THE HIDDEN SIGNAL OF VERIFIER STRICTNESS: CONTROLLING AND IMPROVING STEP-WISE VERIFICATION VIA SELECTIVE LATENT STEERING

Anonymous authors
Paper under double-blind review

ABSTRACT

Generative verifiers have emerged as a promising paradigm for step-wise verification, but their verification behavior is often poorly calibrated: they may be under-critical and miss erroneous steps, or over-critical and reject correct reasoning. We refer to the verifier's tendency toward stricter or more lenient judgments as verifier strictness. We identify hidden-state directions at verification-paragraph boundaries that modulate verifier strictness without fine-tuning the verifier. However, uniform steering induces a trade-off between error detection and correctness certification. To address this, we propose `VerifySteer`, which exploits latent correctness signals for sample-level routing and selectively intervenes on paragraph boundaries. Experiments on ProcessBench and Hard2Verify show that `VerifySteer` outperforms prompt optimization and activation steering baselines, and is competitive with self-consistency while requiring 3–8$\times$ less inference compute. `VerifySteer` is also complementary to verification fine-tuning, providing further gains on top of fine-tuned verifiers. Our code is available at `https://anonymous.4open.science/r/VerifySteer-Anony-9996`.

1 INTRODUCTION

Large Language Models (LLMs) have made remarkable progress in mathematical reasoning. Yet they still make notable errors, including miscalculations and logical mistakes that lead to incorrect conclusions (Yin et al., 2025). Crucially, these errors do not appear only in the final answer. Recent work shows that LLMs can also make step-level mistakes in intermediate reasoning, sometimes even while arriving at the correct final answer (Zheng et al., 2025; Setlur et al., 2025). This challenge motivated step-level verification and Process Reward Models (PRMs; Wang et al., 2024; Lightman et al., 2023) with the aim of detecting errors at the level of reasoning steps. More recently, attention has shifted to *generative verifiers* (Zhang et al., 2025a; Mahan et al., 2024; Liu et al., 2025b). These models treat verification as a next-token prediction problem. Given a problem and a candidate solution, a generative verifier typically generates a verification chain-of-thought (CoT) and then outputs a verdict. This paradigm can be directly applied to general instruction-tuned LLMs through prompting and provides a natural language explanation of the detected error.

Recent benchmark and evaluation studies have revealed important limitations of generative verification. A recurring pattern is a *positivity bias*, where the verifier tends to incorrectly accept flawed solutions. This bias arises in several settings, including self-verification and intra-family verification (Lu et al., 2025; Singh et al., 2026a), as well as when a weaker verifier evaluates solutions produced by a stronger generator (Pandit et al., 2025; Zhou et al., 2026). At the opposite extreme, a verifier may become overly critical and reject reasoning that is actually correct. We view these two failure modes as opposite ends of a behavioral dimension that we call *verifier strictness*: an under-critical verifier is too willing to accept erroneous reasoning, whereas an over-critical verifier rejects valid reasoning.

Fine-tuning specialized verifiers (Xiong et al., 2025; Zha et al., 2025; Xu et al., 2026) can mitigate these failures, but the investment buys less than it appears. The Qwen-Math PRMs require 860K training examples (Zhang et al., 2025b), and the step-level annotation pipeline of Duan et al. (2025) consumes an estimated 17B tokens from an LLM judge, yet Pandit et al. (2025) report that far smaller general instruction-tuned models such as Qwen3 (Yang et al., 2025) already outperform the largest of them, Qwen2.5-Math-PRM-72B, on step-level verification. The returns also decay

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.

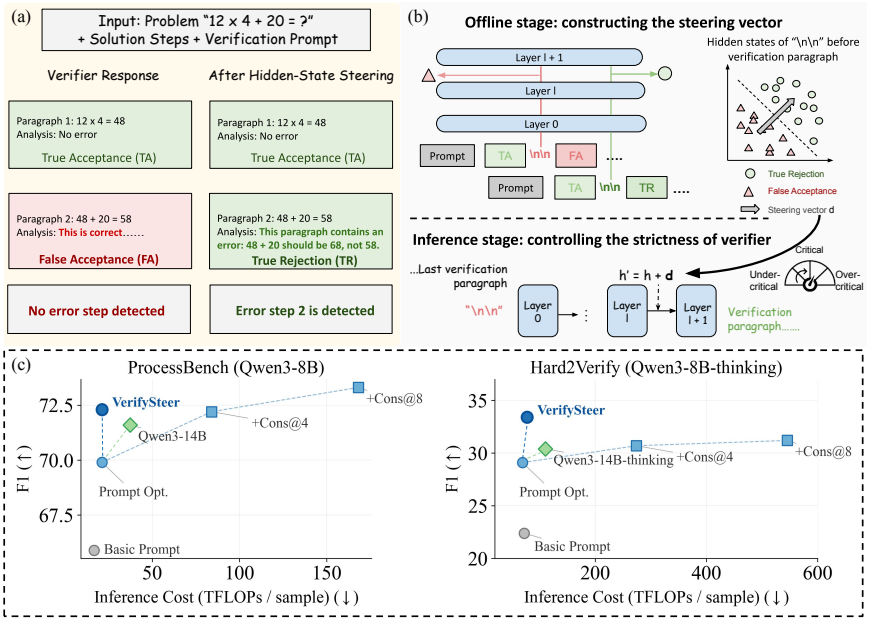


Figure 1: **Overview of our findings on controlling step-wise verification through hidden-state steering.** (a) Effect of steering. The baseline verifier falsely accepts an erroneous step and misses the true error location. After steering, the verifier correctly rejects the erroneous step and identifies the exact error location. (b) Steering pipeline. Offline, we collect hidden states of the paragraph-boundary delimiter token (`\n\n`) at transformer layer l on a validation set. TA, FA, and TR denote true acceptance, false acceptance, and true rejection verification paragraphs. We construct a steering vector $\mathbf{d}$ from the mean difference between true rejection (TR) and false acceptance (FA) states. At inference time, given an unseen sample to verify, we add the steering vector ($\mathbf{d}$) to the delimiter-token hidden states at layer l to shift verifier strictness. (c) Performance-cost tradeoffs on ProcessBench and Hard2Verify for our method `VerifySteer` and baselines. `Cons@N` denotes self-consistency decoding with N sampled verification traces.

over time: evaluators are fine-tuned on solution traces generated by past or current generator models, and Singh et al. (2026b) show that those trained on responses from earlier generations generalize poorly to outputs from newer ones. These observations motivate a lightweight approach for adapting the verification behavior of evolving instruction-tuned models without updating their weights.

Drawing inspiration from recent work in representation engineering (Zou et al., 2023; Wu et al., 2025; Maar et al., 2026; Dong et al., 2025; Kumaran et al., 2026), which suggests that LLM behaviors are encoded in hidden states and can be controlled through targeted interventions, we ask whether targeted hidden-state intervention can provide direct control over step-wise verification behavior, without fine-tuning the verifier. Our key finding is that the verifier's tendency to accept or reject a solution step is reflected in the hidden state of the paragraph-boundary delimiter preceding the corresponding verification paragraph. By intervening on this hidden state at a selected transformer layer, we can directly modulate verifier strictness. When steering corrects a verdict, it also corrects the underlying reasoning: the verifier recalculates the step, catches the error, and states the correct result. An overview of this finding is illustrated in Figure 1. Concretely, this basic intervention applies a contrastive steering vector uniformly to every sample and every generated delimiter token, nudging the verifier toward stricter or more lenient judgment depending on the steering direction.

However, uniform steering exposes a trade-off: increasing strictness improves error detection at the cost of correctness certification, and decreasing strictness has the opposite effect. We therefore propose `VerifySteer`, which makes the intervention adaptive at two levels. At the sample level, a lightweight correctness probe predicts whether the candidate solution is likely fully correct and routes each sample to the appropriate steering direction. The probe provides only a coarse routing signal, and the steered generative verifier remains responsible for examining the reasoning and identifying the first erroneous step. At the delimiter level, `VerifySteer` intervenes only on paragraph-boundary tokens whose hidden states are misaligned with the selected direction. This selective gate avoids unnecessary

perturbations to states that already express the desired level of strictness. Figure 1(c) summarizes the resulting performance-cost profile: `VerifySteer` achieves the highest F1 at inference cost comparable to single-pass baselines, with the steered 8B verifier outperforming a larger 14B one, and is competitive with self-consistency while requiring 3–8× fewer FLOPs. Our contributions are as follows:

- We identify a hidden-state signal associated with verifier strictness: the verifier's tendency to accept or reject a solution step is reflected in the hidden state of the paragraph-boundary delimiter token, and steering this signal can modulate verifier strictness.
- Building on this finding, we propose `VerifySteer`, which introduces sample-level and delimiter-level adaptivity to control verifier strictness, with ablations confirming the importance of each component and its robustness to hyperparameter choices.
- On ProcessBench and Hard2Verify, `VerifySteer` outperforms prompt optimization, activation steering (Rimsky et al., 2024), and probe-only baselines, and is competitive with self-consistency (Wang et al., 2023) at 3–8× less inference compute. These gains hold from 1.7B to 20B parameters, across dense and mixture-of-experts architectures and both thinking and non-thinking modes, and are complementary to verification fine-tuning: applying `VerifySteer` to a fine-tuned verifier, FARE-20B (Xu et al., 2026), raises F1 from 41.5 to 51.1.

2 Related Work

LLM Verification. Generative verifiers have emerged as an evaluation paradigm comparable to scalar reward models (Lightman et al., 2023; Zhang et al., 2025b). They can be broadly categorized into step-level verification (Zheng et al., 2025) and sequence-level verification (Zhang et al., 2025a). Recent evaluation and analysis work has revealed important limitations of prompted approaches, i.e., using general instruction-tuned models as verifiers. In particular, prior work has identified a positivity bias in self-verification (Lu et al., 2025; Mahdavi et al., 2025) and intra-family verification, as well as a tendency for weaker verifiers to overrate generations produced by stronger models (Zhou et al., 2026; Pandit et al., 2025). One line of work attempts to address these limitations by developing specialized verifiers through additional fine-tuning (Zhao et al., 2025b; Chan et al., 2025; Ma et al., 2025; Xiong et al., 2025), and joint training of generators and verifiers (Singh et al., 2026a; Zha et al., 2025). Our work investigates verification bias through the lens of hidden representations, and develops a lightweight approach for improving verification behavior in general instruction-tuned models.

Representation Engineering. Representation engineering (Zou et al., 2023; Wu et al., 2025) treats hidden representations as a central object of analysis and intervention, motivated by the hypothesis that behaviors and concepts correspond to directions in activation space (Park et al., 2023; Nanda et al., 2023; Bereska & Gavves, 2024; Marks & Tegmark, 2024). Activation steering (Turner et al., 2023) has been used to control refusal (Lee et al., 2025; Vu & Nguyen, 2025; Xiong et al., 2026; Sharma & Trivedi, 2026) and reduce hallucination (Wang et al., 2025; You et al., 2026; Zhao et al., 2025c) in question answering. It also extends to long-CoT reasoning, since hidden representations plan future outputs beyond the next token (Maar et al., 2026; Dong et al., 2025; Feng et al., 2026), and prior studies use it to suppress or induce thinking and reflection behaviors for more efficient reasoning (Chen et al., 2025; Liu et al., 2025a; Lin et al., 2025; Azizi et al., 2025; Zhao et al., 2025d; Zhang et al., 2025c). Conditional steering, mainly for safety, uses internal signals to predict unsafe behavior and then switches on a single refusal behavior (Lee et al., 2025; Han et al., 2025). Step-wise verification poses a harder control problem, because a verifier can err in two opposite directions and its errors arise step by step within a long CoT rather than being fixed by the prompt. Our method therefore handles both directions for each sample and re-evaluates its gate at every generated paragraph delimiter. Concurrent with our work, Kumaran et al. (2026) analyze how verbal confidence representations emerge at answer-adjacent positions during self-evaluation. We instead study step-wise verification with distinct generator and verifier models, and turn our analysis into a method that improves verification performance.

3 Preliminaries and Background

Step-Wise Verification. We consider the task of step-level error identification using a generative verifier. Given a math problem p and its step-by-step solution $\mathcal{S} = \{s_0, \ldots, s_{n-1}\}$, the verifier is prompted to predict the index of the first incorrect step in the solution. Specifically, it outputs an index $i \in \{-1, 0, \ldots, n-1\}$, where $i = -1$ indicates that all steps are correct, and $i \geq 0$ indicates that the earliest error occurs at step s_i. Before producing the final verdict, the verifier generates a verification CoT $\mathcal{V}$. We observe that the verification CoT also exhibits a step-wise structure: each

verification step evaluates a single solution step, and adjacent verification steps are often separated by a double-newline token sequence ("\n\n"), as illustrated in Figure 1. This format is the norm across open-source model families (Appendix C). Accordingly, we decompose $\mathcal{V}$ as $\{v_0, \ldots, v_{n-1}\}$, where v_i denotes the verification step evaluating the correctness of s_i.

Evaluation Metrics. Following prior work (Zheng et al., 2025; Pandit et al., 2025), we evaluate step-wise verification using true positive rate (TPR), true negative rate (TNR), and their harmonic mean $\text{F1} = 2\,\text{TPR} \cdot \text{TNR}/(\text{TPR} + \text{TNR})$. TPR measures the verifier's accuracy on fully correct solutions, i.e., whether it correctly accepts all steps, while TNR measures its accuracy on incorrect solutions, i.e., whether it correctly identifies the erroneous step.

Benchmarks and Datasets. We evaluate on ProcessBench (Zheng et al., 2025), with four subsets of increasing difficulty (GSM8K, MATH, OlympiadBench, and Omni-MATH), and Hard2Verify (Pandit et al., 2025), which curates recent competition problems (IMO, Putnam). Both provide human-annotated first-error locations. For steering vector construction and correctness probe training, we subsample ActPRM (Duan et al., 2025), a step-labeled dataset built from NuminaMath-1.5 (LI et al., 2024). Further dataset details are in Appendix D.2.

Categorization of Verification Mistakes. We define *strictness* as the verifier's tendency to accept or reject solution steps. We categorize verification mistakes along this dimension. On an incorrect solution with first error s_{i_e}, an *under-critical mistake* is a *false acceptance* that wrongly concludes all steps are correct, as opposed to the desired *true rejection* at s_{i_e} (Figure 1). The verifier may instead predict a wrong error location $\hat{i} \neq i_e$, which we call *inaccurate step identification*. On a fully correct solution, an *over-critical mistake* is a *false rejection*, as opposed to the desired *true acceptance*. Insufficient strictness thus leads to under-critical mistakes and excessive strictness to over-critical behavior.

4 Dissecting Step-Wise Verification from Latent Perspective

	Solution Step	Baseline Incorrect Verification	Steered ($+\boldsymbol{d}_{\text{strict}}$ / $+\boldsymbol{d}_{\text{lenient}}$)
Under-critical	*After using 25 bandages, the remaining bandages are* $(x - 16 + 100) - 25 = x + 69 - 25 = x + 44$.	**Analysis**: This is correct. They added 100 bandages, used 25, so the remaining is $x - 16 + 100 - 25 = x + 69 - 25 = x + 4$ **No error**.	**Analysis**: This paragraph contains an **error**. The expression $(x - 16 + 100) - 25$ simplifies to $x + 84 - 25 = x + 59$, not $x + 44$. The error is in the arithmetic: $-16 + 100 = 84$, not 69. So the correct remaining bandages should be $x + 59$, not $x + 44$.
Over-critical	*First, calculate the area of the larger rectangle. The larger rectangle has a length of 4 yards and a width of 5 yards.* [*Area of the larger rectangle* $= 4 \times 5 = 20$ *square yards*]	**Analysis**: ...However, the dimensions are not clearly defined in the diagram, and the interpretation of which dimension corresponds to which side is ambiguous. - **Conclusion**: The error is in the **interpretation of the dimensions**, ...	This is also correct. The diagram labels "4 yds" at the top and "5 yds" on the left side, so the larger rectangle is 4 yards wide and 5 yards tall. The area is indeed $4 \times 5 = 20$ square yards.

Table 1: **Representative case studies of verification failures and their correction by hidden-state steering.** Qwen3-8B on ProcessBench. *Top:* an under-critical case where the baseline verifier falsely accepts an erroneous step, corrected by applying $\boldsymbol{d}_{\text{strict}}$. *Bottom:* an over-critical case where the baseline falsely rejects a correct step, corrected by applying $\boldsymbol{d}_{\text{lenient}}$. More examples are in Appendix B.

4.1 Dissecting Verification Failures

We examine verification failure cases of Qwen3-8B on ProcessBench using the basic prompt, with representative examples shown in Table 1 under the column "baseline incorrect verification". More examples are shown in Appendix B Table 6. The baseline verifier tends to produce a superficial CoT that mainly serves to justify a pre-existing verdict: for under-critical cases, the CoT repeats erroneous intermediate steps rather than independently recalculating; for over-critical cases, it elevates superficial stylistic issues into reasoning errors. These patterns suggest that the verifier's accept-or-reject tendency may be determined before the corresponding verification paragraph is generated, rather than arising from step-by-step reasoning within it. This aligns with prior work showing that CoT in problem-solving can be "decorative", with the model's beliefs already encoded in the hidden states (Zhao et al., 2025a; Boppana et al., 2026).

Combined with the sequential structure of step-wise verification shown in Figure 1, where adjacent verification paragraphs are separated by \n\n tokens, this motivates the following hypothesis: the

verifier's tendency to accept or reject a specific solution step is encoded in the hidden state of the \n\n token preceding the corresponding verification paragraph.

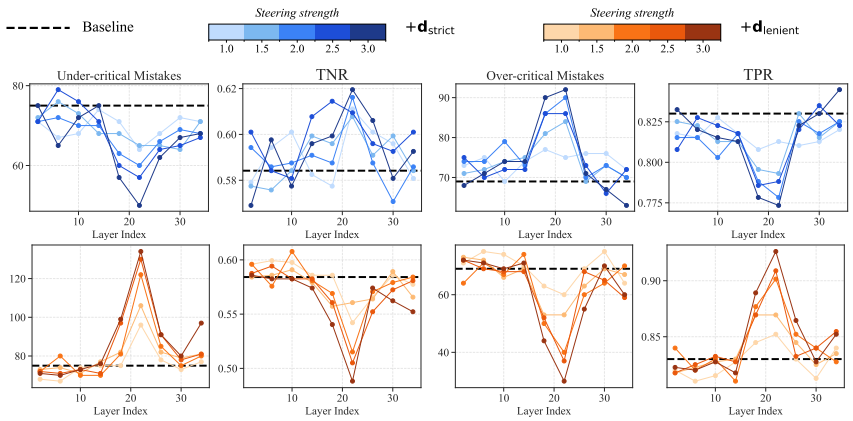


Figure 2: **Verification strictness is adjustable via hidden-state steering.** Results on Qwen3-8B on the Math subset of ProcessBench. Each panel shows how hidden-state steering changes under-critical mistake count (lower is better), TNR (higher is better), over-critical mistake count, and TPR relative to the baseline. *Upper*: applying the strictness-increasing vector $\boldsymbol{d}_{\text{strict}}$ across different layers and steering strengths. *Lower*: applying the leniency-increasing vector $\boldsymbol{d}_{\text{lenient}}$.

4.2 Verification Strictness is Adjustable by Hidden State Steering

To test the hypothesis that verifier strictness is steerable, we develop a verification-specific hidden-state steering framework with two stages: an offline stage for constructing a steering vector, and an online stage for test-time intervention. The full pipeline is shown in Figure 1 (b). We use CAA (Rimsky et al., 2024; Wu et al., 2025) as the steering primitive here and compare it with our proposed method in Section 5.2.

Extraction of Strictness Steering Vector. We first build a validation set from ActPRM (Duan et al., 2025). To construct the steering vector for increasing verifier strictness in this analysis, we use 500 randomly sampled samples labeled as erroneous. For each sample, we use the verifier to generate 16 verification traces with temperature 0.7 and top-p 0.8. Consider an erroneous solution $\mathcal{S}$ for problem p, whose annotated first incorrect step is s_{i_e}. For each sampled verification trace $\mathcal{V}$, we segment the trace into $\{v_0, \ldots, v_{n-1}\}$ using the \n\n, and extract the verifier's final prediction $\hat{i}$. We retain only those erroneous samples whose 16 rollouts contain at least one trace with $\hat{i} = -1$ and at least one trace with $\hat{i} = i_e$. Recall that a verifier output of -1 indicates that all steps are correct. This ensures that, for the same problem-solution pair, the verifier exhibits both false-acceptance and true-rejection behaviors across different rollouts.

For a trace with $\hat{i} = -1$, a false acceptance of s_{i_e}, we localize the corresponding verification segment using paragraph matching (e.g., "Paragraph i") together with acceptance-oriented keywords such as "correct" or "okay", and collect the hidden state of the immediately preceding \n\n token at layer l into the set $\mathcal{H}^l_{\text{FA}}$. Analogously, for a trace with $\hat{i} = i_e$, a true rejection, we localize the segment using rejection-oriented keywords (e.g., "incorrect", "issue") and collect the corresponding hidden states into $\mathcal{H}^l_{\text{TR}}$. The complete list of keywords is provided in Appendix D.1.

The strictness steering vector is the difference between their means, $\boldsymbol{d}^l_{\text{strict}} = \bar{\boldsymbol{h}}^l_{\text{TR}} - \bar{\boldsymbol{h}}^l_{\text{FA}}$, which captures the average hidden-state direction from false acceptance to true rejection. We omit the layer superscript l when it is clear from context. Analogously, from 500 fully correct samples we build the leniency vector from true-acceptance and false-rejection segments, $\boldsymbol{d}_{\text{lenient}} = \bar{\boldsymbol{h}}_{\text{TA}} - \bar{\boldsymbol{h}}_{\text{FR}}$, which captures the direction from false rejection to true acceptance.

Test-time Intervention. After obtaining the two steering directions, we test whether they can modulate verifier strictness on unseen verification samples. Specifically, during inference, we intervene on the hidden representations of all generated \n\n tokens. Given a hidden state $\boldsymbol{h}$ at layer l and a steering direction $\boldsymbol{d}$, we first apply a scaled perturbation and then renormalize the result to

	GSM8K			Math			OlympiadBench			Omni-MATH			Overall	
Metric	**TNR**	**TPR**	**F1 (↑)**	**TNR**	**TPR**	**F1 (↑)**	**TNR**	**TPR**	**F1 (↑)**	**TNR**	**TPR**	**F1 (↑)**	**F1 (↑)**	**TFLOPs (↓)**
Qwen3-1.7B														
Basic Prompt	17.9	96.4	30.2	14.5	84.2	24.7	7.7	78.8	14.1	8.0	82.6	14.7	20.9	2.86
Probe-only	22.2	80.3	34.8	17.8	66.5	28.1	13.2	52.5	21.0	10.4	61.0	17.8	25.4	3.05
Prompt Opt.	22.2	76.7	34.5	18.0	58.4	27.5	13.3	49.6	21.0	10.4	57.3	17.6	25.1	3.19
+Cons@4†	21.6	81.2	34.1	20.3	69.4	31.4	11.0	58.5	18.5	12.0	62.2	20.0	26.0	12.71
+Cons@8†	21.0	83.9	33.6	21.4	71.8	**32.9**	10.9	63.0	18.6	11.4	66.2	19.4	26.1	25.43
VerifySteer-Uni	25.6	77.2	38.5	21.7	61.6	32.1	15.6	56.6	**24.4**	12.6	59.3	20.9	29.0	3.37
VerifySteer-Bi	27.1	83.4	**40.9**	21.5	69.2	**32.9**	15.0	58.4	23.8	12.6	64.3	**21.1**	**29.7**	3.47
Qwen3-8B														
Basic Prompt	54.6	96.4	69.7	58.4	83.0	68.6	54.0	76.1	63.2	53.8	73.4	62.1	65.9	16.71
Probe-only	57.0	94.3	71.1	62.0	87.9	72.7	56.4	79.9	66.2	57.6	76.8	65.8	69.0	18.71
Prompt Opt.	58.5	94.3	72.2	66.8	85.0	74.8	58.4	77.9	66.7	58.6	75.5	66.0	69.9	21.26
+Cons@4†	63.3	95.6	76.2	69.5	86.3	77.0	61.3	77.9	68.7	63.1	71.4	67.0	72.2	84.09
+Cons@8†	63.3	95.9	76.2	70.6	87.1	**78.0**	63.2	79.6	**70.5**	65.0	72.4	68.5	**73.3**	168.18
VerifySteer-Uni	62.3	93.3	74.7	69.0	85.5	76.4	59.6	77.0	67.2	61.9	76.8	68.5	71.7	21.23
VerifySteer-Bi	64.3	93.8	**76.3**	68.7	86.9	76.7	60.1	76.7	67.4	61.8	77.2	**68.6**	72.3	21.23
Qwen3-14B	61.4	98.4	75.6	68.5	86.9	76.6	58.9	74.3	65.7	59.6	80.2	68.4	71.6	37.45
Llama-3.3-70B	68.6	96.9	80.3	46.0	94.3	61.8	32.8	93.2	48.6	29.0	88.0	43.6	58.6	85.45
Qwen2.5-72B	61.8	95.9	75.2	45.8	91.4	61.0	35.9	87.6	50.9	37.5	83.4	51.8	59.7	53.03

Table 2: Comparison of VerifySteer with baselines on ProcessBench. Cons@4 and Cons@8 refer to the self-consistency method with 4 and 8 sampled results respectively, with the final verdict selected by majority voting. † denotes sampling-based decoding with temperature 0.7 and top-p 0.8. We report the mean across 4 runs of Cons@4 and Cons@8. Methods without † use greedy decoding. TFLOPs reports the estimated inference compute per sample. **Bold** and underline mark the best and second-best F1 within each model block. The bottom three rows are reference models and are not ranked.

preserve the original hidden-state ℓ_2 norm:

$$\boldsymbol{h}' = \|\boldsymbol{h}\|_2 \frac{\boldsymbol{h} + \alpha \boldsymbol{d}}{\|\boldsymbol{h} + \alpha \boldsymbol{d}\|_2}, \tag{1}$$

where α controls the steering strength.

On the MATH subset of ProcessBench with Qwen3-8B, we sweep intervention layers every four layers of the 36-layer transformer and vary the steering strength over $\{1.0, 1.5, 2.0, 2.5, 3.0\}$ with greedy decoding.Table 1 shows that strictness is indeed adjustable: $\boldsymbol{d}_{\text{strict}}$ converts false acceptances into true rejections, making the verifier identify the arithmetic error and state the correction, and $\boldsymbol{d}_{\text{lenient}}$ converts false rejections into true acceptances by suppressing harsh judgments over wording. In these examples, steering changes both the verification rationale and the verdict. Figure 2 further shows that the modulation is smooth and continuous: at the most effective setting, $\boldsymbol{d}_{\text{strict}}$ reduces under-critical mistakes by 33.3% and $\boldsymbol{d}_{\text{lenient}}$ reduces over-critical mistakes by 56.5%, with TNR and TPR improving accordingly. The strongest effects occur around layer 22 in this setting.

5 VerifySteer: Adaptive Control of Verification Strictness

5.1 Method

The previous section shows that the strictness-increasing vector $\boldsymbol{d}_{\text{strict}}$ and the leniency-increasing vector $\boldsymbol{d}_{\text{lenient}}$ induce opposite trade-offs between TNR and TPR when applied uniformly to all samples and delimiter tokens. This suggests that a fixed steering policy is suboptimal. At the sample level, different verification inputs require different strictness: some benefit from increased strictness, while others require reduced strictness. At the delimiter-token level, steering is only necessary when the local hidden state is misaligned with the desired direction; intervening on aligned states can cause inaccurate step identification. To address this, we propose VerifySteer, a lightweight conditional steering method that first chooses the steering direction at the sample level and then applies steering selectively at the delimiter-token level.

Sample-level Conditional Steering Policy. Given a verification prompt containing the math problem p and candidate solution $\mathcal{S}$, we extract a prompt-level representation $\boldsymbol{h}_p$ at layer l by pooling the hidden states of the input tokens. Following Li et al. (2026), we consider two pooling strategies: last-token and mean pooling, see Appendix D.3 for details. We feed $\boldsymbol{h}_p$ into a lightweight probe f_ϕ, which outputs a scalar correctness score $q(p, \mathcal{S}) = \sigma(f_\phi(\boldsymbol{h}_p)) \in [0, 1]$, where larger values indicate that the candidate solution is more likely to be correct. We introduce two variants: VerifySteer-Uni

(unidirectional), which applies $\boldsymbol{d}_{\text{strict}}$ when the solution is predicted to be likely incorrect ($q \leq \tau_l$) and does not steer otherwise; and `VerifySteer-Bi` (bidirectional), which additionally applies $\boldsymbol{d}_{\text{lenient}}$ when the solution is predicted to be likely correct ($q \geq \tau_h$), with no steering in the intermediate uncertainty region. Formally, the selected steering direction is:

$$\boldsymbol{d}_{\text{sel}}(p, \mathcal{S}) = \begin{cases} \boldsymbol{d}_{\text{strict}}, & q(p, \mathcal{S}) \leq \tau_l, \\ \boldsymbol{d}_{\text{lenient}}, & q(p, \mathcal{S}) \geq \tau_h \text{ and } \mathsf{A} = \texttt{VerifySteer-Bi}, \\ \mathbf{0}, & \text{otherwise}, \end{cases} \tag{2}$$

where $\tau_l < \tau_h$ and A denotes the `VerifySteer` variant.

Delimiter-level Intervention Gate. After selecting the steering direction $\boldsymbol{d}_{\text{sel}}$ for a sample, we further determine which delimiter tokens should be intervened on. If $\boldsymbol{d}_{\text{sel}} = \mathbf{0}$, we leave all delimiter-token hidden states unchanged. Otherwise, steering is only applied when the local hidden state is sufficiently misaligned with the selected steering direction. For each delimiter token hidden state at layer l $\boldsymbol{h}_k$, we apply steering as:

$$\boldsymbol{h}'_k = (1 - g_k)\, \boldsymbol{h}_k + g_k \, \|\boldsymbol{h}_k\|_2 \frac{\boldsymbol{h}_k + \alpha \boldsymbol{d}_{\text{sel}}}{\|\boldsymbol{h}_k + \alpha \boldsymbol{d}_{\text{sel}}\|_2}, \tag{3}$$

where $g_k = \mathbf{1}[\text{cossim}(\boldsymbol{h}_k, \boldsymbol{d}_{\text{sel}}) < \rho]$ is a binary gate, cossim denotes the cosine similarity. This gate prevents unnecessary intervention when the local hidden state is already sufficiently aligned with the selected steering direction and therefore does not require correction.

5.2 Results

Experimental Setup. We compare `VerifySteer` against baselines on all ProcessBench subsets and Hard2Verify. For ProcessBench, we use Qwen3-1.7B and Qwen3-8B (Yang et al., 2025) in non-thinking mode. For Hard2Verify, we evaluate two verifiers: Qwen3-8B in thinking mode (denoted Qwen3-8B-thinking), and FARE-20B (Xu et al., 2026), a multi-task evaluator fine-tuned from the MoE model gpt-oss-20B (OpenAI, 2025) with 2.5M training samples. We also report results of large instruction-tuned models Qwen3-14B, Llama-3.3-70B, Qwen2.5-72B and gpt-oss-20B (low reasoning) for reference. We report TNR, TPR, and F1 for verification performance, and TFLOPs per sample for inference cost. Following Snell et al. (2024), we approximate inference FLOPs as $2MT_{\text{inf}}$, where M is the number of model parameters and T_{inf} is the number of inference tokens. For the MoE model gpt-oss-20B and FARE-20B, M refers to the number of active parameters per token. TFLOPs measures per-sample test-time inference cost. We separately analyze the one-time development cost of steering vector construction and probe training in Appendix E.

Baselines. We consider five baselines: (i) a *base evaluation prompt*, using the standard ProcessBench prompt (Zheng et al., 2025); (ii) an *optimized prompt*, adapted from Zhang et al. (2025b) for ProcessBench and from Pandit et al. (2025) for Hard2Verify, which instructs the model to be more critical; (iii) *self-consistency* (`Cons@N`), which generates N verification traces via sampling-based decoding and selects the final verdict by majority voting; (iv) a *probe-only* baseline that isolates the effect of the correctness probe from that of steering: it keeps f_ϕ but removes steering, returning the unsteered prompt-optimization output when the solution is predicted likely incorrect and predicting fully correct otherwise; and (v) CAA (Rimsky et al., 2024), the representative activation steering method, which applies the steering vector uniformly to every sample and every delimiter token. All prompts are provided in Appendix A. For the fine-tuned verifier FARE-20B, we use the prompt it was fine-tuned with, denoted as the *FARE Prompt*.

`VerifySteer` Configuration. For our method `VerifySteer`, we use the optimized prompt for each benchmark. We construct validation sets from ActPRM, build the steering vectors from them, and train a three-layer MLP probe f_ϕ on the validation samples. The validation sets are described in Appendix D.2, the probe in Appendix D.3, and the steering layer selection in Appendix D.4. The selection procedure and the resulting values for the steering hyperparameters, such as the steering strength and the delimiter-level gate threshold, are given in Appendix D.5. The sensitivity analysis in Appendix D.5 shows that gains persist across a broad range of steering strengths and thresholds. Our method keeps the verifier frozen. The verifier stays frozen throughout. Building the steering vectors and training the probe requires labeled examples, and the development cost is analyzed in Appendix E.

Main Results. Tables 2 and 3 show that both `VerifySteer` variants outperform the base and optimized prompt baselines across all models and benchmarks, showing that hidden-state steering provides gains additive to prompt-level optimization.

The probe-only baseline improves over prompt optimization on Qwen3-1.7B and Qwen3-8B-thinking at lower inference cost, with the gain coming from generally improved TPR by exploiting the hidden correctness signal. However, `VerifySteer` still outperforms it by a clear margin in both settings, confirming that the steering step, not the probe alone, drives most of the improvement. The gain also exceeds what parameter scaling provides: on ProcessBench it adds 2.4 F1 to Qwen3-8B versus 1.7 from scaling to Qwen3-14B, and on Hard2Verify 4.3 versus 1.3, so the steered 8B model outperforms the prompt-optimized 14B one.

`VerifySteer` is competitive with self-consistency at a fraction of the inference cost. It surpasses `Cons@4`, which requires 3–4× the FLOPs, in all four settings. Against `Cons@8`, at 6–8× the FLOPs, `VerifySteer` wins in three of four settings, by up to 8.4 F1 on FARE-20B Hard2Verify, and trails by 1.0 F1 on Qwen3-8B ProcessBench. A closer look reveals that self-consistency does not effectively improve TNR, suggesting that basic parallel test-time scaling is less effective at mitigating evaluator positivity bias than hidden-state steering.

`VerifySteer` is also compatible with verification fine-tuning. In Table 3, fine-tuning gpt-oss-20B into FARE-20B raises F1 from 31.4 to 41.5, and applying `VerifySteer` on top raises it to 51.1. On ProcessBench, steering Qwen3-1.7B to 29.7 overall F1 also surpasses several fine-tuned PRMs built on earlier-generation models (Zhang et al., 2025b; Xiong et al., 2024), which are 4–5× larger.

Method	TNR	TPR	F1 (↑)[1]	TFLOPs (↓)
Qwen3-8B-thinking				
Basic Prompt	12.8	94.0	22.4	72.64
Probe-only	18.0	80.4	29.3	65.01
Prompt Opt.	18.0	78.6	29.1	69.59
+Cons@4	18.7	88.9	30.7	274.33
+Cons@8	18.8	92.1	<u>31.2</u>	545.19
VerifySteer-Uni	21.5	75.6	**33.4**	77.72
VerifySteer-Bi	21.5	76.2	**33.4**	77.71
FARE-20B				
Probe-only	26.6	90.5	41.1	5.85
FARE Prompt	28.5	76.2	41.5	7.7
+Cons@4	27.4	85.7	41.5	30.9
+Cons@8	28.3	87.3	42.7	62.1
VerifySteer-Uni	35.4	73.8	<u>47.9</u>	9.2
VerifySteer-Bi	37.3	81.0	**51.1**	9.9
gpt-oss-20B	19.0	90.5	31.4	5.0
Qwen3-14B-thinking	18.5	85.4	30.4	110.87
Llama-3.3-70B	4.4	88.1	8.4	71.35
Qwen2.5-72B	10.1	69.0	17.7	53.03

Table 3: Comparison of `VerifySteer` with baselines on Hard2Verify. For Qwen3-8B-thinking and Qwen3-14B-thinking, we use the recommended sampling settings (temperature 0.6 and top-p 0.95) and report the mean over 8 runs. `Cons@4` and `Cons@8` are averaged over 6 self-consistency runs. TFLOPs reports the estimated inference compute per sample. **Bold** and <u>underline</u> mark the best and second-best F1 within each model block. The bottom four rows are reference models and are not ranked.

5.3 Ablation Studies

We examine whether both levels of adaptivity contribute to verification performance, and whether the steering site and paragraph identifier affect the gains.

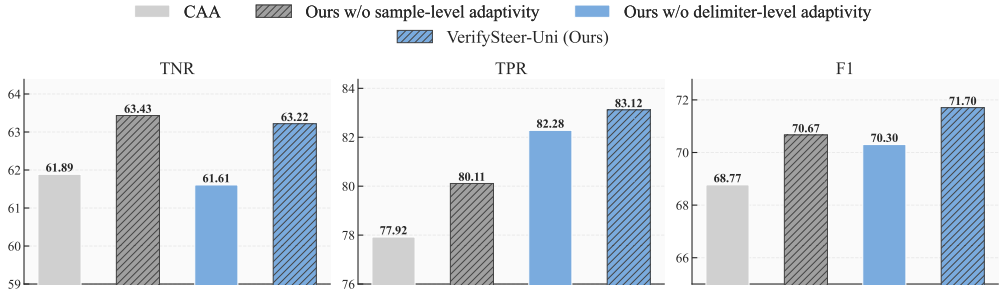


Figure 3: **Ablations of adaptive steering design.** From left to right, the three subplots report TNR, TPR, and F1, averaged over the four ProcessBench subsets. The leftmost gray bar corresponds to the activation steering method CAA. The gray-hatched and blue bars show ablations of `VerifySteer-Uni` without sample-level adaptivity and without delimiter-level adaptivity. The rightmost bar shows the full method. We use Qwen3-8B with greedy decoding.

Both sample-level and delimiter-level adaptivity improve verification performance. Figure 3 compares the full method with variants that remove either adaptive component, averaging results over the four ProcessBench subsets. Without sample-level routing, every sample is evaluated under

[1]TNR, TPR, and F1 are each averaged across runs. Because the harmonic mean is a nonlinear function, the averaged F1 may not exactly equal the harmonic mean of the averaged TNR and TPR.

Steering Site	GSM8K TNR	GSM8K TPR	GSM8K F1	MATH TNR	MATH TPR	MATH F1
Qwen3-1.7B						
Prompt Opt. Baseline	22.2	76.7	34.5	18.0	58.4	27.5
First content token	24.2	77.2	36.8	20.7	56.9	30.4
\n\n token (Ours)	25.6	77.2	**38.5**	21.7	61.6	**32.1**
Qwen3-8B						
Prompt Opt. Baseline	58.5	94.3	72.2	66.8	85.0	74.8
First content token	58.9	94.3	72.5	68.5	85.2	76.0
\n\n token (Ours)	62.3	93.3	**74.7**	69.0	85.5	**76.4**

Table 4: **Ablation of the steering token position.** "Prompt Opt. Baseline" uses the optimized prompt without steering. "First content token" is the first token of text in a verification paragraph, immediately after \n\n. "\n\n token" is the steering token position used in our method.

strictness-increasing steering. TPR falls from 83.1 to 80.1 while TNR remains nearly unchanged (63.2 versus 63.4), resulting in a 1.0-point drop in F1. This supports the effectiveness of adapting strictness to different solutions. Without delimiter-level gating, every delimiter in a selected solution is steered: TNR falls from 63.2 to 61.6 and TPR from 83.1 to 82.3, reducing F1 by 1.4. CAA (Rimsky et al., 2024) applies steering without any adaptivity. Either component alone improves F1 over CAA, and our full method achieves the largest gain of 2.9 points.

Paragraph delimiters provide effective steering sites. Our method steers at the \n\n token that separates verification paragraphs. We study whether the gains depend on this delimiter or also extend to the adjacent content token. To this end, we rebuild and apply the steering vector at the first content token of each verification paragraph, keeping the other hyperparameters fixed. Table 4 shows that this alternative improves F1 over prompt optimization. However, steering at \n\n yields larger gains in all four settings. These results support the delimiter as the intervention site, while showing that useful steering information is also present at the adjacent content token. Furthermore, Appendix C shows that \n\n delimiter are prevalent in step-wise verification outputs across the evaluated open-source model families, supporting the broader applicability of this design.

Keyword-based paragraph identification suffices for effective steering. Our method uses a simple keyword-based heuristic to identify acceptance and rejection verification paragraphs. We quantify its disagreement with an LLM-based identifier and examine whether this potential identification noise affects downstream verification performance. We compare its selections with those of gpt-5-mini (prompt in Appendix A) across 2,617 paragraph identifications from 1K problems. We then replace the keyword identifier with gpt-5-mini and rebuild the steering vector, holding the data, construction procedure, and hyperparameters fixed. Using the LLM identifier as the reference, the heuristic achieves 79.9% agreement between the two identifiers. Despite this disagreement, Table 5 shows that keywords achieve F1 similar to gpt-5-mini (74.7 vs. 74.5), and both outperform prompt optimization (72.2). Thus, the observed differences in paragraph identification have little effect on downstream performance, supporting the use of keyword heuristic.

Identifier	TNR	TPR	F1
Prompt Opt. Baseline	58.5	94.3	72.2
LLM (gpt-5-mini)	61.8	93.8	74.5
Keyword (Ours)	62.3	93.3	**74.7**

Table 5: Ablation of the verification paragraph identifier. Qwen3-8B on ProcessBench GSM8K.

6 Conclusion

We study step-wise verification through hidden-state steering, and find that verifier strictness is encoded in a verification-specific latent signal that can be modulated without fine-tuning. Uniform steering improves either error detection or correctness certification but induces a trade-off, which VerifySteer addresses through sample-level routing and paragraph-level intervention. On ProcessBench and Hard2Verify, VerifySteer outperforms single-pass baselines and is competitive with self-consistency at 3–8× less inference compute. Our results opens a path toward lightweight and adaptive control of verification in instruction-tuned models.

VerifySteer has several limitations. Our evaluation covers only mathematical reasoning, including math word problems and proof-style problems. We focus on this domain because benchmarks with human-annotated step-level errors are readily available. Future work could extend the approach to other domains and explore ways to calibrate verifier strictness without hidden-state access.

AI USE STATEMENT

We used generative AI tools (large language models) to assist with drafting and editing portions of the manuscript text. All AI-assisted text was reviewed, revised, and verified by the authors. We did not use generative AI tools to generate experimental results, design the method, or produce figures. All code for training and evaluation was written by the authors; LLM-generated code snippets used during development were reviewed and tested for correctness. We take responsibility for the final content of this work, including all text, claims, and artifacts.

ETHICS STATEMENT

This work studies step-wise verification of mathematical reasoning using ProcessBench, Hard2Verify, and ActPRM. Better-calibrated verifiers may reduce both the acceptance of erroneous reasoning and the rejection of correct solutions. However, the ability to steer verifier strictness also creates a misuse risk: a party with hidden-state access could deliberately bias an evaluator toward accepting unsound solutions or rejecting valid ones. Our results do not guarantee reliable verification, and applications such as educational grading should retain human oversight. We report both error-detection and correctness-certification performance to make this trade-off explicit, and distinguish test-time inference savings from the one-time development cost.

REPRODUCIBILITY STATEMENT

Sections 4.2 and 5.1 describe steering-vector construction and adaptive intervention, and Section 5.2 specifies the models, baselines, and evaluation setup. Appendix A provides the prompts, while Appendix D documents ActPRM data construction, paragraph-identification rules, correctness-probe training, and steering-layer and hyperparameter selection. The main results table captions specify decoding settings and the number of runs used for averaged results; Appendix E reports development compute separately from inference cost. Code is available at https://anonymous.4open.science/r/VerifySteer-Anony-9996.

REFERENCES

Seyedarmin Azizi, Erfan Baghaei Potraghloo, and Massoud Pedram. Activation steering for chain-of-thought compression. *arXiv preprint arXiv:2507.04742*, 2025.

Leonard Bereska and Stratis Gavves. Mechanistic interpretability for AI safety - a review. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=ePUVetPKu6. Survey Certification, Expert Certification.

Siddharth Boppana, Annabel Ma, Max Loeffler, Raphael Sarfati, Eric Bigelow, Atticus Geiger, Owen Lewis, and Jack Merullo. Reasoning theater: Disentangling model beliefs from chain-of-thought. *arXiv preprint arXiv:2603.05488*, 2026.

Chi-Min Chan, Chunpu Xu, Jiaming Ji, Zhen Ye, Pengcheng Wen, Chunyang Jiang, Yaodong Yang, Wei Xue, Sirui Han, and Yike Guo. J1: Exploring simple test-time scaling for llm-as-a-judge. *arXiv preprint arXiv:2505.11875*, 2025.

Runjin Chen, Zhenyu Zhang, Junyuan Hong, Souvik Kundu, and Zhangyang Wang. Seal: Steerable reasoning calibration of large language models for free. *arXiv preprint arXiv:2504.07986*, 2025.

Zhichen Dong, Zhanhui Zhou, Zhixuan Liu, Chao Yang, and Chaochao Lu. Emergent response planning in LLMs. In *Forty-second International Conference on Machine Learning*, 2025.

Keyu Duan, Zichen Liu, Xin Mao, Tianyu Pang, Changyu Chen, Qiguang Chen, Michael Qizhe Shieh, and Longxu Dou. Efficient process reward model training via active learning. In *Second Conference on Language Modeling*, 2025.

Zijian Feng, Tianjiao Li, Zixiao Zhu, Hanzhang Zhou, Junlang Qian, Li Zhang, Chua Jia Jim Deryl, Mak Lee Onn, Gee Wah Ng, and Kezhi Mao. Fine-grained activation steering: Steering less, achieving more. In *The Fourteenth International Conference on Learning Representations*, 2026.

Peixuan Han, Cheng Qian, Xiusi Chen, Yuji Zhang, Heng Ji, and Denghui Zhang. Safeswitch: Steering unsafe llm behavior via internal activation signals. In *EMNLP (Findings)*, pp. 6936–6955, 2025.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021.

Dharshan Kumaran, Arthur Conmy, Federico Barbero, Simon Osindero, Viorica Patraucean, and Petar Velickovic. How do llms compute verbal confidence. *arXiv preprint arXiv:2603.17839*, 2026.

Bruce W. Lee, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Erik Miehling, Pierre Dognin, Manish Nagireddy, and Amit Dhurandhar. Programming refusal with conditional activation steering. In *The Thirteenth International Conference on Learning Representations*, 2025.

Jia LI, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. Numinamath. [https://huggingface.co/datasets/AI-MO/NuminaMath-1.5](https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf), 2024.

Yichen Li, Zhiting Fan, Ruizhe Chen, Xiaotang Gai, Luqi Gong, Yan Zhang, and Zuozhu Liu. Fairsteer: Inference time debiasing for llms with dynamic activation steering. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 11293–11312, 2025.

Zhuochun Li, Yong Zhang, Ming Li, Yuelyu Ji, Yiming Zeng, Ning Cheng, Yun Zhu, Yanmeng Wang, Shaojun Wang, Jing Xiao, and Daqing He. Rethinking LLM-as-a-judge: Representation-as-a-judge with small language models via semantic capacity asymmetry. In *The Fourteenth International Conference on Learning Representations*, 2026.

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.

Zhengkai Lin, Zhihang Fu, Ze Chen, Chao Chen, Liang Xie, Wenxiao Wang, Deng Cai, Zheng Wang, and Jieping Ye. Controlling thinking speed in reasoning models. *arXiv preprint arXiv:2507.03704*, 2025.

Sheng Liu, Tianlang Chen, Pan Lu, Haotian Ye, Yizheng Chen, Lei Xing, and James Zou. Fractional reasoning via latent steering vectors improves inference time compute. *arXiv preprint arXiv:2506.15882*, 2025a.

Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. Inference-time scaling for generalist reward modeling. *arXiv preprint arXiv:2504.02495*, 2025b.

Jack Lu, Ryan Teehan, Jinran Jin, and Mengye Ren. When does verification pay off? a closer look at llms as solution verifiers. *arXiv preprint arXiv:2512.02304*, 2025.

Ruotian Ma, Peisong Wang, Cheng Liu, Xingyan Liu, Jiaqi Chen, Bang Zhang, Xin Zhou, Nan Du, and Jia Li. S^2r: Teaching llms to self-verify and self-correct via reinforcement learning. *arXiv preprint arXiv:2502.12853*, 2025.

Jim Maar, Denis Paperno, Callum Stuart McDougall, and Neel Nanda. What's the plan? metrics for implicit planning in LLMs and their application to rhyme generation. In *The Fourteenth International Conference on Learning Representations*, 2026.

Dakota Mahan, Duy Van Phung, Rafael Rafailov, Chase Blagden, Nathan Lile, Louis Castricato, Jan-Philipp Fränken, Chelsea Finn, and Alon Albalak. Generative reward models. *arXiv preprint arXiv:2410.12832*, 2024.

Sadegh Mahdavi, Branislav Kisacanin, Shubham Toshniwal, Wei Du, Ivan Moshkov, George Armstrong, Renjie Liao, Christos Thrampoulidis, and Igor Gitman. Scaling generative verifiers for natural language mathematical proof verification and selection. *arXiv preprint arXiv:2511.13027*, 2025.

Samuel Marks and Max Tegmark. The geometry of truth: Emergent linear structure in large language model representations of true/false datasets. In *First Conference on Language Modeling*, 2024.

Neel Nanda, Andrew Lee, and Martin Wattenberg. Emergent linear representations in world models of self-supervised sequence models. In Yonatan Belinkov, Sophie Hao, Jaap Jumelet, Najoung Kim, Arya McCarthy, and Hosein Mohebbi (eds.), *Proceedings of the 6th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pp. 16–30, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.blackboxnlp-1.2. URL https://aclanthology.org/2023.blackboxnlp-1.2/.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Shrey Pandit, Austin Xu, Xuan-Phi Nguyen, Yifei Ming, Caiming Xiong, and Shafiq Joty. Hard2verify: A step-level verification benchmark for open-ended frontier math. *arXiv preprint arXiv:2510.13744*, 2025.

Kiho Park, Yo Joong Choe, and Victor Veitch. The linear representation hypothesis and the geometry of large language models. *arXiv preprint arXiv:2311.03658*, 2023.

Nina Rimsky, Nick Gabrieli, Julian Schulz, Meg Tong, Evan Hubinger, and Alexander Turner. Steering llama 2 via contrastive activation addition. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15504–15522, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.828. URL https://aclanthology.org/2024.acl-long.828/.

Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. Rewarding progress: Scaling automated process verifiers for LLM reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.

Kartik Sharma and Rakshit Trivedi. COLD-steer: Steering large language models via in-context one-step learning dynamics. In *The Fourteenth International Conference on Learning Representations*, 2026.

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*, 2026a.

Janvijay Singh, Austin Xu, Yilun Zhou, Yefan Zhou, Dilek Hakkani-Tür, and Shafiq Joty. On the shelf life of finetuned LLM-judges: Future proofing, backward compatibility, and question generalization. In *The Fourteenth International Conference on Learning Representations*, 2026b.

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.

Alexander Matt Turner, Lisa Thiergart, Gavin Leech, David Udell, Juan J Vazquez, Ulisse Mini, and Monte MacDiarmid. Steering language models with activation engineering. *arXiv preprint arXiv:2308.10248*, 2023.

Hieu M Vu and Tan M Nguyen. Angular steering: Behavior control via rotation in activation space. *arXiv preprint arXiv:2510.26243*, 2025.

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

Weixuan Wang, JINGYUAN YANG, and Wei Peng. Semantics-adaptive activation intervention for LLMs via dynamic steering vectors. In *The Thirteenth International Conference on Learning Representations*, 2025.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Zhengxuan Wu, Aryaman Arora, Atticus Geiger, Zheng Wang, Jing Huang, Dan Jurafsky, Christopher D Manning, and Christopher Potts. Axbench: Steering LLMs? even simple baselines outperform sparse autoencoders. In *Forty-second International Conference on Machine Learning*, 2025.

Chen Xiong, Zhiyuan He, Pin-Yu Chen, Ching-Yun Ko, and Tsung-Yi Ho. Steering externalities: Benign activation steering unintentionally increases jailbreak risk for large language models. *arXiv preprint arXiv:2602.04896*, 2026.

Wei Xiong, Hanning Zhang, Nan Jiang, and Tong Zhang. An implementation of generative prm, 2024.

Wei Xiong, Wenting Zhao, Weizhe Yuan, Olga Golovneva, Tong Zhang, Jason Weston, and Sainbayar Sukhbaatar. Stepwiser: Stepwise generative judges for wiser reasoning. *arXiv preprint arXiv:2508.19229*, 2025.

Austin Xu, Xuan-Phi Nguyen, Yilun Zhou, Chien-Sheng Wu, Caiming Xiong, and Shafiq Joty. Foundational automatic evaluators: Scaling multi-task generative evaluator training for reasoning-centric domains. In *The Fourteenth International Conference on Learning Representations*, 2026.

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.

Zhangyue Yin, YuHong Sun, Xuanjing Huang, Xipeng Qiu, and Hui Zhao. Error classification of large language models on math word problems: A dynamically adaptive framework. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, November 2025.

Zejia You, Chunyuan Deng, and Hanjie Chen. Spherical steering: Geometry-aware activation rotation for language models. *arXiv preprint arXiv:2602.08169*, 2026.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng YU, Zhengying Liu, Yu Zhang, James Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

Kaiwen Zha, Zhengqi Gao, Maohao Shen, Zhang-Wei Hong, Duane S Boning, and Dina Katabi. RL tango: Reinforcing generator and verifier together for language reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. In *The Thirteenth International Conference on Learning Representations*, 2025a.

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

Zhenyu Zhang, Xiaoxia Wu, Zhongzhu Zhou, Qingyang Wu, Yineng Zhang, Pragaash Ponnusamy, Harikaran Subbaraj, Jue Wang, Shuaiwen Leon Song, and Ben Athiwaratkun. Understanding and steering the cognitive behaviors of reasoning models at test-time. *arXiv preprint arXiv:2512.24574*, 2025c.

Jiachen Zhao, Yiyou Sun, Weiyan Shi, and Dawn Song. Can aha moments be fake? identifying true and decorative thinking steps in chain-of-thought. *arXiv preprint arXiv:2510.24941*, 2025a.

Jian Zhao, Runze Liu, Kaiyan Zhang, Zhimu Zhou, Junqi Gao, Dong Li, Jiafei Lyu, Zhouyi Qian, Biqing Qi, Xiu Li, and Bowen Zhou. Genprm: Scaling test-time compute of process reward models via generative reasoning. *arXiv preprint arXiv:2504.00891*, 2025b.

Yu Zhao, Alessio Devoto, Giwon Hong, Xiaotang Du, Aryo Pradipta Gema, Hongru Wang, Xuanli He, Kam-Fai Wong, and Pasquale Minervini. Steering knowledge selection behaviours in LLMs via SAE-based representation engineering. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 5117–5136, Albuquerque, New Mexico, April 2025c. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.264. URL https://aclanthology.org/2025.naacl-long.264/.

Zekai Zhao, Qi Liu, Kun Zhou, Zihan Liu, Yifei Shao, Zhiting Hu, and Biwei Huang. Activation control for efficiently eliciting long chain-of-thought ability of language models. *arXiv preprint arXiv:2505.17697*, 2025d.

Chujie Zheng, Zhenru Zhang, Beichen Zhang, Runji Lin, Keming Lu, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. Processbench: Identifying process errors in mathematical reasoning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1009–1024, 2025.

Yefan Zhou, Austin Xu, Yilun Zhou, Janvijay Singh, Jiang Gui, and Shafiq Joty. Variation in verification: Understanding verification dynamics in large language models. In *The Fourteenth International Conference on Learning Representations*, 2026.

Andy Zou, Long Phan, Sarah Chen, James Campbell, Phillip Guo, Richard Ren, Alexander Pan, Xuwang Yin, Mantas Mazeika, Ann-Kathrin Dombrowski, et al. Representation engineering: A top-down approach to ai transparency. *arXiv preprint arXiv:2310.01405*, 2023.

APPENDIX

A PROMPT TEMPLATES

We use three prompt templates for step-wise verification evaluation, described below.

Basic Prompt. This is the standard evaluation template provided by Zheng et al. (2025), which instructs the verifier to review the solution paragraph by paragraph and return the index of the first erroneous step.

Basic Prompt for ProcessBench

```
### User Prompt
The following is a math problem and a solution (split into paragraphs, enclosed with tags and
indexed from 0):

[Math Problem]

{problem}

[Solution]

{tagged_response}

Your task is to review and critique the solution paragraph by paragraph. Once you identify an error
in a paragraph, return the index of the paragraph where the earliest error occurs. Otherwise, return
the index of -1 (which typically denotes ‘‘not found’’).

Please put your final answer (i.e., the index) in \\boxed{{}}.
```

Optimized Prompt for ProcessBench. This prompt is adapted from Zhang et al. (2025b), which was originally used by LLM-as-judge to construct PRM training data. It extends the basic prompt with instructions for more detailed analysis, including explicit recalculation and reflection. It also instructs the model to be "as critical as possible", aiming to increase verifier strictness and improve error detection. This prompt achieves F1 gains across all models and subsets on ProcessBench.

Optimized Prompt for ProcessBench

```
### User Prompt
The following is a math problem and a solution (split into paragraphs, enclosed with tags and
indexed from 0):

[Math Problem]

{problem}

[Solution]

{tagged_response}

Your task is to review and critique the solution paragraph by paragraph. Once you identify an error
in a paragraph, return the index of the paragraph where the earliest error occurs. Otherwise, return
the index of -1 (which typically denotes ‘‘not found’’).

Review each paragraph carefully for mathematical correctness and logical consistency. Be as critical
as possible.
When you analyze each paragraph, you should use proper verification, recalculation, or reflection to
indicate whether it is logically and mathematically valid. Please elaborate on the analysis process
carefully.

Please put your final answer (i.e., the index) in \\boxed{{}}.
```

Optimized Prompt for Hard2Verify. The ProcessBench optimized prompt does not transfer well to Hard2Verify, where it fails to improve the thinking model Qwen3-8B-thinking. We therefore use a separate prompt adapted from Pandit et al. (2025), which includes a system prompt instructing the model to act as a "strict, reliable math grader" and provides more structured instructions for checking mathematical correctness and logical consistency.

Optimized Prompt for Hard2Verify

```
### System Prompt
You are a strict, reliable math grader performing an Error identification task.
Your task is to identify the first incorrect paragraph in a mathematical solution.
Return your answer using the exact format requested by the user.

### User Prompt
The following is a math problem and a solution (split into paragraphs, enclosed with tags and
indexed from 0):

[Math Problem]

{problem}

[Solution]

{tagged_response}

Your task is to identify the first incorrect paragraph in the solution.

Instructions:
- Review each paragraph carefully for mathematical correctness and logical consistency
- A paragraph is incorrect if it contains mathematical errors, logical inconsistencies, or is based
on errors from previous paragraphs
- Find the FIRST paragraph that is incorrect (0-indexed)
- If ALL paragraphs are correct, return -1

Provide your reasoning and then put your final answer (i.e., the index) in \\boxed{{}}.
```

FARE Prompt

```
### System Prompt
Please act as an impartial judge and evaluate the quality of the response provided by an AI
assistant to the user prompt displayed below. You will be given the assistant's solution to a math
problem, which is split into steps, starting with a <step [step number]> tag, where [step number] is
indexed from 0. Your job is to identify which step an error occurs, if an error is present.
When evaluating the solution, consider each step separately. Evaluate the content of each step for
correctness. If you encounter a mistake at <step [step number]>, output [step number] as your
Verdict. If the full response is error free, then select step number -1. Avoid any biases, such as
length of step, or stylistic elements like formatting.

Here are some rules for evaluation.
(1) The assistant's answer does not need to be complete or arrive at a final solution. You may
receive a partially complete response. Your job is to assess the quality of each step.
(2) When evaluating the assistant's answer, identify any mistakes or inaccurate information. Focus
on the content each step and determine if the step is logically valid.
(3) For each step, you should provide an explanation of your assessment. If you find an error,
describe the nature and cause of the error.
(4) Avoid any biases, such as answer length, or stylistic elements like formatting.

Before providing an your final verdict, think through the judging process and output your thoughts
as an explanation
After providing your explanation, you must output the corresponding step number with an error. Use
the following format:
Explanation: Your explanation here
Verdict: The step number with the error or -1 if no error occurs

### User Prompt
[User Question]
{instruction}

[The Start of Assistant's Answer]
{response}
[The End of Assistant's Answer]
```

Prompt for Verification Paragraph Identification. This prompt is used by the LLM-based identifier (gpt-5-mini) in the paragraph-identification ablation of Section 5.3, as an alternative to the keyword heuristic in Appendix D.1.

Prompt for Paragraph Identification

```
Below is a verifier's critique of a multi-step math solution, split into {len(eligible_segments)}
segments (listed in order).
Steps are labeled paragraph_0, paragraph_1, etc.

Task: Find the index of the FIRST segment where {verdict_condition}.

Rules:
1. Only consider segments that come at or after the first segment that mentions
paragraph_{item_label} or paragraph {item_label} - ignore everything before that point.
2. The verdict must be about THIS specific step (paragraph_{item_label}), not a different step or
the overall solution conclusion.{next_para_note}
3. If a short summary or conclusion line appears after a longer analysis segment that already
qualifies, pick the earlier analysis segment.

Segments:
{segments_text}

Return JSON only: {{segment_index: <int>}}
Use -1 if no segment qualifies.


## if identifying false acceptance paragraphs:
verdict_condition=segment explicitly labels or declares paragraph_{item_label} as correct, valid, or
okay - e.g. using words like 'correct', 'okay', 'valid', 'right', 'accurate', 'no error', or any
similar explicit verdict word. Analytical reasoning that merely implies correctness without an
explicit verdict label does NOT count.

## if identifying true rejection paragraphs:
verdict_condition=segment explicitly labels or declares paragraph_{item_label} as wrong, erroneous,
or problematic - e.g. using words like 'error', 'incorrect', 'mistake', 'flaw', 'inconsistency',
'wrong', 'invalid', 'fails', or any similar explicit verdict word. Analytical reasoning that merely
implies an error through logic, without an explicit verdict label, does NOT count.
```

B Additional Results

Table 6 provides the complete set of case studies summarized in Table 1 of the main paper, including two additional examples (one under-critical and one over-critical). All examples use Qwen3-8B on ProcessBench with the basic prompt.

In the second under-critical case (row 2), the baseline verifier blindly repeats the solution's algebraic expansion without independently checking the intermediate steps, concluding that $a^4 - 152a^2 + 196 = 0$ is correct. After steering with $\boldsymbol{d}_{\text{strict}}$, the verifier performs its own expansion, arrives at $a^4 - 197a^2 + 196 = 0$, and correctly identifies the discrepancy. In the second over-critical case (row 4), the baseline verifier acknowledges that the final result is correct but flags the use of an approximate decimal value as an error in "reasoning and precision." After steering with $\boldsymbol{d}_{\text{lenient}}$, the verifier recognizes that the approximation is consistent with the exact value $\sqrt{8}$ and correctly accepts the step.

C Prevalence of the \n\n Paragraph Format

Step-wise verification outputs use the \n\n paragraph format across open-source model families. Across all 3,400 problems in ProcessBench, the verification CoT contains at least three \n\n separators for almost every output under the optimized prompt: 3,400 for Qwen2.5-72B, 3,392 for Llama-3.3-70B, and 3,399 for Gemma-3-12B. Together with our Qwen3 and GPT-OSS results (Table 3), this spans five open-source families, indicating that \n\n-separated verification paragraphs are the norm rather than an exception.

D Implementation and Hyperparameters

This section describes each component of `VerifySteer` and how its settings were determined. Appendix D.1 gives the keyword heuristic used to label acceptance and rejection verification paragraphs. Appendix D.2 describes the construction of validation set for building steering vector and training

	Solution Step	Baseline Incorrect Verification	Steered ($+\boldsymbol{d}_{\text{strict}}$ / $+\boldsymbol{d}_{\text{lenient}}$)
Under-critical	*After using 25 bandages, the remaining bandages are* $(x-16+100)-25 = x+69-25 = x+44$.	**Analysis**: This is correct. They added 100 bandages, used 25, so the remaining is $x-16+100-25 = x+69-25 = x+44$. **No error**.	**Analysis**: This paragraph contains an **error**. The expression $(x-16+100)-25$ simplifies to $x+84-25 = x+59$, not $x+44$. The error is in the arithmetic: $-16+100 = 84$, not 69. So the correct remaining bandages should be $x+59$, not $x+44$.
	...we get $(49-a^2)(4-a^2) = 144a^2$. *Expanding and simplifying, we have* $196-8a^2+a^4 = 144a^2$, *which simplifies to* $a^4-152a^2+196=0$.	**Correct.** Substituting $b^2 = 49-a^2$ into the equation $b^2(4-a^2) = 144a^2$ gives $(49-a^2)(4-a^2) = 144a^2$. Expanding and simplifying leads to $a^4-152a^2+196=0$.	Analysis: Let's check the substitution and expansion:- Substitute (...). So the equation becomes: $196-53a^2+a^4 = 144a^2$ \n $a^4-197a^2+196=0$ - But the solution says $a^4-152a^2+196=0$, which is incorrect.
Over-critical	*First, calculate the area of the larger rectangle. The larger rectangle has a length of 4 yards and a width of 5 yards.* [*Area of the larger rectangle* $= 4\times5 = 20$ *square yards*]	**Analysis**: ...However, the dimensions are not clearly defined in the diagram, and the interpretation of which dimension corresponds to which side is ambiguous. - **Conclusion**: The error is in the **interpretation of the dimensions**, ...	This is also correct. The diagram labels "4 yds" at the top and "5 yds" on the left side, so the larger rectangle is 4 yards wide and 5 yards tall. The area is indeed $4\times5 = 20$ square yards.
	We are asked to find x^2. *Thus, we calculate:* [$x^2 = (2.8284)^2 = 8$]	Analysis: This is incorrect. ... While ..., the reasoning is slightly misleading because it uses an approximate decimal value instead of an exact expression. However, the final result is still correct. But the earliest error is in the reasoning and precision of the calculation.	This is also correct. $(2.8284)^2$ is approximately 8, and since the coordinates of E were given with high precision, this suggests that the exact value is $\sqrt{8}$, and thus $x^2 = 8$.

Table 6: **Representative case studies of verification failures and their correction by hidden-state steering.** Qwen3-8B on ProcessBench. The top two rows show under-critical cases, where the baseline verifier falsely accepts an erroneous solution step, and the bottom two rows show over-critical cases, where it falsely rejects a correct step. The "Steered" column shows that applying $\boldsymbol{d}_{\text{strict}}$ corrects false acceptances into true rejections, while applying $\boldsymbol{d}_{\text{lenient}}$ corrects false rejections into true acceptances.

probe. Appendix D.3 gives the correctness probe training setup. Appendix D.4 describes how the steering layers are selected per model using the ActPRM development split. Appendix D.5 describes how the scalar steering hyperparameters, such as the steering strength and the delimiter-level gate threshold, are selected on the same development split, once per steering vector.

D.1 Keywords for Identifying Verification Paragraphs

As described in Section 4.2, we use a simple keyword-based heuristic to identify acceptance and rejection verification paragraphs. We first localize the relevant candidate paragraph using the step index, either from the ground-truth first-error annotation or from the verifier's predicted first error, together with the paragraph marker "paragraph i". We then use keyword matching as a consistency check to confirm that the localized paragraph contains the expected verdict cue and to filter out paragraphs with conflicting cues. Specifically, an acceptance paragraph (either false-acceptance or true-acceptance) must contain at least one required acceptance cue and none of the excluded cues; false-rejection and true-rejection paragraphs are identified analogously. We additionally exclude paragraphs containing "let's" or "let me", which frequently appear in thinking-model transition sentences (e.g., "let me check if this is correct") without conveying an actual verification verdict. The full keyword list is shown in Table 7. In Section 5.3, we show that this heuristic is reliable and achieves nearly identical performance to an LLM-based identifier.

D.2 Validation Set Construction.

We construct validation sets from ActPRM for both steering vector extraction and correctness probe training. ActPRM is built on problems from NuminaMath-1.5 (LI et al., 2024), which provides source labels for each problem. We exploit these labels to sample subsets that are aligned with ProcessBench subsets in terms of problem difficulty and type, using fuzzy matching on problem text to identify source-aligned examples. Specifically, for the GSM8K subset, we randomly sample 1,000 math word problems from ActPRM. For the MATH subset, we collect 4,224 examples whose source problems are matched to the MATH training set (Hendrycks et al., 2021) and MetaMath (Yu et al., 2024). For OlympiadBench and Omni-MATH, we sample 5,132 olympiad-level examples. For Hard2Verify,

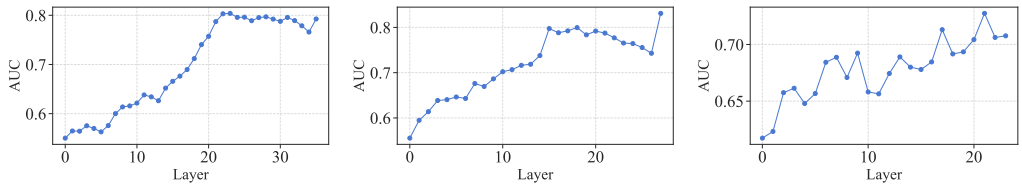
Acceptance	**Required cues**: correct, okay, no error
	Excluded cues: incorrect, the correct, not correct, **not** correct, let me, let's
Rejection	**Required cues**: error, incorrect, issue, mistake, flaw, inconsistency, not correct, wrong
	Excluded cues: no/any error, any explicit/immediate/-mathematical error, no immediate/mathematical error, is logically/mathematically correct, not a mathematical error, does not contain an error, correct, let me, let's

Table 7: Keyword-based rules used to identify acceptance and rejection verification paragraphs. A paragraph is labeled as Acceptance if it contains at least one required acceptance cue and none of the excluded cues; the same rule is applied analogously for Rejection.

we sample 812 competition-level proof problems using keyword matching of competition names (e.g., IMO, Putnam, USAMO) following the problem categorization in Pandit et al. (2025). For each sample, we generate 16 verification rollouts using the verifier model to be steered with the optimized prompt.

D.3 Correctness Probe Training.

The correctness probe f_ϕ is a three-layer MLP with ReLU activations and dropout 0.1. We train with cross-entropy loss using Adam with learning rate 10^{-5}, cosine learning-rate scheduler, weight decay 10^{-2}, batch size 32, and a maximum of 300 epochs. These hyperparameters are shared across all settings. The probe input layer is selected based on validation AUC: layer 23 for Qwen3-8B and Qwen3-8B-thinking, layer 17 for Qwen3-1.7B and FARE-20B. For pooling, we use mean pooling for Qwen3-8B and Qwen3-8B-thinking, and last-token pooling for Qwen3-1.7B. For ProcessBench, we train on 6,190 ActPRM samples with a held-out validation set of 2,310 samples. For Hard2Verify, we train on 2,448 samples with 1,050 validation samples.



(a) Qwen3-8B (b) Qwen3-1.7B (c) FARE-20B

Figure 4: Validation AUC of linear classifier on classifying delimiter tokens before the true rejection and false acceptance paragraphs.

D.4 Steering Layer Selection.

For steering layer selection, we follow the criterion and procedure used in prior work. The criterion is based on the linear separability (Park et al., 2023) of LLM concepts and behaviors in hidden states: Li et al. (2025); Zhang et al. (2025c); Lin et al. (2025) identify the steering layer as the one where the target features are most separable. Concretely, we sample 1,000 problems from ActPRM, generate 16 verification rollouts per problem, and collect delimiter-token hidden states preceding true rejection and false acceptance paragraphs (identified using the keywords in Appendix D.1) to form the training set. We repeat the same collection on a disjoint set of 1,000 problems to form the validation split. Following Li et al. (2025); Lin et al. (2025), for each layer we fit a linear classifier to distinguish true rejection from false acceptance hidden states and report the validation AUC. A higher AUC indicates that the two verification patterns are more separable at that layer. Figure 4 shows the AUC curves across layers for the three models we study. Following Lin et al. (2025), for each model we

first shortlist the top-6 layers by validation AUC, and then fix the two final steering layers based on steered verification performance on the same development split.

D.5 Steering Strength and Threshold Selection.

All selection decisions are made on a held-out ActPRM development split, which is disjoint from the pools used for steering vector construction. The search is a single round, run once per steering vector, and the selected configuration is applied to ProcessBench and Hard2Verify. For τ_l we choose between 0.5 and 0.6, and always set τ_h to 0.7. For α and ρ, we randomly sample five (α, ρ) pairs from a grid, α from 1.0 to 3.0 in steps of 0.5 and ρ from 0.0 to 0.6 in steps of 0.1, evaluate each on the development split and keep the single best.

Table 8 summarizes the steering hyperparameters for each model and benchmark setting. For all settings, `VerifySteer-Uni` uses only $\boldsymbol{d}_{\text{strict}}$ with routing threshold τ_l, while `VerifySteer-Bi` additionally applies $\boldsymbol{d}_{\text{lenient}}$ with routing thresholds (τ_l, τ_h). ρ is the delimiter-level cosine-similarity threshold introduced in Equation (3). Appendix D.5 examines sensitivity to these hyperparameters on ProcessBench GSM8K.

Benchmark	Model	Variant	Layers	α_{strict}	α_{lenient}	τ_l	τ_h	ρ_{strict}	ρ_{lenient}
ProcessBench	Qwen3-1.7B	`VerifySteer-Uni`	{16, 18}	3.0	–	0.6	–	0.6	–
		`VerifySteer-Bi`			0.5		0.7		0.4
	Qwen3-8B	`VerifySteer-Uni`	{22, 23}	{1.0, 2.0}	–	0.5	–	0.0	–
		`VerifySteer-Bi`			1.5		0.7		0.0
Hard2Verify	Qwen3-8B-thinking	`VerifySteer-Uni`	{22, 23}	1.5	–	0.6	–	0.0	–
		`VerifySteer-Bi`			1.5		0.7		0.1
	FARE-20B	`VerifySteer-Uni`	{17, 21}	1.0	–	0.6	–	0.4	-
		`VerifySteer-Bi`			1.0		0.7		0.4

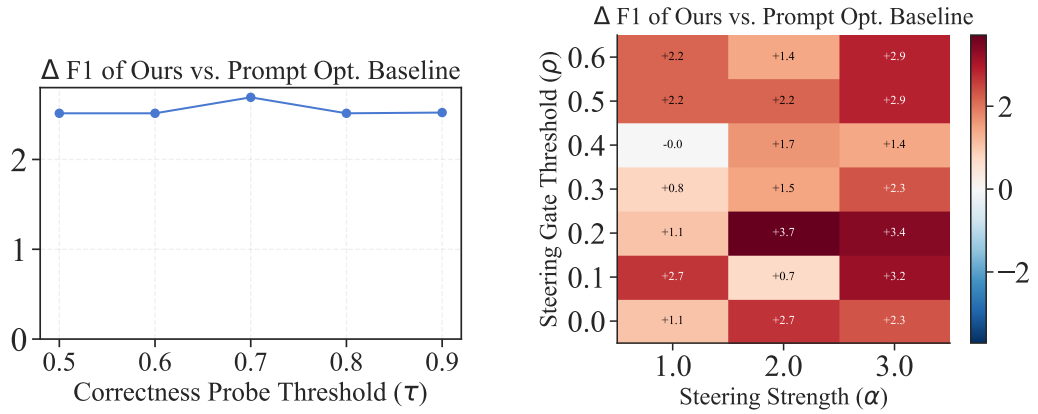
Table 8: Steering hyperparameters for each model and benchmark setting. Selection is run once per vector on a held-out ActPRM split. Qwen3-8B on ProcessBench uses three steering vectors, so α_{strict} takes two distinct values: 2.0 for the GSM8K and MATH vectors, and 1.0 for the vector shared by OlympiadBench and Omni-MATH.

Hyperparameter sensitivity. Figure 5 examines sensitivity of `VerifySteer-Uni` on ProcessBench GSM8K using Qwen3-8B with greedy decoding. Varying the correctness-probe threshold τ_l from 0.5 to 0.9 keeps the F1 gain over prompt optimization between 2.5 and 2.7 points. Across the 3×7 grid of steering strengths $\alpha \in \{1.0, 2.0, 3.0\}$ and gate thresholds $\rho \in \{0.0, 0.1, \ldots, 0.6\}$, 20 of 21 configurations improve F1, with gains up to 3.7 points; the remaining configuration is approximately tied with the baseline at the reported precision. The gains therefore occur across multiple configurations rather than at a single optimum.

E Computational Cost

We distinguish two types of compute for `VerifySteer`: the per-sample test-time inference cost reported in Tables 2 and 3, and a one-time development cost for steering vector construction and correctness probe training. The development cost is paid once and reused for every subsequent verification call, so it does not scale with the number of test instances. As the number of calls grows, `VerifySteer`'s amortized per-call compute converges to the reported per-sample TFLOPs. We report test-time cost because it determines compute at deployment scale and is the cost relevant to the self-consistency comparison. This distinction is standard and also applies to the baselines: the prompt optimization baseline, both alone and with self-consistency, requires a prompt-engineering search whose cost is not reported in the works we adopt the prompts from (Zheng et al., 2025; Zhang et al., 2025b; Pandit et al., 2025).

Probe Training. The correctness probe is a three-layer MLP whose input is a single hidden-state vector, so training is lightweight. For Qwen3-8B, training one probe takes 190.35 seconds on average over four runs on a single A6000 GPU, or about 75 TFLOPs under standard FLOP estimates. Since



(a) Correctness-probe threshold τ_l.

(b) Steering strength α and gate threshold ρ.

Figure 5: F1 gain of `VerifySteer-Uni` over prompt optimization when varying steering hyperparameters. Results use Qwen3-8B with greedy decoding on ProcessBench GSM8K.

one `VerifySteer` verification call costs about 21 TFLOPs (Table 2), this cost is recovered within a few calls.

Steering Vector Construction. Steering vector construction is the main development cost. For Qwen3-8B on GSM8K, we sample 16 verification rollouts for each of 1K ActPRM problems, which takes about one hour on 8 A6000 GPUs with vLLM inference, or about 3.7×10^5 TFLOPs based on generated token counts and a standard inference FLOP estimate. Including this full construction cost, `VerifySteer`'s total compute is lower than that of `Cons@4` after about 6K test-time calls, a small number at deployment scale, after which the amortized per-call cost converges to the value reported in Table 2.

Extending to New Settings. The overhead of applying `VerifySteer` to a new model and domain is the two development costs above, re-evaluated for the new setting. Construction cost scales with the model's parameter count and the number of training problems, and a moderate set is sufficient: around 1K problems is enough, and we obtain the Hard2Verify results using 812 proof problems. Probe training cost scales with the model's hidden dimension. In both cases the overhead is a one-time cost reused across every test instance, so the amortization above applies.

F Downstream Effect of Verifier TPR and TNR on Test-Time Scaling

On Hard2Verify, `VerifySteer` raises TNR while lowering TPR (Table 3). We analyze how this shift in the verifier's operating point affects a downstream test-time scaling (TTS) method, localized rejection sampling. We trace a Pareto curve of TPR against TNR by tuning the steering hyperparameters and measure the TTS accuracy at each operating point.

Simulation. We estimate TTS accuracy by Monte Carlo simulation of localized rejection sampling, with one round given in Algorithm 1. In each round, the solver produces a multi-step solution and the verifier inspects it. If the verifier accepts, the pipeline stops. If it rejects, the verifier identifies the first incorrect step, keeps the prefix up to that step, and the solver regenerates the rest. This repeats until the verifier accepts a full solution or the round budget is exhausted. Each round is modeled from the verifier's measured TPR, TNR, and false-accept rate, so every operating point on the Pareto curve yields a TTS accuracy. We set the solver per-step accuracy p, the number of solution steps H, and the round budget R, and run 60,000 trials per configuration.

Algorithm 1: One trial of the localized rejection sampling simulation with a step-level verifier. The reported accuracy is the mean of the returned indicator over T independent trials.

input : step accuracy p; solution step H; verifier TPR t and TNR n; false accept rate f; round budget R

output : 1 if the final solution is fully correct, else 0

$x \leftarrow (x_1, \ldots, x_H)$ with each $x_j \sim \text{Bernoulli}(p)$ // x_j=1: step j is correct

for $r \leftarrow 1$ **to** R **do**

  draw $u \sim \text{Uniform}(0, 1)$

  if $x_j = 1$ *for all* j **then**

    // solution is fully correct

    if $u < t$ **then return** 1 // correct accept

    $k \leftarrow$ uniform random step in $\{1, \ldots, H\}$ // false alarm

  else

    $e \leftarrow \min\{\, j : x_j = 0 \,\}$ // first incorrect step

    if $u < n$ **then**

      $k \leftarrow e$ // correct localization

    else if $u < n + (1 - n)\, f$ **then**

      return 0 // false-accept

    else

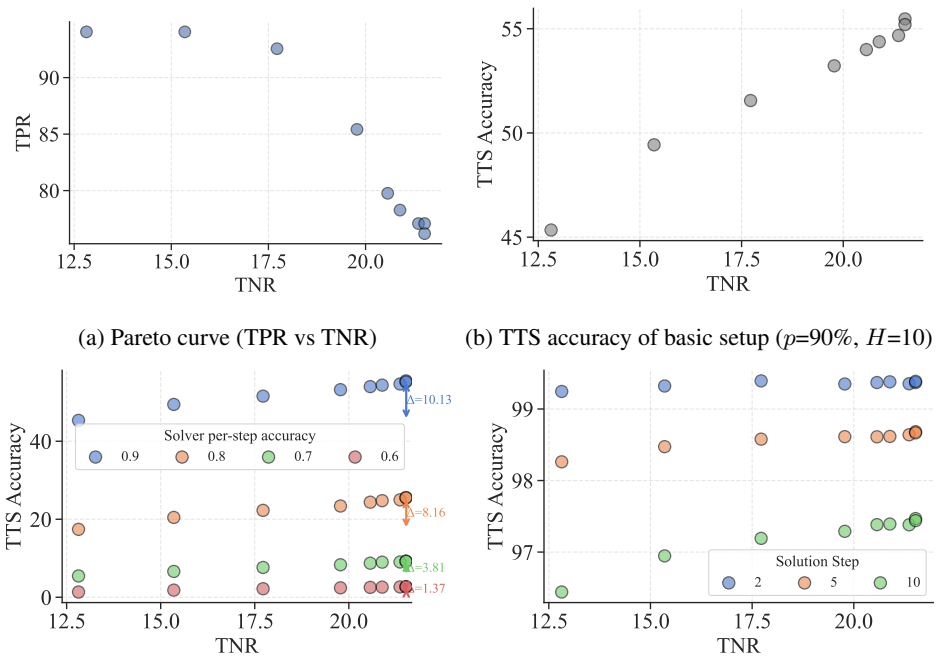
      $k \leftarrow$ uniform random step in $\{1, \ldots, H\} \setminus \{e\}$ // mislocalization

  resample $x_j \sim \text{Bernoulli}(p)$ for $j = k, \ldots, H$ // keep prefix $x_{1:k-1}$, redo the suffix

return $\mathbf{1}[\, x_j = 1 \text{ for all } j \,]$

Pareto curve. To span a range of operating points, we route between the basic-prompt verifier and the steered verifier with a routing threshold τ, swept over $[0, 0.025, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6]$. Here $\tau = 0$ recovers the basic prompt (TPR 94.0, TNR 12.8), and larger τ routes more inputs to the steered verifier. The resulting curve is shown in Figure 6(a). Most of the 18-point TPR span comes from prompt optimization rather than steering: prompt optimization alone lowers TPR from 94.0 to 78.6 and raises TNR from 12.8 to 18.0, while `VerifySteer` on top changes TPR only mildly, from 78.6 to 75.6, and raises TNR to 21.5.

Results. Higher TNR yields higher TTS accuracy in nearly all regimes. Under the common setup of solver per-step accuracy $p = 90\%$ and $H = 10$ steps (Figure 6(b)), accuracy rises steadily from 45.34 to 55.47 as TNR increases, despite the TPR drop, and the same trend holds as solver accuracy varies (Figure 6(c)). The only exception (Figure 6(d)) is a near-perfect solver ($p = 99.5\%$) with short solutions ($H = 2$), where a lower-TNR, higher-TPR verifier wins by 0.04 points (99.39 to 99.35). Higher TNR helps because it reduces the unrecoverable error from false accepts, whereas false rejections remain recoverable through regeneration. The shift is thus beneficial whenever the pipeline can retry after a rejection, and least useful only when the unrecoverable error is already rare.



(a) Pareto curve (TPR vs TNR)

(b) TTS accuracy of basic setup (p=90%, H=10)

(c) Varying solver accuracy p (fixed H=10)

(d) Varying solution steps H (fixed p=99.5%)

Figure 6: **TTS accuracy of localized rejection sampling as the verifier is steered toward higher TNR, for Qwen3-8B-thinking on Hard2Verify.** (a) Sweeping the routing threshold τ increases TNR while lowering TPR. (b) Under the basic setup ($p = 90\%$, $H = 10$), TTS accuracy increases steadily as the verifier is steered toward higher TNR, despite the drop in TPR. (c) Higher TNR yields higher TTS accuracy across all solver accuracy levels. (d) Higher TNR is slightly less beneficial when the solution is short ($H = 2$) and solver accuracy is high, but otherwise it consistently improves accuracy.