CAPS: CASCADED ADAPTIVE PAIRWISE SELECTION FOR EFFICIENT PARALLEL REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Parallel reasoning, where a generator samples many candidate solutions and an aggregator selects the best, is one of the most effective forms of test-time scaling in large language models, and pairwise self-verification has become its strongest aggregation primitive. Yet pairwise verification carries a heavy cost: each judgment reads two complete solutions in full, and existing methods perform tens of such judgments per problem regardless of whether the comparison is informative. We introduce **CAPS** (Cascaded Adaptive Pairwise Selection), an inference-only framework that allocates verifier compute non-uniformly along two orthogonal axes: an *evidence axis* that adapts how much of each candidate the judge sees, and a *distribution axis* that adapts how comparisons are spread across the pool. CAPS instantiates these into a four-stage cascade with an optional rescue subroutine, and admits a closed-form verifier-token cost in which the per-candidate marginal cost is roughly halved relative to uniform full-evidence schedules. On four self-verifying models (Qwen3-14B, GPT-OSS-20B, Qwen3-4B-Instruct/Thinking) and five reasoning benchmarks spanning code (LiveCodeBench-v5/v6, CodeContests) and math (AIME 2025, HMMT 2025), CAPS outperforms the leading pairwise verifier on 14 of 20 suites while using 25.4% of its verifier-token budget on code, and outperforms pointwise self-verification on *all* 20. The trade-off suites admit an interpretable diagnostic in terms of the verifier's accuracy at partial versus full evidence, providing a concrete pre-deployment check for cascade suitability.

1 INTRODUCTION

Inference-time scaling gives language models additional computation to solve reasoning problems (Snell et al., 2024; Wei et al., 2022; Wang et al., 2022; Kojima et al., 2022; Snell et al., 2025). This computation can be spent sequentially, through reflection or revision, or in parallel, by sampling multiple candidate solutions and selecting an answer (Lightman et al., 2023; Jaech et al., 2024; Chow et al., 2024). In *parallel reasoning*, additional samples can increase the chance that the pool contains a correct solution (Brown et al., 2024; Snell et al., 2024; Singh et al., 2026). Realizing that benefit also requires a selector that can identify a correct candidate within the pool.

One approach is *self-verification*, in which the generator also judges its candidates (Weng et al., 2023; Huang et al., 2023; Stechly et al., 2024). Pointwise verification assigns each solution an independent quality score (Venkatraman et al., 2025), but these scores can be difficult to calibrate: ratings may saturate, and the model may favor its own incorrect answers (Weng et al., 2023; Zhuang et al., 2024; Stechly et al., 2024; Madaan et al., 2025). Pairwise verification instead asks the model to compare two solutions, reducing reliance on a shared scale of absolute scores. V1-Infer (Singh et al., 2026), for example, uses an uncertainty-guided Swiss tournament (Csató, 2017) to select among candidates. In the $N = 16$, $3\times$-budget configuration evaluated here, it makes approximately 48 full-evidence comparisons.

Repeated comparisons can require the verifier to read the same long solutions many times. Pair-selection schedules control which candidates meet and how often they are compared (Csató, 2017; Hunter, 2004; Singh et al., 2026); the amount of evidence supplied to each comparison offers another way to reduce verification work. Early comparisons could use short excerpts, with full solutions reserved for later rounds among the remaining candidates. This saves input tokens, but risks removing

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

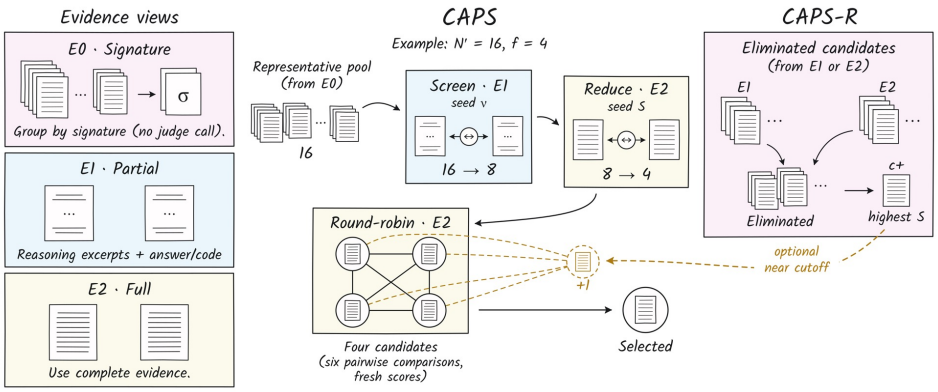


Figure 1: **CAPS overview.** Signature grouping, partial-evidence elimination, full-evidence elimination, and a round-robin among the finalists. Optional rescue allows one eliminated candidate to rejoin the finalist pool.

a correct solution before its full reasoning is examined. We study whether a cascade can reduce repeated full-solution reads while maintaining competitive selection accuracy.

We introduce **CAPS** (Cascaded Adaptive Pairwise Selection), an inference-only framework that allocates verification work through two choices: the evidence shown to the judge and the distribution of comparisons across candidates. CAPS first groups candidates by answer or code signatures, then performs one elimination round using reasoning excerpts together with a code excerpt or final answer. Subsequent elimination rounds and the final round-robin use full solutions. An optional rescue step reintroduces one eliminated candidate based on its accumulated score. We derive the verifier-input cost from the resulting comparison counts and per-call input costs. The procedure concentrates full-evidence comparisons on the surviving pool; its accuracy depends on the judge and on what the partial view retains.

We evaluate four models: Qwen3-14B, Qwen3-4B-Instruct-2507, Qwen3-4B-Thinking-2507 (Yang et al., 2025), and GPT-OSS-20B (Agarwal et al., 2025); on five benchmarks: LiveCodeBench-v5/v6 (Jain et al., 2024), CodeContests (Li et al., 2022), AIME 2025, and HMMT February 2025 (Balunović et al., 2025). Each model serves as both generator and judge. Under the original judge interfaces, CAPS exceeds V1-Infer-3$\times$ on 14 of 20 model–benchmark settings and Pointwise verification on all 20. Averaging the per-setting input-token ratios, CAPS uses 25.4% of V1-Infer-3$\times$'s verifier inputs on code and 50.4% on math. These ratios exclude candidate generation and verifier outputs.

The accuracy gains are concentrated on code. Under both common judge interfaces, the paired 95% confidence intervals for the code accuracy difference remain above zero, while the overall and math intervals include or touch zero. On math, majority voting requires no judge calls, and CAPS does not achieve a statistically significant mean improvement over it. On Qwen3-14B/LiveCodeBench-v6, with the judge interface and comparison count fixed, replacing full evidence with partial evidence in CAPS's first elimination round reduces verifier inputs by 35.9% and Pass@1 by 0.5 percentage points. A deployment test on the same model and benchmark, using the original method interfaces, reduces selection GPU time by 50.8% and achieves $1.92\times$ the throughput of V1-Infer-3$\times$, with smaller gains in single-problem latency.

Contributions.

1. Joint allocation of evidence and comparisons. We study how partial evidence in early comparisons can be combined with a shrinking candidate pool to limit repeated full-solution reads. This design exposes a trade-off between verification inputs and errors introduced by early elimination.

2. A cascaded selector with explicit input-cost analysis. CAPS combines signature grouping, elimination at two evidence levels, and a finalist round-robin, with optional rescue. Its cost analysis accounts for the actual finalist count and the additional comparisons introduced by rescue.

3. Evaluation across four models and five benchmarks. We compare CAPS with pointwise and pairwise verification and with voting on math, and examine judge-interface, comparison-budget, and

evidence-extraction effects. The evaluation reports verifier inputs alongside measured GPU time, throughput, and latency.

2 Related Work

Inference-time scaling encompasses both *sequential* approaches that lengthen a single chain of thought through reflection or revision (Zhang et al., 2024b; Madaan et al., 2023; Qu et al., 2024) and *parallel* approaches that sample multiple independent chains and aggregate their answers (Wei et al., 2022; Wang et al., 2022; Brown et al., 2024; Snell et al., 2024); Self-Consistency (Chen et al., 2023; Wang et al., 2022) and Best-of-N sampling with learned reward models (Lightman et al., 2023; Cobbe et al., 2021) are the canonical aggregation primitives, and the accuracy gap on hard problems motivates stronger selection mechanisms (Brown et al., 2024; Singh et al., 2026). Among these, *self-verification*, which reuses the generator as a judge over its own samples (Weng et al., 2023; Huang et al., 2023; Stechly et al., 2024), has become the dominant primitive in domains with verifiable rewards: pointwise scoring with each candidate rated in isolation suffers from documented calibration failures including score saturation, self-preference bias, and the lack of a globally comparable scale (Stechly et al., 2024; Madaan et al., 2025; Panickssery et al., 2024; Zheng et al., 2023; Venkatraman et al., 2025), and recent work has therefore converged on *pairwise* comparison: V1-Infer (Singh et al., 2026) uses an uncertainty-guided Swiss tournament Csató (2017) to allocate a fixed comparison budget, and Pairwise RM (Liu et al., 2025) trains a dedicated pairwise reward model with a knockout-tournament selection rule. An orthogonal line of work combines candidates rather than selecting among them, ranging from majority voting (Chen et al., 2023) to learned aggregators such as AggLM (Zhao et al., 2025) and Recursive Self-Aggregation (Venkatraman et al., 2025); aggregation methods improve Pass@1 but exhibit diversity collapse, motivating hybrid designs that pair aggregation with verification (Singh et al., 2026). A more detailed discussion is provided in Appendix A.

3 Method

CAPS reduces verification work by using partial evidence for early elimination and full evidence for the remaining comparisons. We describe the selection procedure and count its verifier inputs for a given candidate pool and finalist cap.

3.1 Problem Setup

The selection task. Given a problem q and $N \geq 1$ candidate solutions $\mathcal{C} = \{c_1, \ldots, c_N\}$ sampled from a generator, the selection objective is

$$\hat{\imath} \;\in\; \arg\max_{i \in [N]} \Pr[y_i = 1 \mid q, \mathcal{C}], \tag{1}$$

where $y_i \in \{0, 1\}$ is the unobserved correctness of c_i. A pairwise judge supplies comparisons between candidates (Venkatraman et al., 2025; Singh et al., 2026). In our experiments, the generator also serves as the judge. The pool size N, finalist cap f, and evidence views determine the verification workload.

Evidence map. We expose three views of a candidate:

- $\phi_0(c) \in \Sigma$: a normalized boxed answer for math or a hash of normalized code text, used for signature grouping without a judge call.
- $\phi_1(c) \in \mathcal{V}_1$: the first and last 50 reasoning words, followed by the first 500 code characters for code or the final answer for math.
- $\phi_2(c) = c$: the full solution.

Extraction and normalization are specified in Appendix E.2. We write T_1 and T_2 for representative per-call verifier-input costs and $\rho = T_1/T_2$. Partial evidence saves inputs when $\rho < 1$; the ratio depends on the prompts and candidate lengths.

Pairwise judge. At evidence level $e \in \{1, 2\}$, the judge returns

$$J\big(q, \phi_e(c_i), \phi_e(c_j)\big) \;\longmapsto\; (v_{ij}, w_{ij}), \tag{2}$$

Algorithm 1 CAPS: Cascaded Adaptive Pairwise Selection

Require: problem q, nonempty pool $\mathcal{C}$, judge J, integer cap $f \geq 1$, margin δ
Ensure: selected candidate $c^\star$

$(\mathcal{C}', \nu) \leftarrow \textsc{Dedup}(\mathcal{C}, \phi_0)$ ▷ Stage 0
if $|\mathcal{C}'| = 1$ **then**
  return the sole candidate in $\mathcal{C}'$
end if
$S(c) \leftarrow 0$ for $c \in \mathcal{C}'$; $\mathcal{T} \leftarrow (S, \nu)$
$\mathcal{C}_A \leftarrow \textsc{Eliminate}(q, \mathcal{C}', J, 1, \nu, \mathcal{T})$ ▷ one E1 round
$\mathcal{C}_B \leftarrow \textsc{Eliminate}(q, \mathcal{C}_A, J, 2, S, \mathcal{T}, f)$ ▷ E2; skip if $|\mathcal{C}_A| \leq f$
$F \leftarrow |\mathcal{C}_B|$
$\widetilde{\mathcal{C}}_B \leftarrow \mathcal{C}_B$
if rescue is enabled **then**
  $\widetilde{\mathcal{C}}_B \leftarrow \textsc{Rescue}(\mathcal{C}_B, \mathcal{C}' \setminus \mathcal{C}_B, \delta, \mathcal{T})$
end if
if $|\widetilde{\mathcal{C}}_B| = 1$ **then**
  return the sole candidate in $\widetilde{\mathcal{C}}_B$
end if
for each unordered pair $\{c_i, c_j\} \subset \widetilde{\mathcal{C}}_B$ **do** ▷ Stage C
  Record $(v_{ij}, w_{ij}) \leftarrow J(q, \phi_2(c_i), \phi_2(c_j))$
end for
return $\arg\max_{c \in \widetilde{\mathcal{C}}_B} s_C(c)$ using Eq. equation 4 and its tie-breaks

where $v_{ij} \in \{0, \frac{1}{2}, 1\}$ denotes a loss, tie, or win from i's perspective, and $w_{ij} \in [\tau_w, 1]$ is the confidence weight. We use $\tau_w = 0.05$; the mapping from judge outputs to these quantities appears in Appendix E.3. During elimination, each candidate has a cumulative score $S(c)$, initially zero. Each comparison updates

$$S(c_i) \mathrel{+}= w_{ij} v_{ij}, \qquad S(c_j) \mathrel{+}= w_{ij}(1 - v_{ij}). \tag{3}$$

Low-confidence comparisons contribute less to the score, but an incorrect judgment can still affect subsequent elimination.

3.2 The CAPS Pipeline

CAPS groups candidates by signature (Stage 0), performs one elimination round at E1 (Stage A), reduces the surviving pool at E2 (Stage B), and runs a full-evidence round-robin (Stage C). Algorithm 1 gives the procedure; Appendix B specifies the helpers. We distinguish the chosen cap f from the actual number F of finalists before rescue.

Stage 0: Signature grouping. We group candidates with equal ϕ_0 values and keep one representative per group, yielding $N' \leq N$ candidates. The group size $\nu(c)$ is used for initial seeding and tie-breaking, but is not added to $S(c)$. Equal signatures do not guarantee equal semantics or correctness. On code pools with distinct signatures, this stage leaves the pool unchanged. If no reliable signature is available, each candidate can be assigned a distinct key.

Stage A: Halving at E1. We sort candidates by ν and pair the i-th seed with the $(N' + 1 - i)$-th seed, following slaughter pairing (Sziklai et al., 2022). Each pair is judged at E1; after updating S, the higher-scoring candidate advances, with ties handled as in Appendix B.1.3. An unpaired candidate receives a bye. The round uses $\lfloor N'/2 \rfloor$ calls and leaves $a = \lceil N'/2 \rceil$ candidates. The seeding rule is motivated by separating candidates with different signature frequencies. When all groups are singletons, it provides no ranking information and the initial pairing is distributionally equivalent to random pairing.

Stage B: Reduction at E2. Stage B uses full evidence and seeds each round by the current cumulative score S. It is skipped when $a \leq f$. Otherwise, halving rounds continue while more than f candidates remain. Let F be the resulting count; $F \leq f$, and complete halving can give $F < f$. Each comparison updates S before choosing the survivor, so advancement depends on the accumulated score, not just the latest verdict. For $N' = 16$, $f = 4$, this stage uses one round of four E2 calls.

Stage C: Round-robin at E2. After optional rescue, let $\widetilde{\mathcal{C}}_B$ be the finalist set and $\widetilde{F} = |\widetilde{\mathcal{C}}_B| \in \{F, F+1\}$. We compare every unordered finalist pair once at E2, using $\binom{\widetilde{F}}{2}$ calls. For $\widetilde{F} \geq 2$, the final score is

$$s_C(c) = \frac{\sum_{c' \in \widetilde{\mathcal{C}}_B \setminus \{c\}} w(c, c') v(c, c')}{\sum_{c' \in \widetilde{\mathcal{C}}_B \setminus \{c\}} w(c, c')}, \tag{4}$$

using only Stage-C judgments. Ties are resolved by the pre-Stage-C score S, then ν, then random choice. A single remaining candidate is returned directly. Round-robin compares all finalist pairs (Shreedhar & Varghese, 1996; Fürnkranz, 2002); with a small finalist pool, this requires few full-evidence calls.

3.3 CAPS-R: Rescue of Eliminated Candidates

Early elimination can discard a correct candidate. CAPS-R permits one eliminated candidate to rejoin the finalist pool. If the eliminated set is nonempty, let c^+ be its highest-scoring candidate and $c_{\min}$ the lowest-scoring finalist. Rescue is triggered when

$$\underbrace{|S(c^+) - S(c_{\min})| \leq \delta}_{\text{score proximity}} \quad \text{or} \quad \underbrace{\nu(c^+) = 1 \text{ and } |S(c^+) - S(c_{\min})| \leq 2\delta}_{\text{relaxed threshold for a singleton}}. \tag{5}$$

We use $\delta = 0.15$. The first condition measures proximity to the finalist boundary in cumulative score; the second uses a wider threshold for a singleton. Neither condition identifies correctness. A rescued candidate participates in the Stage-C round-robin, adding F E2 calls. Rescue is skipped when the eliminated set is empty. It is enabled by default in the experiments, but does not guarantee recovery of an early error.

3.4 Verifier-Input Cost Analysis

We first count calls using a common per-call input cost T_e within each evidence level. Stage A uses $\lfloor N'/2 \rfloor$ E1 calls. Stage B removes $a - F$ candidates, one per comparison, and Stage C uses $\binom{F}{2}$ E2 calls without rescue. Thus

$$T_{\text{CAPS}}(N', F) = \lfloor N'/2 \rfloor T_1 + \left[\lceil N'/2 \rceil - F + \binom{F}{2} \right] T_2. \tag{6}$$

This count uses the actual finalist number and also covers pools that do not reach exactly f. For fixed f, with $F \leq f$ and fixed T_1, T_2, it gives

$$T_{\text{CAPS}}(N', F) = \frac{N'}{2}(T_1 + T_2) + O(1). \tag{7}$$

The coefficient describes the leading growth with pool size; individual increments can differ because of rounding and changes in F. Appendix C.2 gives the exact parity correction. Adding one rescued candidate costs FT_2 in this model.

For $N' = 16$, $f = 4$, we have $F = 4$ and, without rescue,

$$T_{\text{CAPS}}(16, 4) = 8T_1 + 4T_2 + 6T_2 = (10 + 8\rho)T_2. \tag{8}$$

At the illustrative value $\rho = 0.12$ used in the parameter cost scan, this is $10.96T_2$. Measured input totals instead sum the actual token lengths of all calls, including cached prefixes. Generation, verifier outputs, and runtime are accounted for separately in Section 4. The analysis counts verification work; it provides no guarantee that early elimination preserves a correct candidate.

4 Experiments

4.1 Experimental Setup

Models and benchmarks. We evaluate Qwen3-14B, Qwen3-4B-Instruct-2507, Qwen3-4B-Thinking-2507 (Yang et al., 2025), and GPT-OSS-20B (Agarwal et al., 2025), using each model as both generator and judge. The code benchmarks are LiveCodeBench-v5 (279 problems), LiveCodeBench-v6 (131 problems) (Jain et al., 2024), and CodeContests (165 problems) (Li et al., 2022). For math, we

use AIME 2025 and HMMT February 2025, each with 30 problems (Balunović et al., 2025). Unless specified otherwise, all methods select from the same pools of $N = 16$ candidates. Code solutions must pass all hidden tests; math solutions are evaluated by final-answer equivalence. Generation settings, prompts, and evaluation details appear in Appendix E.

Methods and interfaces. We compare unaided sampling (**Vanilla**), **Pointwise** scoring, **V1-Infer** (Singh et al., 2026), and **CAPS**. V1-Infer uses its $3\times$ budget, approximately 48 full-evidence calls at $N = 16$. CAPS uses $f = 4$, rescue margin $\delta = 0.15$, and confidence floor $\tau_w = 0.05$, with rescue enabled unless specified otherwise. The original V1-Infer prompt requests two 1–10 ratings; CAPS requests a Winner/Tie verdict with HIGH/LOW confidence. We test both methods with a common interface in Section 4.3. On math, we also compare **majority voting** over normalized final answers and **weighted voting**, which sums pointwise ratings within each answer cluster. Majority voting uses no judge calls; weighted voting requires N pointwise calls.

Table 1: **Pass@1 (%) at $N = 16$.** V1-Infer uses its $3\times$ budget and rating interface; CAPS uses its verdict interface and rescue. Bold marks the highest displayed accuracy in each row. Averages weight the model–benchmark suites equally. Table **??** adds the math voting baselines.

Model	Benchmark	Vanilla	Pointwise	V1-Infer	CAPS
Qwen3-14B	LCB-v5	26.2	29.3	34.8	**37.6**
	LCB-v6	50.7	53.2	58.8	**59.5**
	CodeContests	21.7	24.5	30.9	**35.8**
	AIME 2025	70.4	73.3	76.7	**80.0**
	HMMT 2025	43.6	46.5	**53.3**	50.0
Qwen3-4B-Instruct	LCB-v5	35.3	38.0	42.3	**48.7**
	LCB-v6	29.8	33.0	37.4	**38.9**
	CodeContests	1.3	4.5	7.9	**14.5**
	AIME 2025	66.6	70.0	**76.7**	73.3
	HMMT 2025	47.4	50.0	**56.7**	53.3
Qwen3-4B-Thinking	LCB-v5	36.9	39.8	45.2	**51.6**
	LCB-v6	44.0	46.5	51.9	**52.7**
	CodeContests	31.4	34.1	40.6	**44.8**
	AIME 2025	43.7	46.7	53.3	**56.7**
	HMMT 2025	20.8	23.3	30.0	**33.3**
GPT-OSS-20B	LCB-v5	49.8	52.8	58.4	**61.6**
	LCB-v6	55.5	58.2	**64.1**	62.3
	CodeContests	21.4	24.5	**30.9**	29.4
	AIME 2025	37.4	40.0	46.7	**50.0**
	HMMT 2025	20.4	23.3	**33.3**	30.0
Code average (12 suites)		33.7	36.5	41.9	**44.8**
Math average (8 suites)		43.8	46.6	**53.3**	**53.3**
Overall average (20 suites)		37.7	40.6	46.5	**48.2**

Metrics and aggregation. We report Pass@1 (%) and total verifier input tokens per benchmark. Macro averages weight each model–benchmark suite equally. Paired 95% bootstrap confidence intervals for the CAPS–V1-Infer difference cluster outcomes by benchmark and problem ID, keeping the four models' paired outcomes together. Token counts include cached prefixes and measure logical input length. They exclude generation cost; the deployment experiment reports end-to-end costs separately.

4.2 Main Results and Voting Baselines

Comparison with V1-Infer. CAPS exceeds V1-Infer on 14 of 20 suites and Pointwise on all 20 (Table 1). With the original judge interfaces, the mean gain over V1-Infer is 1.7 percentage points (pp), with a paired 95% confidence interval of $[0.3, 3.1]$. The gain is concentrated on code: 2.9pp, $[1.1, 4.6]$, versus approximately zero on math, $[-2.9, 2.8]$. V1-Infer performs better on two GPT-OSS code suites and four math suites. None of the individual 30-problem math comparisons is significant after Holm correction.

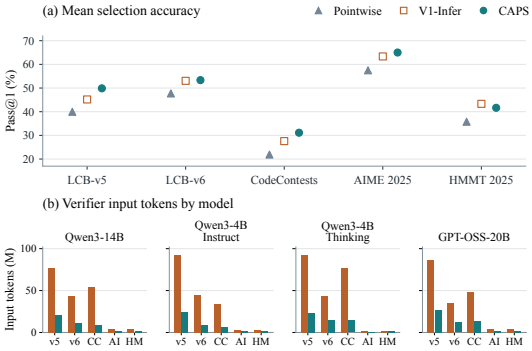


Figure 2: **Accuracy and verifier-input cost at $N = 16$.** (a) Pass@1 averaged equally over the four models for each benchmark, calculated from the displayed values in Table 1. (b) Benchmark-total logical verifier input tokens for each model.

Figure 2 summarizes mean accuracy across the four models and verifier-input cost for each model–benchmark setting. CAPS has higher mean accuracy than V1-Infer on the three code benchmarks and AIME, but lower mean accuracy on HMMT. It uses fewer verifier input tokens in all 20 settings. The code gains vary across benchmarks: CAPS improves on V1-Infer for all four models on LCB-v5, whereas the mean difference on LCB-v6 is only 0.3pp.

Comparison with voting. On the same math candidate pools, CAPS beats majority voting in five of eight settings, ties in one, and loses in two (Table 2). The mean differences are not statistically significant. Weighted voting narrows the gap further, although it requires pointwise scoring. Majority voting is therefore a strong baseline on these tasks, with no verification cost. Results for each model appear in Appendix D.2.

4.3 Controlling the Judge Interface and Selection Budget

Common-interface comparison. With a common rating prompt, the overall CAPS–V1-Infer gap falls from 1.7 to 0.9pp; with a common verdict prompt, it is 1.3pp. Both overall intervals include zero at the reported precision. The code intervals remain positive under either interface, while the math intervals span zero (Table 2, bottom). Prompt choice accounts for part of the original gap, but the code advantage persists. Appendix D.3 gives the full 2×2 comparison. These interface controls retain each method's comparison budget. The following experiment additionally matches the number of calls, allowing a comparison of the selection procedures with full evidence.

Fixed interface and comparison count. On Qwen3-14B/LCB-v6, we fix the candidate pools, seeds, judge model, rating prompt, and comparison count. V1-Infer stops after the number of comparisons used by CAPS on each problem, averaging 18.5 calls including rescue. With full evidence, CAPS reaches 59.1 Pass@1 and V1-Infer reaches 56.0 (Table 2, top). Using partial evidence in CAPS's first elimination round reduces accuracy to 58.6 and input tokens from 17.0M to 10.9M. In this setting, E1 saves 35.9% of verifier inputs at a cost of 0.5pp. The full-evidence comparison tests the selection procedures as a whole, including their aggregation rules.

Table 2: **Judge-interface and comparison-budget controls.** Top: Qwen3-14B/LCB-v6 at $N = 16$, with a common rating prompt and 18.5 mean calls per problem. Bottom: CAPS–V1-Infer differences in pp, averaged over the indicated suites, with paired 95% confidence intervals. The two panels use separate experimental protocols.

Selection procedure	Evidence	Input (M)	Pass@1
V1-Infer	All E2	17.0	56.0
CAPS	All E2	17.0	59.1
CAPS	E1 then E2	10.9	58.6

Judge interfaces	Overall (20)	Code (12)	Math (8)
Original	$+1.7\ [0.3, 3.1]$	$+2.9\ [1.1, 4.6]$	$0.0\ [-2.9, 2.8]$
Common rating	$+0.9\ [-0.3, 2.1]$	$+2.1\ [0.5, 3.7]$	$-0.9\ [-3.4, 1.6]$
Common verdict	$+1.3\ [0.0, 2.6]$	$+2.5\ [0.8, 4.1]$	$-0.5\ [-3.0, 2.0]$

Verification cost at fixed N. CAPS uses an average of 25.4% of V1-Infer-3×'s verifier input tokens on code and 50.4% on math, averaging the per-suite ratios. Figure 3(a) compares the measured configurations on Qwen3-14B/LCB-v6 at $N = 16$. Default CAPS reaches 59.5 Pass@1 with 11.2M input tokens, exceeding all three tested V1-Infer budgets at lower input cost. Increasing the finalist cap from $f = 4$ to $f = 8$ raises Pass@1 to 60.2 but increases inputs to 29.6M. These comparisons use each method's original judge interface; Table 7 gives the complete measurements.

Increasing the candidate pool. Figure 3(b) varies N while keeping CAPS at $f = 4$ and V1-Infer at its 3× budget. CAPS reaches 63.4 Pass@1 at $N = 32$ with 24.5M verifier input tokens, and 65.0 at $N = 64$ with 51.2M. At $N = 32$, it exceeds V1-Infer-3× at $N = 16$ by 4.6pp while using about 44% fewer verifier input tokens. The larger candidate pool incurs additional generation cost, which is excluded from this comparison. At $N = 32$, $f = 4$, CAPS also exceeds the accuracy at $N = 16$, $f = 8$ while using fewer verifier inputs. In this comparison, expanding the pool is more effective than retaining more finalists, although it requires additional candidate generation.

Table 3: **Runtime on Qwen3-14B/LCB-v6, $N = 16$.** Both methods use the same hardware and candidate pools. Token totals cover all 131 problems; GPU time and latency are per problem. Input counts include cached prefixes.

Metric	V1-Infer-3×	CAPS
Mean judge calls/problem	48.0	18.5
Verifier input tokens (M)	44.0	11.2
Verifier output tokens (M)	2.8	1.2
Selection GPU-seconds/problem	18.7	9.2
Selection latency P50 / P95 (s)	12.6 / 20.4	10.9 / 18.1
Steady-state throughput (selections/min)	6.0	11.5
End-to-end GPU-seconds/problem	68.8	59.3
End-to-end latency P50 / P95 (s)	49.4 / 78.6	47.7 / 76.4

Table 4: **Runtime on Qwen3-14B/LCB-v6,** $N = 16$**.** Both methods use the same hardware and candidate pools. Token totals cover all 131 problems; GPU time and latency are per problem. Input counts include cached prefixes.

Metric	V1-Infer-3×	CAPS
Mean judge calls/problem	48.0	18.5
Verifier input tokens (M)	44.0	11.2
Verifier output tokens (M)	2.8	1.2
Selection GPU-seconds/problem	18.7	9.2
Selection latency P50 / P95 (s)	12.6 / 20.4	10.9 / 18.1
Steady-state throughput (selections/min)	6.0	11.5
End-to-end GPU-seconds/problem	68.8	59.3
End-to-end latency P50 / P95 (s)	49.4 / 78.6	47.7 / 76.4

4.4 Deployment Cost, Throughput, and Latency

We measure runtime on the same 131 Qwen3-14B/LCB-v6 candidate pools over three runs, using two A100-80GB GPUs, SGLang, BF16, chunked prefill, warm prefix caching, and fixed concurrency. Latency includes queueing, retries, parsing, and stage barriers. Both methods retain their original judge interfaces.

CAPS reduces verifier output tokens by 57.1% and selection GPU-seconds by 50.8% (Table 4). Throughput rises from 6.0 to 11.5 selections/minute (1.92×), while selection P50 and P95 latency fall by 13.5% and 11.3%. The smaller latency gain reflects the dependencies between cascade stages. Including generation, the reductions are 13.8% in GPU-seconds and 3.4% in median latency. The shared generation workload is unchanged, so savings in selection account for a smaller fraction of end-to-end GPU time.

Where the verification budget is spent. Figure 5 illustrates the cost decomposition in Eq. equation 8. With $N' = 16$ and $F = 4$, both All-E2 and CAPS make 18 judge calls: eight for screening, four for reduction, and six for the final round-robin. The savings come from shortening the screening inputs, yielding a reduction of $8(T_2 - T_1)$. At the illustrative ratio $\rho = 0.12$, this reduces verifier-input cost by 39.1% relative to All-E2, while the later stages retain their full-evidence comparison counts. If rescue adds one candidate, the finalist pool grows from four to five, increasing the round-robin from six to ten comparisons. The resulting CAPS-R cost is $14.96T_2$, or 16.9% below the All-E2 baseline. This last value is conditional on rescue firing; the average overhead across problems depends on the rescue activation rate.

4.5 Sensitivity and Applicability

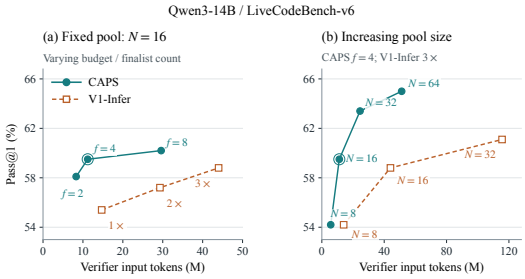


Figure 3: **Verifier-input cost and accuracy on Qwen3-14B/LiveCodeBench-v6.** (a) Fixed $N = 16$, varying V1-Infer's comparison budget and CAPS's finalist cap. (b) Varying N, with CAPS $f = 4$ and V1-Infer's 3× budget. Both panels use the original method-specific judge interfaces.

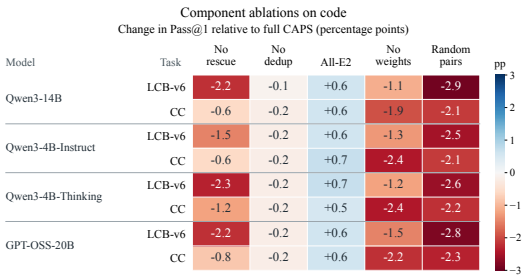


Figure 4: **Component ablations on code.** Each cell gives the change in Pass@1, in percentage points, from full CAPS, calculated from the displayed absolute accuracies in Table 10. Red indicates lower accuracy and blue higher accuracy under the original CAPS verdict interface.

Component ablations. Figure 4 compares five component variants across eight code settings. Removing rescue or confidence weighting lowers accuracy in every setting, by 0.6–2.3pp and 1.1–2.4pp, respectively. These declines exceed the 0.1–0.2pp change from removing deduplication. All-E2 gains 0.5–0.7pp, so the default partial view incurs a small accuracy loss in these ablations; its input-cost trade-off is examined separately in Table 2. Random pairing lowers accuracy by 2.1–2.9pp, but changes the pairing configuration as a whole and does not isolate cluster-size seeding. Table 10 reports the absolute accuracies.

Evidence extraction. The default E1 view contains the first and last 50 reasoning words, followed by the first 500 code characters for code or the final answer for math. On Qwen3-4B-Thinking/LCB-v6, a thinking-aware extractor improves Pass@1 from 52.7 to 54.2 while reducing inputs from 14.7M to 13.5M. This variant retains the final answer when present and the last approximately 500 tokens. All-E2 reaches 53.4 at 16.7M. Across four models on LCB-v6, code windows of 250, 500, and 1,000 characters give a mean accuracy spread of 0.6pp and a maximum of 1.1pp. Appendix D.5 gives the extractor comparisons. All-E2 is not the most accurate evidence configuration on Qwen3-4B-Thinking/LCB-v6. On math, answer-only views score 1.8pp below views retaining terminal reasoning on average, while 300- and 500-token windows differ by 0.7pp. With only 30 problems per benchmark, these are sensitivity observations rather than evidence that the two window lengths are equivalent.

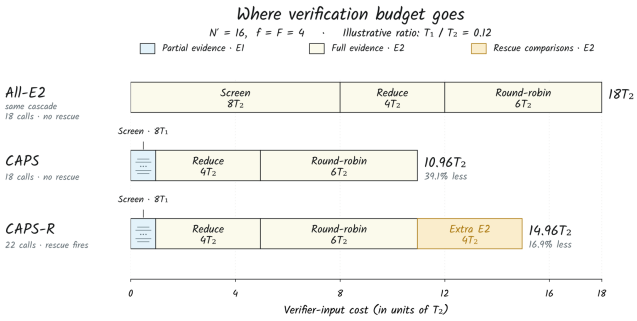


Figure 5: **Verifier-input cost decomposition.** For $N' = 16$ and $F = f = 4$, we compare All-E2, which uses full evidence throughout the same cascade without rescue, with CAPS and CAPS-R. Bar lengths follow the illustrative per-call cost ratio $\rho = T_1/T_2 = 0.12$. CAPS replaces the eight E2 screening calls with E1 calls, reducing the total cost from $18T_2$ to $10.96T_2$. When rescue fires, four additional Stage-C comparisons increase the cost to $14.96T_2$. Blue, pale yellow, and gold denote E1 inputs, E2 inputs, and the additional E2 comparisons induced by rescue, respectively.

Cascade choices. On Qwen3-14B, the mean Pass@1 over LCB-v6, CodeContests, AIME, and HMMT is 55.2, 56.3, and 56.8 for $f = 2, 4, 8$. The analytical cost ratios are $0.73\times$, $1.00\times$, and $2.64\times$ at $N' = 16$ and $\rho = 0.12$, excluding rescue. Increasing f from 4 to 8 therefore adds 0.5pp at more than twice the analytical cost. Changing the rescue margin within 0.10–0.20 gives 56.0–56.3, while confidence floors within 0.02–0.10 give 56.2–56.3. Full scans and component ablations appear in Appendices D.6 and D.7.

Rescue sensitivity. In the same four-task scan, disabling rescue lowers mean Pass@1 from 56.3 to 54.9. Rescuing two or four candidates gives 56.5 or 56.4, but raises reported relative input cost to $1.06\times$ or $1.23\times$ the one-candidate default. Increasing δ from 0.15 to 0.30 doubles the trigger rate from 12% to 24%, while accuracy falls to 55.9 (Appendix D.6). Rescuing more candidates therefore yields little further improvement in this scan. These measurements do not establish that rescue preserves a correct candidate in much larger pools with sparse correct solutions.

5 Conclusion

In this work, we introduce **CAPS**, an inference-only framework for pairwise self-verification that allocates verifier compute non-uniformly along two orthogonal axes: an *evidence axis* that adapts how much of each candidate the judge sees, and a *distribution axis* that concentrates full-evidence comparisons on the strongest candidates; through a four-stage cascade with an optional rescue subroutine. We derive a closed-form characterization of CAPS's verifier-token cost showing that the per-candidate marginal cost is roughly halved relative to uniform full-evidence schedules, and our experiments confirm that the empirical efficiency is structural rather than a hyperparameter effect. Across four self-verifying models and five reasoning benchmarks, CAPS outperforms V1-Infer at $3\times$ budget on 14 of 20 settings while using a fraction of its verifier-token cost.

AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic data sets, design or provide feedback on research methodology or experiments, implement methods, assist with translation, and clean and reformat dataset, edit a research paper to improve readability. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, support qualitative and thematic data analysis, or interpret results.

Additionally, we used generative AI tools to create or modify scientific figures or images and create or edit software code. We have reviewed all AI-assisted work. All LLM-generated code was thoroughly tested and verified by three authors. Additionally, all formatted tables within the manuscript were manually cross-validated against the raw data by two authors to guarantee accuracy. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions. *arXiv preprint arXiv:2505.23281*, 2025.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Xinyun Chen, Renat Aksitov, Uri Alon, Jie Ren, Kefan Xiao, Pengcheng Yin, Sushant Prakash, Charles Sutton, Xuezhi Wang, and Denny Zhou. Universal self-consistency for large language model generation. *arXiv preprint arXiv:2311.17311*, 2023.

Yinlam Chow, Guy Tennenholtz, Izzeddin Gur, Vincent Zhuang, Bo Dai, Sridhar Thiagarajan, Craig Boutilier, Rishabh Agarwal, Aviral Kumar, and Aleksandra Faust. Inference-aware fine-tuning for best-of-n sampling in large language models. *arXiv preprint arXiv:2412.15287*, 2024.

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

László Csató. On the ranking of a swiss system chess team tournament. *Annals of Operations Research*, 254(1):17–36, 2017.

Zilin Dai, Lehong Wang, Fangzhou Lin, Yidong Wang, Zhigang Li, Kazunori D Yamada, Ziming Zhang, and Wang Lu. A language anchor-guided method for robust noisy domain generalization. *arXiv preprint arXiv:2503.17211*, 2025.

Johannes Fürnkranz. Round robin classification. *Journal of Machine Learning Research*, 2(Mar): 721–747, 2002.

Jiaxin Huang, Shixiang Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 1051–1068, 2023.

David R Hunter. Mm algorithms for generalized bradley-terry models. *The annals of statistics*, 32(1): 384–406, 2004.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Lingyu Jiang, Yuping Wang, Yao Su, Shuo Xing, Wenjing Chen, Xin Zhang, Zhengzhong Tu, Ziming Zhang, Fangzhou Lin, Michael Zielewski, et al. Kanmixer: Can kan serve as a new modeling core for long-term time series forecasting? *arXiv preprint arXiv:2508.01575*, 2025a.

Lingyu Jiang, Lingyu Xu, Peiran Li, Qianwen Ge, Dingyi Zhuang, Shuo Xing, Wenjing Chen, Xiangbo Gao, Ting-Hsuan Chen, Xueying Zhan, et al. Timepre: Bridging accuracy, efficiency, and stability in probabilistic time-series forecasting. *arXiv preprint arXiv:2511.18539*, 2025b.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

Peiran Li, Fangzhou Lin, Shuo Xing, Jiashuo Sun, Dylan Zhang, Siyuan Yang, Chaoqun Ni, and Zhengzhong Tu. Let the abyss stare back adaptive falsification for autonomous scientific discovery. *arXiv preprint arXiv:2603.29045*, 2026a.

Peiran Li, Fangzhou Lin, Shuo Xing, Xiang Zheng, Xi Hong, Siyuan Yang, Jiashuo Sun, Zhengzhong Tu, and Chaoqun Ni. Bibagent: An agentic framework for traceable miscitation detection in scientific literature. *arXiv preprint arXiv:2601.16993*, 2026b.

Peiran Li, Jiashuo Sun, Fangzhou Lin, Shuo Xing, Tianfu Fu, Suofei Feng, Chaoqun Ni, and Zhengzhong Tu. Traversal-as-policy: Log-distilled gated behavior trees as externalized, verifiable policies for safe, robust, and efficient agents. *arXiv preprint arXiv:2603.05517*, 2026c.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

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 *The twelfth international conference on learning representations*, 2023.

Fangzhou Lin, Qianwen Ge, Lingyu Xu, Peiran Li, Xiangbo Gao, Shuo Xing, Kazunori Yamada, Ziming Zhang, Haichong Zhang, and Zhengzhong Tu. Position: Human-centric ai requires a minimum viable level of human understanding. *arXiv preprint arXiv:2602.00854*, 2026a.

Fangzhou Lin, Peiran Li, Shuo Xing, Siyuan Yang, Qianwen Ge, Kazunori Yamada, Ziming Zhang, Haichong Zhang, and Zhengzhong Tu. Adaptfuse: Training-free sequential preference learning via externalized bayesian inference. *arXiv preprint arXiv:2604.03925*, 2026b.

Yantao Liu, Zijun Yao, Rui Min, Yixin Cao, Lei Hou, and Juanzi Li. Pairwise rm: Perform best-of-n sampling with knockout tournament. *arXiv e-prints*, pp. arXiv–2501, 2025.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

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. *arXiv preprint arXiv:2510.01123*, 2025.

Jiayi Pan, Xiuyu Li, Long Lian, Charlie Snell, Yifei Zhou, Adam Yala, Trevor Darrell, Kurt Keutzer, and Alane Suhr. Learning adaptive parallel reasoning with language models. *arXiv preprint arXiv:2504.15466*, 2025.

Rui Pan, Shuo Xing, Shizhe Diao, Wenhe Sun, Xiang Liu, KaShun Shum, Jipeng Zhang, Renjie Pi, and Tong Zhang. Plum: Prompt learning using metaheuristics. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 2177–2197, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.129. URL https://aclanthology.org/2024.findings-acl.129/.

Arjun Panickssery, Samuel R Bowman, and Shi Feng. Llm evaluators recognize and favor their own generations. *Advances in Neural Information Processing Systems*, 37:68772–68802, 2024.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve. *Advances in Neural Information Processing Systems*, 37:55249–55285, 2024.

Amrith Setlur, Nived Rajaraman, Sergey Levine, and Aviral Kumar. Scaling test-time compute without verification or rl is suboptimal. *arXiv preprint arXiv:2502.12118*, 2025.

Madhavapeddi Shreedhar and George Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, 4(3):375–385, 1996.

Harman Singh, Xiuyu Li, Kusha Sareen, Monishwaran Maheswaran, Sijun Tan, Xiaoxia Wu, Junxiong Wang, Alpay Ariyak, Qingyang Wu, Samir Khaki, et al. v_1: Unifying generation and self-verification for parallel reasoners. *arXiv preprint arXiv:2603.04304*, 2026.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. On the self-verification limitations of large language models on reasoning and planning tasks. *arXiv preprint arXiv:2402.08115*, 2024.

Balázs R Sziklai, Péter Biró, and László Csató. The efficacy of tournament designs. *Computers & Operations Research*, 144:105821, 2022.

Amir Taubenfeld, Tom Sheffer, Eran Ofek, Amir Feder, Ariel Goldstein, Zorik Gekhman, and Gal Yona. Confidence improves self-consistency in llms. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 20090–20111, 2025.

Siddarth Venkatraman, Vineet Jain, Sarthak Mittal, Vedant Shah, Johan Obando-Ceron, Yoshua Bengio, Brian R Bartoldson, Bhavya Kailkhura, Guillaume Lajoie, Glen Berseth, et al. Recursive self-aggregation unlocks deep thinking in large language models. *arXiv preprint arXiv:2509.26626*, 2025.

Xinglin Wang, Shaoxiong Feng, Yiwei Li, Peiwen Yuan, Yueqi Zhang, Chuyi Tan, Boyuan Pan, Yao Hu, and Kan Li. Make every penny count: Difficulty-adaptive self-consistency for cost-efficient reasoning. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 6904–6917, 2025a.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Xuezhi Wang, Jason Wei, Dale Schuurmans, and Quoc V Le. 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*, volume 1, pp. 2, 2023.

Ziqi Wang, Boye Niu, Zipeng Gao, Zhi Zheng, Tong Xu, Linghui Meng, Zhongli Li, Jing Liu, Yilong Chen, Chen Zhu, et al. A survey on parallel reasoning. *arXiv preprint arXiv:2510.12164*, 2025b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. Large language models are better reasoners with self-verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 2550–2575, 2023.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yun Yue, Fangzhou Lin, Guanyi Mou, and Ziming Zhang. Understanding hyperbolic metric learning through hard negative sampling. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 1891–1903, 2024.

Zhiyuan Zeng, Qinyuan Cheng, Zhangyue Yin, Yunhua Zhou, and Xipeng Qiu. Revisiting the test-time scaling of o1-like models: Do they truly possess test-time scaling capabilities? In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4651–4665, 2025.

Xuan Zhang, Chao Du, Tianyu Pang, Qian Liu, Wei Gao, and Min Lin. Chain of preference optimization: Improving chain-of-thought reasoning in llms. *Advances in Neural Information Processing Systems*, 37:333–356, 2024a.

Yusen Zhang, Ruoxi Sun, Yanfei Chen, Tomas Pfister, Rui Zhang, and Sercan Ö Arık. Chain of agents: Large language models collaborating on long-context tasks. *Advances in Neural Information Processing Systems*, 37:132208–132237, 2024b.

Ziming Zhang, Yuping Shao, Yiqing Zhang, Fangzhou Lin, Haichong Zhang, and Elke Rundensteiner. Deep loss convexification for learning iterative models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 47(3):1501–1513, 2024c.

Ziming Zhang, Fangzhou Lin, Haotian Liu, Jose Morales, Haichong Zhang, Kazunori Yamada, Vijaya B Kolachalama, and Venkatesh Saligrama. Gps: A probabilistic distributional similarity with gumbel priors for set-to-set matching. In *The Thirteenth International Conference on Learning Representations*, 2025.

Wenting Zhao, Pranjal Aggarwal, Swarnadeep Saha, Asli Celikyilmaz, Jason Weston, and Ilia Kulikov. The majority is not always right: Rl training for solution aggregation. *arXiv preprint arXiv:2509.06870*, 2025.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

Shengyao Zhuang, Honglei Zhuang, Bevan Koopman, and Guido Zuccon. A setwise approach for effective and highly efficient zero-shot ranking with large language models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 38–47, 2024.

A Extended Related Work

This appendix expands on the related-work overview from Section 2 of the main text, providing additional context for parallel reasoning, test-time scaling, self-verification, and self-aggregation.

Parallel reasoning. Parallel reasoning in LLMs generates multiple independent solution attempts to a problem and selects or combines among them, in contrast to sequential approaches that produce a single chain of thought, possibly with refinement (Wei et al., 2022; Wang et al., 2022; Kojima et al., 2022; Snell et al., 2025; Jiang et al., 2025a; Lin et al., 2026a; Jiang et al., 2025b; Pan et al., 2024). The basic premise dates to early ensemble decoding (Li et al., 2022; Chen et al., 2023; Wang et al., 2023; Yue et al., 2024; Zhang et al., 2025) and has been refined through methods that exploit the diversity of independent samples to cover a larger fraction of the problem's solution space than any single chain can (Brown et al., 2024; Wang et al., 2025b; Zhang et al., 2024c). Self-Consistency (Chen et al., 2023; Wang et al., 2025a; Taubenfeld et al., 2025) demonstrated that majority voting over independently sampled chains substantially improves arithmetic reasoning. Best-of-N sampling with learned reward models (Lightman et al., 2023; Jaech et al., 2024; Chow et al., 2024; Li et al., 2026c) extends this to settings without an objective ground truth answer. More recent work has shown that the gap between Pass@1 and Pass@N, the probability that the correct answer exists somewhere in the candidate pool, remains large for hard problems even at moderate N (Brown et al., 2024; Snell et al., 2024; Singh et al., 2026), motivating the development of stronger selection mechanisms that close this gap.

Test-time scaling. A complementary line of work treats inference-time compute as a continuous axis along which model performance can be improved without changing model weights (Zeng et al., 2025; Snell et al., 2025; Lin et al., 2026b; Li et al., 2026a). Two families have emerged: *sequential scaling*, in which a model performs longer chains of reasoning, reflection, or revision (Zhang et al., 2024b;a; Madaan et al., 2023; Qu et al., 2024; Dai et al., 2025), and *parallel scaling*, in which multiple independent attempts are aggregated (Cobbe et al., 2021; Pan et al., 2025; Setlur et al., 2025; Li et al., 2026b). The two are largely orthogonal and can be combined (Snell et al., 2024; Brown et al., 2024). Empirical scaling laws relate test-time compute to performance, and Snell et al. (Snell et al., 2024) showed that, for a fixed compute budget, the optimal allocation between train-time and test-time compute is task-dependent and often favors the latter for hard problems. Within parallel scaling, the principal compute budget consists of two components: the cost of generation (sampling N candidates) and the cost of selection (verifying them). The relative weight of the two has shifted as generation has become cheaper through speculative decoding, batch inference, and small-model speedups, with verification cost now frequently dominating the test-time budget for verifiable-reward settings such as code and math.

Self-verification. Self-verification methods use the same model that generated the candidates to assess their correctness, avoiding the need for a separately trained reward model or external verifier (Weng et al., 2023; Huang et al., 2023; Stechly et al., 2024). Early work used pointwise scoring, prompting the model to assign an absolute quality score to each candidate (Weng et al., 2023; Zhuang et al., 2024). Recent studies have systematic failure modes of pointwise self-verification: scores lack a globally comparable scale (Stechly et al., 2024; Madaan et al., 2025; Venkatraman et al., 2025), models exhibit a self-preference bias toward their own samples even when those samples are incorrect (Panickssery et al., 2024; Zheng et al., 2023), and verifiers saturate at the top of the scale, assigning near-maximum scores to most candidates and losing discriminative power. The current state of the art is V1-Infer (Singh et al., 2026), which replaces pointwise scoring with *pairwise* comparison: candidates are compared head-to-head, and an uncertainty-guided Swiss-system tournament dynamically allocates the comparison budget to the most ambiguous pairs. V1-Infer reports up to $+10\%$ Pass@1 over pointwise verification on code and math benchmarks, and the same paper introduces V1-PairRL, an RL framework that co-trains a single model as both generator and pairwise verifier so that the verifier remains in-distribution as the generator evolves. Pairwise RM (Liu et al., 2025) similarly trains a dedicated pairwise reward model and applies a knockout tournament with answer-based grouping for Best-of-N selection ().

Self-aggregation. Self-aggregation methods take a different approach to using a candidate pool: rather than selecting one of the existing candidates, they prompt the model to combine or refine the

pool into a new solution (Madaan et al., 2025; Venkatraman et al., 2025). Majority voting (Chen et al., 2023) is the simplest instance, applicable when the answer admits an exact-match comparison. Beyond majority, recent work has explored learned aggregation: AggLM (Zhao et al., 2025) trains an aggregator via reinforcement learning to synthesize a final answer from a pool, and Recursive Self-Aggregation (RSA) (Venkatraman et al., 2025) iteratively consolidates groups of k candidates into single solutions over multiple rounds, producing an evolutionary refinement loop. RSA and similar methods improve Pass@1 in many settings but exhibit *diversity collapse*: Pass@N monotonically decreases with aggregation steps as correct outlier solutions are absorbed into majority-aligned refinements (Venkatraman et al., 2025; Singh et al., 2026). This has motivated hybrid designs that combine aggregation with verification, using the latter as a fitness signal that preserves Pass@N across aggregation rounds. The two paradigms, selection-based and aggregation-based, are largely orthogonal and can be combined when verification is reliable.

B Method: Helper Procedures and Implementation Details

This appendix specifies the helper procedures in Algorithm 1, the evidence views, and the verifier-input cost calculation.

B.1 Helper Procedures

Shared state. The state $\mathcal{T} = (\{S(c)\}_{c\in\mathcal{C}'}, \{\nu(c)\}_{c\in\mathcal{C}'})$ stores each candidate's cumulative confidence-weighted score $S(c)$ and Stage 0 cluster size $\nu(c)$. Scores start at zero. The call UpdateScore$(\mathcal{T}, i, j, v_{ij}, w_{ij})$ applies Eq. equation 3 in place. The procedures assume a nonempty candidate pool and an integer finalist cap $f \geq 1$.

B.1.1 Dedup: Stage 0

Dedup groups candidates by their E0 signatures and retains one representative per cluster. For code, the signature is a hash of normalized text; for math, it is the normalized final answer. These signatures do not establish semantic equivalence or equal correctness. Cluster sizes are used for initial seeding and tie-breaking, but do not enter S.

Algorithm 2 Dedup$(\mathcal{C}, \phi_0)$: signature-based grouping

Require: nonempty candidate set $\mathcal{C} = \{c_1, \ldots, c_N\}$, signature map ϕ_0
Ensure: representative pool $\mathcal{C}'$, cluster sizes $\nu(\cdot)$
Compute $\sigma_i \leftarrow \phi_0(c_i)$ for $i = 1, \ldots, N$
Partition $\mathcal{C}$ into clusters $K_1, \ldots, K_M$ of equal signatures
$\mathcal{C}' \leftarrow \emptyset$
for $m = 1, \ldots, M$ **do**
    $r_m \leftarrow \arg\max_{c\in K_m} |c|$ ▷ longest candidate by token count
    $\nu(r_m) \leftarrow |K_m|$
    $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{r_m\}$
end for
return $\mathcal{C}'$, $\nu(\cdot)$

Representative selection is described in Section B.2.2; the normalization rules are in Section E.2.

B.1.2 Slaughter: Pairing rule

Slaughter pairs candidates from opposite ends of the seed ordering. Stage A seeds by cluster size ν; Stage B seeds by the current cumulative score S.

Seed separation need not imply a difference in correctness. In particular, when every cluster is a singleton, ν provides no ordering information and the first-round pairing has the same distribution as random pairing. Later rounds use comparison scores, which can also be wrong.

Algorithm 3 SLAUGHTER$(\mathcal{X}, \mathrm{key})$: pairing by seed order

Require: candidate set $\mathcal{X}$, seeding key $\mathrm{key} : \mathcal{X} \to \mathbb{R}$
Ensure: disjoint pair set $\mathcal{P}$, optional bye candidate b

$\pi \leftarrow$ sort $\mathcal{X}$ by descending $\mathrm{key}(\cdot)$, breaking key ties uniformly at random
$n \leftarrow |\mathcal{X}|$; $\mathcal{P} \leftarrow \emptyset$
for $i = 1, \ldots, \lfloor n/2 \rfloor$ **do**
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{(\pi_i, \pi_{n+1-i})\}$
end for
if n is odd **then**
    $b \leftarrow \pi_{\lceil n/2 \rceil}$ ▷ bye candidate skips this round
else
    $b \leftarrow \perp$
end if
return $\mathcal{P}$, b

B.1.3 ELIMINATE: ELIMINATION ROUNDS

Stage A calls ELIMINATE once at E1, seeded by ν. Stage B uses E2, seeded by S, and repeats full halving rounds while more than f candidates remain. The procedure updates $\mathcal{T}$ in place and returns the surviving set.

Algorithm 4 ELIMINATE$(q, \mathcal{X}, J, e, \mathrm{key}, \mathcal{T}, [\mathrm{stop_at} = f])$

Require: problem q, nonempty pool $\mathcal{X}$, judge J, evidence level $e \in \{1, 2\}$, seeding key key, state $\mathcal{T}$, optional stop count $f \geq 1$
Ensure: surviving pool $\mathcal{X}_{\mathrm{out}}$; updated scores in $\mathcal{T}$

$\mathcal{X}_{\mathrm{cur}} \leftarrow \mathcal{X}$
if stop_at is specified **and** $|\mathcal{X}_{\mathrm{cur}}| \leq$ stop_at **then**
    return $\mathcal{X}_{\mathrm{cur}}$
end if
repeat
    $\mathcal{P}, b \leftarrow$ SLAUGHTER$(\mathcal{X}_{\mathrm{cur}}, \mathrm{key})$
    $\mathcal{W} \leftarrow \emptyset$
    for $(c_i, c_j) \in \mathcal{P}$ **do**
        $(v_{ij}, w_{ij}) \leftarrow J\big(q, \phi_e(c_i), \phi_e(c_j)\big)$
        UPDATESCORE$(\mathcal{T}, i, j, v_{ij}, w_{ij})$
        if $S(c_i) > S(c_j)$ **or** $(S(c_i) = S(c_j)$ **and** $\nu(c_i) > \nu(c_j))$ **then**
            $\mathcal{W} \leftarrow \mathcal{W} \cup \{c_i\}$
        else
            $\mathcal{W} \leftarrow \mathcal{W} \cup \{c_j\}$
        end if
    end for
    if $b \neq \perp$ **then**
        $\mathcal{W} \leftarrow \mathcal{W} \cup \{b\}$ ▷ bye advances at its current score
    end if
    $\mathcal{X}_{\mathrm{cur}} \leftarrow \mathcal{W}$
until stop_at is unspecified **or** $|\mathcal{X}_{\mathrm{cur}}| \leq$ stop_at
return $\mathcal{X}_{\mathrm{cur}}$

With no stop count, Stage A performs one round and leaves $\lceil N'/2 \rceil$ candidates. Stage B returns immediately if its input size is at most f. Otherwise, it performs $r_B = \max\{0, \lceil \log_2(|\mathcal{X}|/f) \rceil\}$ rounds. We denote its actual output size by $F \leq f$; a full halving round can pass below the target. For $N' = 16$, $f = 4$, Stage A leaves eight candidates and Stage B leaves four after one round.

Selection within each pair. The updated cumulative score S determines which candidate advances; the current judge verdict alone need not do so. Ties in S favor the larger cluster. If both S and ν are equal, the procedure retains c_j in the seed ordering shown above. Randomness enters when equal

seeding keys are sorted. Cluster size is used for Stage A seeding and score ties, not as an additive contribution to S.

B.1.4 RESCUE: OPTIONAL FINALIST EXPANSION

RESCUE considers the highest-scoring eliminated candidate. It compares that candidate's cumulative score with the lowest finalist score, using the trigger in Eq. equation 5. This gap is not the margin from the comparison in which the candidate was eliminated.

Algorithm 5 RESCUE$(\mathcal{C}_B, \mathcal{E}, \delta, \mathcal{T})$: optional CAPS-R subroutine

Require: nonempty finalist set $\mathcal{C}_B$, eliminated pool $\mathcal{E} = \mathcal{C}' \setminus \mathcal{C}_B$, margin δ, state $\mathcal{T}$
Ensure: expanded set $\widetilde{\mathcal{C}}_B$ with $\widetilde{F} \in \{F, F+1\}$, where $F = |\mathcal{C}_B|$
if $\mathcal{E} = \emptyset$ **then**
  return $\mathcal{C}_B$
end if
$c^+ \leftarrow \arg\max_{c \in \mathcal{E}} S(c)$
$c_{\min} \leftarrow \arg\min_{c \in \mathcal{C}_B} S(c)$
$\Delta \leftarrow |S(c^+) - S(c_{\min})|$
if $\Delta \le \delta$ **then**
  return $\mathcal{C}_B \cup \{c^+\}$ ▷ margin condition
else if $\nu(c^+) = 1$ **and** $\Delta \le 2\delta$ **then**
  return $\mathcal{C}_B \cup \{c^+\}$ ▷ singleton condition
else
  return $\mathcal{C}_B$
end if

At most one candidate is added, increasing Stage C by F E2 comparisons. The default margin is $\delta = 0.15$. Rescue can reconsider an early elimination, but does not guarantee that an eliminated correct candidate returns.

B.1.5 COMPOSITION: TRACING THE FULL ALGORITHM

For $N' = 16$ representatives and the default cap $f = 4$, the procedure is:

1. DEDUP$(\mathcal{C}, \phi_0)$ returns $\mathcal{C}'$ and cluster sizes ν; initialize S to zero.
2. ELIMINATE$(q, \mathcal{C}', J, 1, \nu, \mathcal{T})$ performs eight E1 comparisons and leaves eight candidates.
3. ELIMINATE$(q, \mathcal{C}_A, J, 2, S, \mathcal{T}, \text{stop_at}{=}4)$ performs four E2 comparisons and leaves $F = 4$ finalists.
4. If enabled, RESCUE$(\mathcal{C}_B, \mathcal{C}' \setminus \mathcal{C}_B, 0.15, \mathcal{T})$ returns an expanded set $\widetilde{\mathcal{C}}_B$; otherwise $\widetilde{\mathcal{C}}_B = \mathcal{C}_B$.
5. Stage C compares every pair in $\widetilde{\mathcal{C}}_B$ at E2 and selects by Eq. equation 4. If only one candidate remains, it is returned without a comparison or score normalization.

Without rescue, this gives eight E1 calls and ten E2 calls: 18 calls in total, with cost $8T_1 + 10T_2$. A rescue adds four E2 calls in this configuration. If grouping reduces N' below 16, the counts follow Section C.1.

B.2 IMPLEMENTATION DETAILS

B.2.1 PER-CALL TOKEN COST DECOMPOSITION

For a pair (c_i, c_j), the logical verifier-input length is

$$T_e(i, j, q) = |\phi_e(c_i)| + |\phi_e(c_j)| + T_{\text{ovhd},e}(q), \tag{9}$$

where lengths are measured in tokens and $T_{\text{ovhd},e}(q)$ includes the problem statement, system prompt, instructions, and formatting for the chosen interface. Cached prefixes count toward logical input length. Generation tokens and verifier outputs are excluded from this metric; their separate accounting is described in Section E.6.

The cost analysis uses representative per-call values T_1, T_2 and their ratio $\rho = T_1/T_2$. These values depend on the task, model, and solution lengths. The example $\rho = 0.12$ illustrates the analytical cost formula; it is not a shared measured constant across all settings. Reported token totals sum the actual input lengths of the calls.

B.2.2 Stage 0 Representative Selection

Within each signature cluster, we retain the longest candidate by token count, $\mathrm{rep}(K_m) = \arg\max_{c \in K_m} |c|$. This is a heuristic preference for retaining more of the generated reasoning. Length alone does not establish completeness or correctness.

B.3 Design of the Evidence Cascade

B.3.1 Why three evidence levels

E0 groups candidates without a judge call. E1 reduces the input length of the first elimination round, while E2 exposes the full solutions in subsequent rounds. This allocation saves verifier-input tokens when $T_1 < T_2$, at the risk of discarding a useful candidate before the full solution is inspected. The All-E2 ablation in Table 8 measures this trade-off for the reported extractor setting.

B.3.2 Content of the partial view ϕ_1

The default E1 view contains the first and last 50 words of the reasoning. For code, it also includes the first 500 characters of the final code; for math, it includes the final boxed answer. Section E.2 specifies extraction and normalization. These views can omit decisive details, so their usefulness depends on the model's output format as well as the task.

The first-20-code-line and thinking-aware extractors are evaluated as alternatives to the default E1 view in Table 8. The latter retains the final answer when present and the last approximately 500 tokens.

B.3.3 Rescue trigger

The margin condition admits the highest-scoring eliminated candidate when its score is within δ of the lowest finalist score. The singleton condition widens this threshold to 2δ when $\nu(c^+) = 1$. This is a heuristic allowance for a candidate that has no duplicate support; the signature does not reveal whether it is correct. The trigger uses an OR because either condition can admit the candidate. With the default $\delta = 0.15$, at most one candidate is rescued. Sensitivity to the margin and rescue count is reported in Table 9.

C Verifier-Input Cost Analysis: Derivation and Asymptotics

The following counts assume a uniform input cost T_1 for each E1 call and T_2 for each E2 call. They describe the selection schedule; actual token totals account for variation in input length as described in Section B.2.1.

C.1 General-N' Cost Formula

Let $F = |\mathcal{C}_B|$ be the actual number of finalists before rescue, and $\widetilde{F} = |\widetilde{\mathcal{C}}_B|$ the number after rescue. We first count calls without rescue.

Stage A. One elimination round makes $\lfloor N'/2 \rfloor$ E1 calls and leaves $m_0 = \lceil N'/2 \rceil$ candidates.

Stage B. Let $m_r = \lceil m_{r-1}/2 \rceil$ be the number remaining after round r, and define

$$r_B = \min\{r \geq 0 : m_r \leq f\} = \max\{0, \lceil \log_2(m_0/f) \rceil\}, \qquad F = m_{r_B}.$$

Thus $r_B = 0$ when Stage A already leaves at most f candidates. Each comparison removes one candidate, so the number of E2 calls telescopes exactly:

$$\sum_{r=1}^{r_B} \left\lfloor \frac{m_{r-1}}{2} \right\rfloor = \sum_{r=1}^{r_B} (m_{r-1} - m_r) = m_0 - F. \tag{10}$$

Stage C. The round-robin makes $\binom{F}{2}$ E2 calls. For $F = 1$, Stage C directly returns the sole candidate.

Total. The verifier-input cost is

$$T_{\mathrm{CAPS}}(N', F) = \left\lfloor \frac{N'}{2} \right\rfloor T_1 + \left[\left\lceil \frac{N'}{2} \right\rceil - F + \binom{F}{2} \right] T_2, \tag{11}$$

which is Eq. equation 6 in the main text. This count uses the actual F, rather than assuming that every halving schedule ends at exactly f.

Standard configuration. For $N' = 16, f = 4$, Stage A leaves eight candidates and Stage B leaves $F = 4$. Hence

$$T_{\mathrm{CAPS}}(16, 4) = 8T_1 + (4 + 6)T_2 = (10 + 8\rho)T_2. \tag{12}$$

C.2 Asymptotic Form via Telescoping

Write $\epsilon_{N'} = N' \bmod 2 \in \{0, 1\}$. Expanding the floor and ceiling terms in Eq. equation 11 gives the exact expression

$$T_{\mathrm{CAPS}}(N', F) = \frac{N'}{2}(T_1 + T_2) + \left[\binom{F}{2} - F \right] T_2 + \frac{\epsilon_{N'}}{2}(T_2 - T_1). \tag{13}$$

For a fixed cap f, the bound $1 \le F \le f$ makes the last two terms bounded independently of N'. With fixed T_1, T_2, the cost is therefore $\frac{N'}{2}(T_1 + T_2) + O(1)$. The coefficient describes the leading growth with pool size; rounding and changes in F can produce different costs for individual increments of N'.

For comparison, a schedule with kN' full-evidence pairwise calls costs $kN'T_2$. The ratio of its leading coefficient to CAPS's is $2k/(1 + \rho)$. This comparison concerns verifier-input cost under the stated call-count model; it does not imply equal selection accuracy or the same ratio in runtime.

C.3 Expected Cost Under CAPS-R

Let $R \in \{0, 1\}$ indicate whether rescue adds a candidate, so $\widetilde{F} = F + R$. The extra Stage C cost is

$$\Delta T_{\mathrm{rescue}} = \left[\binom{F + R}{2} - \binom{F}{2} \right] T_2 = RFT_2. \tag{14}$$

With $p_R(F) = \Pr(R = 1 \mid F)$, the conditional expected overhead is $p_R(F)FT_2$. In general,

$$\mathbb{E}[T_{\mathrm{CAPS+R}}] = \mathbb{E}[T_{\mathrm{CAPS}}(N', F)] + \mathbb{E}[RFT_2]. \tag{15}$$

For a fixed finalist count $F = f$, this reduces to an added cost of $p_R f T_2$. The rescue probability depends on the candidate pool, judge, and configuration; no common activation rate is assumed.

C.4 Summary of Cost Results

Without rescue, $N' = 16, F = 4$ requires $8T_1 + 10T_2$, or $10.96T_2$ when $\rho = 0.12$. For fixed f and per-call costs, the leading term is $\frac{N'}{2}(T_1 + T_2)$. Rescue adds FT_2 on the problems where it fires. The finalist cap f and rescue margin δ are configurable parameters; their effect is examined in Table 9.

D Experimental Details and Extended Results

Unless stated otherwise, accuracy is Pass@1 in percent and token counts refer to verifier inputs.

D.1 Evaluation Protocol and Accounting

Shared candidate pools. All selection methods receive the same sampled candidates within each model–benchmark setting, with $N = 16$ unless otherwise specified. Appendix E gives the generation settings and prompts. CAPS includes rescue and uses $f = 4$ finalists, rescue margin $\delta = 0.15$, confidence floor $\tau_w = 0.05$, and at most one rescued candidate. The sensitivity experiments vary one parameter at a time.

Judge interfaces. The original V1-Infer configuration elicits two ratings on a 1–10 scale; CAPS elicits a winner/tie verdict and HIGH/LOW confidence. Both use the generator model as the judge, with different prompts and parsers. Appendix D.3 compares the methods under a common interface. Appendices E.1 and E.3 give the prompt templates and verdict conversion.

Evidence and cost. The default partial view contains the first and last 50 reasoning words, together with the first 500 code characters for code or the extracted final answer for math. Code signatures use normalized text, not AST equivalence. Input-token totals sum the logical prompt length of every verification call, including instructions, the problem statement, candidate views, and cached prefixes. They exclude generation and verifier output tokens. The deployment experiment in the main text measures output tokens and runtime separately. Because generation work varies with N, the cross-N results do not compare equal end-to-end compute budgets.

D.2 Voting Baselines on Mathematics

Majority voting selects the most frequent normalized boxed answer in the same $N = 16$ candidate pool and requires no judge calls. Ties between maximum-frequency answers use fixed random seeds. Confidence-weighted voting sums the pointwise ratings of candidates in each answer cluster; obtaining those ratings requires N pointwise calls. Table 5 reports all eight model–benchmark settings, using the original interfaces for V1-Infer and CAPS.

Table 5: **Voting and verification on mathematics.** Pass@1 (%) on shared $N = 16$ pools. Means are over four models. Majority uses zero verifier calls; weighted voting requires 16 pointwise ratings.

Benchmark	Model	Majority	Weighted	V1-Infer	CAPS
AIME 2025	Qwen3-14B	83.3	83.3	76.7	80.0
	Qwen3-4B-Instruct	70.0	73.3	76.7	73.3
	Qwen3-4B-Thinking	53.3	53.3	53.3	56.7
	GPT-OSS-20B	46.7	46.7	46.7	50.0
	Mean	**63.3**	**64.2**	**63.4**	**65.0**
HMMT 2025	Qwen3-14B	53.3	53.3	53.3	50.0
	Qwen3-4B-Instruct	50.0	53.3	56.7	53.3
	Qwen3-4B-Thinking	30.0	30.0	30.0	33.3
	GPT-OSS-20B	30.0	30.0	33.3	30.0
	Mean	**40.8**	**41.7**	**43.3**	**41.7**

CAPS exceeds majority voting in five settings, ties in one, and falls below it in two, but its average gain is not statistically significant. Weighted voting further narrows the difference. Voting is thus a competitive baseline for these math tasks, where candidates can be grouped by their final answers.

D.3 Judge-Interface Controls

Table 6 crosses the two selection methods with the two judge interfaces. CAPS-rating keeps the CAPS candidate pools, evidence views, hyperparameters, and selection procedure, and replaces its verdict prompt with V1-Infer's rating prompt, using $|r_A - r_B|/9$ as the confidence margin. Conversely, V1-Infer-verdict keeps the V1-Infer selection procedure and uses the CAPS verdict interface. These controls retain each method's comparison budget; they do not match the number of calls between methods.

With the rating interface, CAPS exceeds V1-Infer by 0.9 points on average and wins on $12/20$ settings. Switching CAPS from ratings to verdicts adds 0.8 points; the gap between methods under

Table 6: **Selection method × judge interface.** Macro-average Pass@1 (%) over the same 20 model–benchmark settings. The diagonal cells are the original configurations.

Selection method	Rating interface	Verdict interface
V1-Infer	46.5	46.9
CAPS	47.4	48.2

the verdict interface is 1.3 points. Judge-interface choice therefore accounts for part of the original gap. In the paired analysis in the main text, the code confidence intervals are positive under both common interfaces. The overall intervals include or touch zero, and the math intervals cross zero. The separate control with matched comparison counts also uses the rating interface: its CAPS All-E2 accuracy is 59.1, compared with 60.1 under the original verdict interface in Table 10.

D.4 Budget and Candidate-Count Measurements

Table 7 compares verifier budgets on Qwen3-14B/LCB-v6, using the original rating interface for V1-Infer and verdict interface for CAPS. At $N = 16$, CAPS with $f = 4$ has higher accuracy and lower verifier-input cost than each of the three measured V1-Infer budgets. Increasing f from 4 to 8 adds 0.7 accuracy points while increasing inputs from 11.2M to 29.6M.

Table 7: **Measured verifier budgets on Qwen3-14B/LCB-v6.** Original method-specific judge interfaces. Tokens are benchmark-total logical verifier inputs; generation and output tokens are excluded.

Method	N	Budget / finalists	Input tokens (M)	Pass@1
V1-Infer	8	3×	14.0	54.2
V1-Infer	16	1×	14.7	55.4
V1-Infer	16	2×	29.3	57.2
V1-Infer	16	3×	44.0	58.8
V1-Infer	32	3×	115.6	61.1
CAPS	8	$f = 4$	5.7	54.2
CAPS	16	$f = 2$	8.3	58.1
CAPS	16	$f = 4$	11.2	59.5
CAPS	16	$f = 8$	29.6	60.2
CAPS	32	$f = 4$	24.5	63.4
CAPS	64	$f = 4$	51.2	65.0

At $N = 32$, CAPS reaches 63.4 with 24.5M verifier-input tokens, compared with 58.8 and 44.0M for V1-Infer-3× at $N = 16$. This comparison excludes the additional generation cost of the larger candidate pool. Increasing CAPS to $N = 64$ raises accuracy to 65.0 in this setting.

D.5 Partial-Evidence Extractors

Table 8 varies the evidence view on Qwen3-4B-Thinking/LCB-v6. The default extractor combines a head–tail reasoning summary with the first 500 code characters. The thinking-aware variant uses the final answer, when present, plus the last approximately 500 tokens. All variants retain the CAPS selection procedure and original verdict interface.

Table 8: **Evidence extraction on Qwen3-4B-Thinking/LCB-v6.** Pass@1 (%) and measured verifier-input totals. All-E2 replaces partial evidence with full evidence in every stage.

Evidence configuration	Pass@1	Input tokens (M)
First 20 code lines only	51.6	13.2
Default head–tail extractor	52.7	14.7
Thinking-aware extractor	54.2	13.5
All-E2	53.4	16.7

The thinking-aware view improves on the default by 1.5 points while using fewer input tokens. It also exceeds All-E2, which gains 0.7 points over the default despite using full evidence at every stage.

With the reasoning summary fixed, code windows of 250, 500, and 1,000 characters have a mean accuracy spread of 0.6 points and a maximum spread of 1.1 across the four models on LCB-v6. On math, an answer-only view is 1.8 points below a view retaining a terminal reasoning window on average; 300- and 500-token windows differ by 0.7 points on average. The math benchmarks contain only 30 problems each, so these small differences do not establish statistical equivalence.

D.6 Sensitivity to Cascade Hyperparameters

Table 9 varies one parameter at a time on Qwen3-14B and reports mean Pass@1 over LCB-v6, CodeContests, AIME, and HMMT. The default configuration scores 56.3. In the rescue-count scan, a rescue adds the k highest-scoring eliminated candidates. Finalist-count costs are calculated analytically for $N' = 16$ and $\rho = 0.12$, excluding rescue; Table 7 instead reports measured per-benchmark costs. Actual finalist counts are capped by the number of available candidates.

Table 9: **One-factor sensitivity on Qwen3-14B.** Bold settings are defaults. Costs are relative to the default: finalist-count costs are analytical; the rescue-count scan gives reported relative costs. Dashes denote unreported quantities.

Parameter	Setting	Mean Pass@1	Relative cost	Rescue trigger (%)
Finalists f	2	55.2	0.73× (analytical)	—
	4	56.3	1.00× (analytical)	—
	8	56.8	2.64× (analytical)	—
Rescue margin δ	Disabled	54.9	—	0
	0.05	55.5	—	4
	0.10	56.0	—	8
	0.15	56.3	—	12
	0.20	56.2	—	16
	0.30	55.9	—	24
Confidence floor τ_w	0.01	56.1	—	—
	0.02	56.2	—	—
	0.05	56.3	—	—
	0.10	56.2	—	—
	0.20	56.0	—	—
Rescued candidates k	0	54.9	0.96× (reported)	—
	1	56.3	1.00× (reported)	—
	2	56.5	1.06× (reported)	—
	4	56.4	1.23× (reported)	—

Accuracy changes little near the default rescue margin and confidence floor. Relative token costs in the rescue-margin scan range from 0.97× to 1.05×; costs for individual settings are unavailable. Rescuing more than one candidate adds at most 0.2 points to mean accuracy while increasing cost.

D.7 Extended Component Ablation

Table 10 gives component ablations under the original CAPS interface. All-E2 is 0.5–0.7 points above full CAPS in all eight settings. Its cost increase varies by model and workload: on Thinking/LCB-v6, inputs rise from 14.7M to 16.7M; in the common-rating Qwen3-14B control, they rise from 10.9M to 17.0M. In the latter control, using E1 reduces input tokens by 35.9% at a cost of 0.5 accuracy points.

Removing deduplication changes code accuracy by only 0.1–0.2 points; removing rescue or confidence weighting lowers accuracy in every setting. These are point estimates, without confidence intervals or stage-wise survival measurements. The random-pairing ablation changes the pairing configuration as a whole and cannot isolate the contribution of cluster-size seeding. In particular, when every signature is unique, all clusters have the same size and the initial ordering provides no ranking information.

Table 10: **Component ablation.** Absolute Pass@1 (%). CC denotes CodeContests. "Random pairs" replaces the slaughter-pairing configuration; it does not isolate cluster-size seeding alone.

Model	Task	Full	No rescue	No dedup	All-E2	No weights	Random pairs
Qwen3-14B	LCB-v6	59.5	57.3	59.4	60.1	58.4	56.6
	CC	35.8	35.2	35.6	36.4	33.9	33.7
Qwen3-4B-Instruct	LCB-v6	38.9	37.4	38.7	39.5	37.6	36.4
	CC	14.5	13.9	14.3	15.2	12.1	12.4
Qwen3-4B-Thinking	LCB-v6	52.7	50.4	52.5	53.4	51.5	50.1
	CC	44.8	43.6	44.6	45.3	42.4	42.6
GPT-OSS-20B	LCB-v6	62.3	60.1	62.1	62.9	60.8	59.5
	CC	29.4	28.6	29.2	30.0	27.2	27.1

D.8 Scope of the Controls

The interface and call-budget controls compare complete selection procedures. Verifier accuracy on pairs selected by different methods also depends on which pairs are selected, so it cannot isolate the effect of partial evidence or define a deployment threshold. Early elimination may discard a correct candidate, and rescue need not recover it. The $N \leq 64$ experiments do not test this failure mode in much larger pools with only one or two correct solutions.

E Implementation Details and Prompt Templates

The templates below are the original generation and judge prompts. The common-interface experiments exchange the rating and verdict interfaces as described below. Unless stated otherwise, CAPS includes rescue.

E.1 Prompt Templates

E.1.1 Generation Prompts

Code generation (default, used for GPT-OSS). We pass the benchmark problem statement, including its input/output format, constraints, and examples, directly to the model.

Code Generation Prompt (Default)

```
{original_problem}
```

Code generation (Instruct/Thinking, used for Qwen models). For Qwen models, we append the following instruction to the problem statement. Model-specific reasoning settings are listed in Table 15.

Code Generation Prompt (Instruct/Thinking)

```
**Problem**
{problem}

Think and reason step by step before coding the final solution
for the problem above.  Put your reasoning and any draft coding
solutions between <thinking> ...  </thinking> tags.  After the
reasoning (i.e.  after the </thinking> tag), use the format
provided in the problem above (code-block with backticks) to format
your final code solution.  Do not include any thinking within the
code block.

### Answer:
```

Math generation. We append a boxed-answer instruction to each math problem.

Math Generation Prompt

```
{problem_statement} Let's think step by step and output the final
answer within \boxed{}.
```

E.1.2 Pointwise Verification Prompts

Pointwise selection scores each candidate with the following prompts. Weighted voting uses these same ratings, summed within each normalized answer cluster.

Pointwise code verification.

Pointwise Code Verification Prompt

```
You are an expert code reviewer.  Rate the correctness of a
solution to a programming problem.

**Evaluation Guidelines:**
- Analyze the problem's requirements and constraints.
- Mentally trace the solution with test cases (including edge cases)
to verify correctness.
- Give a higher score if the solution is robust and fault-tolerant.

**Problem**
{problem}

**Solution**
{code}

**Output Format:**
First, provide your step-by-step reasoning.  Then, on a new line,
provide your final rating using the EXACT tags below.  Add no other
text after the tags.
<rating>INTEGER_1_TO_10</rating>

**Rating Rules:**
- Rate correctness on a 1-10 scale (10 = correct & robust, 5 =
borderline, 1 = incorrect).
Please provide your analysis now.
```

Pointwise math verification.

Pointwise Math Verification Prompt

```
You are an expert math contest grader.  Rate the correctness of a
submission based solely on the final answer.

**Evaluation Guidelines:**
- Extract the submission's final answer.  Use any provided reasoning
only to help you assess whether the stated final answer is
trustworthy.  Do not award credit for method quality or rigor.
- Carefully analyze the problem statement and the submission to
assess whether the final answer is correct.  Grade only the final
answer.

**Problem**
{problem}

**Solution**
{solution}

**Output Format:**
First, provide your reasoning (what checks you performed).  Then,
on a new line, provide your final rating using the EXACT tag below.
Add no other text after the tag.
<rating>INTEGER_1_TO_10</rating>

**Rating Rules:**
- Rate correctness on a 1-10 scale (10 = certainly correct, 8 = very
```

likely correct, 5 = uncertain/borderline, 3 = likely incorrect, 1 = certainly incorrect).
Please provide your analysis now.

E.1.3 V1-Infer Pairwise Verification Prompts

V1-Infer requests two ratings on a 1–10 scale. Their order determines the winner, and the normalized absolute difference gives the confidence margin before flooring. The common-rating controls use this interface for both methods.

V1 Pairwise Code Verification Prompt

You are an expert code reviewer. Compare two solutions to a programming problem and rate their correctness.

****Evaluation Guidelines:****
- Analyze the problem's requirements and constraints.
- Mentally trace each solution with test cases (including edge cases) to verify correctness.
- If both solutions appear correct, prefer the more robust and fault-tolerant one.

****Problem****
{problem}

****Solution A****
{code_A}

****Solution B****
{code_B}

****Output Format:****
First, provide your step-by-step reasoning. Then, on separate new lines, provide your final ratings using the EXACT tags below. Add no other text after the tags.
<rating_A>INTEGER_1_TO_10</rating_A>
<rating_B>INTEGER_1_TO_10</rating_B>

****Rating Rules:****
- Rate correctness on a 1-10 scale (10 = correct & robust, 5 = borderline, 1 = incorrect).
- The higher rating wins. Equal ratings imply a tie.
Please provide your analysis now.

V1 Pairwise Math Verification Prompt

You are an expert math contest grader. Compare two submissions and rate correctness based solely on the final answer.

****Evaluation Guidelines:****
- Extract each submission's final answer. Use any provided reasoning only to help you assess whether the stated final answer is trustworthy. Do not award credit for method quality or rigor.
- Carefully analyze the problem statement and the submissions to assess whether each final answer is correct. Grade only the final answer.

****Problem****
{problem}

****Solution A****
{sol_A}

****Solution B****
{sol_B}

****Output Format:****

```
First, provide your reasoning (what checks you performed). Then,
on separate new lines, give ratings using the EXACT tags below.
Add no other text after the tags.
<rating_A>INTEGER_1_TO_10</rating_A>
<rating_B>INTEGER_1_TO_10</rating_B>

**Rating Rules:**
- Rate correctness on a 1-10 scale (10 = certainly correct, 8 = very
likely correct, 5 = uncertain/borderline, 3 = likely incorrect, 1 =
certainly incorrect).
- Higher rating wins. Equal ratings imply a tie.
Please provide your analysis now.
```

E.1.4 CAPS Pairwise Verification Prompts

The CAPS judge returns a winner/tie verdict and HIGH/LOW confidence at E1 and E2. E0 computes signatures without a judge call. Table 12 maps the verdicts to numerical ratings for the confidence-weighted score update.

In CAPS-rating, the candidate pools, evidence views, selection procedure, and hyperparameters are retained while the judge uses the V1-Infer rating interface. V1-Infer-verdict retains the V1-Infer selection procedure and uses the CAPS verdict interface. These interface controls retain each method's comparison budget. The separate comparison-count control on Qwen3-14B/LCB-v6 uses the rating interface for both methods and stops V1-Infer at the CAPS call count for each problem, including rescue (Table 2).

Stage A prompts (E1, partial evidence). Stage A uses the partial views defined in Section E.2.

CAPS E1 Code Prompt (Reasoning Summary + Truncated Code)

```
You are an expert code reviewer. Compare two solutions to a
programming problem using their reasoning and truncated code. Rate
their correctness.

**Problem**
{problem}

**Solution A (summary)**
{evidence_A}

**Solution B (summary)**
{evidence_B}

**Output Format:**
Provide your step-by-step reasoning. Then on a new line output
your verdict using the EXACT tags below:
<winner>A or B or TIE</winner>
<confidence>HIGH or LOW</confidence>

HIGH confidence = you are fairly sure. LOW = close call.
Provide your analysis now.
```

CAPS E1 Math Prompt (Reasoning Summary + Final Answer)

```
You are an expert math contest grader. Compare two submissions
using their reasoning summaries and final answers.

**Problem**
{problem}

**Submission A (summary)**
{evidence_A}

**Submission B (summary)**
{evidence_B}
```

```
**Output Format:**
Provide your reasoning.  Then on a new line output your verdict
using the EXACT tags below:
<winner>A or B or TIE</winner>
<confidence>HIGH or LOW</confidence>

HIGH confidence = you are fairly sure.  LOW = close call.
Provide your analysis now.
```

Stages B and C prompts (E2, full evidence). The judge sees the complete solution in Stage B and the Stage C round-robin. Both stages use the same full-evidence prompt.

CAPS E2 Code Prompt (Full Solution)

```
You are an expert code reviewer.  Compare two solutions to a
programming problem and determine which is more correct.

**Evaluation Guidelines:**
- Analyze the problem's requirements and constraints.
- Mentally trace each solution with test cases (including edge
cases) to verify correctness.
- If both solutions appear correct, prefer the more robust and
fault-tolerant one.

**Problem**
{problem}

**Solution A**
{code_A}

**Solution B**
{code_B}

**Output Format:**
Provide your step-by-step reasoning.  Then on separate new lines
output:
<winner>A or B or TIE</winner>
<confidence>HIGH or LOW</confidence>

HIGH = clear winner.  LOW = very close.
Provide your analysis now.
```

CAPS E2 Math Prompt (Full Solution)

```
You are an expert math contest grader.  Compare two submissions and
determine which is more correct, based on the final answer.

**Evaluation Guidelines:**
- Extract each submission's final answer.
- Use reasoning to assess trustworthiness of the answer.
- Grade based on the final answer correctness.

**Problem**
{problem}

**Submission A**
{sol_A}

**Submission B**
{sol_B}

**Output Format:**
Provide your reasoning.  Then on separate new lines output:
<winner>A or B or TIE</winner>
<confidence>HIGH or LOW</confidence>

HIGH = clear winner.  LOW = very close or both seem
```

```
correct/incorrect.
Provide your analysis now.
```

E.2 Evidence Extraction Details

Table 11 specifies the default evidence views. E0 is computed locally; E1 and E2 are passed to the judge together with the problem statement and instructions.

Table 11: **Default evidence views.** T_1 and T_2 denote verifier-input cost per call at E1 and E2. Their ratio $\rho = T_1/T_2$ varies with the model, task, and solution length.

Level	Code evidence	Math evidence	Verifier input
ϕ_0 (E0)	Hash of normalized code	Normalized boxed answer	0 (no call)
ϕ_1 (E1)	First and last 50 reasoning words; first 500 code characters	First and last 50 reasoning words; final boxed answer	$T_1 = \rho T_2$
ϕ_2 (E2)	Complete solution (reasoning + code)	Complete solution with all reasoning	T_2

Stage 0 (E0) signatures. For code, we strip import statements, comments, blank lines, and leading whitespace, then hash the normalized text with SHA-256. The first 16 hexadecimal digits form the cluster key. For math, we extract the last `\boxed{}` answer, lowercase it, strip whitespace, and remove LaTeX spacing commands such as `\,` and `\!`. These signatures group candidates heuristically; candidates in the same cluster need not have the same semantics or correctness. The no-deduplication ablation assigns each candidate a distinct key.

Stage A (E1) reasoning summary. For code, the default E1 view contains:

- the first 50 words and the last 50 words of the reasoning span (between `<thinking>` and `</thinking>` tags or, for non-instruct models, before the final code block), separated by a `[...reasoning truncated...]` marker;
- the first 500 characters of the final code, beginning at the opening of the code block.

For math, the second component is replaced by the extracted final answer.

Alternative views. The extractor comparison in Table 8 replaces the default view with either the first 20 code lines, a thinking-aware view containing the final answer when present and the last approximately 500 tokens, or full evidence at every stage (All-E2). The code-window scan changes only the code excerpt to 250, 500, or 1,000 characters. The math-window scan compares an answer-only view with terminal reasoning windows of 300 or 500 tokens. These variants are evaluated separately from the default extractor.

E.3 Verdict-to-Rating Conversion

The mapping in Table 12 converts CAPS verdicts into a pair of ratings. From candidate A's perspective, the outcome is 1, 0, or $1/2$ for an A win, B win, or tie. The confidence weight is

$$w_{AB} = \max\left(\tau_w, \frac{|r_A - r_B|}{9}\right), \qquad \tau_w = 0.05.$$

The same margin rule applies to ratings returned directly by the rating interface.

After flooring, HIGH and LOW wins have weights $2/3$ and $2/9$, and ties have weight 0.05. This conversion specifies the judge output used by the scoring rule; it does not make the two methods' selection and aggregation procedures identical.

E.4 Hyperparameters

Generation hyperparameters. All selection methods use the same generated candidate pools within a model–benchmark setting. Table 13 gives the default sampling configuration; the candidate-

Table 12: **Verdict-to-rating conversion.** Raw margins are shown before applying the confidence floor.

CAPS verdict	Rating A	Rating B	Raw margin
A + HIGH	9	3	0.67
A + LOW	7	5	0.22
B + HIGH	3	9	0.67
B + LOW	5	7	0.22
TIE (any conf.)	5	5	0.00

count experiments vary N. For code, the Qwen models use temperature 0.6 and GPT-OSS-20B uses its model-default temperature. Reasoning settings vary by model, as listed in Table 15.

Table 13: **Default generation hyperparameters.** The code temperature is 0.6 for Qwen and model-default for GPT-OSS. The pool-size experiments vary N.

Parameter	Code	Math
Temperature	0.6 (Qwen); default (GPT-OSS)	1.0
Top-p	0.95	0.95
Top-k	disabled (-1)	disabled (-1)
Candidates N	16	16
Max length (tok.)	32,768	32,768
Seed	1234	1234

Verification hyperparameters. Table 14 lists the default configurations of V1-Infer (Singh et al., 2026) and CAPS. The budget experiments vary V1-Infer's multiplier; the interface controls replace the judge interface; the sensitivity experiments vary one CAPS parameter at a time.

Table 14: **Default verification hyperparameters.** CAPS includes rescue unless an ablation disables it.

Parameter	V1-Infer	CAPS
Budget multiplier k	3.0	—
Coverage strategy	min-degree	cascade + rescue
Min degree $d_{\min}$	2	—
Swiss window h	3	—
Finalist count f	—	4
Judge output	Two 1–10 ratings	A/B/TIE + HIGH/LOW
Rescue enabled	—	true
Rescue margin δ	—	0.15
Maximum rescued candidates	—	1
Stage 0 deduplication	—	true
Confidence floor τ_w	0.05	0.05

Model-specific configuration. Table 15 records the model-specific settings. GPT-OSS-20B receives the code problem directly and uses reasoning effort `medium`. The Qwen models receive the code instruction shown above.

E.5 Datasets and Evaluation Criteria

Datasets. We use 279 problems from LiveCodeBench-v5 (August 2024–February 2025) and 131 from LiveCodeBench-v6 (February–May 2025) (Jain et al., 2024), together with 165 CodeContests problems (Li et al., 2022). AIME 2025 and HMMT February 2025 each contain 30 problems, obtained from MathArena (Balunović et al., 2025). Code evaluation uses the benchmark test suites; math evaluation uses the official answer keys.

Table 15: **Model-specific configuration.**

Model	Precision	Prompt Template	Thinking Mode
Qwen3-14B	bfloat16	instruct	default
Qwen3-4B-Instruct	bfloat16	instruct	disabled
Qwen3-4B-Thinking	bfloat16	instruct	always-on
GPT-OSS-20B	bfloat16	default	medium

Evaluation criteria. Code solutions must pass all hidden tests. We execute them in a sandboxed Python environment with a 10-second timeout per test. For math, we extract the last `\boxed{}` answer and compare it with the reference using SymPy symbolic equivalence, with a LaTeX-normalization fallback when parsing fails. Correctness evaluation is separate from the heuristic signature used to cluster candidates.

E.6 Inference and Response Parsing

Inference stack. The local models use SGLang with bfloat16 precision, GPU memory utilization 0.92, and chunked prefill. Maximum concurrency is configured per model within the range 40–100. Generation and judging use the same server endpoints; judge decoding is greedy ($T_{\text{judge}} = 0$).

Deployment measurements. The runtime experiment uses the same 131 Qwen3-14B/LCB-v6 candidate pools over three runs on two A100-80GB GPUs, with warm prefix caching and fixed concurrency. Both methods use their original judge interfaces. Selection latency includes queueing, retries, parsing, and stage barriers; end-to-end measurements also include generation. The three runs refer to these runtime measurements.

Token accounting. Verifier-input totals sum the logical input length of all verification calls, including prompt instructions, problem statements, candidate evidence, and cached prefixes. They exclude generation and verifier outputs, which are reported separately in the runtime experiment. Cached prefixes count toward logical input length even when the server reuses their computation. Token ratios and runtime ratios therefore measure different costs.

Response parsing. Pointwise ratings use `<rating>`; pairwise ratings use separate `<rating_A>` and `<rating_B>` fields. Parsed ratings are clamped to $[1, 10]$. CAPS uses `<winner>` with values A, B, or TIE, and `<confidence>` with values HIGH or LOW. Parsing first checks the tagged fields, with case-insensitive matching, then uses unstructured fallbacks. If the winner or confidence cannot be recovered, the corresponding field defaults to TIE or LOW.

Selection configurations. Table 16 lists the configuration names used in the implementation. The reported CAPS results use `caps_r`; `caps` is the no-rescue ablation.

Table 16: **Selection configurations.** Random pairing changes the pairing configuration as a whole.

Configuration	Selection procedure
`min_degree`	V1-Infer Swiss tournament with minimum degree $d_{\min} = 2$
`caps`	CAPS without rescue
`caps_r`	CAPS with rescue (default)
`random`	Random-pairing ablation