LANGUAGE MODELS DO NOT RE-VERIFY COMPUTED VALUES IN THEIR OWN REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Language models exhibit increasingly remarkable capabilities on reasoning tasks, but it is unclear whether they spontaneously check values that are already part of their reasoning. In particular, we inject a single numerical error into model-generated execution traces for short mathematical programs across 14 language models from 5 developers, and study whether the models detect and correct the error or continue reasoning from it. Errors that require recomputation propagate at much higher rates than errors that can be caught by comparison alone. When the planted value explicitly contradicts a value written earlier in the trace, detection and correction rise up to 96-99% for stronger models. In contrast, when detecting the error requires even one arithmetic operation, unprompted continuation by the base models instead propagates the error in 88–100% of continuations, despite the models being able to compute the correct value when asked directly. This failure persists on most models even when they are given verification instructions and have both the information and computational ability needed to check the value, and it is not repaired by the activation-steering interventions we test. In contrast, fine-tuning on a few hundred planted-error examples are enough to teach some models to recheck previously computed values, and this behavior generalizes to harder errors than those seen during training. Our results expose a fragility in downstream self-correction even in highly capable models, and challenge the assumption that an incorrect intermediate computation will have a meaningful chance of being independently rechecked later in the reasoning process.

1 INTRODUCTION

Language models increasingly solve problems by producing long chains of intermediate steps (Nye et al., 2021; Wei et al., 2022; Kojima et al., 2022). Later steps depend on values produced earlier, which means that a wrong intermediate value can affect everything that follows. This raises a simple question that is rarely measured directly: when a false value is already present in a model's own reasoning, does the model continue from it?

Two trends make this question increasingly important. Reasoning models spend additional test-time computation reconsidering and revising their work (DeepSeek-AI et al., 2025; Snell et al., 2024). Agent systems routinely pass the output of one step into the next (Yao et al., 2023; Kapoor et al., 2024), so an unchecked error can affect later steps. Prior work on self-correction gives reasons for doubt (Huang et al., 2024; Stechly et al., 2024; Tsui, 2025), and related work has studied corrupted reasoning under a range of prompting setups. We first study ordinary continuations, where the model is given no indication that anything has changed, and then test whether verification instructions and other interventions can restore checking.

A model solves a short program by writing its execution trace one line at a time. We take a correct trace it produced, replace one intermediate value with a false one, and let the model continue. We compare errors that can be caught by rereading an earlier value with matched errors that require recomputation. Because the program is executable, we can determine whether the planted value propagates to the final answer without relying on a human or model judge (Section 3).

When the correct value can be recovered by comparing against something already written in the trace, strong models usually avoid propagating the error. When recovering it requires even one

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.

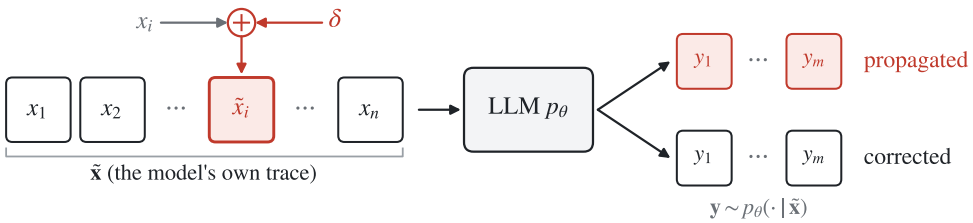


Figure 1: The model first produces a correct trace τ for a program. We change the value x_i at one intermediate step to $\tilde{x}_i$ and ask the model to continue from the modified trace. We then check whether the planted error propagates to the final answer or whether the model corrects it.

arithmetic operation, the same models almost always continue from the planted value instead. For Haiku 4.5, this transition occurs between zero and one operation.

This is not simply because the model lacks the information or ability to perform the arithmetic. Showing the operands beside the planted value does not restore checking, even when everything needed to detect the error appears on the same line. Explicit verification instructions also have little effect on most models we test. Together, these results suggest a gap between being able to verify an intermediate value and spontaneously doing so during continuation.

We make four main contributions:

- We introduce a controlled way to measure whether a planted error propagates through a model's previously correct reasoning, with outcomes determined by program execution rather than a learned judge.
- Across fourteen models, we find a sharp difference between errors that can be caught by reading an earlier value and errors that require recomputation.
- We show that, in our matched program traces, this difference can appear after a single arithmetic operation even when the operands are directly visible.
- We test interventions at the input, prompt, inference-time reasoning, representation, and training levels, including fine-tuning that substantially improves checking.

2 Related Work

Prefilling. If you force a model to begin its answer with a few tokens you chose, it will often continue as though it had written those tokens itself. This is what makes prefilling and priming attacks so effective (Qi et al., 2025; von Recum et al., 2026). The same behavior raises a natural question for reasoning: if a model is given a wrong prefix as its own, will it notice? Yee et al. (2024) studied a closely related question on GSM8K and found that models often recover from planted calculation errors, especially copying errors. We find much lower recovery once catching the error requires recomputation. Two differences help explain the gap. First, recovery in their setup depends partly on the particular problems and numbers: renumbering their GSM8K examples lowers GPT-4o's recovery from 66% to 52%. Second, their corrupted prefixes often stop in the middle of an unfinished calculation. Letting the model complete the statement before continuation lowers recovery by about 9 points. We instead perturb completed intermediate values and ask whether the model revisits them while continuing its reasoning.

Compounding error. Errors in intermediate reasoning matter because they can affect everything that comes after them. Hallucinations can snowball (Zhang et al., 2024), errors compound in multi-step tasks (Dziri et al., 2023) and during next-token generation (Bachmann & Nagarajan, 2024), and long reasoning traces become less reliable as they grow (Panda et al., 2026). This is especially important for agent systems, where the output of one step is often passed directly into the next (Yao et al., 2023; Kapoor et al., 2024), and for chain-of-thought monitoring, which relies on mistakes leaving some observable trace in the model's reasoning (Baker et al., 2025; Korbak et al., 2025). Prior work has studied several related questions. Lanham et al. (2023) corrupted chains of thought and measured whether the final answer changed. Other work found that invalid demonstrations can

have surprisingly little effect on accuracy (Wang et al., 2023; Schaeffer et al., 2023), or perturbed problem statements and measured the resulting change in performance (Shi et al., 2023; Chen et al., 2024; Mirzadeh et al., 2025). These studies largely ask how changes to the model's context affect its final answer. We instead ask whether a model notices an error inside the reasoning trace it is currently continuing. The closest work is the Self-Verification Dilemma (Long et al., 2026), which found that open models often revisit their previous reasoning but usually confirm their original answer. We study this question with planted errors whose detectability can be controlled directly.

Reasoning capabilities. Reasoning models are explicitly trained to inspect and revise their own work (DeepSeek-AI et al., 2025; Snell et al., 2024), so they might seem especially likely to catch a planted error. Prior work on self-correction gives reasons to be skeptical. Models asked to review their own answers often fail to improve them (Huang et al., 2024). Even when told that an error is present, they struggle to locate it (Tyen et al., 2024). Similar failures appear in planning (Valmeekam et al., 2023; Stechly et al., 2024) and on benchmarks designed specifically to test error correction (Tsui, 2025; Huang, 2026; Sun et al., 2026; Somov et al., 2026). These failures do not appear to reflect a simple lack of capability. Trained verifiers can detect computation errors reliably (Cobbe et al., 2021; Uesato et al., 2022; Lightman et al., 2024; Wang et al., 2024; Saunders et al., 2022; McAleese et al., 2024), and prompting models to explicitly check their work can reduce errors (Madaan et al., 2023; Dhuliawala et al., 2024; Ling et al., 2023). This suggests an important distinction between being able to verify a computation and spontaneously deciding to do so. Related work also finds that models can internally represent whether a reasoning step is correct without changing their behavior accordingly (Bertolazzi et al., 2025), and that models can defer to information already present in context even when it is wrong (Sharma et al., 2024). Our experiments focus on this gap: when does a model actually use its verification ability while continuing its own reasoning? Because stated reasoning can diverge from what produces the final answer (Turpin et al., 2023; Arcuschin et al., 2025; Chen et al., 2025), we treat reasoning traces as supporting evidence rather than the basis of our main claims.

3 Problem setup: adversarial injections into a model's own reasoning

In this section, we present the setup considered in this paper and we introduce the research question that is the object of study. Specifically, given a language model, we ask it to solve a short program step by step under a trace perturbation. The trace perturbation entails taking a correct trace the model previously wrote, and changing one of the intermediate values before handing the trace back for the model to finish (Figure 1). We ask the question of how robust models are to these types of perturbations.

We study generations that consist of a program P of n integer assignments (Algorithm 1). Each line sets a variable to a constant or to a linear combination of earlier variables, represented generically by $x_i \leftarrow w_{i-1}^i x_{i-1} + \cdots + w_0^i x_0 + c^i$ at each line i where each $w_j^k \in \{0, 1\}$ for all $j, k = 0, \ldots, i-1$, $c^i \in \mathbb{Z}$ for all i, and the last line prints the result, $y \leftarrow x_n$. The model solves P step by step and writes a trace $\tau = (t_1, \ldots, t_n)$, with one line per statement and its computed value. We keep only traces where each line i has the correct value for x_i and the correct output y at the end of the program.

Algorithm 1 Program P as a sequence of computations

Input: $x_0,\ c^1, \ldots, c^n,\ w_0^1, \ldots, w_{n-1}^n$ **Output:** y

1: $x_1 \leftarrow w_0^1 x_0 + c^1$

2: $x_2 \leftarrow w_0^2 x_0 + w_1^2 x_1 + c^2$

3: $x_3 \leftarrow w_0^3 x_0 + w_1^3 x_1 + w_2^3 x_2 + c^3$

$\vdots$

n: $y \leftarrow w_0^n x_0 + \cdots + w_{n-1}^n x_{n-1} + c^n$

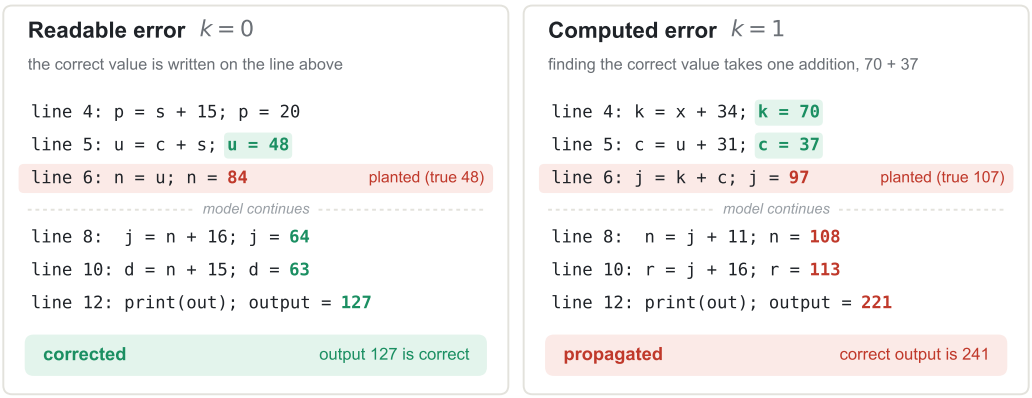


Figure 2: Two real examples illustrating the difference between readable and computed errors.

Given the sequence of computations in problem P, we study how robust models are to an adversarial disturbance in the computation. To that end, we introduce a disturbance as an additive perturbation to a value in an arbitrary line, i.e., $x_i \leftarrow w^i_{i-1}x_{i-1} + \cdots + w^i_0 x_0 + c^i + \delta$, where $\delta \in \mathbb{Z} \setminus \{0\}$ and line i in Algorithm 1 is chosen so that detecting the perturbation requires the desired verification depth. We define the *verification depth* k as the number of arithmetic operations required to recover x_i from values earlier in the trace. Given the structure of P, this can lead to two distinct types of errors:

- *Readable Error:* Error that has $k = 0$. The correct value already appears earlier in the trace, so the model can catch the error by comparison alone.
- *Computed Error:* Error that has $k \geq 1$. The correct value is not written anywhere in the trace, so catching the error requires recomputation.

In this paper, we study the robustness of models to this type of additive perturbation in simple arithmetic programs. To that end, we take problems that the models initially solved correctly, and we study the outcome of adding an additive perturbation somewhere along the computation. We distinguish three observed outcomes:

- **Explicitly corrected:** the model flags line i (where the disturbance was injected) as corrupted and reassigns its original value x_i. This outcome means that the perturbation was explicitly detected and corrected by the model.
- **Correct output:** the model reaches the same final value y it would have reached when correctly solving the undisturbed program P. A correct final output alone does not imply that the model noticed and repaired the disturbed line, since it may have reached the answer through another route.
- **Propagated:** the model reaches the final value $\tilde{y}$ obtained by carrying the perturbed value through the remainder of the computation. This outcome means that the perturbation propagated to the final answer.

We note that the population being studied is traces the model initially solved correctly, which is an estimation of the robustness but conditions comparisons across models and depths on different subsets of programs.

4 Methods to Elicit Recovery to Adversarial Injections

In this section, we test six interventions aimed at reducing the propagation of planted errors.

4.1 Showing the arithmetic

One possibility is that the model simply does not look back far enough to recover the inputs. If so, putting the arithmetic directly on the changed line should help. We vary how much of the

computation the line shows. In the *bare* form, it gives the operation and reported value, `V = a + b; V =` $\tilde{x}_i$. The *full* form also shows the operands, `V = a + b = 32 + 58; V =` $\tilde{x}_i$. The *partial* form shows only the last one or two operands. Because the operands are correct, the full line contains everything needed to see that the reported value is wrong.

We also ask the model directly, in the same context, `What is a + b?`, which tests whether it can perform the required computation at all. Two further controls, marking the planted value as unverified and printing the correct sum next to the wrong one, are reported in Appendix B.

4.2 Who wrote the trace?

The model may be more willing to trust a value that is described as coming from its own earlier reasoning. We test this by leaving the trace unchanged and changing only its stated author: the model itself, another language model, or a human. This manipulation tests whether the attribution label affects continuation. The actual provenance of the trace does not change.

4.3 Asking the model to check

Perhaps the model can catch the error but does not check unless it is told to. We compare four instruction conditions: no added instruction; a generic request to work carefully and verify; a targeted instruction to recompute a value before using it and correct it if necessary; and the same instruction with a worked example showing a wrong value being caught and fixed. The instruction is present both when the model first solves the program and when it continues the changed trace, so the continuation is not singled out by a new prompt suggesting that something is wrong. There is a tension between the baseline instruction to output only the remaining lines and the targeted instruction to revisit and correct an earlier value. We therefore report explicit corrections, correct final outputs without an explicit correction, and unresolved outputs separately for each prompt condition. Exact prompts are in Appendix B.

4.4 Giving the model time to think

Or catching the error may require more computation than ordinary continuation provides. We test this by varying inference-time reasoning within the same model while keeping everything else fixed. For an API model, we compare reasoning off with reasoning on. For an open-weight reasoning model, we compare direct continuation with continuation through its thinking channel. We disturb only the committed trace, never the model's deliberation. We also record whether the deliberation recomputes the planted value, but treat this only as supporting evidence because written reasoning need not faithfully reflect what determined the answer.

4.5 Activation steering

It is also possible that the model represents the difference between a correct and a planted value but does not use that information when it continues. We build pairs of traces that differ only in the value on line i, one correct and one planted, and train a linear probe w_ℓ on the residual stream h_ℓ at each layer ℓ to distinguish them. We then steer the model along the learned direction during continuation,

$$h_\ell \leftarrow h_\ell + \alpha w_\ell,$$

and test whether this reduces error propagation. We sweep a range of layers and steering strengths (Appendix B). We only consider settings under which the model still continues correct traces normally, so reducing propagation by simply disrupting generation does not count as a successful intervention. A failure of this intervention to reduce propagation does not imply that the relevant information is absent from the model. It only shows that the steering procedure we test does not successfully change the behavior.

4.6 Teaching the model to check

Finally, the model may simply not have learned to revisit values once they have been written. We fine-tune it on paired examples that differ only in whether the value on line i is correct. When the value is wrong, the target recomputes it, points out the mismatch, corrects the line, and continues

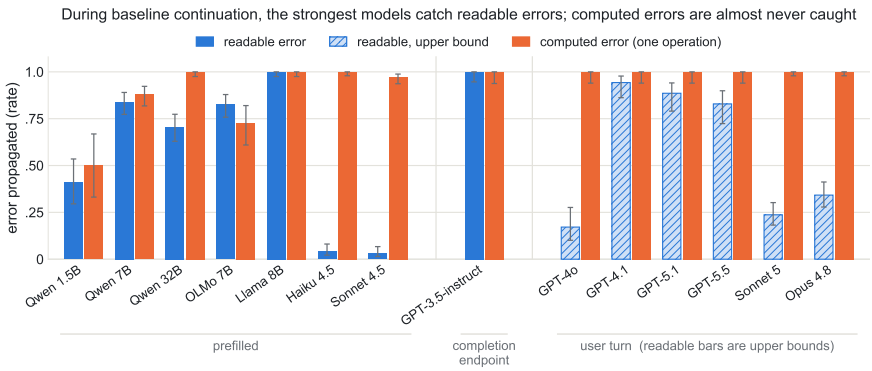


Figure 3: Propagation of a readable error (blue) and a one-operation computed error (orange), planted by the same edit, for fourteen models grouped by developer. Hatched readable bars are models measured through the user-turn method, which inflates this condition (Section 5.1); they are upper bounds. Whiskers are 95% Wilson intervals.

the trace. When the value is correct, the target simply continues. The model is therefore trained to check selectively rather than to always insert a verification step.

We train two versions on 300 programs each, yielding 600 training examples per version: one correct and one perturbed example for each program. One sees only errors at depth one; the other sees equal numbers at depths one, two, and three. We evaluate both on held-out programs across all depths, including depths not seen during training. The held-out programs use the same program templates and operand range as the training programs. Our primary evaluation uses the same set of possible line positions, and we separately test generalization to perturbations at line positions not seen during training. We also test whether the model starts flagging correct traces. Fine-tuning and optimization details are in Appendix B.

5 Experiments

We report results for fourteen models from five developers, ranging from a 1.5-billion-parameter open-weight model to frontier systems. The open-weight models include members of the Llama, Qwen, and OLMo families (Grattafiori et al., 2024; Qwen et al., 2024; OLMo Team, 2024). When the interface allows it, we place the perturbed trace directly in the assistant turn and continue from it. Otherwise, we place the trace in the user turn with a fixed instruction to continue. On Haiku 4.5 and Sonnet 4.5, which support both formats, computed-error propagation is 1.00 under either. One model is available only through a completion endpoint, where direct continuation is the native interface.

We attempted three additional models but could not obtain comparable measurements. Fable 5 and Opus 5 refuse the task before producing a continuation, while Fable 5.1 refuses a subset of the changed traces. (Appendix B).

Across verification depths, we hold both trace length and the distance from the changed line to its inputs fixed. Increasing depth therefore changes how much computation is required to check the value, not how far back the model must look.

5.1 Models catch readable errors and propagate computed ones

Across model families, developers, and scales, computed errors almost always propagate. At depth one, propagation is 100% for all five OpenAI models, three of four frontier Anthropic models, Llama-3.1-8B, and Qwen2.5-32B; Sonnet 4.5 is at 97% and Qwen2.5-7B at 88%(Figure 3).

Readable errors behave very differently. Stronger models usually catch them, while weaker open-weight models still propagate many.

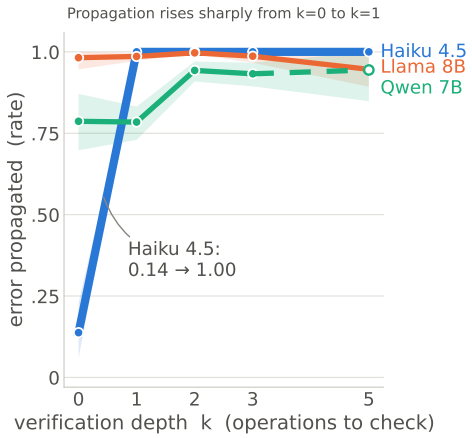


Figure 4: Propagation against verification depth k. For Haiku 4.5, propagation rises from 0.14 at $k = 0$ to 1.00 at $k = 1$. Bands show program-level 95% intervals.

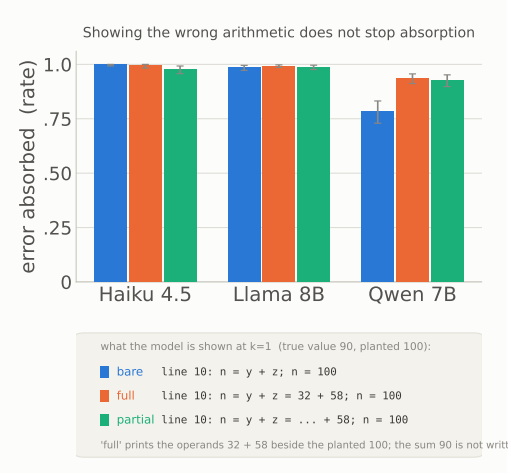


Figure 5: Propagation at $k = 1$ when the changed line shows the value only (bare), the full arithmetic, or the last operand(s) (partial). Showing the arithmetic does not reduce propagation: Haiku 4.5 and Llama 8B remain above 0.97, while Qwen 7B rises from 0.78 to 0.94. Whiskers show program-level 95% intervals.

For Haiku 4.5, error propagation rises from 14% at depth zero to 100% at depth one (Figure 4). This points to a sharp boundary between rereading a value and recomputing it: a single operation is enough for checking to disappear.

The remaining experiments test why this happens and whether any of the interventions in Section 4 restore checking.

5.2 Showing the arithmetic does not help

If the model propagates computed errors because it cannot retrieve the inputs, putting those inputs directly on the changed line should help. It does not (Figure 5). For Haiku 4.5, propagation remains above 97% in the bare, partial, and full forms at every depth, even when the operands on the line visibly add up to a different value than the one reported. Llama-3.1-8B is similarly unchanged at 99% across all three forms.

GPT-5.5 is the only model for which visibility makes a noticeable difference, and the effect is limited. With all operands shown, propagation is 95% at depths one and two; with only the final operands shown, it falls to 82% and 87%. By depths three and five, propagation returns to 100% in every form.

The model can do the arithmetic. Asked directly, in the same context, for the sum of the changed line's operands, it answers correctly in 100%, 100%, and 96% of cases for Haiku 4.5, Llama-3.1-8B, and Qwen2.5-7B, respectively. Two additional controls show the same pattern (Appendix B). The computation is available when asked for directly, but is rarely performed while the model continues the trace.

5.3 Stated authorship has little effect on computed-error propagation

Changing the stated author of an otherwise identical trace leaves computed-error propagation between 98% and 100% across all three models we test: Opus 4.8, Sonnet 5, and GPT-4.1. Attribution does affect readable errors. On Opus 4.8, propagation rises from 13% when the trace is attributed to the model itself to 32% when it is attributed to a human. We do not observe a similarly large effect for computed errors. However, because propagation is already near 100% in these conditions, the experiment has limited sensitivity to small differences in attribution.

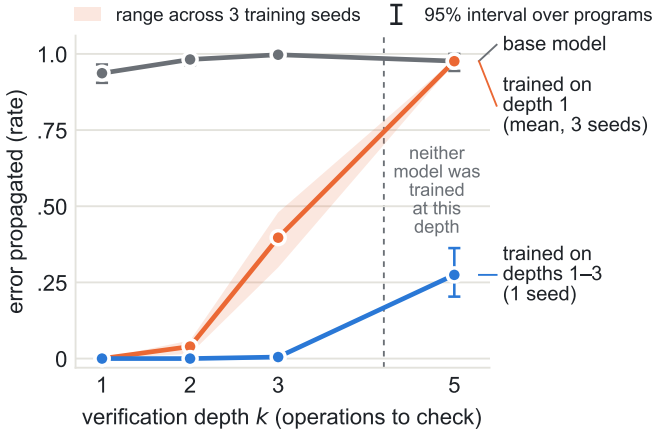


Figure 6: Error propagation after fine-tuning Qwen2.5-32B-Instruct. Training only at depth one eliminates propagation at depth one but generalizes only partially to deeper errors. Training across depths one to three generalizes much further, reducing propagation even at the unseen depth five.

5.4 Prompting helps GPT-5.5 only at shallow depth

Prompting barely changes the behavior of Haiku 4.5, Llama-3.1-8B, or Qwen2.5-7B. Even after being explicitly told to recompute values before using them, or shown an example in which a planted error is caught and fixed, the models propagate computed errors at essentially the same rate as before.

GPT-5.5 is the only model where prompting helps. At depth one, a targeted instruction cuts propagation to 50%, and adding a worked example nearly eliminates it, dropping propagation to 2%. In that condition, 53 of 60 continuations silently correct the planted value. But the improvement does not extend to deeper errors: at depth five, propagation remains between 98% and 100% under the same prompts. GPT-5.1 shows no such effect and error propagation stays at 100% at all depths.

5.5 Reasoning often skips the planted value

Turning on reasoning does not make checking systematic. For GPT-5.1, readable-error propagation drops from 92% with reasoning off to 27% with reasoning on, and five-operation propagation drops from 100% to 88%. One-operation errors barely change, from 100% to 99%. Interestingly, the model also spends substantially more reasoning tokens on five-operation errors than on one-operation errors.

DeepSeek-R1-Distill-7B shows a similarly small effect on computed errors. Propagation falls from 98% to 93% at depth one and from 95% to 93% at depth five when its reasoning channel is enabled.

5.6 Activation Steering does not restore checking

On Qwen2.5-7B, correct and planted values are distinguishable only in the final three of 28 layers, with held-out AUCs of 0.80, 0.77, and 0.98. Steering along these directions does not restore checking: propagation stays near the 90% baseline, and explicit checking never exceeds 5%.

5.7 Fine-tuning teaches checking that generalizes

Fine-tuning changes the behavior substantially on Qwen2.5-32B-Instruct (Figure 6). Training only on depth-one errors reduces propagation from 94% to 0% at depth one across all three training seeds on the matched evaluation cohort. This behavior generalizes beyond the training depth, but only partially: propagation remains between 1.5% and 5.9% at depth two and between 29.9% and 48.0% at depth three, before returning to 97.0–98.2% at depth five.

Training on errors from depths one, two, and three generalizes much further. In the single mixed-depth run, propagation is 0% at depths one and two, 0.5% at depth three, and 27.5% at the unseen depth five. At depth five, 71.7% of continuations explicitly reconstruct all six operands and compute the correct value. The learned behavior is also selective: across 720 correct traces, the model inserts an unnecessary check only three times, and none of these changes the correct final answer. Because we have only one training seed for the mixed-depth condition, these results do not provide an estimate of seed-to-seed variability.

The effect is also model-dependent. A single mixed-depth run on Llama-3.1-8B-Instruct eliminates propagation at depths one through three. In contrast, the same mixed-depth recipe does not improve Qwen2.5-7B-Instruct: depth-one propagation is 97.2% in a single run on nine matched programs, and the model also produces substantially more unnecessary checks on correct traces.

These results show that fine-tuning can teach a model to revisit previously computed values and that the behavior can generalize beyond the depths seen during training. They also show that a few hundred examples are not sufficient in general: the extent of generalization depends on both the training distribution and the model.

All results use the final training checkpoint; we do not select among intermediate checkpoints or use a validation set for checkpoint selection. The Qwen2.5-32B model was chosen before the fine-tuning experiments, while the mixed-depth recipe and cross-model follow-up experiments were added after observing earlier fine-tuning results. Exact model revisions, training hyperparameters, and the full selection timeline are reported in Appendix B.

6 Discussion and Conclusion

Our results suggest a practical precaution for systems that pass computed text between model steps: the receiving model should not be assumed to spontaneously audit values that already appear settled in the context. We do not directly test tool outputs, agent handoffs, or monitoring systems, so claims about those settings require separate experiments.

Several mechanisms could explain the behavior we observe. Models may learn that completed computations in a prefix are usually trustworthy, they may allocate little attention to recomputing settled values, or they may simply prefer to preserve and continue the given prefix. Our results do not distinguish among these accounts. The fine-tuning experiments show, however, that the behavior is not fixed: on Qwen2.5-32B, a few hundred planted-error examples substantially increase checking and generalize beyond the training depths, while the same recipe does not work on Qwen2.5-7B. A useful next experiment would hold the task and format fixed while varying the training distribution of correct and incorrect intermediate values.

Related behavior also appears outside our synthetic programs, but with meaningful variation across models. On word problems, computed-error propagation remains high overall, although Sonnet 4.5 propagates only 69% of worked errors and GPT-5.1 behaves atypically on readable errors (Appendix B). We also observe naturally occurring mistakes: among 1,396 such cases, 10 end with the correct final answer, while 20 contain an explicit correction remark (Appendix B). These measure different events, since an explicit remark need not produce the correct final answer and a correct final answer need not involve an explicit correction.

More broadly, these results separate having the capability to verify a value from actually using that capability during continuation. A model can have all the information needed to identify an error and still fail to invoke the relevant computation at the moment it matters. Future work should test whether this gap persists in richer reasoning domains and whether training can make verification reliable without causing unnecessary checking.Overall, models are much more likely to catch an error when the correct value is already available to compare against than when checking requires recomputation. Until this behavior is better understood, systems that pass computed values between model steps should treat those values as potentially unchecked rather than assuming the model will verify them on its own.

AI USE STATEMENT

We used generative AI tools (primarily Claude, Anthropic) to assist with code development, synthetic data generation, experiment design, analysis, literature search, figure generation, and drafting and editing the manuscript. All AI-assisted work, including experimental results, citations, and manuscript text, was reviewed and approved by the authors. Outcomes were graded mechanically by re-executing the programs rather than by a language model. The authors take responsibility for the final content of the paper.

REFERENCES

DeepSeek-AI, Daya Guo, Dejian Yang, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, 2025. arXiv:2501.12948.

Qwen, :, An Yang, et al. Qwen2.5 technical report, 2024. arXiv:2412.15115.

Iván Arcuschin, Jett Janiak, Robert Krzyzanowski, Senthooran Rajamanoharan, Neel Nanda, and Arthur Conmy. Chain-of-thought reasoning in the wild is not always faithful, 2025. arXiv:2503.08679.

Gregor Bachmann and Vaishnavh Nagarajan. The pitfalls of next-token prediction. In *International Conference on Machine Learning (ICML)*, 2024. arXiv:2403.06963.

Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation, 2025. arXiv:2503.11926.

Leonardo Bertolazzi, Philipp Mondorf, Barbara Plank, and Raffaella Bernardi. The validation gap: A mechanistic analysis of how language models compute arithmetic but fail to validate it. In *Proceedings of EMNLP*, 2025. arXiv:2502.11771.

Xinyun Chen, Ryan A. Chi, Xuezhi Wang, and Denny Zhou. Premise order matters in reasoning with large language models. In *International Conference on Machine Learning (ICML)*, 2024. arXiv:2402.08939.

Yanda Chen, Joe Benton, Ansh Radhakrishnan, et al. Reasoning models don't always say what they think, 2025. arXiv:2505.05410.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. arXiv:2110.14168.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models. In *Findings of the Association for Computational Linguistics: ACL*, 2024. arXiv:2309.11495.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, et al. Faith and fate: Limits of transformers on compositionality. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2305.18654.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. The Llama 3 herd of models, 2024. arXiv:2407.21783.

Fan Huang. ReFlect: An effective harness system for complex long-horizon LLM reasoning, 2026. arXiv:2605.05737.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.01798.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter, 2024. arXiv:2407.01502.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. arXiv:2205.11916.

Tomek Korbak, Mikita Balesni, Elizabeth Barnes, et al. Chain of thought monitorability: A new and fragile opportunity for AI safety, 2025. arXiv:2507.11473.

Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, Kamilė Lukošiūtė, Karina Nguyen, Newton Cheng, Nicholas Joseph, Nicholas Schiefer, Oliver Rausch, Robin Larson, Sam McCandlish, Sandipan Kundu, Saurav Kadavath, Shannon Yang, Thomas Henighan, Timothy Maxwell, Timothy Telleen-Lawton, Tristan Hume, Zac Hatfield-Dodds, Jared Kaplan, Jan Brauner, Samuel R. Bowman, and Ethan Perez. Measuring faithfulness in chain-of-thought reasoning, 2023. arXiv:2307.13702.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2305.20050.

Zhan Ling, Yunhao Fang, Xuanlin Li, Zhiao Huang, Mingu Lee, Roland Memisevic, and Hao Su. Deductive verification of chain-of-thought reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2306.03872.

Quanyu Long, Kai Jie Jiang, Jianda Chen, Xu Guo, Leilei Gan, and Wenya Wang. Self-verification dilemma: Experience-driven suppression of overused checking in LLM reasoning, 2026. arXiv:2602.03485.

Aman Madaan, Niket Tandon, Prakhar Gupta, et al. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2303.17651.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. LLM critics help catch LLM bugs, 2024. arXiv:2407.00215.

Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. GSM-Symbolic: Understanding the limitations of mathematical reasoning in large language models. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.05229.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2021. arXiv:2112.00114.

OLMo Team. 2 OLMo 2 furious, 2024. arXiv:2501.00656.

Sailesh Panda, Pritam Kadasi, Abhishek Upperwal, and Mayank Singh. When LLMs stop following steps: A diagnostic study of procedural execution in language models, 2026. arXiv:2605.00817.

Xiangyu Qi, Ashwinee Panda, Kaifeng Lyu, Xiao Ma, Subhrajit Roy, Ahmad Beirami, Prateek Mittal, and Peter Henderson. Safety alignment should be made more than just a few tokens deep. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2406.05946.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. Self-critiquing models for assisting human evaluators, 2022. arXiv:2206.05802.

Rylan Schaeffer, Kateryna Pistunova, Samar Khanna, Sarthak Consul, and Sanmi Koyejo. Invalid logic, equivalent gains: The bizarreness of reasoning in language model prompting, 2023. arXiv:2307.10573. ICML 2023 Workshop: Knowledge and Logical Reasoning in the Era of Data-driven Learning.

Mrinank Sharma, Meg Tong, Tomasz Korbak, et al. Towards understanding sycophancy in language models. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.13548.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning (ICML)*, 2023. arXiv:2302.00093.

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

Oleg Somov, Mikhail Chaichuk, Gleb Ershov, Karim Vafin, Mikhail Seleznyov, Alexander Panchenko, and Elena Tutubalina. Breaking the chain: A causal analysis of LLM faithfulness to intermediate structures, 2026. arXiv:2603.16475.

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

Mingzhong Sun, Teresa Yeo, Armando Solar-Lezama, and Tan Zhi-Xuan. An enigma of artificial reason: Investigating the production-evaluation gap in large reasoning models, 2026. arXiv:2606.01462.

Ken Tsui. Self-correction bench: Uncovering and addressing the self-correction blind spot in large language models. COLM 2026, 2025. arXiv:2507.02778.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2305.04388.

Gladys Tyen, Hassan Mansoor, Victor Cărbune, Peter Chen, and Tony Mak. LLMs cannot find reasoning errors, but can correct them given the error location. In *Findings of the Association for Computational Linguistics: ACL*, 2024. arXiv:2311.08516.

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

Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. Can large language models really improve by self-critiquing their own plans?, 2023. arXiv:2310.08118.

Alexander von Recum, Leander Girrbach, and Zeynep Akata. Are reasoning LLMs robust to interventions on their chain-of-thought? In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2602.07470.

Boshi Wang, Sewon Min, Xiang Deng, Jiaming Shen, You Wu, Luke Zettlemoyer, and Huan Sun. Towards understanding chain-of-thought prompting: An empirical study of what matters. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023. arXiv:2212.10001. Cite key predates verified pub year (ACL 2023 camera-ready); arXiv v1 was Dec 2022.

Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. arXiv:2312.08935.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. arXiv:2201.11903.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.03629.

Evelyn Yee, Alice Li, Chenyu Tang, Yeon Ho Jung, Ramamohan Paturi, and Leon Bergen. Faithful and unfaithful error recovery in chain of thought. In *First Conference on Language Modeling*, 2024. URL `https://openreview.net/forum?id=IPZ28ZqD4I`.

Muru Zhang, Ofir Press, William Merrill, Alisa Liu, and Noah A. Smith. How language model hallucinations can snowball. In *International Conference on Machine Learning (ICML)*, 2024. arXiv:2305.13534.

A EXTERNAL VALIDITY CHECKS

Five checks probe the wall outside the main protocol: whether error salience rescues checking, whether the dissociation transfers to natural-language word problems, how models treat computed errors they produced themselves, whether telling the model who wrote the trace changes absorption, and a renumbered replication of the nearest prior protocol. Every number below is recorded in the released results registry and computed from the run artifacts.

A.1 SALIENCE DOES NOT RESCUE CHECKING

If absorption reflected a failure to notice small discrepancies, larger discrepancies should be caught more often. We re-ran the computed-error condition on two open-weight models with the planted offset in three bands (last-digit-preserving ± 1; ± 10–20; large offsets of 100–400) crossed with whether the planted line shows its expression or only the value, 1,350 rows per condition. No correction is ever flagged (flag rate 0.000 in all twelve conditions) and silent correction stays at or below 1.4% in every condition. For Llama-3.1-8B-Instruct absorption spans 0.95–0.99 across the six conditions. For Qwen2.5-7B-Instruct the large-offset conditions show lower raw absorption (0.57–0.72), but the shortfall is unresolved continuations rather than corrections (silent correction 0.3–1.4% there), and an audit of values landing within a few units of a wrong label caps possible mislabeling at 0.07. A larger error is not a caught error; at worst it is a derailed continuation.

A.2 TRANSFER TO NATURAL-LANGUAGE WORD PROBLEMS

We rebuilt the protocol on two-quantity word problems: two named parties hold o_1 and o_2 items, their total $V = o_1 + o_2$ feeds two downstream quantities, and the output is the sum of those. Each model solves 240 problems in prose; a solution enters the sample only if its final answer is correct and it states V (the correctness gate). We then plant a false value in the model's own solution and let it continue, in four conditions: *readable* corrupts a restated given before V is computed; *computed-worked* replaces V where the solution shows its equation, so the false arithmetic is visible; *computed-bare* states the false total without the equation; and a *true control* plants the correct value. In more than 2,700 correct solutions, no model ever stated the computed total without also showing the arithmetic in the same sentence, so the bare condition replaces the model's arithmetic sentence with an authored equation-free restatement (the worked condition's rewrite sentence with the equation removed); the worked and bare prefixes are otherwise identical, forming a minimal pair, and rewritten rows are flagged in the manifests. Grading follows the main protocol, with two independently implemented parsers; rows where they disagree or where the answer is hedged are excluded (at most two per model).

model	readable absorbed	computed-bare absorbed	computed-worked absorbed	control pass
Llama-3.1-8B	0.60	0.89	0.97	0.97
OLMo-2-7B	0.96	1.00	1.00	1.00
Qwen2.5-1.5B	0.67	0.55	0.82	0.88
Qwen2.5-7B	0.29	0.84	0.99	1.00
Qwen2.5-32B	0.04	0.70	1.00	1.00
Claude Haiku 4.5	0.00	0.90	1.00	1.00
Claude Sonnet 4.5	0.01	0.33	0.69	1.00
Claude Opus 4.8	—	0.45	0.81	1.00
Claude Sonnet 5	0.00	0.66	0.89	1.00
GPT-4o	0.10	0.85	0.92	1.00
GPT-4.1	0.18	0.82	0.90	1.00
GPT-5.1	0.97	0.84	0.85	0.95

Table 1: Word-problem task: absorption of planted errors by condition ($n = 150$ per condition for API models, up to 240 for open-weight). Opus 4.8 never restates a given before computing, so its readable condition is empty because no such solutions exist; Sonnet 5 fills it rarely ($n = 26$).

Three regularities. First, the wall transfers: with the false equation displayed, absorption is 0.69–1.00 for frontier models and 0.82–1.00 for open-weight ones. Second, the displayed equation is itself load-bearing: removing it lowers absorption in the strongest Anthropic models (Sonnet 4.5

falls from 0.69 to 0.33, Opus 4.8 from 0.81 to 0.45, Sonnet 5 from 0.89 to 0.66) while GPT-series models barely move (0.82–0.85 bare against 0.85–0.92 worked). A visibly worked derivation shields a false value from checking that a bare assertion sometimes triggers, consistent with the deference reading in the discussion; the corrections this unlocks are silent, recomputing the total from the problem statement without ever referencing the contradicting value. Third, the readable–computed dissociation replicates in prose for every frontier model except GPT-5.1, which absorbs corrupted restatements of given quantities (0.97) that every other frontier model corrects by re-reading the problem; it treats its own prior restatement as authoritative over the problem statement.

A.3 Naturally occurring computed slips persist

The solving stage also yields natural errors: attempts in which the model mis-computes an intermediate in its own trace with no perturbation applied. Re-executing each failed attempt against ground truth identifies the first self-produced computed slip. Across the five open-weight solving runs, 1,775 traces contain one, and in 1,396 the slip changes the final answer if carried forward. In 10 of the 1,396 (0.7%) the continuation nevertheless reaches the correct output, and an explicit correction remark appears in 20 traces. The usual fate of a model's own computed slip is to be carried silently into the answer, matching the planted-error rates and confirming that the wall is not an artifact of planting.

A.4 Attribution does not move the wall

Planting raises the objection that the model did not produce the false step itself, so absorption might reflect trust in its own past work rather than a general failure to check. We test this by varying who the trace is said to come from while holding its bytes fixed. Each perturbed trace from the fourteen-model comparison's readable and one-operation conditions, and from the adjacent-note control, is presented under four framings: the standard continuation with no attribution, and prefaces stating that the model itself produced the trace, that a different language model produced it, or that a human wrote it by hand. The test runs on three models measured under the user-turn method, where a preface about the trace's origin is natural to add (Claude Opus 4.8, Claude Sonnet 5, GPT-4.1; 60 programs per condition, 59 in Sonnet 5's one-operation condition).

Attribution changes nothing at the wall. The one-operation computed error is absorbed at 0.98–1.00 under every framing on every model, and the adjacent-note control is caught under every framing (absorbed ≤ 0.02). The framing does register where it can: the readable error moves with attribution for Opus 4.8 (0.27 unattributed, 0.13 self, 0.15 other model, 0.32 human), so where provenance can matter, the protocol registers it; GPT-4.1 absorbs the readable error at 0.88–0.97 under all framings, an upper bound under this method. On committed computed values, attribution is irrelevant: a model told outright that the trace is not its own work absorbs the computed falsehood at the same ceiling.

A.5 Renumbered replication of the Yee et al. protocol

We re-ran the protocol of Yee et al. (2024) on its GSM8K random-perturbation condition with all quantities renumbered in the style of GSM-Symbolic (Mirzadeh et al., 2025), preserving solvability, on 106 paired items. GPT-4o recovers 66.0% of original items and 51.9% of renumbered ones (McNemar $p = 0.004$). Of its 70 original-number recoveries, 50 survive renumbering, 19 flip to propagation, and 1 to another wrong answer; in the other direction, 2 propagated and 3 otherwise-wrong items become recoveries, giving the 55 renumbered recoveries. Roughly a quarter (19 of 70) of the original recoveries flip to propagation. GPT-4.1 is unaffected (94.3% against 93.4%) but restarts the solution from scratch on all 106 items, original and renumbered, rather than continuing the corrupted chain, so it re-solves rather than repairs and cannot exhibit the memorization signature. The test covers one model with an intact repair paradigm, one dataset, and 106 pairs; we report it as evidence that stimulus memorization inflates recovery in this protocol, not as a full decomposition of it.

Statement-boundary test. To test whether resuming mid-statement is what lets their models recover, we took the 45 original-number items from the renumbering replication whose original chain continues past the corrupted numeral without another number before the end of the sentence, and their 45 renumbered twins. For each we built two prefixes identical up to the corrupted numeral: one

ending on the numeral, as in their protocol, and one ending after the sentence's own closing words taken verbatim from the model's original chain (median one word). Same model, prompt, and decoding as the replication; uncorrupted prefixes at both boundaries as controls; 360 calls. GPT-4o recovers 55.6 percent of original items at the numeral boundary and 46.7 percent at the statement boundary; on renumbered items, 40.0 and 31.1 percent. The paired difference is 8.9 points in both numberings (pooled over 90 items: 31 recovered at both boundaries, 12 only at the numeral, 4 only after the statement, 43 at neither; exact McNemar $p = 0.08$). Propagation rises by the same amount. Uncorrupted prefixes are unaffected (93.3 and 97.8 percent correct). Completing the statement therefore moves recovery toward propagation, in the registered direction, but by far less than the gap between their recovery rates and ours; the boundary is part of the explanation, not the whole of it. Reading the continuations shows the in-flight mechanism directly: at the numeral boundary the model sometimes finishes the number itself, writing the correct value's remaining digits after the planted one, which no statement-boundary continuation can do. The test was registered before the run (specification and amendment in the experiment archive).

B BOOKKEEPING AND PROVENANCE

This appendix records the sampling, calibration, and protocol details behind the numbers in the main text. It is a provenance register, not a second results section, and each entry states what was run and where the value comes from.

Prompts. Every run uses the same user message: the instruction below, two worked exemplar programs with their traces, then the target program and the word "Trace:". The instruction reads, verbatim: "You will be given a short Python program with numbered lines. Trace its execution line by line. For each line, write exactly one trace line in the format `line N: <code> ; <var> = <value>`, where `<value>` is the value the variable has after that line runs. For the final print line, write `line N: print(<var>); output = <value>`. Write one trace line per program line, in order, and nothing else." The first exemplar is a seven-line program (`a = 7; b = 12; x = a + b; t = x; q = t - 4; out = q + a; print(out)`) with its trace ending `output = 22`; the second is a ten-line program with a six-operand sum. Under prefilling, the perturbed trace is placed in the assistant turn and the model continues it; under the user-turn method, the same message continues after the program with, verbatim: "A partial trace of this program has already been written; it is shown below under 'Trace:'. Continue it. Write only the remaining trace lines, in order, in the exact same format, picking up immediately after the last line shown. Do not repeat lines that are already written." followed by the perturbed trace. The instruction arms prepend one of the following to the user message, with the prefill unchanged: generic, "Work carefully and verify your work as you go."; targeted, "Before using any already-computed value in a later step, re-verify it by redoing its computation. If a value is wrong, correct it and continue from the corrected value."; demonstrated, the targeted text followed by "Worked example (illustration only; not part of the task below and unrelated to it): an earlier step recorded total_apples = 6 + 5 = 12. Before reusing total_apples, re-add: 6 + 5 = 11, not 12, so the recorded value is wrong. Correct it to total_apples = 11, then continue every later step from 11 (e.g. if the next step is baskets = total_apples - 3, compute 11 - 3 = 8, not 9)." The direct-probe condition replaces the continuation request with the question "What is $a + b$?" for the planted line's operands. The adjacent-note control plants a line `note: <var> = <value>` after the model's own line.

Sampling depth. The instruction-experiment specification (E12) called for $R = 8$ sampled continuations per program plus one greedy continuation, and the Haiku 4.5 runs are reported at that registered depth. They were rerun at $R = 8$ plus greedy, so each depth condition contributes $n = 540$ continuations over 60 programs, nine per program, and the instruction experiment was likewise rerun at $R = 8$. The initial $R = 3$ runs are retained for comparison; they reproduce every condition within its confidence interval except depth-one partial opacity (0.996 at $R = 3$ against 0.976 $[0.957, 0.993]$ at $R = 8$), which is why we report the $R = 8$ run. The open-model depth runs use $R = 8$ plus greedy and contribute 1350 continuations over 150 programs at depths zero through three. The depth-five conditions are limited by how few programs are solved: 504 continuations over 56 programs for Llama-3.1-8B and 198 over 22 for Qwen2.5-7B, because few base traces are solved at depth five. Error bands on the depth and opacity figures are program-cluster bootstrap 95 percent intervals (1000 resamples; the cluster is the program), since continuations of the same pro-

gram are correlated; where the bootstrap degenerates for a condition at the ceiling we report Wilson intervals. The wall figure uses Wilson intervals, since each program contributes one continuation per condition. Wilson intervals, where used, are over continuations.

experiment	condition	programs	continuations
fourteen-model comparison (Figs 1, 2)	readable, computed, adjacent-note control; per model	30–190	one per program
depth and opacity, Haiku 4.5	each depth × opacity	60	540
depth and opacity, Llama-3.1-8B	depths 0–3, each opacity	150	1350
depth and opacity, Llama-3.1-8B	depth 5	56	504
depth and opacity, Qwen2.5-7B	depths 0–3, each opacity	150	1350
depth and opacity, Qwen2.5-7B	depth 5	22	198
instruction arms, Haiku 4.5	each arm × depth	60	540
instruction arms, open models	each arm, depths 0–1	100	700
matched controls (E13)	per model	225–279	225–1350
attribution (E13c)	each framing × condition	59–60	one per program
reasoning arm, GPT-5.1	readable / computed	70 / 60	280 / 240
reasoning arm, R1-Distill	readable / computed	70 / 60	280 / 240
roster refresh, GPT-5.5	each depth × opacity; each instruction × depth	60	one per program

Table 2: Programs and continuations behind each reported condition; per-model values for the fourteen-model comparison are in the registry's `fig2` block. Continuations pool one greedy and R sampled rollouts per program where $R > 0$ (a designed mixture, reported as such).

Matched controls (E13). Three controls, run on the same programs with the same correctness gates and grader as the depth experiments, separate the format of the planted line from the cost of checking it. The first is the copy-line readable error on the depth programs, which supplies the depth-zero points of Figure 4: absorption is 0.98 for Llama-3.1-8B ($n = 279$), 0.79 for Qwen2.5-7B ($n = 225$), and 0.14 for Haiku 4.5 ($n = 240$), against 0.69, 0.67, and 0.03 under the adjacent-note control in the same runs. In the second, the depth-one full-opacity line also prints its operands' true sum: absorption falls to 0.47 for Llama, 0.38 for Qwen, and 0.57 for Haiku. In the third, a grid crossing truth, order, and format on readable errors, Haiku absorbs the false line at 0.45 when the true value precedes it as a note, against 0.008 in the adjacent-contradiction corner. Conditions pool greedy and sampled continuations as in the main experiments; per-condition records are under e13/results in the experiment archive and in the paper registry. Two further E13 conditions probe the same boundary. A tentative-marker condition appends "(unverified)" to the planted line's value: absorption is 0.92 for Haiku 4.5 ($n = 240$), 0.97 for Llama-3.1-8B ($n = 396$), and 0.90 for Qwen2.5-7B ($n = 324$). A direct-probe condition replaces the continuation request with the question "What is $a + b$?" for the planted line's operands: the correct sum appears in 100, 100, and 96 percent of replies respectively, and the planted value is echoed in at most 10 percent; replies typically open by restating the operands, so we report the answer-anywhere rate rather than a first-token match.

Adjacent-note control. In an earlier form of the protocol the readable error was planted not by overwriting a copy line but as a separate line, `note: <var> = <value>`, immediately after the model's own line for that variable, so the contradiction is between two adjacent lines about the same variable: the easiest check the trace can pose. Under it, absorption is 0.00 to 0.01 for the four frontier Anthropic models, GPT-4o, GPT-4.1, and GPT-5.5, 0.09 for GPT-5.1, 0.06 for Qwen2.5-32B, 0.15 for GPT-3.5-turbo-instruct, and 0.54 to 0.73 for the small open-weight models (n = 61 to 190). Same-run comparisons on the depth programs give 0.03 against 0.14 for Haiku 4.5, 0.69 against 0.98 for Llama-3.1-8B, and 0.67 against 0.79 for Qwen2.5-7B (note against copy line). We report the copy-line condition as the readable error throughout because it has the same form as the computed error; the note shows that the errors are catchable when the check is trivial, and that weaker models' catching depends on how trivial it is.

Probes and steering (E16). The probes read the residual stream at the first position after the planted line; at the planted value itself the linear probe reaches held-out AUC 1.00 at layer 27. The 150 steering conditions comprise 73 single-layer, 53 multi-layer, and 24 generation-position conditions, each at 40 greedy continuations plus 20 unperturbed controls. The only conditions that produced more checking language than baseline had also destroyed ordinary continuation (control

validity 0.0), and the rare "checked" labels elsewhere proved on reading to be classifier noise. A fuller account of this representation is future work.

Reasoning-model protocol. The reasoning-channel results come from two runs kept separate from the fourteen-model roster. The GPT-5.1 run uses the OpenAI Responses API at reasoning effort medium with summarized reasoning; because the interface returns only a summary of the reasoning rather than the raw trace, the channel statistics we report from it are lower bounds on how often the model re-derives a value. Its reasoning-off baseline, run at effort none, reproduced its user-turn row from the fourteen-model comparison, so the on-versus-off contrast is a pure reasoning effect. The open-weight run uses DeepSeek-R1-Distill-7B, whose correct base traces are generated through its own think channel rather than by raw completion, because raw completion rarely solves the five-operation programs, which the model solves only with that channel engaged. Both are evaluated on the readable, one-operation, and five-operation conditions at the sample sizes recorded in the results registry. The 85 percent figure for GPT-5.1 is among continuations with a non-empty reasoning summary; it is 92 percent if empty summaries are counted as containing no re-derivation. When the true value surfaces in a five-operation deliberation, the final answer still builds on the planted value in 2 of 9 such continuations for R1-Distill and 7 of 20 for GPT-5.1; the counts are small but the direction is the same for both models.

Roster refresh (GPT-5.5). GPT-5.5 (April 2026) was added after the main runs, through the user-turn method with reasoning effort none, on the same programs, solvability gate, and grader as the other user-turn rows (absorption: readable 0.83, one-operation 1.00, five-operation 1.00, adjacent-note control 0.00; 70, 60, 60, 70 programs). Its depth, opacity, and instruction experiments use an adapter that replaces only the assistant-turn prefill of the Haiku 4.5 runs with the user-turn continuation request; injector, grader, solvability gate, and programs are unchanged, at 60 programs per condition. Two departures from the Haiku 4.5 runs are disclosed: the OpenAI API accepts no sampling temperature with reasoning off, so each program contributes one sample rather than one greedy plus eight sampled continuations, and the decoding is not deterministic (re-querying fourteen partial-opacity prompts flipped seven labels), so condition rates estimate per-program probabilities and intervals are Wilson over programs. The runner for the depth and opacity conditions archived graded labels but not continuation text; the instruction conditions and the fourteen re-queried continuations are archived in full, and an independent re-grade of every archived continuation with a separately written parser and executor agrees with the grader on all 980. The specification, two pre-run amendments, and a protocol confound check on GPT-5.1 are in the experiment archive.

Anthropic models released after the main runs. Claude Opus 5 refuses the task: every request under the shipped prompt (130 of 130 in July, 3 of 3 in September) and under a plain rewording (3 of 3) returns a refusal before any text. Claude Fable 5.1, which cannot run without thinking, completes the base traces (189 of 190 at the lowest effort) but refuses a share of the perturbed continuations that tracks how visible the planted inconsistency is: 46 of 70 readable, 34 of 70 adjacent-note control, 11 of 59 one-operation, 9 of 60 five-operation, and the refusals are stable on re-send (12 of 12). Among the continuations it does write, absorption is 0.00, 0.00, 0.98, and 0.39 (n = 24, 36, 48, 51), with the five-operation survivors including visible recomputation. Because the refusals select on the perturbed prompts, these rates are not comparable to the roster and are not pooled into Figure 3.

Solvability gate. Two open-weight models failed the solvability gate on the hardest conditions. Qwen2.5-1.5B solved the base program at a rate of 0.105 and OLMo-2-7B at 0.369, so after we keep only correctly solved base traces their deepest computed conditions rest on small samples. We report those conditions with widened Wilson intervals and do not treat them as settled points.

User-turn versus prefill calibration. The user-turn method and the prefill method were cross-calibrated on the models that support both. The computed-error conditions coincide across the two methods, so the computed-side result does not depend on the method. The readable (copy-line) condition inflates under the user-turn method: on the same 70 programs Haiku 4.5 absorbs it at 0.04 under prefilling and 0.36 under the user-turn method, and Sonnet 4.5 at 0.03 and 0.09, while both models' computed conditions are 1.00 under either method. Readable-side rates of user-turn models are therefore reported as upper bounds. Reporting the computed series as the one-operation error alone is a deviation from the E10 registration, which pooled the copy and computed conditions

into one recompute-side rate; we separated them when calibration showed the copy condition is method-sensitive.

Fable 5 refusal. Fable 5 returned no usable data. Its safety classifier refused the task format on all 360 of 360 attempted continuations, categorized as cyber, under both the thinking and the non-thinking conditions, at a total spend of $2.93. We report Fable 5 as unmeasured rather than as a zero or as a result.

Protocol constants. The protocol constants are fixed across conditions within each task format. In the depth and instruction experiments (E11 and E12), each program renders to a trace of 20 lines carrying 14 arithmetic operations in total, with the planted line at line 10 (fractional position 0.5) and its first downstream read two lines later. The programs behind Figures 1 and 2 have 12 lines, with the planted line at line 6, likewise fixed across its own conditions. Operands of the planted line are two-digit integers drawn from 11 to 99, filler constants are drawn from 2 to 60, and intermediate values are capped at 999. Verification depth takes values in $\{0, 1, 2, 3, 5\}$ and the planted line's opacity in $\{\text{bare}, \text{full}, \text{partial}\}$. Programs are generated deterministically from an integer seed; the base seed is 0 and the worked example uses seed 507786. Sampling uses one greedy continuation at temperature 0 followed by R sampled continuations at temperature 0.7. These values are taken from the E11 generator header (`gen_depth.py`) and the E10, E11, and E12 specifications. API models are called by these identifiers: claude-haiku-4-5, claude-sonnet-4-5, claude-sonnet-5, claude-opus-4-8, gpt-4o, gpt-4.1, gpt-5.1, gpt-3.5-turbo-instruct; open models are Qwen2.5-1.5B/7B/32B-Instruct, OLMo-2-1124-7B-Instruct, Meta-Llama-3.1-8B-Instruct, and DeepSeek-R1-Distill-Qwen-7B, at the Hugging Face revisions recorded in the run manifests.

C FINE-TUNING DETAILS

This appendix describes the fine-tuning runs of Section 5.7. All runs use the same training script and the same hyperparameters. The hyperparameters were written down and committed before the first run, and they were not tuned afterwards.

Training data. We build two training sets with the same program generator and settings as the evaluation programs, but from different random seeds, so no training program appears in any evaluation set. The *depth-one* set has 300 programs at depth one. The *mixed-depth* set has 100 programs at each of depths one, two, and three. Every program has 20 lines, and the planted line is line 10. The planted value differs from the true value by ± 10 or ± 20. Each program gives two training examples, so each set has 600:

- a *perturbed* example: the prefix ends with the planted line, and the target is one `re-check` line, the corrected line 10, and the rest of the true trace;
- a *control* example: the prefix ends with the true line 10, and the target is the rest of the true trace, with no check.

Half of the examples therefore contain no check, so the model is trained to check only when the value is wrong. We verified all 600 examples in each set in three ways: with assertions in the data builder, with the grader used for evaluation, and with an independent parser that executes each program. Figure 7 shows one perturbed example.

Input format and loss. Training uses exactly the input the model sees at evaluation: the model's chat template, the user message of Appendix B (instruction, two exemplars, and the program), and an assistant turn made of the trace prefix followed by the target. The loss is computed only on the target tokens and the end-of-turn token. The prompt and the prefix are masked. With the Qwen tokenizer, examples average 1,279 tokens, of which 215 are target tokens. The longest example has 1,329 tokens, below the 1,536-token limit, so no example is truncated.

Models. Table 3 lists the three models. Each is fine-tuned with LoRA adapters, so less than 1% of the parameters are trained.

```
PREFIX (last two lines; the true value of y is 121)
line 9: e = 38; e = 38
line 10: y = t + j + e; y = 111

TARGET
re-check line 10: y = t + j + e = 50 + 33 + 38 = 121, not 111
line 10: y = t + j + e; y = 121
line 11: u = 49; u = 49
line 12: z = y + 2; z = 123
...
line 20: print(out); output = 252
```

Figure 7: A perturbed training example from the mixed-depth set, at depth two. The check re-adds the three operands, which were set on lines 7–9. The control example for the same program ends its prefix with `y = 121`, and its target starts at line 11.

Hugging Face repository	Revision	Trainable parameters
`Qwen/Qwen2.5-32B-Instruct`	`5ede1c97`†	134.2M (0.41%)
`NousResearch/Meta-Llama-3.1-8B-Instruct`	`d10aef79`	41.9M (0.52%)
`Qwen/Qwen2.5-7B-Instruct`	`a09a3545`	40.4M (0.53%)

Table 3: Models used for fine-tuning. Revisions are the first eight characters of the Hugging Face commit. †Not pinned in the code; this was the only snapshot in the local cache, downloaded before these runs.

Optimization. Table 4 gives the settings, which are identical for every run. Training uses a plain PyTorch loop (PyTorch 2.11.0, Transformers 5.12.1, PEFT 0.21.0) on one NVIDIA H200 GPU per run.

Setting	Value
LoRA rank / scaling α / dropout	16 / 32 / 0.05
LoRA target modules	query, key, value, output, gate, up, down projections
Optimizer	AdamW, $\beta = (0.9, 0.999)$, weight decay 0
Learning rate	10^{-4} peak; linear warm-up over 5 updates, then cosine decay to 0
Batch	2 examples $\times$ 8 accumulation steps = 16 examples per update
Epochs / updates	3 / 114
Gradient clipping	norm 1.0
Precision	bfloat16 weights, gradient checkpointing
Maximum sequence length	1,536 tokens

Table 4: Fine-tuning hyperparameters, fixed before the first run and shared by all runs.

Runs and checkpoints. Table 5 lists all six runs. The random seed sets the LoRA initialization and the order of examples in each epoch. The training data are the same across seeds. We evaluate the adapter from the end of training, after the last update. We save no intermediate checkpoints, hold out no validation set, and do no early stopping or checkpoint selection. For evaluation, the adapter is merged into the base weights in bfloat16 and served with vLLM 0.24.0. As a check that training fit the data, we ran the first run (Qwen2.5-32B, depth one, seed 0) on 30 of its own training programs. On the 20 of these that both the base and the fine-tuned model traced correctly, depth-one propagation fell from 0.925 to 0. This check was not repeated for the other runs.

Order of the runs. Each run was written down, with its predictions, before it started. The first run was Qwen2.5-32B on the depth-one set with seed 0. After its result, we added seeds 1 and 2, the mixed-depth run, and the position test below. After the mixed-depth result, we applied the same recipe, unchanged, to Llama-3.1-8B-Instruct and Qwen2.5-7B-Instruct. Qwen2.5-7B replaced a planned GPT-4.1 run after the OpenAI fine-tuning API refused the job. The mixed-depth recipe was run with one seed per model.

Model	Training set	Seed	Updates	Time (min)	Loss, first	Loss, final
Qwen2.5-32B	depth one	0	114	26.7	0.34	0.0006
Qwen2.5-32B	depth one	1	114	26.5	0.24	0.0007
Qwen2.5-32B	depth one	2	114	26.4	0.31	0.0008
Qwen2.5-32B	mixed depth	0	114	26.9	0.33	0.0007
Llama-3.1-8B	mixed depth	0	114	7.8	0.27	0.0024
Qwen2.5-7B	mixed depth	0	114	9.2	0.37	0.0066

Table 5: All fine-tuning runs. Time is the training loop only, without model loading or merging. Loss is the cross-entropy per target token; "first" is the first update and "final" is the mean over the last ten updates.

Evaluation. Fine-tuned models are evaluated with the harness and grader of the depth experiments, on 180 held-out programs per depth (depths 0, 1, 2, 3, and 5). As in the main experiments, each model first writes its own trace of every program, and programs it traces incorrectly are dropped. The error is then planted on the model's own line 10. For each program we take one greedy continuation and three continuations sampled at temperature 0.7, each of at most 384 new tokens. The base model is run on the same programs with the same settings. Fine-tuning raises the share of programs traced correctly (for Qwen2.5-32B, from 0.53–0.62 per depth to 0.56–0.72), so each comparison uses only the programs that both the base and the fine-tuned model traced correctly. Intervals are program-cluster bootstrap 95% intervals with 1,000 resamples.

Unnecessary checks. We give each model the correct trace through line 10 for the 720 programs at depths one, two, three, and five, take one greedy continuation, and count continuations that contain a line starting with `re-check`.

Six-operand checks. A depth-five continuation counts as reconstructing all six operands when it contains a line of the form `re-check line 10: v = a + ... =` n_1 `+ ... +` n_6 `= S, not P`, in which $n_1, \ldots, n_6$ are the six operands of the planted line and S is the true value.

Position test. We built 100 more depth-one programs with the planted line at line 14 instead of line 10. None was used in training. On the programs that both models traced correctly, propagation is 0.018 for the seed-0 depth-one model (4 of 224 continuations, 56 programs) and 0 for the mixed-depth model (0 of 216 continuations, 54 programs). The base model propagates 0.97 on the same programs.

Per-run results. Table 6 gives propagation for every run.

Model	Training	$k=1$	$k=2$	$k=3$	$k=5$	Checks on correct traces
Qwen2.5-32B	none	.937	.982	.997	.977	0 / 720
Qwen2.5-32B	depth one, seed 0	.000	.044	.412	.976	0 / 720
Qwen2.5-32B	depth one, seed 1	.000	.059	.480	.970	not measured
Qwen2.5-32B	depth one, seed 2	.000	.015	.299	.982	not measured
Qwen2.5-32B	mixed depth, seed 0	.000	.000	.005	.275	3 / 720
Llama-3.1-8B	none	.992	.994	1.000	.750[a]	0 / 720
Llama-3.1-8B	mixed depth, seed 0	.000	.000	.000	.250[a]	82 / 720
Qwen2.5-7B	none	.750	.909	1.000	–	0 / 720
Qwen2.5-7B	mixed depth, seed 0	.972	1.000	1.000	–[b]	188 / 720

Table 6: Propagation in the bare form, by verification depth k. Each fine-tuned row uses the programs that both it and its base model traced correctly. The Qwen2.5-32B base row uses the programs traced correctly by all five Qwen2.5-32B models, as in Figure 6; on each fine-tuned model's own programs the base rate differs from this row by less than 0.01. The matched programs number 79–105 per depth for Qwen2.5-32B, 32–40 at depths 1–3 for Llama-3.1-8B, and 9–11 for Qwen2.5-7B. [a]Four programs. On its own correctly traced programs, the fine-tuned Llama-3.1-8B propagates .206 at depth five (37 of 180 continuations, 45 programs). [b]The fine-tuned Qwen2.5-7B traced no depth-five program correctly.