditioned on compact summaries distilled from prior rollouts (Section 2.4).

We perform a set of preliminary experiments to formulate our inference scaling method. First, we find that compact summaries outperform raw trajectories as objects of comparison during parallel aggregation. We also observe that recursive selections from small groups outperform flatter selections from large groups (Section 3.2.1). Moreover, we find that sequential refinement functions better by conditioning on multiple prior summaries instead of a single prior attempt. We further observe that the quality of the refinement context is strongly related with the performance of the subsequent rollouts (Section 3.2.2).

Building on these results, we combine **RTV** and **PDR** into a unified test-time scaling recipe for agentic coding. Our recipe consistently improves over single-attempt baselines across frontier LLMs {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash, GPT-5-0825} and agentic coding benchmarks {SWE-Bench Verified, Terminal-Bench v2.0}. Using our method, on SWE-Bench Verified, Claude-4.5-Opus improves from 70.9% to 77.6% and Gemini-3.1-Pro from 72.3% to 76.6%, and on Terminal-Bench v2.0, Claude-4.5-Opus improves from 46.9% to 59.1% and Gemini-3.1-Pro from 52.5% to 64.8% (Section 3.3). Our results suggest that scaling long-horizon agentic systems depend not only on stronger base models, but also on high-quality trajectory representations and robust inference techniques.

2 Methodology

2.1 Problem formulation

We scale test-time compute for *agentic coding* tasks, where a language model must solve a coding problem by interacting with an external bash environment over multiple steps. Given an agentic coding problem P_{in} , an agent Π_{LM} produces a *rollout* $\mathcal{R}$, a trajectory

consisting of a series of interleaved pairs of *actions* $\mathcal{A}_i$ taken in the bash environment $\mathcal{E}$, and the associated *observation* $\mathcal{O}_i$ returned by the terminal upon executing the agent’s action in $\mathcal{E}$, for a given step index i . Each action consists of (1) the agent’s *thought* $\mathcal{T}_i$ which details the agent’s cognitive process, and (2) one or more bash commands $\mathcal{B}_i = \{b_1, \dots, b_{|\mathcal{B}_i|}\}$ to be executed in the terminal, generated by Π_{LM} . Denoting the context accumulated from the previous step $i - 1$ as $\mathcal{C}_{i-1}$ and the action generation prompt as $\mathcal{P}_{\text{action}}$, we have:

$$\mathcal{A}_i = (\mathcal{T}_i, \mathcal{B}_i) = \Pi_{\text{LM}}[\mathcal{P}_{\text{action}}(P_{\text{in}}; \mathcal{C}_{i-1})] \quad (1)$$

After parsing and executing the agent’s bash command in the environment $\mathcal{E}$, we gather the observation $\mathcal{O}_i$ consisting of both the expected outputs and unexpected error messages:

$$\mathcal{O}_i = \mathcal{E}(\mathcal{C}_{i-1}; \mathcal{B}_i) \quad (2)$$

We then update Π_{LM} ’s context $\mathcal{C}_{i-1}$ by appending the (action, observation) pair for step i .

$$\mathcal{C}_i = [\mathcal{C}_{i-1}; (\mathcal{A}_i, \mathcal{O}_i)] \quad (3)$$

Our goal is to improve performance by scaling the number of rollouts rather than the number of tokens in a single generation. We treat a rollout as the primary scaling unit since it interleaves model outputs with observations, making token-level scaling difficult to interpret consistently across steps. We scale our rollouts along two orthogonal dimensions:

- **Parallel scaling:** execute multiple rollouts in separate containers and run selection.
- **Sequential scaling:** execute new sets of rollouts in freshly initialized containers while conditioning on information extracted from earlier attempts.

2.2 Rollout summaries as reusable representations

A rollout trajectory is typically too long and noisy to compare or reuse directly. Given a rollout $\mathcal{R}_i$, we therefore convert it into a compact, structured summary:

$$S_i = \Pi_{\text{LM}}[\mathcal{P}_{\text{sum}}(\mathcal{R}_i)] \quad (4)$$

where $\mathcal{P}_{\text{sum}}$ is a summarization prompt. These summaries serve as the common interface for both components of our framework. In the parallel setting, they provide bounded objects over which multiple rollouts can be compared. In the sequential setting, they provide reusable context that can guide future rollouts without replaying full interaction histories.

2.3 Parallel selection via Recursive Tournament Voting

We first study the *parallel* dimension of inference-time scaling. Suppose we execute N parallel, independent rollouts with Π_{LM} for the given agentic coding problem P_{in} :

$$\mathcal{P}_0 = \{\mathcal{R}_1, \dots, \mathcal{R}_N\}$$

Our goal is to select the highest-quality rollout possible *without* access to any ground-truth outcomes, test cases or test samples.

We introduce **Recursive Tournament Voting (RTV)** to perform this selection in three stages:

1. Execute N parallel rollouts.
2. Summarize each rollout into a compact, structured summary.
3. Recursively compare summaries in small groups until a single rollout remains.

After generating summaries $\{S_1, \dots, S_N\}$, **RTV** executes stage 3 as a recursive procedure consisting of multiple *rounds* indexed with r , where each round reduces a population of rollouts into a subset by dividing the population into groups of size G and selecting a rollout from each group. More formally, for each round r , **RTV** begins with a population of $N^{(r)}$ remaining rollouts such that $N^{(0)} = N$. Within each group indexed by j , **RTV** applies a comparison prompt $\mathcal{P}_{\text{comp}}$ and aggregates V comparison votes to select one summary:

$$g_j^{(r)} = \arg \max_{g \in \{1, \dots, G\}} \sum_{v=1}^V \mathbb{1} \left[\Pi_{\text{LM}} \left[\mathcal{P}_{\text{comp}}(P_{\text{in}}; S_{(j,1)}^{(r)}, \dots, S_{(j,G)}^{(r)}) \right] = g \right], \quad (5)$$

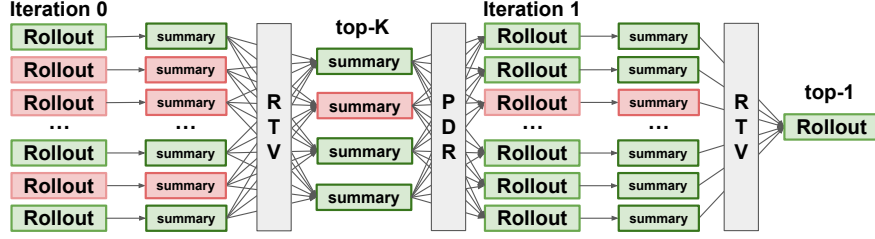


Figure 2: **Unified PDR + RTV inference-time scaling recipe for agentic coding.** The agent first executes N parallel rollouts, and each rollout is converted into a compact summary. **RTV** is then applied to select the top- K summaries, which define the refinement context for the next iteration. The agent then executes a fresh set of N rollouts in newly initialized environments. Finally, **RTV** aggregates the refined rollouts and returns the top-1 rollout.

where $S_{(j,g)}^{(r)}$ denotes the g -th summary in group j at round r .

The selected rollouts form the population for the next round:

$$\mathcal{P}_{r+1} = \left\{ \mathcal{R}_k \mid k = (j-1)G + g_j^{(r)} \text{ for } j \in \left\{ 1, \dots, \left\lceil \frac{|\mathcal{P}_r|}{G} \right\rceil \right\} \right\}. \quad (6)$$

After iterating Eqs. 5 and 6, **RTV** selects the final remaining rollout as its output.

2.4 Sequential reuse via Parallel-Distill-Refine

We next study the *sequential* dimension of inference-time scaling. Here, the objective is to improve future rollouts using information extracted from earlier ones.

To implement sequential refinement, we apply the random- K variant of **PDR** to an agentic setting. We refer to the execution of parallel rollouts as an *iteration*, such that iteration-1 rollouts are refined from iteration-0 rollouts. Starting from an iteration-0 population

$$\mathcal{P}_0 = \{\mathcal{R}_1, \dots, \mathcal{R}_N\},$$

we first summarize each rollout into $\{S_1^{(0)}, \dots, S_N^{(0)}\}$ where each $S_i^{(t)}$ denotes the summary of the i th rollout of iteration t . Then, for each rollout in the next iteration, we sample K summaries from the previous iteration and use them as refinement context.

More formally, for rollout i in iteration $t+1$, let

$$J_i^{(t+1)} \subseteq \{1, \dots, N\}, \quad |J_i^{(t+1)}| = K \quad (7)$$

denote the set of indices associated with the selected summaries, and define the refinement context as a concatenation of the K -selected summaries:

$$\mathcal{C}_i^{(t+1)} = \{S_j^{(t)} \mid j \in J_i^{(t+1)}\} \quad (8)$$

PDR then executes each next-iteration rollout in a *freshly initialized environment*, generating the first action conditioned on both the original problem and this distilled prior experience:

$$\mathcal{A}_{i,0}^{(t+1)} = \Pi_{\text{LM}} \left[\mathcal{P}_{\text{action}}(P_{\text{in}}; \mathcal{C}_i^{(t+1)}) \right], \quad (9)$$

where $\mathcal{A}_{i,m}^{(t+1)}$ is the m th action taken for the i th rollout in iteration $t+1$. Subsequent actions follow the usual rollout dynamics from Eqs. 1 to 3, except with the refinement context added. As **PDR** selects its final rollout by randomly sampling from its final-iteration rollouts, in this work we estimate its performance by averaging the scores of the N final-iteration rollouts.

2.5 A unified pipeline: selection, then reuse, then selection

RTV and **PDR** address complementary aspects of inference-time scaling. **RTV** improves *selection* within a population of rollouts. **PDR** improves *reuse* of information across iterations. Our full recipe integrates these operators into a single pipeline. The full pipeline consists of:

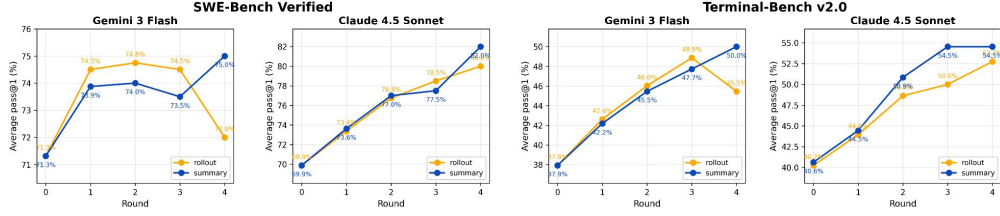


Figure 3: Parallel aggregation results based on generating structured summaries (blue) vs. directly using rollout traces (orange) on SWE-Bench Verified and Terminal-Bench v2.0.

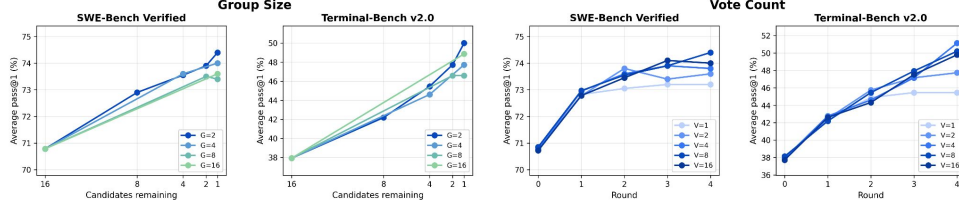


Figure 4: (Left) Effect of scaling the group size $G \in \{16, 8, 4, 2\}$, and (Right) Effect of scaling the vote count $V \in \{1, 2, 4, 8, 16\}$, on final RTV performance using Gemini-3-Flash.

1. **Iteration 0:** execute N independent rollouts and perform summarization;
2. **Select-K:** apply RTV to obtain a high-quality subset of K summaries;
3. **Iteration 1:** execute N fresh rollouts conditioned on the selected summaries;
4. **Final RTV:** apply RTV to the refined rollouts and return the final top-1 rollout.

In a way, we maintain a balance of *exploitation* and *exploration* – exploitation by narrowing down the search space to a subset of $K < N$ higher-quality rollouts using RTV, and exploration by using $K > 1$ rollouts to maintain diversity and cross-pollinating potentially diverse solution approaches for executing the next-iteration rollouts. The final RTV exploits this residual diversity among the refined rollouts. Figure 2 illustrates the full pipeline.

3 Results and analyses

We evaluate our method in three stages: (1) present our *method ablations and design choices*, (2) report our *main results*, and (3) analyze the *sequential & parallel dynamics* of our method.

3.1 Experimental setup

We evaluate on two agentic coding benchmarks – SWE-Bench Verified (Jimenez et al., 2024) and Terminal-Bench v2.0 (Merrill et al., 2026) – using Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash and GPT-5-0825. Our main experiments in Section 3.3 use $N = 16$, $T = 2$, $K = 4$, $G = 2$, and $V = 8$. For SWE-Bench Verified, we evaluate on the full test set. For Terminal-Bench v2.0, we evaluate on 88 of the 89 tasks available to us. Refer to Appendix A for more details on the model capabilities and Appendix B for their head-to-head comparisons.

3.2 Method ablations and design choices

3.2.1 Parallel aggregation ablations

We first identify the best substrate for comparing agentic rollouts during RTV. A direct approach is to compare the full rollout traces. Alternatively, the rollouts can be represented as compact, structured summaries. Figure 3 shows the result of our investigation on SWE-Bench Verified and Terminal-Bench v2.0 using Claude-4.5-Sonnet and Gemini-3-Flash.

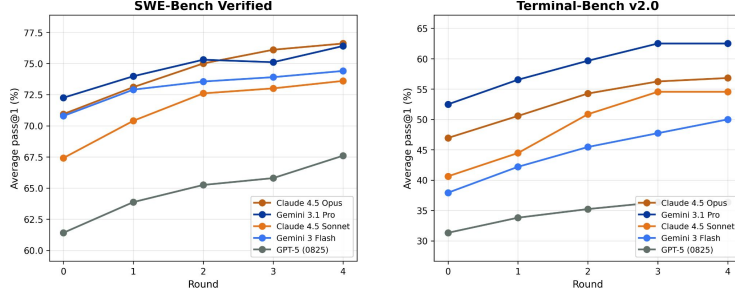


Figure 5: Main **RTV** results with Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash and GPT-5-0825 on SWE-Bench Verified and Terminal-Bench v2.0.

Model	Single-rollout		Random- K rollout		Select- K rollout	
	Iter 0	Iter 1	Iter 0	Iter 1	Iter 0	Iter 1
Claude-4.5-Sonnet	69.87	70.87	69.87	75.06	69.87	78.06
Gemini-3.1-Pro	72.69	73.75	72.69	76.94	72.69	79.25

Table 1: Comparison of refinement methods on 100 randomly sampled tasks from SWE-Bench Verified, based on the average pass rate (%) across $N = 16$ rollouts for iterations 0 and 1.

Finding 1. Compact, structured summaries function as better substrates than full rollout trajectories for selecting among long-horizon agentic rollouts.

Across both models and both benchmarks, using compact, structured summaries consistently outperforms directly comparing full rollout trajectories – full trajectories are too long and noisy to serve as reliable objects of comparison, whereas bounded structured summaries preserve the decisive information while discarding low-signal detail.

We next investigate the architecture of **RTV**. A natural baseline is to load all N summaries into context and perform a single comparison. Alternatively, our proposed **RTV** architecture decomposes the selection procedure into repeated smaller comparisons. Figure 4 reports the effect of varying the group size $G \in \{16, 8, 4, 2\}$ and the number of votes $V \in \{1, 2, 4, 8, 16\}$.

Finding 2. Selection of agentic rollouts is best achieved through recursive small-group comparisons with vote aggregation, rather than flat comparisons over many candidates.

Figure 4 (Left) shows that smaller-group recursive comparison performs better than flatter selection schemes, with pairwise comparisons ($G = 2$) yielding the strongest results. This suggests that selecting among long-horizon trajectories is easier when decomposed into a sequence of local decisions rather than a single global ranking over many candidates. Meanwhile, Figure 4 (Right) shows that vote aggregation improves the reliability of these local decisions, with clear gains as V increases and diminishing returns beginning around $V = 8$. These observations motivate our default **RTV** configuration of $G = 2$ and $V = 8$.

We then evaluate **RTV** as a standalone selection mechanism with $N = 16$, with the results in Figure 5. Across all five models, **RTV** improves performance over the average rollout in the initial population on both SWE-Bench Verified and Terminal-Bench v2.0, with the largest gains in Terminal-Bench v2.0. For example, Claude-4.5-Sonnet improves from $67.4\% \rightarrow 73.6\%$ on SWE-Bench Verified and from $40.6\% \rightarrow 54.6\%$ on Terminal-Bench v2.0. These results show the strength of parallel selection even without any sequential refinement.

3.2.2 Sequential refinement ablations

We next probe our design choices for sequential refinement. First, we study the effects of the prior rollout composition in the refinement context on the next-iteration rollouts. Table 1 compares three methods on 100 SWE-Bench Verified tasks using Claude-4.5-Sonnet and Gemini-3.1-Pro: {single-rollout, random- K , select- K }-refinement. We use $K=4$ prior rollouts.

Model	PDR (random- K) refinement					PDR + RTV (select- K) refinement				
	# pass out of K – pass rate (# avg. tasks)					# pass out of K – pass rate (# avg. tasks)				
	0/4	1/4	2/4	3/4	4/4	0/4	1/4	2/4	3/4	4/4
Claude-4.5-Sonnet	2.5 (17.8)	47.1 (6.5)	80.4 (6.7)	88.8 (16.2)	98.1 (52.8)	1.2 (16)	12.5 (3)	- (0)	90.5 (19)	97.3 (62)
Gemini-3.1-Pro	1.9 (16.2)	50.0 (6.4)	66.4 (7.2)	90.6 (10.6)	99.1 (58.5)	2.2 (17)	40.6 (2)	52.1 (3)	81.2 (7)	99.7 (71)

Table 2: Average iteration-1 pass@1 (%) scores stratified by the number of passing rollouts among the $K = 4$ iteration-0 rollouts used as the refinement context, for **PDR** (random- K) vs. **PDR+RTV** (select- K) on 100 SWE-Bench Verified tasks.

Model	SWE-Bench Verified				Terminal-Bench v2.0			
	Iter 0	Sel-K	Iter 1	Final	Iter 0	Sel-K	Iter 1	Final
Claude 4.5 Opus	70.94	75.00	76.04	77.60	46.95	54.26	52.49	59.09
Gemini 3.1 Pro	72.25	75.30	76.16	76.60	52.49	59.66	56.89	64.77
Claude 4.5 Sonnet	67.41	72.60	74.01	75.60	40.62	50.85	50.00	56.82
Gemini 3 Flash	70.79	73.55	74.28	76.00	37.93	45.45	43.68	48.86
GPT-5 (0825)	61.41	65.25	67.73	69.80	31.32	35.23	35.30	38.64

Table 3: Main results for **PDR+RTV** on SWE-Bench Verified and Terminal-Bench v2.0. We report the average pass@1 score at each stage: Iter 0 (iteration-0 rollouts), Select-K (top- K selected rollouts), Iter 1 (iteration-1 rollouts), and Final (final RTV-selected rollout).

Finding 3. Sequential refinement benefits from conditioning on multiple prior rollouts, and improves further when those rollouts are *selected* rather than sampled at random.

For building the refinement context, using $K=4$ prior rollouts clearly outperforms using a single prior rollout. For example, Gemini-3.1-Pro improves only from 72.69% $\rightarrow$ 73.75% under single-rollout refinement, but improves from 72.69% $\rightarrow$ 76.94% under random- K refinement. Moreover, selecting the $K=4$ rollouts using **RTV** (select- K) further improves performance, reaching 79.25% for Gemini-3.1-Pro and 78.06% for Claude-4.5-Sonnet.

Figure 9 in Appendix C.1 provides a more detailed view by comparing the pass-count distributions under single-rollout and random- K refinement. Across both Claude-4.5-Sonnet and Gemini-3.1-Pro, the distribution under random- K refinement shifts toward larger numbers of passing rollouts relative to single-rollout refinement, indicating that using multiple prior trajectories improves not only the mean but the full distribution of next-iteration outcomes.

We also study how the refinement context quality affects next-iteration rollouts. Table 2 stratifies the results for random- K and select- K , according to how many of the $K=4$ context rollouts succeed. The iteration-1 rollouts’ success rates increase monotonically with the success rate of the refinement context, rising from near-zero for 0/4 success-rate contexts to 97–99% for 4/4 success-rate contexts. It also shows that **RTV** yields more 4/4 passing rollouts than random selection, such as 62 vs. 52.8 tasks for Claude-4.5-Sonnet. Figure 13 in Appendix D.2 shows the stratified pass rate distributions. These results motivate combining **RTV** with **PDR**, so that the refinement context is constructed to be maximally informative.

3.3 Main benchmark results

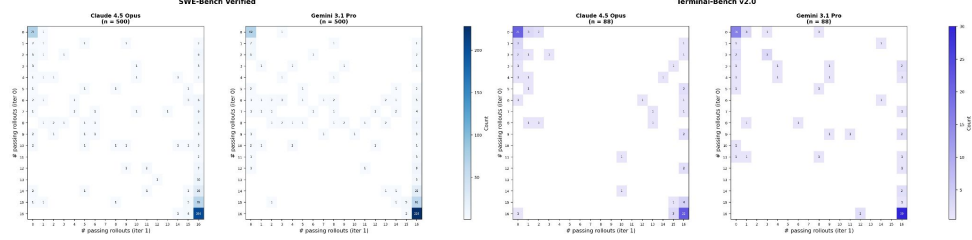
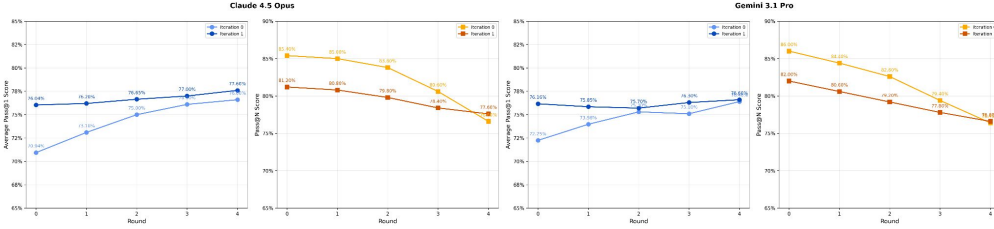
We now turn to the full **PDR+RTV** pipeline. Figure 1 and Table 3 summarize the main results across benchmarks and models.

On SWE-Bench Verified, our method yields consistent gains over the single-attempt baseline, with Claude-4.5-Opus from 70.94% $\rightarrow$ 77.60% (+6.66), Gemini-3.1-Pro from 72.25% $\rightarrow$ 76.60% (+4.35), Claude-4.5-Sonnet from 67.41% $\rightarrow$ 75.60% (+8.19), Gemini-3-Flash from 70.79% $\rightarrow$ 76.00% (+5.21), and GPT-5-0825 from 61.41% $\rightarrow$ 69.80% (+8.39).

On Terminal-Bench v2.0, the gains are even larger in absolute terms, consistent with the benchmark’s higher variance and longer-horizon search difficulty. Our method improves Claude-4.5-Opus from 46.95% $\rightarrow$ 59.09% (+12.14), Gemini-3.1-Pro from 52.49% $\rightarrow$ 64.77% (+12.28), Claude-4.5-Sonnet from 40.62% $\rightarrow$ 56.82% (+16.20), and Gemini-3-Flash from

Model	SWE-Bench Verified						Terminal-Bench v2.0					
	Iter 0			Iter 1			Iter 0			Iter 1		
	All	Pass	Fail	All	Pass	Fail	All	Pass	Fail	All	Pass	Fail
Claude-4.5-Opus	41.23	33.83	59.30	14.31	13.16	17.97	24.43	24.66	24.23	12.14	10.96	13.45
Gemini-3.1-Pro	35.56	33.88	39.92	17.95	17.05	20.82	21.57	17.47	26.09	10.95	9.20	13.25
Claude-4.5-Sonnet	49.24	46.13	55.67	25.02	24.39	26.84	21.74	19.47	23.30	7.78	6.29	9.26
Gemini-3-Flash	51.10	48.39	57.65	28.80	27.37	32.94	16.01	15.68	16.22	7.80	6.16	9.07

Table 4: Average number of steps taken by the agent for each model and benchmark.

Figure 6: Transition matrices of pass-count transitions from iteration 0 to 1 in our **PDR+RTV** main experiments on SWE-Bench Verified (Left) and Terminal-Bench v2.0 (Right).Figure 7: Average pass@1 and pass@N scores across **RTV** rounds for both iterations of our method, for Claude-4.5-Opus (Left) and Gemini-3.1-Pro (Right) on SWE-Bench Verified.

37.93% $\rightarrow$ 48.86% (+10.93). These larger gains suggest that the combination of refinement and selection is especially valuable in settings with substantial rollout-level heterogeneity.

We also analyze the efficiency of the refined rollout populations. Table 4 reports the average number of interaction steps in iteration 0 and iteration 1, both overall and separately for passing and failing rollouts. Across both benchmarks and all models, iteration-1 rollouts are substantially shorter than iteration-0 rollouts. For example, Claude-4.5-Opus drops from 41.23 $\rightarrow$ 14.31 steps on SWE-Bench Verified and from 24.43 $\rightarrow$ 12.14 on Terminal-Bench v2.0, while Gemini-3.1-Pro drops from 35.56 $\rightarrow$ 17.95 and from 21.57 $\rightarrow$ 10.95, respectively. This indicates that refinement improves not only success rate but also *search efficiency* – the agent reaches useful states more quickly using the refinement context.

3.4 Sequential Refinement Dynamics

We next analyze how the rollout populations change from iteration 0 to iteration 1 under the full **PDR+RTV** pipeline. Figure 6 visualizes pass-count transition matrices for Claude-4.5-Opus and Gemini-3.1-Pro on both benchmarks. Each row corresponds to the number of passing rollouts out of $N = 16$ in iteration 0, and each column corresponds to the number of passing rollouts in iteration 1. The mass above the diagonal corresponds to tasks with improvements, while mass below corresponds to tasks with regressions. Across both SWE-Bench Verified and Terminal-Bench v2.0, the matrices exhibit a pronounced upward shift, indicating that sequential refinement improves the overall pass rates for a substantial fraction of tasks. We show transition matrices for all models in Figure 12 in Appendix D.

To better understand the factors leading to improved pass rates upon sequential refinement, we bucket tasks by the number of passing rollouts among the $K = 4$ selected iteration-0 summaries used in the refinement context, and then compute the resulting iteration-1 pass@1 in each bucket. Table 5 reports the results for the full-scale main experiments.

Model	SWE-Bench Verified					Terminal-Bench v2.0				
	# pass out of K – pass rate (# tasks)					# pass out of K – pass rate (# tasks)				
	0/4	1/4	2/4	3/4	4/4	0/4	1/4	2/4	3/4	4/4
Claude-4.5-Opus	0.1 (81)	33.4 (29)	55.5 (25)	85.4 (39)	99.2 (326)	1.8 (31)	31.2 (4)	43.0 (8)	78.5 (9)	94.1 (36)
Gemini-3.1-Pro	0.6 (87)	36.9 (22)	38.4 (22)	87.0 (36)	99.8 (333)	3.8 (23)	37.5 (7)	45.8 (12)	57.5 (5)	93.1 (41)
Claude-4.5-Sonnet	1.7 (94)	18.8 (16)	65.4 (28)	88.1 (68)	99.7 (294)	3.0 (33)	30.2 (6)	58.0 (7)	76.4 (9)	91.7 (33)
Gemini-3-Flash	0.0 (93)	34.7 (18)	73.1 (20)	88.1 (63)	96.4 (306)	1.0 (37)	23.2 (7)	68.1 (9)	60.0 (5)	91.0 (30)

Table 5: Average iteration-1 pass@1 scores (%) of our main experiments, stratified by the number of passing rollouts among the $K = 4$ selected iteration-0 rollouts during sequential refinement on SWE-Bench Verified (**Left**) and Terminal-Bench v2.0 (**Right**).

Finding 4. The quality of the selected refinement context is strongly predictive of the quality of the next-iteration rollouts.

Across all models and benchmarks, iteration-1 performance increases sharply as the quality of the selected context improves. For example, on SWE-Bench Verified Claude-4.5-Opus rises from 0.1% when the selected context contains 0/4 passing rollouts to 99.2% when it contains 4/4 passing rollouts. The same pattern holds for all other models, and also on Terminal-Bench v2.0. These results strongly indicate that *selected prior experience directly drives next-iteration performance*. Refer to Figures 10 and 11 in Appendix C.2 for the full pass-count distribution shifts from iteration 0 to iteration 1 for each model on both benchmarks. Meanwhile, Appendix G dives into examples of several tasks where there are no successful iteration-0 rollouts but a successful iteration-1 rollout.

3.5 Parallel Aggregation Dynamics

We analyze the parallel selection dynamics in our main experiments. In particular, we investigate how **RTV** continues to improve performance even after sequential refinement. We visualize the round-by-round dynamics of **RTV** on SWE-Bench Verified using Claude-4.5-Opus and Gemini-3.1-Pro in Figure 7 while providing the full results in Figures 14 and 15 in Appendix E.1. The blue lines report the pass@1 scores across tournament rounds, while the orange lines report the pass@N scores, for both **RTV** performed after iterations 0 and 1.

Finding 5. Parallel aggregation remains valuable after refinement because refined rollout populations retain useful intra-task diversity exploited by recursive selection.

Figure 7 shows that the second instance of **RTV** over the iteration-1 rollouts still yields improvements, albeit at a smaller rate than the iteration-0 **RTV**. The margin of improvement decreases in iteration 1, because the number of tasks with mixed pass/fail test outcomes significantly decreases, leaving less room for improvement. Yet the improvement persists, as refinement raises the average quality of the rollouts without eliminating the intra-task heterogeneity. For many of the tasks, some refined rollouts remain substantially stronger than others because of differences in local exploration, diagnosis, or recovery from intermediate errors – this leaves meaningful headroom for the final selection mechanism.

Moreover, we measure the accuracy of the groupwise selections performed during both iterations of **RTV**. Within each **RTV** and for each tournament round we extract all groups with both successful *and* failing rollouts, and we measure the selection accuracy. Table 9 in Appendix E.2 presents the results of our analysis. We observe that there is room for improving groupwise selection – as such, we expect that training Π_{LM} to make better group-level rollout selections via SFT or RL would significantly improve the **RTV** performance.

3.6 Compute-matched baselines

We finally investigate whether our gains merely reflect the use of additional test-time compute. Using Claude-4.5-Opus, we compare our method against two selection baselines: **Best-of-N (BoN)** (Kang et al., 2025), where Π_{LM} scores each rollout in isolation from scores of 0 to 5 and we select the rollout with the highest average score across $V=8$ votes, and

Method	SWE-Bench Verified			Terminal-Bench v2.0		
	Eff. tok.	Total tok.	Score	Eff. tok.	Total tok.	Score
BoN ($N=16$)	17,735	989,164	76.0	24,536	978,411	52.3
BoN ($N=32$)	18,671	1,982,793	75.8	26,468	1,963,029	53.4
BoN ($N=48$)	18,771	2,973,310	76.2	27,809	2,953,686	53.2
SC ($N=16$)	53,376	463,618	75.4	61,594	524,525	55.7
RTV (ours)	10,555	1,649,078	76.6	17,648	1,664,141	56.8
PDR + RTV (ours)	14,883	2,796,805	77.6	27,012	2,845,318	59.1

Table 6: Compute-matched comparison using Claude-4.5-Opus. **Eff. tok.** is the longest single token sequence within a method, bounding the context the model must handle, while **Total tok.** sums all rollouts, summaries and judgments, bounding the dollar cost.

self-consistency (SC; Wang et al., 2023), where Π_{LM} ingests all N summaries at once and performs majority voting for the rollout that represents the most common solution approach. We scale $N \in \{16, 32, 48\}$ for BoN to match the total token budget of **PDR+RTV**, and report SC only at $N=16$ since its effective context already reaches 62K tokens.

Finding 6. Our gains are not due to extra compute – at matched or larger token budgets, Best-of- N and self-consistency selection both underperform **RTV** and **PDR+RTV**.

Table 6 shows that scaling N does not help BoN ($76.0 \rightarrow 75.8 \rightarrow 76.2$ on SWE-Bench Verified and $52.3 \rightarrow 53.4 \rightarrow 53.2$ on Terminal-Bench v2.0), as additional rollouts introduce low-quality candidates as often as high-quality ones. In contrast, **RTV** alone already dominates BoN ($N=48$) on both axes, scoring 0.4 and 3.6 points higher while consuming 44% and 37% fewer effective tokens, and **PDR+RTV** is the strongest method overall. Meanwhile, SC is *context-bound* rather than *cost-bound*: it consumes only 28–32% of the total tokens of **RTV** but $3.5\text{--}5.1\times$ its effective tokens, and cannot scale to $N=32$ without overflowing the context window. Finally, **RTV** does not rely on within-family judging, as a stronger cross-family judge improves it further (Appendix E.3).

4 Conclusion

We propose a new test-time scaling method for agentic coding. For agentic tasks, we find that the key bottleneck of leveraging test-time compute is the *representation* of prior experience. We address this bottleneck by converting each rollout into a structured summary which functions as an interface for **Recursive Tournament Voting (RTV)** which performs parallel selection, and **Parallel-Distill-Refine (PDR)** which performs sequential reuse. Combining these two operators yields a unified pipeline that balances exploitation and exploration. Across frontier LLMs {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash, GPT-5-0825} and agentic coding benchmarks {SWE-Bench Verified, Terminal-Bench v2.0}, our proposed method consistently improves over single-attempt baselines and performs rollouts that consume about 50% less steps with improved success rates over the initial rollouts. Our results show that, for long-horizon agents, test-time scaling works when prior experience is transformed into representations that support both *selection* and *reuse*.

5 Limitations

Our margin on SWE-Bench Verified over the strongest compute-matched baseline is modest, which we attribute to its lower rollout-level heterogeneity: SWE-Bench Verified has fewer mixed pass/fail tasks relative to its size than Terminal-Bench v2.0, leaving less headroom for selection. Our method is also not cheap, as judgments account for 88–95% of the total tokens consumed; being parallel and short-context, they inflate total but not effective tokens. We also explore refinement only up to $T=2$: mixed-outcome tasks collapse after one round and pass@16 drops, suggesting diminishing returns as refinement raises the mean but contracts diversity – scaling T further likely requires explicit diversity preservation.

References

- Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate swe patches. *arXiv preprint arXiv:2502.05368*, 2025.
- Anthropic. Introducing claude 4. <https://www.anthropic.com/news/claude-4>, May 2025.
- Antonis Antoniadis, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*, 2024.
- Yifeng Ding and Lingming Zhang. Swe-replay: Efficient test-time scaling for software engineering agents. *CoRR*, abs/2601.22129, 2026. doi: 10.48550/ARXIV.2601.22129. URL <https://doi.org/10.48550/arXiv.2601.22129>.
- 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.
- Tony Feng, Trieu H Trinh, Garrett Bingham, Dawsen Hwang, Yuri Chervonyi, Junehyuk Jung, Joonkyung Lee, Carlo Pagano, Sang-hyun Kim, Federico Pasqualotto, et al. Towards autonomous mathematics research. *arXiv preprint arXiv:2602.10177*, 2026.
- Google. Gemini 3 deep think: Advancing science, research and engineering. <https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-deep-think/>, February 2026.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024*, 2024. URL <https://openreview.net/forum?id=VTF8yNQm66>.
- Zhewei Kang, Xuandong Zhao, and Dawn Song. Scalable best-of-n selection for large language models via self-certainty. *CoRR*, abs/2502.18581, 2025. doi: 10.48550/ARXIV.2502.18581. URL <https://doi.org/10.48550/arXiv.2502.18581>.
- Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, pp. 15964–15978. Association for Computational Linguistics, 2025. URL <https://aclanthology.org/2025.findings-emnlp.865/>.
- Xiaochuan Li, Weixi Feng, Meng Cao, Reno Kriz, Bhuwan Dhingra, Yixin Nie, Emma Strubell, William Yang Wang, and Daniel Fried. Benchmark test-time scaling of general LLM agents. *CoRR*, abs/2602.18998, 2026. doi: 10.48550/ARXIV.2602.18998. URL <https://doi.org/10.48550/arXiv.2602.18998>.
- Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as LLM summarization for agent context management. *CoRR*, abs/2508.21433, 2025. doi: 10.48550/ARXIV.2508.21433. URL <https://doi.org/10.48550/arXiv.2508.21433>.
- Miao Lu, Weiwei Sun, Weihua Du, Zhan Ling, Xuesong Yao, Kang Liu, and Jiecao Chen. Scaling LLM multi-turn RL with end-to-end summarization-based context management. *CoRR*, abs/2510.06727, 2025. doi: 10.48550/ARXIV.2510.06727. URL <https://doi.org/10.48550/arXiv.2510.06727>.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegraffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In Alice Oh,

- Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.
- Lovish Madaan, Aniket Didolkar, Suchin Gururangan, John Quan, Ruan Silva, Ruslan Salakhutdinov, Manzil Zaheer, Sanjeev Arora, and Anirudh Goyal. Rethinking thinking tokens: LLMs as improvement operators. *CoRR*, abs/2510.01123, 2025. doi: 10.48550/ARXIV.2510.01123. URL <https://doi.org/10.48550/arXiv.2510.01123>.
- Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *CoRR*, abs/2601.11868, 2026. doi: 10.48550/ARXIV.2601.11868. URL <https://doi.org/10.48550/arXiv.2601.11868>.
- OpenAI. Introducing chatgpt pro. <https://openai.com/index/introducing-chatgpt-pro/>, December 2024.
- Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/639d992f819c2b40387d4d5170b8ffd7-Abstract-Conference.html.
- Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *CoRR*, abs/2408.03314, 2024. doi: 10.48550/ARXIV.2408.03314. URL <https://doi.org/10.48550/arXiv.2408.03314>.
- 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, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL <https://openreview.net/forum?id=1PL1NIMMrw>.
- Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Improving the efficiency of LLM agent systems through trajectory reduction. *CoRR*, abs/2509.23586, 2025. doi: 10.48550/ARXIV.2509.23586. URL <https://doi.org/10.48550/arXiv.2509.23586>.
- Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Zhihan Guo, Yufei Wang, Irwin King, Xue Liu, and Chen Ma. What, how, where, and how well? A survey on test-time scaling in large language models. *CoRR*, abs/2503.24235, 2025. doi: 10.48550/ARXIV.2503.24235. URL <https://doi.org/10.48550/arXiv.2503.24235>.
- Wenting Zhao, Pranjal Aggarwal, Swarnadeep Saha, Asli Celikyilmaz, Jason Weston, and Ilia Kulikov. The majority is not always right: RL training for solution aggregation. *CoRR*, abs/2509.06870, 2025. doi: 10.48550/ARXIV.2509.06870. URL <https://doi.org/10.48550/arXiv.2509.06870>.
- King Zhu, Hanhao Li, Siwei Wu, Tianshun Xing, Dehua Ma, Xiangru Tang, Minghao Liu, Jian Yang, Jiaheng Liu, Yuchen Eleanor Jiang, Changwang Zhang, Chenghua Lin, Jun Wang, Ge Zhang, and Wangchunshu Zhou. Scaling test-time compute for LLM agents. *CoRR*, abs/2506.12928, 2025. doi: 10.48550/ARXIV.2506.12928. URL <https://doi.org/10.48550/arXiv.2506.12928>.

Yufan Zhuang, Chandan Singh, Liyuan Liu, Yelong Shen, Dinghuai Zhang, Jingbo Shang, Jianfeng Gao, and Weizhu Chen. Test-time recursive thinking: Self-improvement without external feedback. *CoRR*, abs/2602.03094, 2026. doi: 10.48550/ARXIV.2602.03094. URL <https://doi.org/10.48550/arXiv.2602.03094>.

A Rollout statistics

We compute the rollout statistics for all of our models {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash, GPT-5-0825} for our main experiments and share the results in Tables 7 and 8 for SWE-Bench Verified and Terminal-Bench v2.0, respectively.

We measure, for each iteration of our main experiment, (1) the average pass@1 score, (2) the pass@16 score ($N=16$), and (3) the number of tasks which contain mixed passing and failing rollouts for the given iteration. Note that for SWE-Bench Verified we use the bash-only mini-SWE-agent harness, and for Terminal-Bench v2.0 we use the Terminus 1 harness.

Model	SWE-Bench Verified					
	Iteration 0			Iteration 1		
	Avg Pass@1	Pass@16	# Mixed	Avg Pass@1	Pass@16	# Mixed
Claude-4.5-Opus	70.94	85.40	218	76.04	81.20	56
Gemini-3.1-Pro	72.25	86.00	200	76.16	82.00	59
Claude-4.5-Sonnet	67.41	83.40	259	74.01	79.20	71
Gemini-3-Flash	70.79	84.00	251	74.28	79.80	179
GPT-5-0825	61.41	79.00	257	67.73	73.40	96

Table 7: Rollout statistics for SWE-Bench Verified based on our main experiments.

Model	Terminal-Bench v2.0					
	Iteration 0			Iteration 1		
	Avg Pass@1	Pass@16	# Mixed	Avg Pass@1	Pass@16	# Mixed
Claude-4.5-Opus	46.95	70.45	36	52.49	65.91	22
Gemini-3.1-Pro	52.49	76.14	35	56.89	72.73	23
Claude-4.5-Sonnet	40.62	67.05	38	50.00	62.50	19
Gemini-3-Flash	37.93	60.23	37	43.68	56.82	18
GPT-5-0825	31.32	51.14	30	35.30	43.18	10

Table 8: Rollout statistics for Terminal-Bench v2.0 based on our main experiments.

We find that Gemini-3.1-Pro performs the best across both benchmarks, closely followed by Claude-4.5-Opus. Gemini-3-Flash outperforms Claude-4.5-Sonnet in SWE-Bench Verified but Claude-4.5-Sonnet outperforms Gemini-3-Flash in Terminal-Bench v2.0. GPT-5-0825 performs the worst, which is expected given its earlier release date compared to the others.

B Model comparisons

We compare the capabilities of the frontier language models used in our experiments, {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash, GPT-5-0825}, for both SWE-Bench Verified and Terminal-Bench v2.0. To this end, for each pair of models (M_i, M_j) we compute the number of tasks where given $N=16$ parallel rollouts in iteration 0, M_i is able to succeed in at least one of its rollouts while M_j fails all of its rollouts.

Figure 8 visualizes the results of our investigation. We find that Gemini-3.1-Pro is overall the most competitive across both benchmarks in terms of win rates, closely followed by Claude-4.5-Opus. Gemini-3-Flash has slightly better win rates than Claude-4.5-Sonnet on SWE-Bench Verified but does worse than Claude-4.5-Sonnet on Terminal-Bench v2.0. GPT-5-0825 is the least competitive, which is expected given its earlier release date than the other models. Meanwhile, we also observe that smaller models from the same family struggle more to outperform their larger counterparts, such as Gemini-3-Flash winning over Gemini-3.1-Pro for only 10/500 tasks on SWE-Bench Verified and only 1/88 tasks on Terminal-Bench v2.0. We observe similar patterns with Claude-4.5-Sonnet and Claude-4.5-Opus, but not as pronounced as the Gemini models.

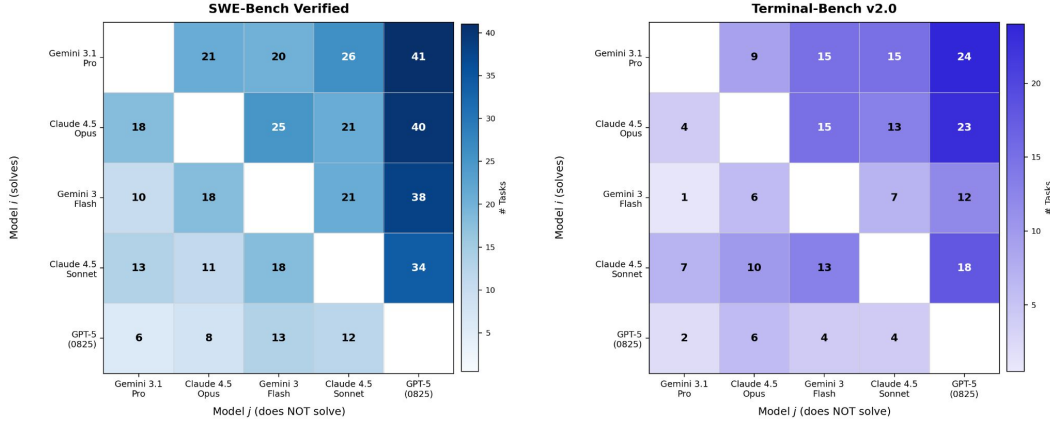


Figure 8: Pairwise model comparisons for iteration 0 with $N=16$ rollouts. Each cell reports the number of tasks for which model M_i produces at least one successful rollout while model M_j fails all of its rollouts (left: SWE-Bench Verified; right: Terminal-Bench v2.0).

C Pass count distributions

We provide more detailed results regarding the pass count distributions for our experiments involving sequential refinement.

C.1 Single vs. Random-K refinement

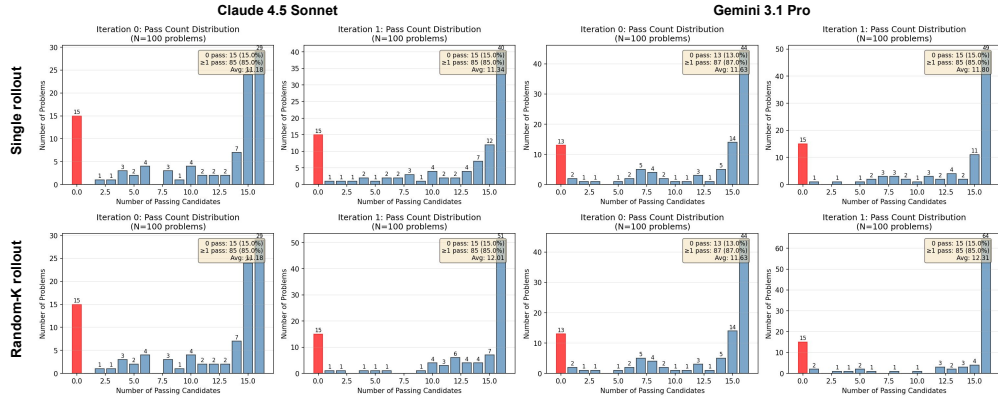


Figure 9: Pass count distributions for iterations 0 & 1 under single-rollout vs. K -rollout refinement for Claude-4.5-Sonnet and Gemini-3.1-Pro, for 100 SWE-Bench Verified tasks.

In Section 3.2.2 we investigate the effect of grounding refinement rollouts based on either a single previous-iteration rollout, or a group of K random previous-iteration rollouts. Figure 9 shows the pass count distributions observed for Claude-4.5-Sonnet and Gemini-3.1-Pro for a subset of 100 randomly-sampled tasks from SWE-Bench Verified.

C.2 Select-K (RTV) refinement

In Section 3.3 we run our end-to-end PDR+RTV using Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash and GPT-5-0825 on both SWE-Bench Verified and Terminal-Bench v2.0. Figures 10 and 11 below show the pass count distributions of the first four models (i.e., excluding GPT-5-0825 for space) for iterations 0 and 1 on SWE-Bench Verified and Terminal-Bench v2.0, respectively.

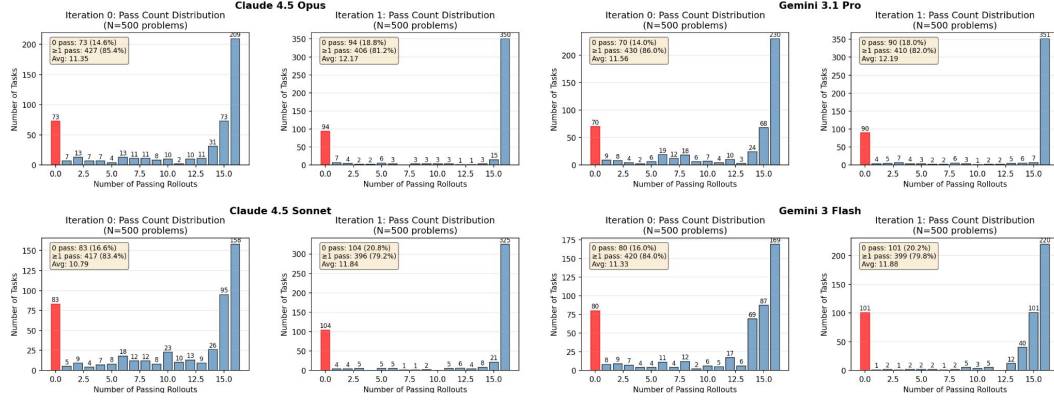


Figure 10: Pass count distributions for iterations 0&1 for {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash} for PDR+RTV, across all SWE-Bench Verified tasks.

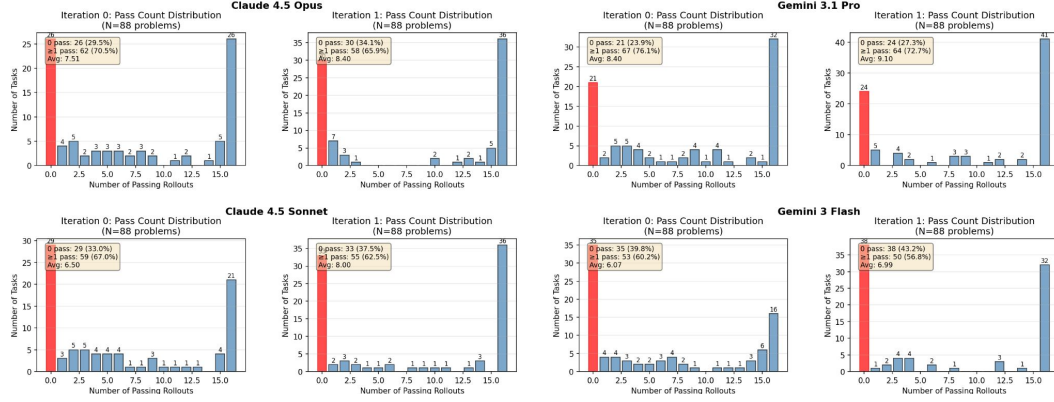


Figure 11: Pass count distributions for iterations 0&1 for {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash} for PDR+RTV, across all Terminal-Bench v2.0 tasks.

D Sequential refinement dynamics

We provide additional results for the sequential refinement analyses performed for our PDR+RTV main experiments, as done in Section 3.4.

D.1 Pass count transition analysis

In Figure 6 of Section 3.4 we visualize the transition matrices of Claude-4.5-Opus and Gemini-3.1-Pro from iteration-0 to iteration-1 rollouts on SWE-Bench Verified and Terminal-Bench v2.0. Note that the mass above the diagonal corresponds to tasks with improvements, while mass below corresponds to tasks with regressions. Figure 12 provides all of the transition matrices spanning Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet and Gemini-3-Flash for both benchmarks.

D.2 Context quality analysis

In Section 3.2.2, Table 2 shows the effect of refinement context management on the performance of the next-iteration rollouts, comparing random- K refinement with select- K (RTV) refinement, by stratifying the tasks by the number of passing rollouts in the context windows. The results in the table show that the select- K refinement yields more refinement contexts with all-passing rollouts, which contributes to the improved average pass@1 score compared to the rollouts executed via random- K refinement.

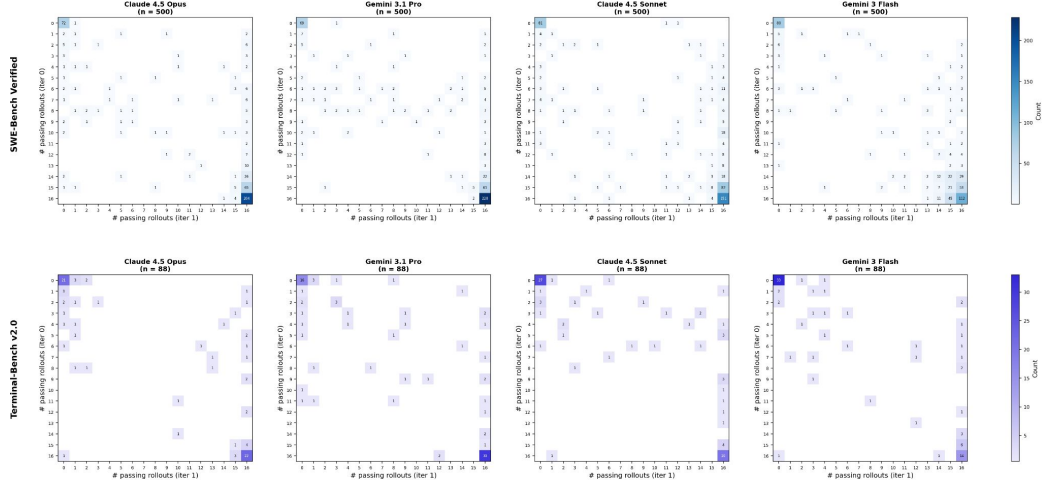


Figure 12: All transition matrices of pass-count transitions from iteration 0 to 1 in our **PDR+RTV** main experiments on SWE-Bench Verified (top row) and Terminal-Bench v2.0 (bottom row), for each model in {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash}. Each cell counts tasks whose number of passing rollouts (out of $N = 16$) changes from the row value (Iter 0) to the column value (Iter 1); the mass above the diagonal indicates net improvement upon refinement, while the mass below indicates regression.

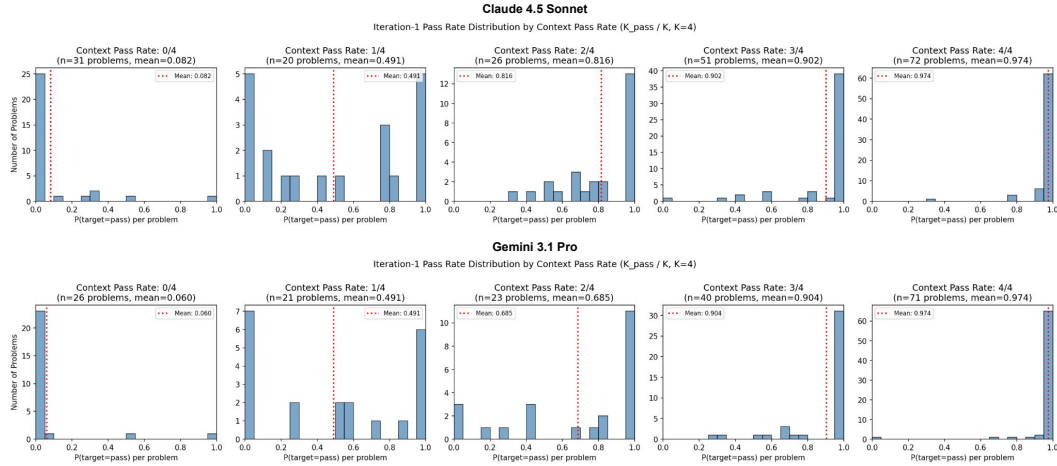


Figure 13: Impact of the average performance of the K rollouts selected for the **PDR** refinement context on the iteration-1 rollout pass rates, executed for 100 tasks from SWE-Bench Verified, based on Claude-4.5-Sonnet and Gemini-3.1-Pro.

In Figure 13 we explore this further and visualize the task-level pass rate distribution of 100 randomly-sampled tasks from SWE-Bench Verified, for Claude-4.5-Sonnet and Gemini-3.1-Pro. Again, for each model we stratify the tasks by the number of passing rollouts in the refinement context, but this time we measure the per-bucket distribution of tasks in terms of their average iteration-1 pass rates.

E Parallel aggregation dynamics

We provide additional results for the parallel aggregation analyses performed for our **PDR+RTV** main experiments, as done in Section 3.5.

E.1 Tournament voting analysis

Figure 7 in Section 3.5 visualizes the evolution of average pass@1 and pass@N scores during the tournament rounds of **RTV** on SWE-Bench Verified, using Claude-4.5-Opus and Gemini-3.1-Pro. The blue lines report the average pass@1 scores across tournament rounds, while the orange lines report the pass@N scores, for both instances of **RTV** performed after iterations 0 and 1. Note that we proceed with iteration-0 **RTV** to the final rollout instead of $K = 4$ for analysis purposes.

Meanwhile, Figures 14 and 15 provide the results for four frontier models – Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet and Gemini-3-Flash – on SWE-Bench Verified and Terminal-Bench v2.0, respectively.

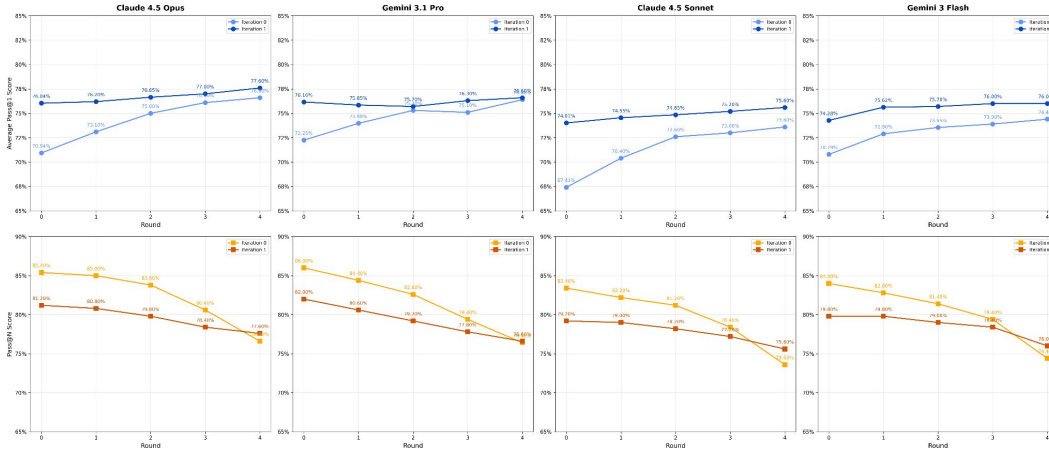


Figure 14: (Top) Average pass@1 scores across **RTV** rounds for both iterations of our **PDR+RTV** method, across {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash}. (Bottom) Pass@N scores across **RTV** rounds for both iterations of our **PDR+RTV** method, across {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash}. All experiments are conducted for SWE-Bench Verified.

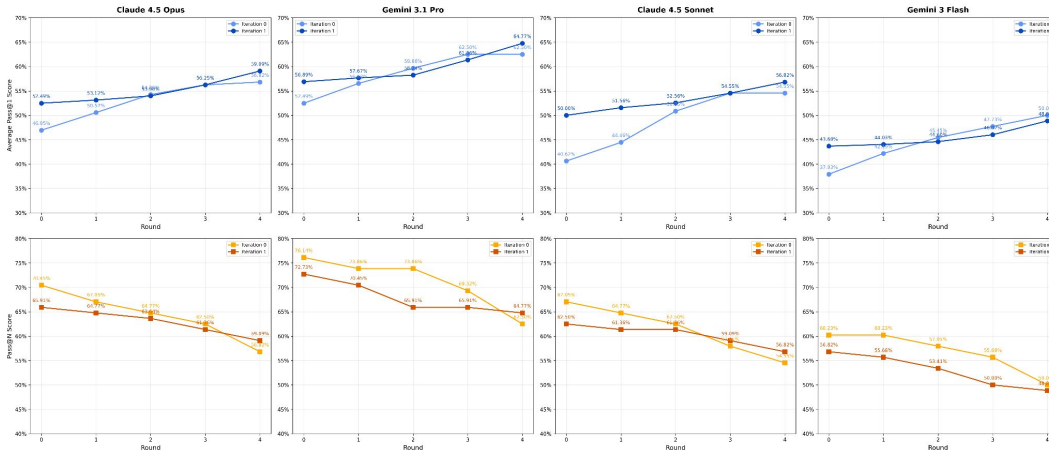


Figure 15: (Top) Average pass@1 scores across **RTV** rounds for both iterations of our **PDR+RTV** method, across {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash}. (Bottom) Pass@N scores across **RTV** rounds for both iterations of our **PDR+RTV** method, across {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash}. All experiments are conducted for Terminal-Bench v2.0.

Iter	Model	SWE-Bench Verified					Terminal-Bench v2.0				
		R0	R1	R2	R3	Avg	R0	R1	R2	R3	Avg
0	Claude-4.5-Opus	69.1	68.3	60.6	55.6	67.0	81.4	78.9	67.9	64.3	77.9
	Gemini-3.1-Pro	66.9	64.5	51.1	65.1	64.5	84.2	81.4	72.2	62.5	80.7
	Claude-4.5-Sonnet	70.6	66.2	53.3	55.6	66.6	80.5	87.0	78.6	62.5	81.7
	Gemini-3-Flash	66.3	55.6	53.3	54.5	61.3	85.1	79.4	80.0	73.7	82.3
1	Claude-4.5-Opus	54.4	62.9	60.0	71.4	58.2	75.4	74.3	84.2	83.3	77.2
	Gemini-3.1-Pro	43.4	47.4	65.8	60.0	48.3	77.4	68.6	85.7	88.9	76.7
	Claude-4.5-Sonnet	63.2	57.5	59.0	60.0	60.9	78.9	77.4	76.2	77.8	78.0
	Gemini-3-Flash	67.3	51.5	55.8	50.0	62.1	77.9	71.1	72.4	90.9	75.8

Table 9: Groupwise selection accuracy (%) across benchmarks, models, iterations, and **RTV** rounds for each Π_{LM} functioning as a LLM-as-a-Judge over the summaries of its own rollouts. Note that the judgment accuracies are *not* directly comparable with each other since the judgments are performed over different summaries of different rollout traces, with significantly different distributions of the input rollouts across the models and benchmarks.

E.2 Groupwise selection analysis

Table 9 provides the results for the analysis of the groupwise selections performed during both iterations of **RTV** in our main experiments. Within each **RTV** and for each tournament round we extract all groups with both successful *and* failing rollouts, and we measure the ratio of groups from which a successful rollout is selected instead of a failing rollout.

Note that the group selection accuracies in Table 9 should not be interpreted as a controlled, head-to-head comparison across judges/models. Each judge is applied to *different* sets of candidate rollouts produced by *different* generator models (i.e., itself), and it makes decisions based on summaries that are themselves generated from those rollouts. As a result, differences in the selection accuracies not only involve the judge’s decision quality, but also (i) the difficulty and diversity of the underlying rollout pools, and (ii) the informativeness of the corresponding summaries.

E.3 Cross-family judge

The selection accuracies in Table 9 are measured with each Π_{LM} judging the summaries of its own rollouts, which leaves open the possibility that **RTV** benefits from a judge scoring rollouts from its own model family. To test this, we hold the policy fixed at Claude-4.5-Opus and swap only the judge on SWE-Bench Verified, keeping the iteration-0 rollouts and their summaries identical across conditions. Unlike Table 9, this comparison is therefore controlled: the candidate pools and the summaries are shared, and only the judge changes. Table 10 reports the results.

Policy	Judge	Selection accuracy	Final pass@1
Claude-4.5-Opus	Claude-4.5-Opus (self)	67.04	76.60
Claude-4.5-Opus	GPT-5-0825	65.05	76.00
Claude-4.5-Opus	GPT-5.5	69.64	77.20

Table 10: Effect of the judge model on **RTV** with the policy fixed at Claude-4.5-Opus on SWE-Bench Verified. Selection accuracy is measured as in Table 9, over the groups that contain both passing and failing rollouts.

The self-judge row reproduces the iteration-0 average of Claude-4.5-Opus in Table 9 (67.0%). Swapping in GPT-5.5 improves both the selection accuracy and the final pass rate, whereas GPT-5-0825 degrades both. Since the capability ordering of these models, consistent with their release order and reported benchmark results, is $GPT-5-0825 < Claude-4.5-Opus < GPT-5.5$, this indicates that **RTV** does not rely on within-family judging: its selection quality tracks the capability of the judge rather than its family, and the judge can therefore be upgraded independently of the policy.

F Token cost breakdown

We break down the token consumption of our **PDR+RTV** main experiments into the three components of our pipeline – *rollout* execution, *summary* generation, and *judgment* for groupwise selection. Tables 11 and 12 report the per-task token counts for SWE-Bench Verified and Terminal-Bench v2.0, respectively, where each cell contains the total count followed by the iteration-0 and iteration-1 counts in parentheses.

Category	Claude-4.5-Opus	Gemini-3.1-Pro	Claude-4.5-Sonnet	Gemini-3-Flash
Rollout	127,324 (82,739 / 44,585)	225,170 (147,248 / 77,922)	217,386 (131,871 / 85,515)	264,637 (182,131 / 82,506)
Summary	20,636 (16,551 / 4,085)	20,481 (15,969 / 4,512)	34,310 (26,575 / 7,735)	25,088 (12,170 / 12,918)
Judgment	2,610,532 (1,561,948 / 1,048,584)	2,627,481 (1,573,926 / 1,053,555)	2,918,547 (1,815,004 / 1,103,543)	2,712,643 (1,533,085 / 1,179,558)

Table 11: Token counts per task for **PDR+RTV** on SWE-Bench Verified, reported as total (iteration 0 / iteration 1).

Category	Claude-4.5-Opus	Gemini-3.1-Pro	Claude-4.5-Sonnet	Gemini-3-Flash
Rollout	217,914 (132,926 / 84,988)	336,625 (233,734 / 102,891)	209,591 (131,963 / 77,628)	197,676 (130,252 / 67,424)
Summary	21,927 (13,099 / 8,828)	21,616 (12,037 / 9,579)	30,761 (18,781 / 11,980)	16,857 (9,964 / 6,893)
Judgment	2,565,446 (1,516,711 / 1,048,735)	2,606,624 (1,546,635 / 1,059,989)	2,871,454 (1,775,444 / 1,096,010)	2,527,602 (1,507,903 / 1,019,699)

Table 12: Token counts per task for **PDR+RTV** on Terminal-Bench v2.0, reported as total (iteration 0 / iteration 1).

Judgment dominates the cost across every model and both benchmarks, accounting for 88–95% of the tokens spent on these three components, while rollouts consume 5–11% and summaries under 2%. This is a direct consequence of the **RTV** configuration rather than of long judgments: with $N=16$, $G=2$ and $V=8$, each **RTV** stage performs $8 + 4 + 2 + 1 = 15$ groupwise comparisons and repeats each one $V=8$ times, so our two **RTV** stages issue 240 judge calls per task while the pipeline executes only 32 rollouts. The number of votes V is therefore the cheapest lever on cost, and Figure 4 (Right) shows that $V=4$ retains most of the gain of $V=8$ at half the judgment cost. We also note that judgments are short and independent, so they are issued in parallel over bounded summaries: they inflate the total token count without increasing the effective context length that the model must handle.

G New solution discoveries

We identify the tasks which the frontier models used in our experiments, {Claude-4.5-Opus, Gemini-3.1-Pro, Claude-4.5-Sonnet, Gemini-3-Flash, GPT-5-0825}, are initially unable to solve across its $N=16$ parallel rollouts, but are able to solve upon performing the select- K refinement during our **PDR+RTV** method. In other words, despite being provided a refinement context with the summaries of all-failing rollouts, the models produce next-iteration rollouts that pass all test cases.

Tables 13 and 14 list the task IDs for each model that discover new solutions upon refinement for SWE-Bench Verified and Terminal-Bench v2.0, respectively.

Model	Task IDs w/ fail → pass
Claude-4.5-Opus	django__django-11951
Gemini-3.1-Pro	sphinx-doc__sphinx-9602
Claude-4.5-Sonnet	django__django-13964, scikit-learn__scikit-learn-25102
Gemini-3-Flash	N/A
GPT-5-0825	pydata__xarray-4687

Table 13: List of tasks in SWE-Bench Verified where each model fails to solve the task in iteration 0 across all $N=16$ parallel rollouts, but discovers a solution in iteration 1 in at least one of the $N = 16$ rollouts (i.e., succeeds after the select- K refinement during **PDR+RTV**).

We observe that despite there being almost six times as many tasks in SWE-Bench Verified compared to Terminal-Bench v2.0, there are 13 tasks with $0 \rightarrow 1$ improvement capabilities in Terminal-Bench v2.0 across all models compared to 5 tasks for SWE-Bench Verified.

Model	Task IDs w/ fail $\rightarrow$ pass
Claude-4.5-Opus	caffe-cifar-10, chess-best-move, gpt2-codegolf, nginx-request-logging, vulnerable-secret
Gemini-3.1-Pro	configure-git-webserver, gcode-to-text, git-leak-recovery, large-scale-text-editing, openssl-selfsigned-cert
Claude-4.5-Sonnet	mcmc-sampling-stan, sparql-university
Gemini-3-Flash	mcmc-sampling-stan, regex-chess
GPT-5-0825	mcmc-sampling-stan, schemelike-metacircular-eval, vulnerable-secret

Table 14: List of tasks in Terminal-Bench v2.0 where each model fails to solve the task in iteration 0 across all $N=16$ parallel rollouts, but discovers a solution in iteration 1 in at least one of the $N = 16$ rollouts (i.e., succeeds after the select- K refinement during **PDR+RTV**).

Moreover, we analyze whether the tasks improving from fail $\rightarrow$ pass from iteration 0 to 1 had already been solved by other models during their iteration-0 rollouts. Tables 15 and 16 show the results of our analysis across all the frontier models on SWE-Bench Verified and Terminal-Bench v2.0, respectively.

Task ID	4.5 Opus	3.1 Pro	4.5 Sonnet	3 Flash	GPT-5
django__django-11951	×	×	✓	✓	✓
sphinx-doc__sphinx-9602	×	×	✓	✓	×
django__django-13964	✓	✓	×	✓	✓
scikit-learn__scikit-learn-25102	✓	×	✓	×	✓
pydata__xarray-4687	✓	✓	✓	✓	×

Table 15: Per-task “new solution discovery” matrix for SWE-Bench Verified indicating which models are already able to solve each task. A ✓ indicates the model fails all iteration-0 rollouts for the task but succeeds in at least one iteration-1 rollout; × indicates it does not.

Task ID	4.5 Opus	3.1 Pro	4.5 Sonnet	3 Flash	GPT-5
caffe-cifar-10	×	✓	×	✓	✓
chess-best-move	×	✓	✓	✓	✓
configure-git-webserver	×	×	✓	×	✓
gcode-to-text	✓	×	×	×	×
git-leak-recovery	×	×	✓	×	×
gpt2-codegolf	×	✓	×	×	×
large-scale-text-editing	×	×	×	×	×
mcmc-sampling-stan	×	✓	×	×	×
nginx-request-logging	×	✓	×	×	×
openssl-selfsigned-cert	✓	×	×	×	×
regex-chess	×	×	✓	×	×
schemelike-metacircular-eval	✓	✓	✓	✓	×
sparql-university	✓	✓	×	✓	×
vulnerable-secret	×	✓	×	×	×

Table 16: Per-task “new solution discovery” matrix for Terminal-Bench v2.0 indicating which models are already able to solve each task. A ✓ indicates the model fails all iteration-0 rollouts for the task but succeeds in at least one iteration-1 rollout; × indicates it does not.

We find that all the Terminal-Bench v2.0 tasks that Claude-4.5-Opus can self-improve are already solved by Gemini-3.1-Pro in at least one of its iteration-0 rollouts. At the same time, all but one of the Terminal-Bench v2.0 tasks that Gemini-3.1-Pro can self-improve are already solved either by Claude-4.5-Opus or Claude-4.5-Sonnet in at least one of their iteration-0 rollouts. Interestingly, the mcmc-sampling-stan task self-improved by Claude-4.5-Sonnet, Gemini-3-Flash and GPT-5-0825 are originally only solved by Gemini-3.1-Pro. Finally, while none of our models are able to solve large-scale-text-editing initially, Gemini-3.1-Pro can leverage **PDR+RTV** to execute a successful rollout during iteration 1.