Why We Think

This PDF conversion (v1) was generated on June 21, 2025*

Online version: https://lilianweng.github.io/posts/2025-05-01-thinking/
Date: May 1, 2025 **Estimated Reading Time:** 40 min **Author:** Lilian Weng

Special thanks to John Schulman for a lot of super valuable feedback and direct edits on this post.

Test time compute (Graves et al. 2016, Ling, et al. 2017, Cobbe et al. 2021) and Chain-of-thought (CoT) (Wei et al. 2022, Nye et al. 2021), have led to significant improvements in model performance, while raising many research questions. This post aims to review recent developments in how to effectively use test-time compute (i.e. "thinking time") and why it helps.

[0]This file was prepared by Sakura (bili_sakura@zju.edu.cn). Please contact for any issues or copyright concerns. The original source of this file is https://lilianweng.github.io/posts/2025-05-01-thinking.

Contents

1 Motivation

Enabling models to think for longer can be motivated in a few different ways.

1.1 Analogy to Psychology

The core idea is deeply connected to how humans think. We humans cannot immediately provide the answer for "What's 12345 times 56789?". Rather, it is natural to spend time pondering and analyzing before getting to the result, especially for complex problems. In *Thinking, Fast and Slow* (Kahneman, 2013), Daniel Kahneman characterizes human thinking into two modes, through the lens of the dual process theory:

- **Fast thinking (System 1)** operates quickly and automatically, driven by intuition and emotion while requiring little to no effort.
- **Slow thinking (System 2)** demands deliberate, logical thought and significant cognitive efforts. This mode of thinking consumes more mental energy and requires intentional engagement.

Because System 1 thinking is fast and easy, it often ends up being the main decision driver, at the cost of accuracy and logic. It naturally relies on our brain's mental shortcuts (i.e., heuristics) and can lead to errors and biases. By consciously slowing down and taking more time to reflect, improve and analyze, we can engage in System 2 thinking to challenge our instincts and make more rational choices.

1.2 Computation as a Resource

One view of deep learning is that neural networks can be characterized by the amount of computation and storage they can access in a forward pass, and if we optimize them to solve problems using gradient descent, the optimization process will figure out how to use these resources—they'll figure out how to organize these resources into circuits for calculation and information storage. From this view, if we design an architecture or system that can do more computation at test time, and we train it to effectively use this resource, it'll work better.

In Transformer models, the amount of computation (flops) that the model does for each generated token is roughly 2 times the number of parameters. For sparse models like mixture of experts (MoE), only a fraction of the parameters are used in each forward pass, so computation $= 2 \times$ parameters/sparsity, where sparsity is the fraction of experts active.

On the other hand, CoT enables the model to perform far more flops of computation for each token of the answer that it is trying to compute. In fact, CoT has a nice property that it allows the model to use a variable amount of compute depending on the hardness of the problem.

1.3 Latent Variable Modeling

A classic idea in machine learning is to define a probabilistic model with a latent (hidden) variable z and a visible variable y, where y is given to our learning algorithm. Marginalizing (summing) over the possible values of the latent variable allows us to express a rich

distribution over the visible variables,

$$P(y) = \sum_{z \sim P(z)} P(y \mid z).$$

For example, we can model the distribution over math problems and solutions by letting x denote a problem statement, y be ground truth answer or proof, and z as a free-form thought process that leads to the proof. The marginal probability distribution to optimize would be

$$P(y \mid x) = \sum_{z \sim p(z|x)} P(y \mid x, z)$$

The latent variable perspective is particularly useful for understanding methods that involve collecting multiple parallel CoTs or searching over the CoT—these algorithms can be seen as sampling from the posterior $P(z \mid x, y)$. This view also suggests the benefits of using the log loss $\log P(y \mid x)$ as the target objective to optimize, as the log loss objective has been so effective in pretraining.

2 Thinking in Tokens

The strategy of generating intermediate steps before generating short answers, particularly for math problems, was explored by Ling et al. 2017, who introduced the AQUA-RAT dataset, and then expanded by Cobbe et al. 2021, who introduced the Grade School Math (GSM) dataset. Cobbe et al. trained a generator with supervised learning on human-written solutions and verifiers that predict the correctness of a candidate solution; they can then search over these solutions. Nye et al. (2021) experimented with intermediate thinking tokens as "scratchpads" and Wei et al. (2022) coined the now-standard term **chain-of-thought** (CoT).

Early work on improving CoT reasoning involved doing supervised learning on human-written reasoning traces or model-written traces filtered for answer correctness, where the latter can be seen as a rudimentary form of reinforcement learning (RL). Some other work found that one could significantly boost math performance of instruction-tuned models by prompting them appropriately, with "`think step by step`" (Kojima et al. 2022) or more complex prompting to encourage the model to reflect on related knowledge first (Yasunaga et al. 2023).

Later work found that the CoT reasoning capabilities can be significantly improved by doing reinforcement learning on a dataset of problems with automatically checkable solutions, such as STEM problems with short answers, or coding tasks that can be checked with unit tests (Zelikman et al. 2022, Wang et al., 2023, Liu et al., 2023). This approach rose to prominence with the announcement of o1-preview, o3, and the R1 tech report (DeepSeek-AI, 2025), which showed that a simple recipe where a policy gradient algorithm could lead to strong performance.

3 Branching and Editing

The fundamental intent of test-time compute is to adaptively modify the model's output distribution at test time. There are various ways of utilizing test-time resources for decoding to select better samples and thus alter the model's predictions towards a

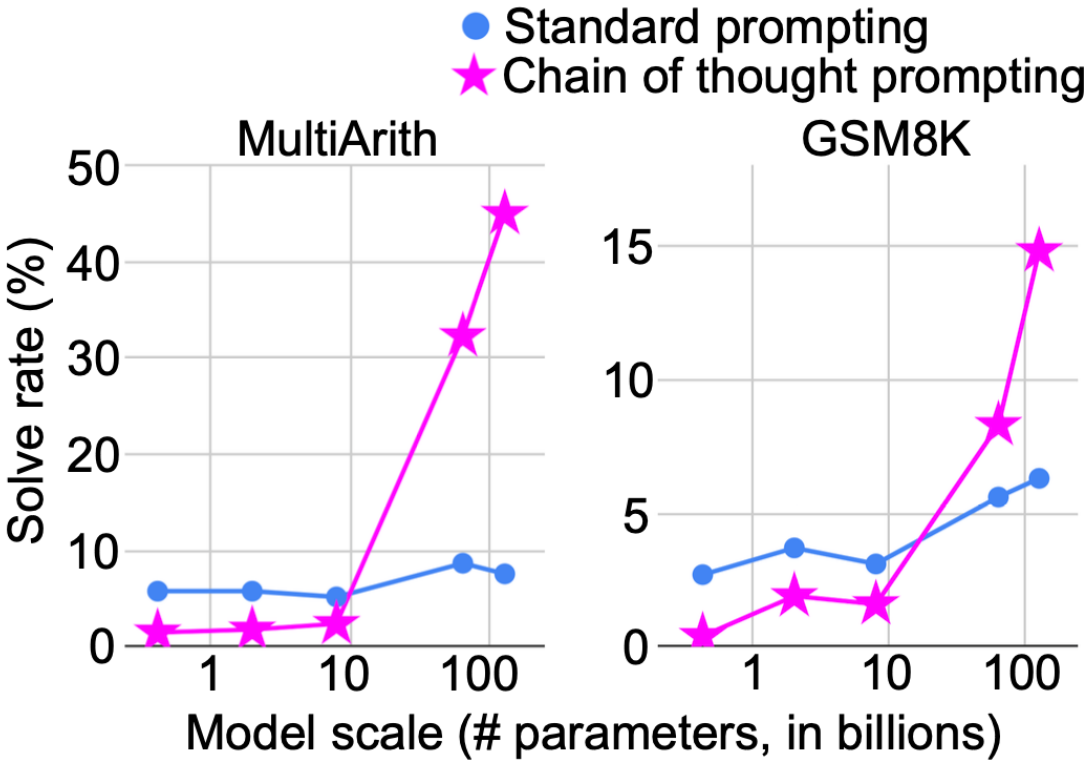


Figure 1: Chain-of-thought prompting leads to higher success rate of solving math problems. Larger models benefit more from thinking time. (Image source: Wei et al. 2022)

more desired distribution. Two main approaches for improving the decoding process are **parallel sampling** and **sequential revision**.

- **Parallel sampling** generates multiple outputs simultaneously, meanwhile providing guidance per step with process reward signals or using verifiers to judge the quality at the end. It is the most widely adopted decoding method to improve test-time performance, such as best-of-N or beam search. Self-consistency (Wang et al. 2023) is commonly used to select the answer with majority vote among multiple CoT rollouts when the ground truth is not available.
- **Sequential revision** adapts the model's responses iteratively based on the output in the previous step, asking the model to intentionally reflect on its existing response and correct mistakes. The revision process may have to rely on a fine-tuned model, as naively relying on the model's intrinsic capability of self-correction without external feedback may not lead to improvement (Kamoi et al. 2024, Huang et al. 2024).

Parallel sampling is simple, intuitive, and easier to implement, but is bounded by the model's capability of whether it can achieve the correct solution in one go. Sequential revision explicitly asks the model to reflect on mistakes, but it is slower and requires extra care during implementation, as it does run the risk of correct predictions being modified to be incorrect or introducing other types of hallucinations. These two methods can be used together. Snell et al. (2024) showed that easier questions benefit from purely sequential test-time compute, whereas harder questions often perform best with an optimal ratio of sequential to parallel compute.

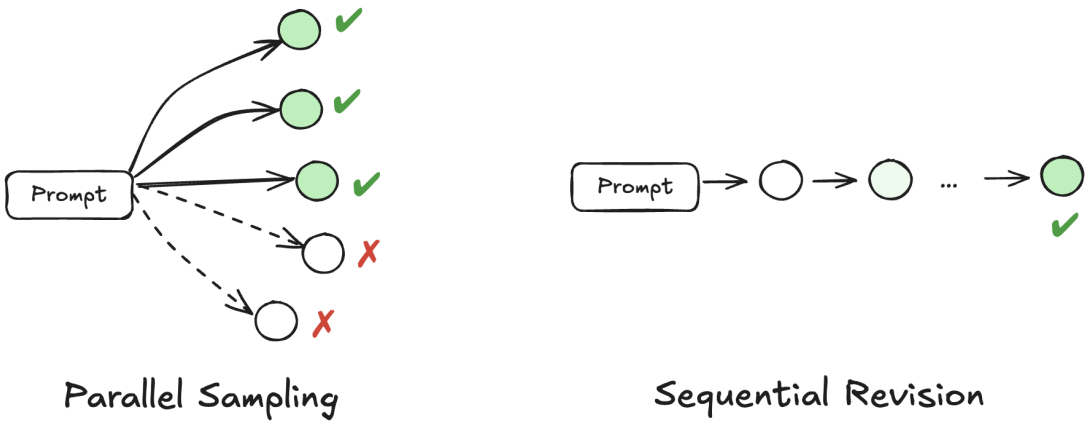


Figure 2: Illustration of parallel sampling vs sequential revision.

3.1 Parallel Sampling

Given a generative model and a scoring function that we can use to score full or partial samples, there are various search algorithms we can use to find a high scoring sample. Best-of-N is the simplest such algorithm: one just collects N independent samples and chooses the highest-ranking sample according to some scoring function. Beam search is a more sophisticated search algorithm that makes the search process more adaptive, spending more sampling computation on more promising parts of the solution space.

Beam search maintains a set of promising partial sequences and alternates between extending them and pruning the less promising ones. As a selection mechanism, we can use a process reward model (PRM; Lightman et al. 2023) to guide beam search candidate selection. Xie et al. (2023) used LLMs to evaluate how likely their own generated reasoning step is correct, formatted as a multiple-choice question, and found that per-step self-evaluation reduces accumulative errors in multi-step reasoning during beam search decoding. Besides, during sampling, annealing the temperature helps mitigate aggregated randomness. These experiments by Xie et al. achieved 5–6% improvement on few-shot GSM8k, AQuA, and StrategyQA benchmarks with the Codex model. Reward balanced search ("REBASE"; Wu et al. 2025) separately trained a process reward model (PRM) to determine how much each node should be expanded at each depth during beam search, according to the softmax-normalized reward scores. Jiang et al. (2024) trained their PRM, named "RATIONALYST", for beam search guidance on synthetic rationales conditioned on a large amount of unlabelled data. Good rationales are filtered based on whether they help reduce the negative log-probability of true answer tokens by a threshold, when comparing the difference between when the rationale is included in the context vs not. At inference time, RATIONALYST provides process supervision to the CoT generator by helping estimate log-prob of next reasoning steps ("implicit") or directly generating next reasoning steps as part of the prompt ("explicit").

Interestingly, it is possible to trigger the emergent chain-of-thought reasoning paths *without* explicit zero-shot or few-shot prompting. Wang & Zhou (2024) discovered that if we branch out at the first sampling tokens by retaining the top k tokens with highest confidence, measured as the difference between top-1 and top-2 candidates during sampling, and then continue these k sampling trials with greedy decoding onward, many of these sequences natively contain CoT. Especially when CoT does appear in the context,

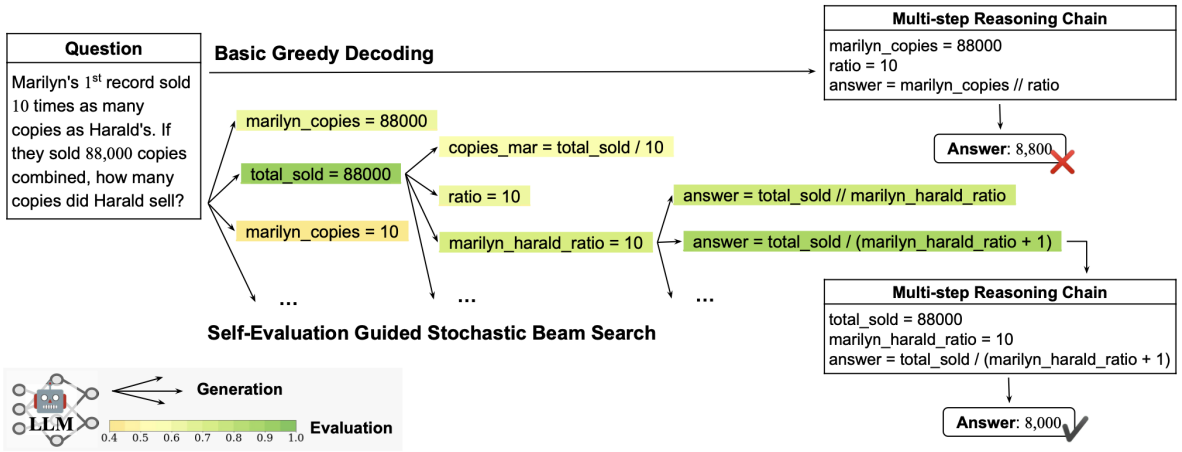


Figure 3: Beam search decoding guided by LLM self-evaluation per reasoning step. (Image source: Xie et al. 2023)

it leads to a more confident decoding of the final answer. To calculate the confidence of the final answer, the answer span needs to be identified by task-specific heuristics (e.g., last numerical values for math questions) or by prompting the model further with "So the answer is …". The design choice of only branching out at the first token is based on the observation that early branching significantly enhances the diversity of potential paths, while later tokens are influenced a lot by previous sequences.

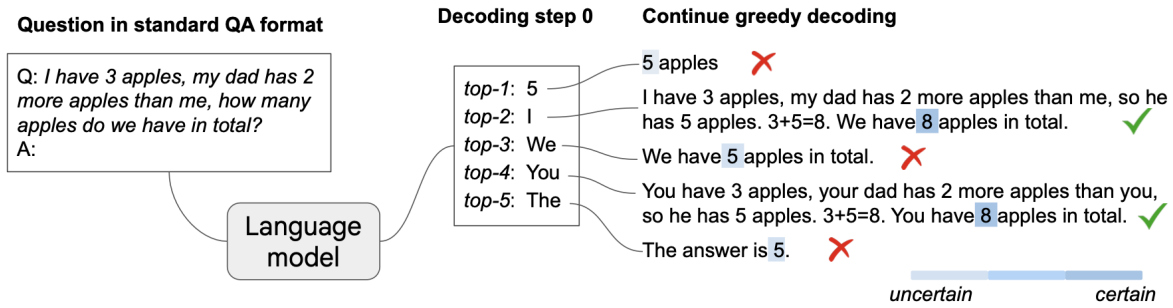


Figure 4: Top-k decoding, k refers to the number of candidates at the first sampling step. (Image source: Wang & Zhou, 2024)

3.2 Sequential Revision

If the model can reflect and correct mistakes in past responses, we would expect the model to produce a nice sequence of iterative revision with increasing quality. However, this self-correction capability turns out to not exist intrinsically among LLMs and does not easily work out of the box, due to various failure modes, such as: (1) hallucination, including modifying correct responses to be incorrect; (2) behavior collapse to non-correcting behavior, e.g., making minor or no modification on the first incorrect responses; or (3) failure to generalize to distribution shift at test time. Experiments by Huang et al. (2024) showed that naively applying self-correction leads to worse performance and external feedback is needed for models to self-improve, which can be based on matching ground truths, heuristics and task-specific metrics, unit test results for coding

questions (Shinn et al. 2023), a stronger model (Zhang et al. 2024), as well as human feedback (Liu et al. 2023).

Self-correction learning (Welleck et al. 2023) aims to train a corrector model $P_\theta(y \mid y_0, x)$ given a fixed generator model $P_0(y_0 \mid x)$. While the generator model remains generic, the corrector model can be task-specific and only does generation conditioned on an initial model response and additional feedback (e.g., a sentence, a compiler trace, unit test results; can be optional):

1. Self-correction learning first generates multiple outputs per prompt in the data pool;
2. Then creates value-improving pairs by pairing two outputs for the same prompt together if one has a higher value than the other, (prompt x, hypothesis y, correction y').
3. These pairs are selected proportional to their improvement in value, $v(y') - v(y)$, and similarity between two outputs, $\text{Similarity}(y, y')$, to train the corrector model.
4. To encourage exploration, the corrector provides new generations into the data pool as well. At inference time, the corrector can be used iteratively to create a correction trajectory of sequential revision.

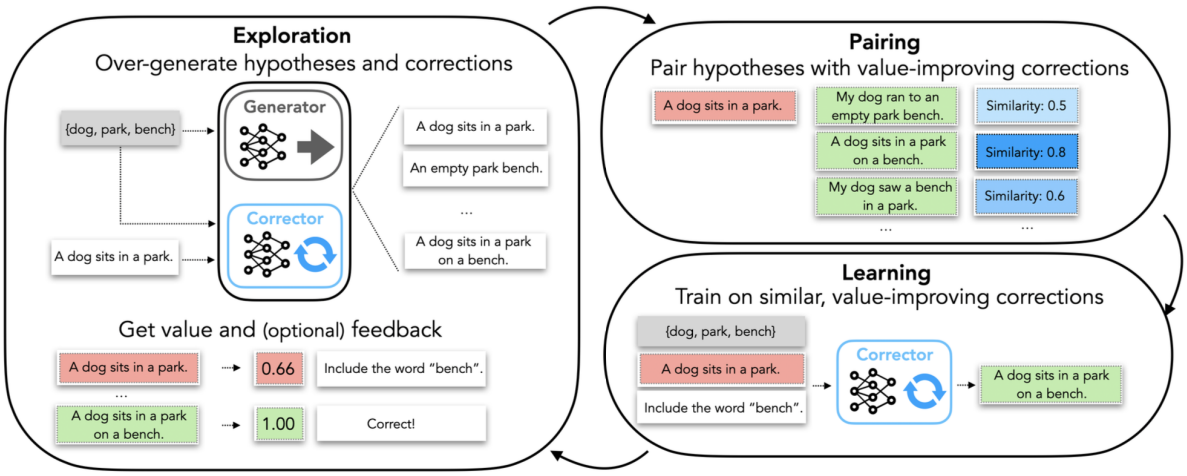


Figure 5: Illustration of self-correction learning by matching model outputs for the same problem to form value-improving pairs to train a correction model. (Image source: Welleck et al. 2023)

Recursive inspection (Qu et al. 2024) also aims to train a better corrector model but with a single model to do both generation and self-correction.

SCoRe (Self-Correction via Reinforcement Learning; Kumar et al. 2024) is a multi-turn RL approach to encourage the model to do self-correction by producing better answers at the second attempt than the one created at the first attempt. It composes two stages of training: stage 1 only maximizes the accuracy of the second attempt while enforcing a KL penalty only on the first attempt to avoid too much shifting of the first-turn responses from the base model behavior; stage 2 optimizes the accuracy of answers produced by both the first and second attempts. Ideally, we do want to see performance at both first and second attempts to be better, but adding stage 1 prevents the behavior

collapse where the model does minor or no edits on the first response, and stage 2 further improves the results.

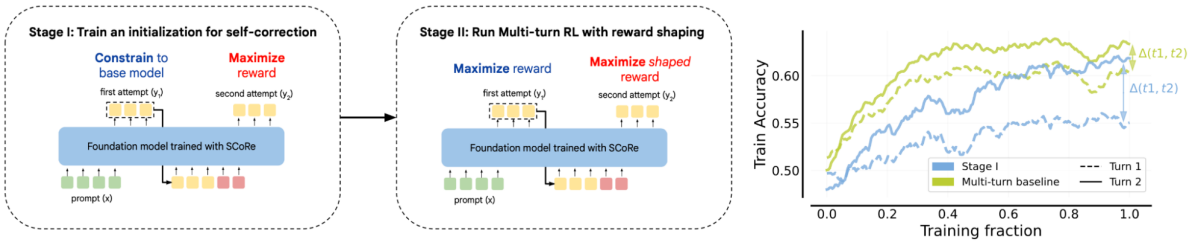


Figure 6: Explicit training setup to improve self-correction capabilities by doing two-staged RL training. (Image source: Kumar et al. 2024)

3.3 RL for Better Reasoning

There has been a lot of recent success in using RL to improve the reasoning ability of language models, by using a collection of questions with ground truth answers (usually STEM problems and puzzles with easy to verify answers), and rewarding the model for getting the correct answer. Recent activity in this area was spurred by strong performance of the o-series models from OpenAI, and the subsequent releases of models and tech reports from DeepSeek.

`DeepSeek-R1` (DeepSeek-AI, 2025) is an open-source LLM designed to excel in tasks that require advanced reasoning skills like math, coding and logical problem solving. They run through 2 rounds of SFT-RL training, enabling R1 to be good at both reasoning and non-reasoning tasks.

1. **Cold-start SFT** is to fine-tune the `DeepSeek-V3-Base` base model on a collection of thousands of cold-start data. Without this step, the model has issues of poor readability and language mixing.

2. **Reasoning-oriented RL** trains a reasoning model on reasoning-only prompts with two types of rule-based rewards:

    - **Format rewards:** The model should wrap CoTs by `<thinking>... </thinking>` tokens.
    - **Accuracy rewards:** Whether the final answers are correct. The answer for math problems needs to be present in a specific format (e.g. in a box) to be verified reliably. For coding problems, a compiler is used to evaluate whether test cases pass.

3. **Rejection-sampling + non-reasoning SFT** utilizes new SFT data created by rejection sampling on the RL checkpoint of step 2, combined with non-reasoning supervised data from `DeepSeek-V3` in domains like writing, factual QA, and self-cognition, to retrain `DeepSeek-V3-Base`.

    - Filter out CoTs with mixed languages, long paragraphs, and code blocks.
    - Include non-reasoning tasks using DeepSeek-V3 (DeepSeek-AI, 2024) pipeline.

- For certain non-reasoning tasks, call DeepSeek-V3 to generate potential CoTs before answering the question by prompting. But for simpler queries like "hello", CoT is not needed.
- Then fine-tune the DeepSeek-V3-Base on the total 800k samples for 2 epochs.

4. The final **RL** stage trains the step 3 checkpoint on both reasoning and non-reasoning prompts, improving helpfulness, harmlessness and reasoning.

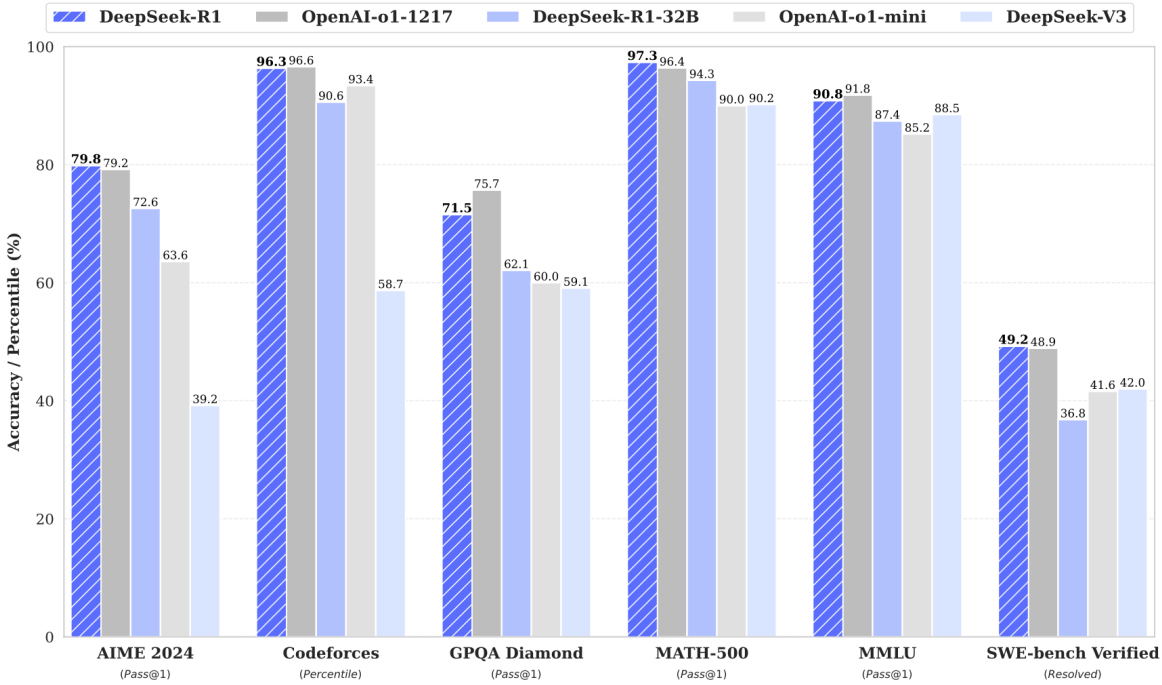
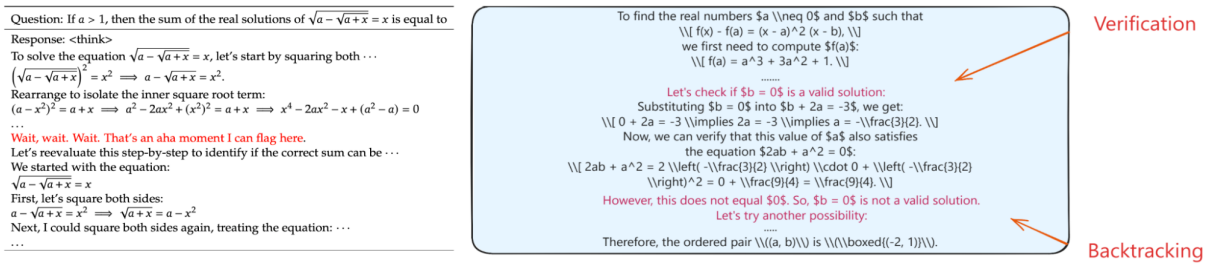


Figure 7: `DeepSeek-R1` performs comparable to OpenAI `o1-preview` and `o1-mini` on several widely used reasoning benchmarks. `DeepSeek-V3` is the only non-reasoning model listed. (Image source: DeepSeek-AI, 2025)

Interestingly, the DeepSeek team showed that with pure RL, no SFT stage, it is still possible to learn advanced reasoning capabilities like reflection and backtracking ("Aha moment"). The model naturally learns to spend more thinking tokens during the RL training process to solve reasoning tasks. The "aha moment" can emerge, referring to the model reflecting on previous mistakes and then trying alternative approaches to correct them. Later, various open source efforts happened for replicating R1 results like Open-R1, SimpleRL-reason, and TinyZero, all based on Qwen models. These efforts also confirmed that pure RL leads to great performance on math problems, as well as the emergent "aha moment".

The DeepSeek team also shared some of their unsuccessful attempts. They failed to use process reward model (PRM) as it is hard to define per-step rubrics or determine whether an intermediate step is correct, meanwhile making the training more vulnerable to reward hacking. The efforts on MCTS (Monte Carlo Tree Search) also failed due to the large search space for language model tokens, in comparison to, say, chess; and training the fine-grained value model used for guiding the search is very challenging too. Failed attempts often provide unique insights and we would like to encourage the research community to share more about what did not work out.



(a) "aha moment" in DeepSeek-R1 (b) "aha moment" in simpleRL-reason

Figure 8: Examples of the model learning to reflect and correct mistakes. (Image source: (left) DeepSeek-AI, 2025; (right) Zeng et al. 2025)

3.4 External Tool Use

During the reasoning steps, certain intermediate steps can be reliably and accurately solved by executing code or running mathematical calculations. Offloading that part of reasoning components into an external code interpreter, as in PAL (Program-Aided Language Model; Gao et al. 2022) or Chain of Code (Li et al. 2023), can extend the capability of LLMs with external tools, eliminating the need for LLMs to learn to execute code or function as calculators themselves. These code emulators, like in Chain of Code, can be augmented by an LLM such that if a standard code interpreter fails, we have the option of using the LLM to execute that line of code instead. Using code to enhance reasoning steps is especially beneficial for mathematical problems, symbolic reasoning, and algorithmic tasks. These unit tests may not exist as part of the coding questions, and in those cases, we can instruct the model to self-generate unit tests for it to test against to verify the solution (Shinn et al. 2023).

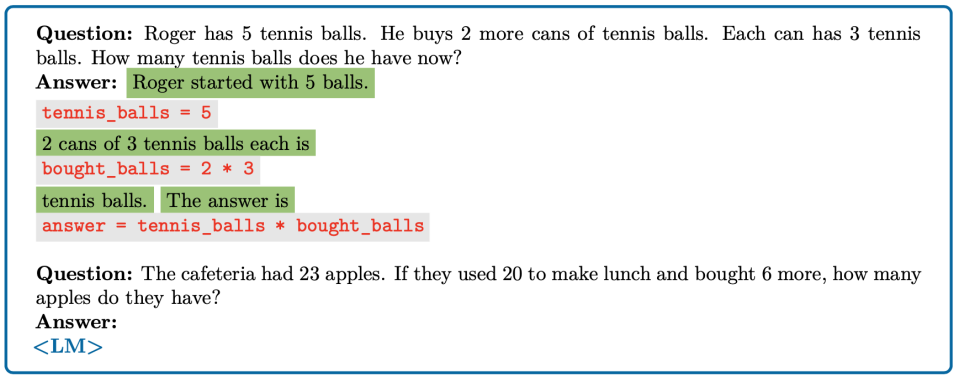


Figure 9: An example of program-aided language model prompting. (Image source: Gao et al. 2022)

ReAct (Reason+Act; Yao et al. 2023) combines the action of searching the Wikipedia API and generation of reasoning traces, such that reasoning paths can incorporate external knowledge.

o3 & o4-mini, recently released by OpenAI, are another two good examples where the reasoning process involves tool use like Web search, code execution, and image processing. The team observed that large-scale reinforcement learning exhibits the same trend as in the GPT paradigm that "more compute = better performance".

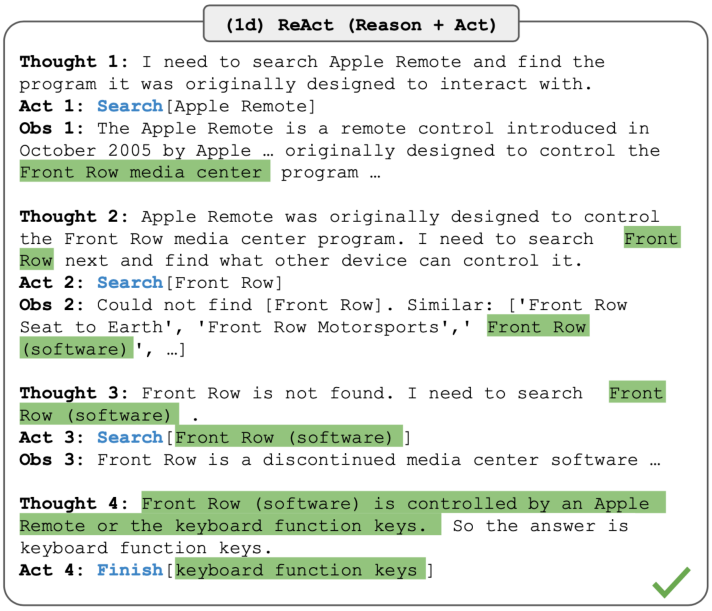


Figure 10: An example of the ReAct prompting method to solve a HotpotQA question, using Wikipedia search API as an external tool to help with reasoning. (Image source: Yao et al. 2023)

3.5 Thinking Faithfully

Deep learning models are often treated as black boxes and various interpretability methods have been proposed. Interpretability is useful for a couple of reasons: first, it gives us an extra test to determine if the model is misaligned with its creators' intent, or if it's misbehaving in some way that we can't tell by monitoring its actions. Second, it can help us determine whether the model is using a sound process to compute its answers. Chain of thought provides an especially convenient form of interpretability, as it makes the model's internal process visible in natural language. This interpretability, however, rests on the assumption that the model truthfully describes its internal thought processes.

Recent work showed that monitoring CoT of reasoning models can effectively detect model misbehavior such as reward hacking, and can even enable a weaker model to monitor a stronger model (Baker et al. 2025). Increasing test time compute can also lead to improved adversarial robustness (Zaremba et al. 2025); this makes sense intuitively, because thinking for longer should be especially useful when the model is presented with an unusual input, such as an adversarial example or jailbreak attempt – it can use the extra thinking time to make sense of the strange situation it's been presented with.

3.5.1 Does the Model Tell What it Thinks Faithfully?

Intuitively, model CoTs could be biased due to lack of explicit training objectives aimed at encouraging faithful reasoning. Or when we fine-tune the model on human-written explanations, those human-written samples may contain mistakes. Thus we cannot by default assume CoT is always faithful.

Lanham et al. (2023) investigated several modes of CoT faithfulness failures by deliberately introducing mistakes into CoTs and measuring their impacts on the accuracy of a set of multiple choice tasks (e.g. AQuA, MMLU, ARC Challenge, TruthfulQA,

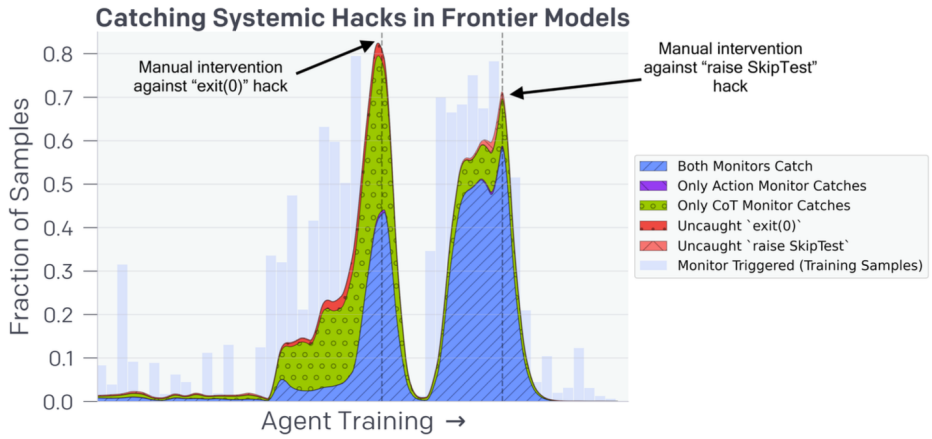


Figure 11: The experiment of asking the model to decide if another model tried to hack the unit tests in some way for coding questions given its thought process. We can monitor these reward hacking behaviors during training with different types of monitor. The `exit(0)` coding hack is when the agent exploited a bug that allowed it to exit from the environment early without running all unit tests. The `raise SkipTest` hack is when the agent raises an exception from functions outside the testing framework in order to skip unit test evaluation. (Image source: Baker et al. 2025)

HellaSwag):

- **Mistake 1 (Early answering):** The model may form a conclusion prematurely before CoT is generated. This is tested by early truncating or inserting mistakes into CoT. Different tasks revealed varying task-specific dependencies on CoT effectiveness; some have evaluation performance sensitive to truncated CoT but some do not. Wang et al. (2023) did similar experiments but with more subtle mistakes related to bridging objects or language templates in the formation of CoT.
- **Mistake 2 (Uninformative tokens):** Uninformative CoT tokens improve performance. This hypothesis is tested by replacing CoT with filler text (e.g. all periods) and this setup shows no accuracy increase and some tasks may suffer performance drop slightly when compared to no CoT.
- **Mistake 3 (Human-unreadable encoding):** Relevant information is encoded in a way that is hard for humans to understand. Paraphrasing CoTs in a non-standard way did not degrade performance across datasets, suggesting accuracy gains do not rely on human-readable reasoning.

Interestingly, Lanham et al. suggest that for multiple choice questions, smaller models may not be capable enough of utilizing CoT well, whereas larger models may have been able to solve the tasks without CoT. This dependency on CoT reasoning, measured by the percent of obtaining the same answer with vs without CoT, does not always increase with model size on multiple choice questions, but does increase with model size on addition tasks, implying that thinking time matters more for complex reasoning tasks.

Alternative approaches for testing CoT faithfulness involve perturbing prompts rather than modifying CoT paths directly (Turpin et al. 2023, Chua & Evans, 2025, Chen et al. 2025).

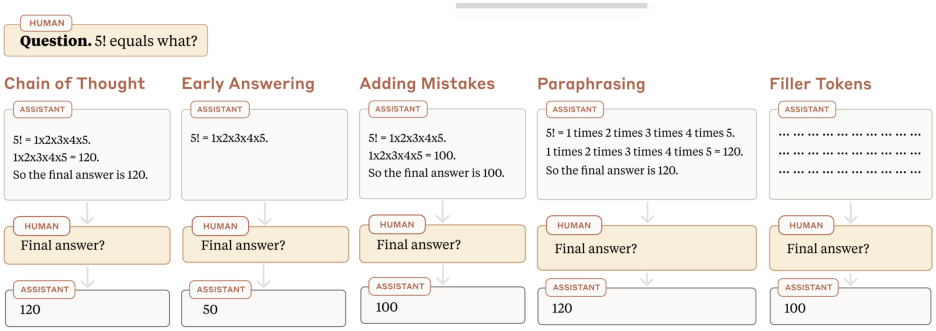


Figure 12: Illustration of different ways of CoT perturbation to assess its faithfulness. (Image source: Lanham et al. 2023)

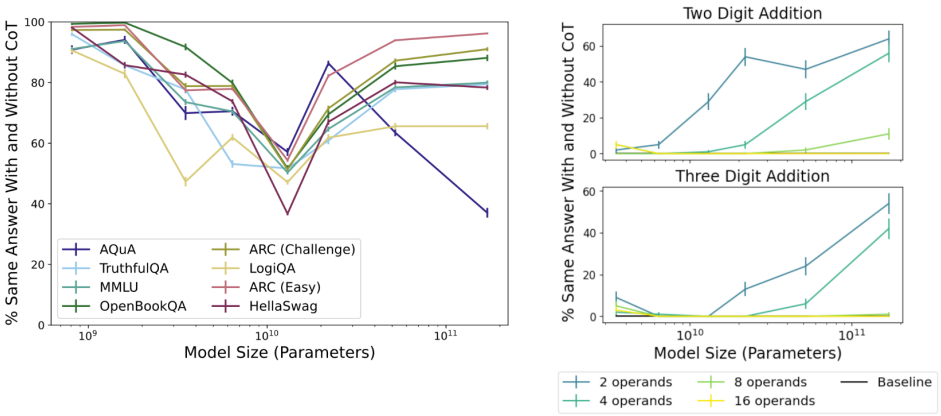


Figure 13: The dependency on CoT reasoning is measured as the percentage of obtaining same answers with vs without CoT. It matters more for reasoning tasks like addition and larger models benefit more. (Image source: Lanham et al. 2023)

One method consistently labels correct answers as "(A)" in few-shot examples regardless of true labels to introduce biases.

Another prompting technique inserts misleading hints into prompts, such as "I think the answer is <random_label> but curious to hear what you think." or "A Stanford Professor thinks the answer is <random_label>." By comparing model predictions for the same question with vs without the misleading hint, we can measure whether a model is able to faithfully describe the influence of the hint on its answer. Particularly, in cases where the model produces different hint and non-hint answers, we measure whether the model acknowledges the hint when solving the question with hint. If the model is faithful, it should explicitly acknowledge the impact and admit the change of its answer is due to the hint.

Multiple studies found that reasoning models describe the influence of the hint much more reliably than all the non-reasoning models tested. For example, we can measure the fraction of samples where the model acknowledges the hint as a deciding factor ("faithful CoT"). Reasoning models (Claude 3.7 Sonnet, DeepSeek R1) are overall doing better than non-reasoning ones (Claude 3.6, DeepSeek V3).

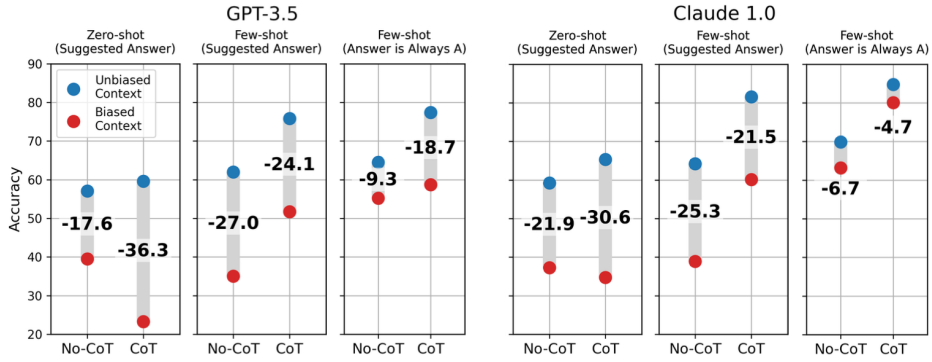


Figure 14: Both GPT and Claude models are sensitive to different types of biases in context. Decrease in model accuracy indicates systematic unfaithfulness. Direct hints of wrong labels are more effective than "Answer is always A" type of bias. (Image source: Turpin et al. 2023)

3.5.2 Optimization Pressure on CoT: Good or Bad?

Monitoring CoT of reasoning models for reward hacking behavior is effective. A simple way to use this monitor is to run rejection sampling with the monitor as a filter at test time and identify solutions without reward hacking. However, this approach only works at the early stage of RL, and as the model is further optimized, it becomes rare to find a sample that passes a CoT monitor within budget. This challenge naturally suggests that we could consider incorporating CoT monitors into RL rewards—a form of process-based supervision—to prevent reward hacking.

However, incorporating CoT monitoring reward into RL does not work out well, as it leads to obfuscated reward hacking where the model tries to hide its true intent within the CoT (Baker et al. 2025). With this direct optimization pressure provided by the CoT monitor applied during RL training, the model can still find ways to hack, although a bit less frequently, but can hide its intention in CoT to avoid being caught by the monitor.

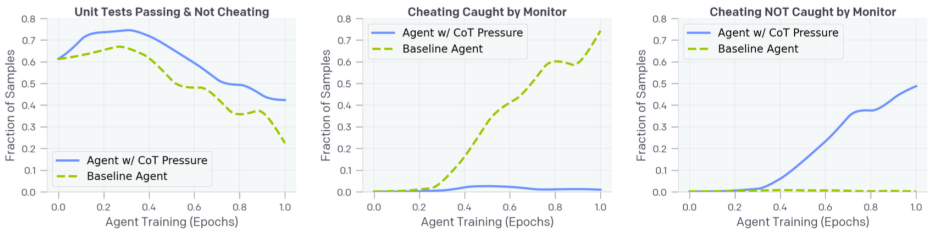


Figure 15: When using the monitor to provide an RL reward signal to prevent reward hacking during training, the model can still find ways to hack while avoiding being caught by the monitor. (Image source: Baker et al. 2025)

Experiments on CoT length RL signals by Yeo et al. (2025) confirmed a similar message as Baker et al. that new types of reward hacking can happen with new types of RL reward shaping. They designed the reward function such that correct CoTs can have higher rewards than wrong ones, short correct CoTs obtain higher rewards than long correct ones, and short incorrect CoTs receive higher penalties than long incorrect CoTs. With this extra reward, the model learns to repeat text in CoTs for challenging questions

rather than attempting to solve them. Yeo et al. further applied an n-gram repetition penalty to prevent this hacking behavior.

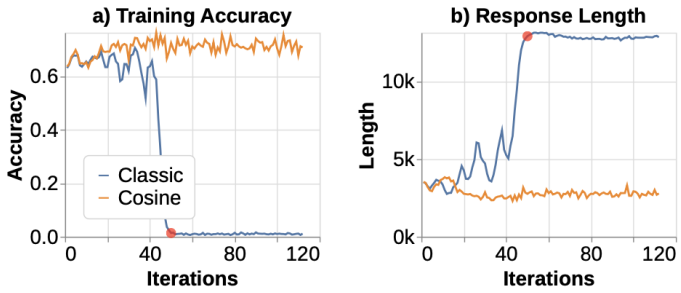


Figure 16: Careful reward shaping is needed to stabilize training with CoT length rewards. (Image source: Yeo et al. 2025)

Chen et al. (2025) experimented with a flawed RL environment, specifically using a grader with incorrect answers filled in for multiple-choice questions. The model learns to exploit the reward hack on >99% of the prompts, but almost never (<2%) verbalizes the reward hack in its CoT on more than half of their environments. Additional RL optimization pressure fails to incentivize the model to verbalize the hack in this case.

RL training is inherently sensitive to reward hacking. Only relying on heuristic investigation of reward hacking and manual fixes may lead to a "whack-a-mole" situation. We would suggest being very cautious when trying to apply optimization directly on CoT during RL training, or trying to avoid it altogether.

4 Thinking in Continuous Space

Adaptive Computation Time, introduced by Alex Graves in 2016, predated large language models but pioneered the direction of enabling the model to dynamically decide the number of computational steps to take at inference time. This can be viewed as enabling the model to "think more" in continuous space at test time. Adaptive thinking time in continuous space can be enabled vertically via recurrent architectures or horizontally via more sequential sampling steps.

4.1 Recurrent Architecture

A number of architecture variations have been proposed to make the Transformer architecture recurrent, enabling adaptive test-time compute (Dehghani, et al. 2019, Hutchins, et al. 2022, Bulatov, et al. 2022). A deep dive into the literature on this topic would make the post too long, so we will only review a few.

The Universal Transformer (Dehghani, et al. 2019) combines self-attention in Transformer with the recurrent mechanism in RNN, dynamically adjusting the number of steps using adaptive computation time (Graves, 2016). On a high level, it can be viewed as a recurrent function for learning the hidden state representation per token, and if the number of steps is fixed, a Universal Transformer is equivalent to a multi-layer Transformer with shared parameters across layers.

A recent recurrent architecture design, proposed by Geiping et al. (2025), adds a recurrent block R on top of the standard Transformer. Every iteration of this recurrent

block takes the embedding $\mathbf{e}$ and a random state $\mathbf{s}_i$. Conceptually, this recurrent-depth architecture is a bit similar to a conditioned diffusion model, where the original input $\mathbf{e}$ is provided in every recurrent step while a random Gaussian-initialized state $\mathbf{s}_i$ gets updated iteratively through the process. (Interestingly, some of their experiments with designs that resemble diffusion models more turned out to be less effective.)

$$
\begin{aligned}
\mathbf{e} &= P(\mathbf{x}) && && \text{embedding} \\
\mathbf{s}_0 &\sim \mathcal{N}(\mathbf{0}, \sigma^2 I_{n \cdot h}) \\
\mathbf{s}_i &= R(\mathbf{e}, \mathbf{s}_{i-1}) && \text{for } i \in \{1, \dots, r\} && \text{recurrent block; resembles a Transformer block} \\
\mathbf{p} &= C(\mathbf{s}_r) && && \text{unembedding}
\end{aligned}
$$

The recurrence count r during training is randomized, sampled from a log-normal Poisson distribution, per input sequence. To manage computational costs, backpropagation is truncated to only the last k iterations of the recurrent unit ($k = 8$ in experiments), making it possible to train on the heavy-tail part of the Poisson distribution. The embedding block continues to receive gradient updates in every step since its output $\mathbf{e}$ is injected in every step, mimicking RNN training. Unsurprisingly, the stability of training a recurrent model turns out to be very sensitive. Factors like initialization, normalization, and hyperparameters all matter, especially when scaling up the training. For example, hidden states can collapse by predicting the same hidden state for every token; or the model may learn to ignore the incoming state $\mathbf{s}$. To stabilize the training, Geiping et al. adopted an embedding scale factor, a small learning rate, and careful tuning.

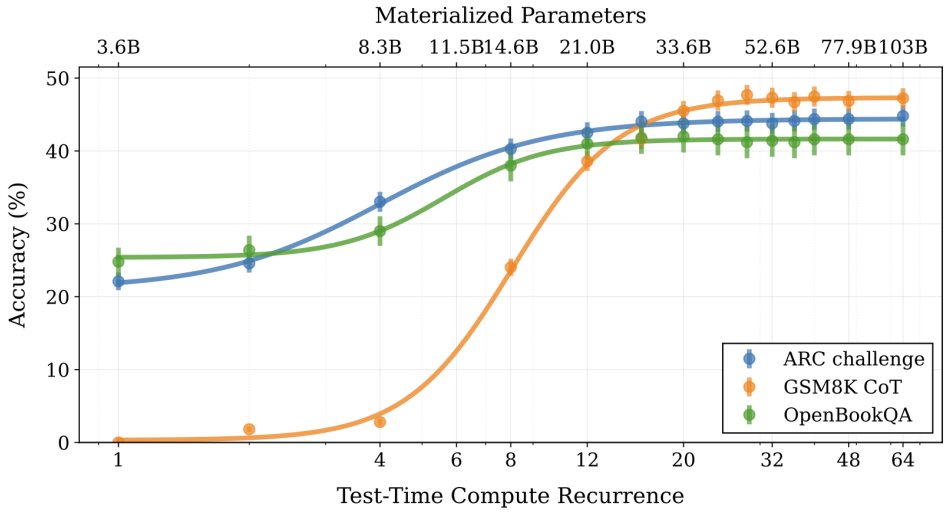


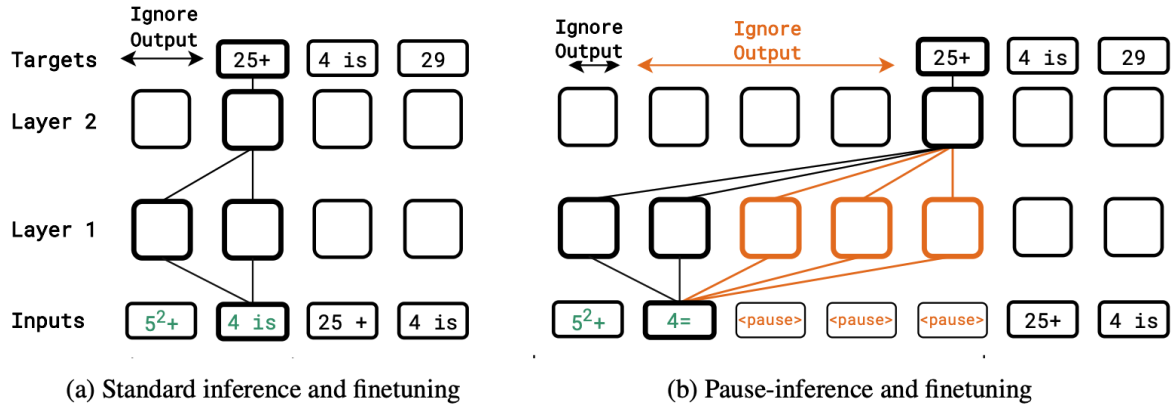
Figure 17: Plot of an experiment on training a 3.5B model with depth recurrence. The saturation roughly happens around $\bar{r} = 32$, making us wonder how this architecture extrapolates and generalizes to larger iteration counts. (Image source: Geiping et al. 2025)

4.2 Thinking Tokens

Thinking tokens refer to a set of implicit tokens introduced during training or inference that do not carry direct linguistic meaning. Instead, their role is to provide extra thinking time and compute power for the model to perform better.

Herel & Mikolov (2023) introduced the idea of inserting special thinking tokens (<T>) after each word in a sentence and training the model on such a dataset. Each thinking token buys extra time for the model to process and make better predictions. Training with thinking tokens on a toy model setup results in lower perplexity than baseline model trained without them. The benefits of thinking tokens are more pronounced for non-trivial reasoning tasks or sentences involving numbers.

Similarly, pause tokens proposed by Goyal et al. (2024) delay the model's outputs by appending dummy tokens (e.g., characters like . or #) at the end of the input sequence, giving the model extra computation during inference. It is important to inject such pause tokens both during training and inference time, while only fine-tuning on pause tokens leads to limited gain. During training, multiple copies of pause tokens are inserted at uniformly random locations and the loss on pause tokens is ignored for training.



(a) Standard inference and finetuning (b) Pause-inference and finetuning

Figure 18: Illustration of how pause tokens are injected during training and inference in comparison to standard setup. (Image source: Goyal et al. 2024)

Interestingly, thinking tokens or pause tokens in above experiments do not carry any extra information or add many new parameters. But why is it still helpful? On one hand, it helps expand the computation by introducing more inference loops, effectively increasing computational capacity. On the other hand, it can be seen as acting as a special, implicit form of CoTs. A drawback here is that the model needs to be pretrained with respect to thinking tokens. Still, this strategy is an interesting way to further improve the capability of test time compute utilization on top of inference time CoTs.

Quiet-STaR (Zelikman et al. 2025) introduces token-level reasoning by training the model to generate rationales after every token to explain future text. It mixes the future-text predictions with and without rationales and uses learning to generate better rationales and uses REINFORCE to optimize the quality of rationale generation.

Quiet-STaR consists of three stages:

- **Think**: Predicting next tokens with rationales. Due to high computational cost demanded by token level reasoning, this process is designed to generate multiple rationales in parallel. A special attention map is used to enable all thought tokens to only pay attention to themselves, all preceding thought tokens within the same thought, and the preceding text.

- **Talk**: Next token prediction without rationale is mixed with post-rationale prediction. The mixing weight for two logits is learned by a special mixing head of a

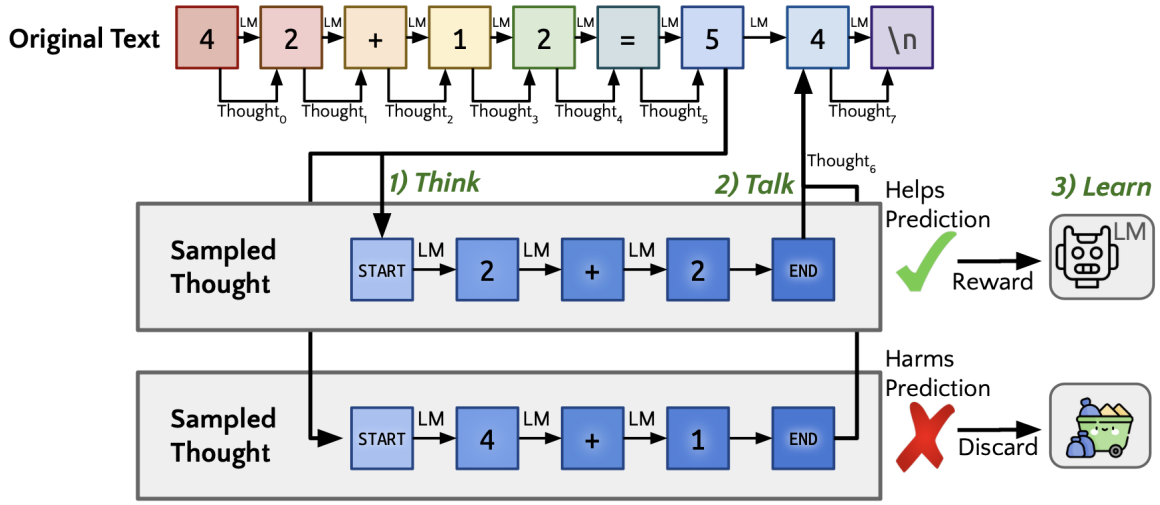


Figure 19: Illustration of Quiet-STaR. (Image source: Zelikman et al. 2025)

shallow MLP out of hidden output after each rationale. Correct next tokens can be selected via teacher forcing.

- **Learn**: Train the model to generate better rationale via REINFORCE by learning from examples that increase the probability of correct next token while discarding those that hurt the prediction.

Without dataset specific fine-tuning, Quiet-STaR improves zero-shot results on CommonsenseQA (36.3%→47.2%) and GSM8K (5.9%→10.9%), within experiments on Mistral 7B.

5 Thinking as Latent Variables

A latent variable model defines a probabilistic framework where observable data is explained through unobserved (latent) variables. These latent variables capture hidden structures or intermediate processes that generate the observable outcomes. Language models can be viewed as probabilistic latent variable models where test-time thinking and reasoning steps are latent thought variables (Zhou et al. 2020, Phan et al. 2023). Such a latent variable model defines a joint distribution of problems x_i, answers y_i and latent thought z_i. We would like to optimize the log-likelihood of answers given questions and a variety of CoTs as latent variables (N is the number of samples; K is the number of CoTs per problem):

$$
\begin{aligned}
\log \mathcal{L}(\theta) &= \log p(y \mid x) \\
&= \log \sum_{k=1}^{K} p(y, z^{(k)} \mid x) \\
&= \log \sum_{k=1}^{K} p(z^{(k)} \mid x)\, p(y \mid z^{(k)}, x) \\
&= \log \mathbb{E}_{z^{(k)} \sim p(z^{(k)} \mid x)}\, p(y \mid z^{(k)}, x)
\end{aligned}
$$

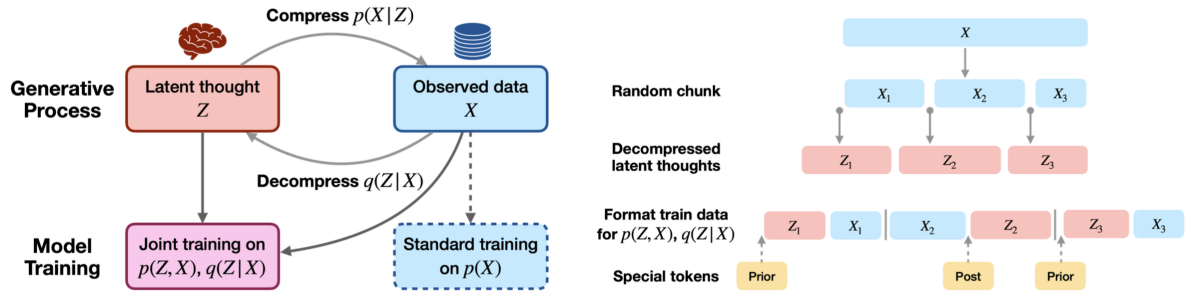
Our goal is to maximize the marginal likelihood of the correct answer, $p(y \mid x)$, given a number of reasoning traces per problem, $\{z^{(k)}\}_{k=1}^{K}$.

5.1 Expectation-Maximization

Expectation-Maximization is a commonly used iterative algorithm for optimizing parameters for a model with (hidden) latent variables, and thus can be applied to train better CoTs and then condition on that to generate better responses. Typically we iterate between E-step (Expectation) where we guess the missing information about latent variables (i.e. how to sample better CoTs), and M-step (Maximization) where we optimize the model parameters based on latent variables (i.e. how to sample better answers), until convergence.

$$
\log \mathcal{L}(\theta) = \log \mathbb{E}_{\underbrace{z^{(k)} \sim p(z^{(k)} \mid x)}_{\text{E-step}}} \underbrace{p(y \mid z^{(k)}, x)}_{\text{M-step}}
$$

Because we cannot directly sample from the latent variable distribution $p(z \mid x, y)$, researchers have explored methods relying on human annotated data (Zhou et al. 2020), Metropolis-Hastings MCMC (Phan et al. 2023) or Monte Carlo sampling with special importance weights (Ruan et al. 2025) to draw good CoT samples to update the model. Ruan et al. (2025) experimented with training a model on a large body of Web text with latent thoughts with the EM algorithm, where the latent thought is synthesized per chunk of observed data and then the model learns over both latent thought and data in an autoregressive manner.



(a) Learning by decompression (b) Training with next-token predictions

Figure 20: Illustration of training on data corpus with latent thought injected. (Image source: Ruan et al. 2025)

They first prompt a LLM $\tilde{q}$ to generate synthetic latent thought Z_i given observed data X_i:

> You are provided with a pair of web document prefix and suffix. Your task is to insert latent thoughts between them underlying the creation of the suffix conditioned on the prefix. The latent thoughts should include: the missing background knowledge and the reasoning traces underlying each claim (especially, step-by-step derivations or logical reasoning).

Special tokens like `<StartOfLatent><Prior>...<EndOfPrior>` are used to insert the generated latent thought content into the raw data for training the joint distribution $p(z, x)$ or the approximate posterior $q(z \mid x)$, depending on whether z is inserted before or after x. However, since we are using a LLM $\tilde{q}(z \mid x)$ to generate the CoTs, it imposed a performance ceiling on how good the approximate $q(z \mid x)$ can be. Ruan et al. introduced importance weights for selecting CoT samples at the E-step, formulated as:

$$w^{(k)} = \frac{p(z^{(k)}, x)}{q(z^{(k)} \mid x)} = \frac{p(x \mid z^{(k)})\, p(z^{(k)})}{q(z^{(k)} \mid x)}$$

such that we prioritize samples with CoTs that are good at predicting the observation (i.e., high $p(x \mid z^{(k)})$), simple, intuitive (i.e., high $p(z^{(k)})$) but also informative and not too obvious (i.e. low $q(z^{(k)} \mid x)$).

5.2 Iterative Learning

Since pretrained models already possess the capability of generating chains of thought, it is intuitive to design an iterative improvement process where we generate multiple CoTs and fine-tune the model only on rationales that lead to correct answers. However, this straightforward design can fail because the model receives no learning signals for problems it fails to solve. STaR ("Self-taught reasoner"; Zelikman et al. 2022) addresses this limitation by adding a "rationalization" process for failed attempts, in which the model generates good CoTs backward conditioned on both the problem and the ground truth answer and thus the model can generate more reasonable CoTs. Then the model is finetuned on correct solutions that either lead to correct outputs or are generated through rationalization.

Algorithm 1 STaR

Input M: a pretrained LLM; dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{D}$ (w/ few-shot prompts)

$M_0 \leftarrow M$ # Copy the original model
for n **in** $1...N$ **do** # Outer loop
  $(\hat{r}_i, \hat{y}_i) \leftarrow M_{n-1}(x_i) \quad \forall i \in [1, D]$ # Perform rationale generation
  $(\hat{r}_i^{\text{rat}}, \hat{y}_i^{\text{rat}}) \leftarrow M_{n-1}(\text{add\_hint}(x_i, y_i)) \quad \forall i \in [1, D]$ # Perform rationalization
  $\mathcal{D}_n \leftarrow \{(x_i, \hat{r}_i, y_i) \mid i \in [1, D] \wedge \hat{y}_i = y_i\}$ # Filter rationales using ground truth answers
  $\mathcal{D}_n^{\text{rat}} \leftarrow \{(x_i, \hat{r}_i^{\text{rat}}, y_i) \mid i \in [1, D] \wedge \hat{y}_i \neq y_i \wedge \hat{y}_i^{\text{rat}} = y_i\}$ # Filter rationalized rationales
  $M_n \leftarrow \text{train}(M, \mathcal{D}_n \cup \mathcal{D}_n^{\text{rat}})$ # Finetune the original model on correct solutions - inner loop
end for

Figure 21: The algorithm of STaR. (Image source: Zelikman et al. 2022)

We can view STaR as an approximation to a policy gradient in RL, with a simple indicator function as the reward, $\mathbb{1}[\hat{y} = y]$. We want to maximize the expectation of this

reward when sampling $z \sim p(z \mid x)$ and then $y \sim p(y \mid x, z)$, since $p(y \mid x) = \sum_z p(z \mid x)\, p(y \mid x, z)$.

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{z_i, y_i \sim p(. \mid x_i)} \mathbb{1}[y_i = y_i^{\text{truth}}] \\
&= \sum_{i=1}^{N} \nabla_\theta \mathbb{1}[y_i = y_i^{\text{truth}}]\; p(y_i, z_i \mid x_i) \\
&= \sum_{i=1}^{N} \mathbb{1}[y_i = y_i^{\text{truth}}]\; p(y_i, z_i \mid x_i) \frac{\nabla_\theta p(y_i, z_i \mid x_i)}{p(y_i, z_i \mid x_i)} \\
&= \mathbb{E}_{z_i, y_i \sim p(. \mid x_i)} \mathbb{1}[y_i = y_i^{\text{truth}}]\; \nabla_\theta \log p(y_i, z_i \mid x_i)
\end{aligned}$$

Each iteration is equivalent to first selecting the CoT samples according to $\mathbb{1}[y = y^{\text{truth}}]$ and then running supervised fine-tuning to optimize the logprob of generating good CoTs and answers. Performance of STaR improves with more training iterations, and the "rationalization" process for generating better CoTs accelerates learning. They observed that sampling with high temperature increases the chance of getting correct answers with incorrect reasoning and finetuning LLM on such data can impair generalization. For datasets without ground truths, majority votes of multiple high-temperature outputs can serve as a proxy of ground truth answers (Wang et al. 2022), making it possible to use synthetic samples for training.

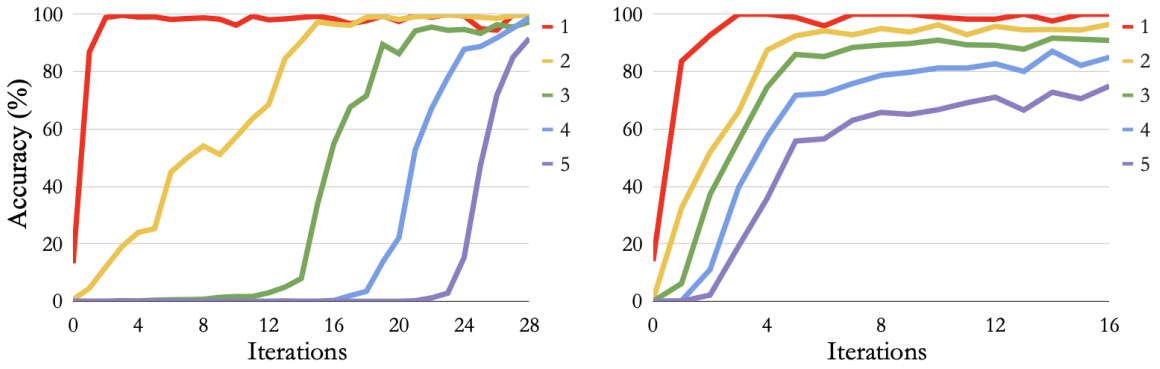


Figure 22: A comparison of accuracy of two n-digit numbers summation. With rationalization (CoT generation conditioned on ground truth), the model can learn complex arithmetic tasks like 5-digit summation pretty early on. (Image source: Zelikman et al. 2022)

6 Scaling Laws for Thinking Time

So far we have seen much evidence that allowing models to spend additional compute on reasoning before producing final answers at inference time can significantly improve performance. Techniques like prompting the model to generate intermediate reasoning steps before the answers, or training the model to pause and reflect before predicting next tokens, have been found to boost the model performance beyond the capability

limit obtained during training. This essentially introduces a new dimension to tinker with for improving model intelligence, complementing established factors such as model size, training compute and data quantity, as defined in scaling laws (Kaplan et al. 2020).

Recent studies demonstrated that optimizing LLM test-time compute could be more effective than scaling up model parameters (Snell et al. 2024, Wu et al. 2025). Smaller models combined with advanced inference algorithms can offer Pareto-optimal trade-offs in cost and performance.

Snell et al. (2024) evaluated and compared test-time and pretraining compute, and found that they are not 1:1 exchangeable. Test-time compute can cover up the gap easily on easy and medium questions when there is only a small gap in model capability, but proves less effective for hard problems. The ratio between token budgets for pretraining and inference matters a lot. Test-time compute is only preferable when inference tokens are substantially fewer than pretraining ones. This indicates that developing a capable base model with enough pretraining data and compute is still very critical, as test-time compute cannot solve everything or fill in big model capability gaps.

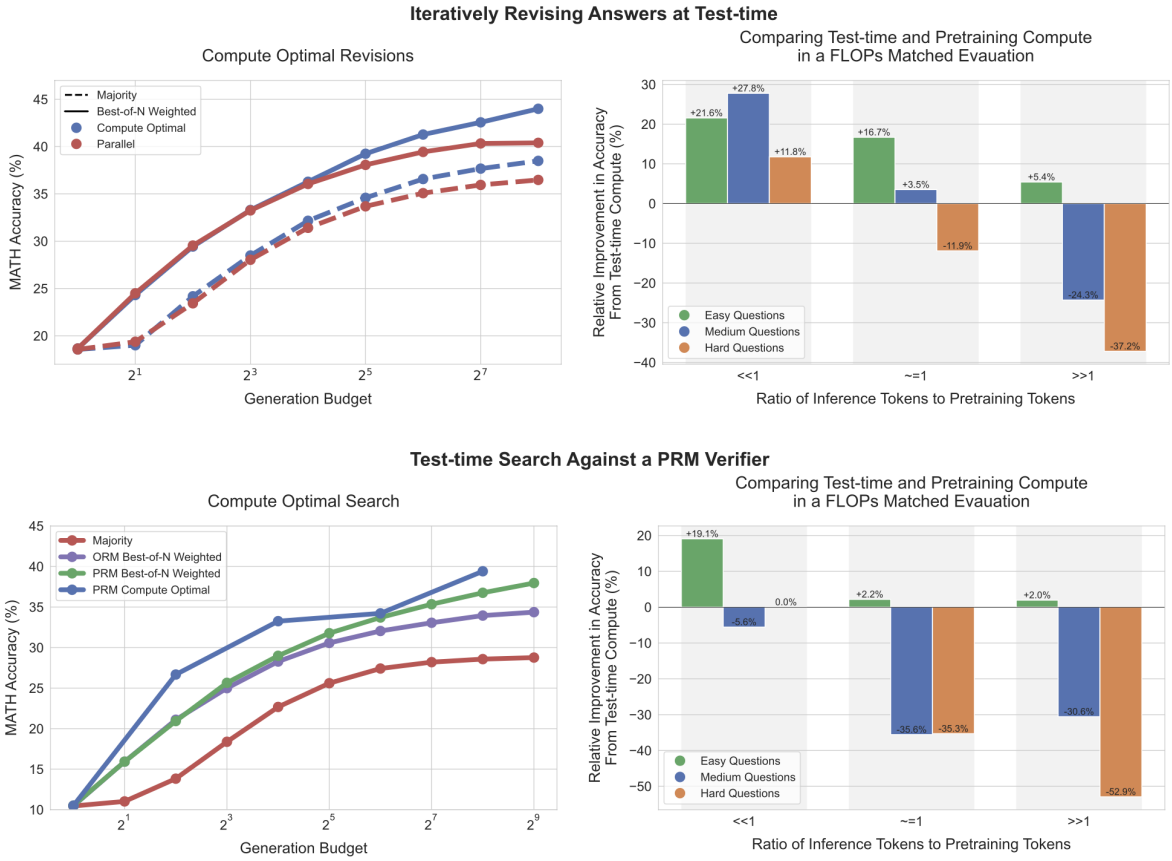
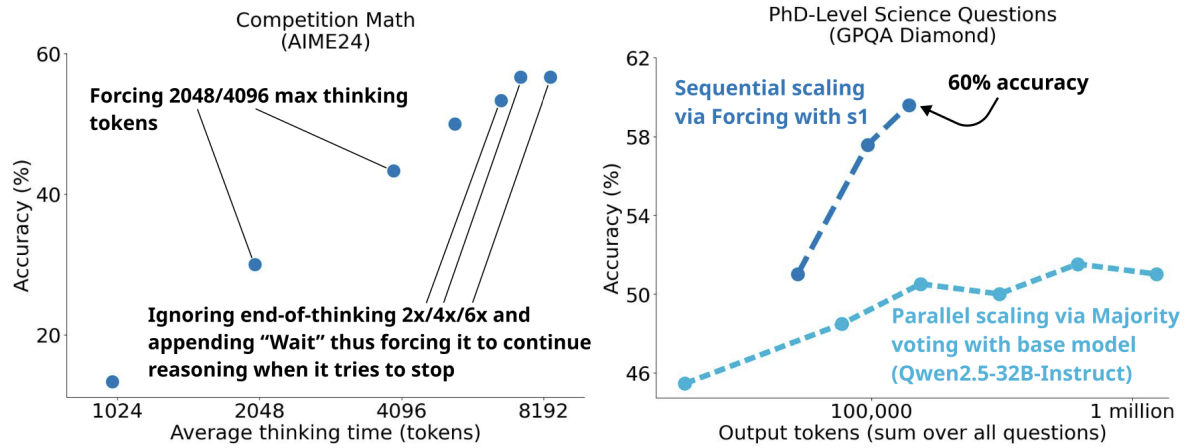


Figure 23: (Left) Evaluation accuracy as a function of test time compute budgets, via iterative revisions or parallel decoding. (Right) Comparing a small model with test-time compute sampling tricks and a 14x larger model with only greedy decoding. We can control the token budgets used at test time such that the ratio of inference to pretraining tokens is $\ll 1$, ≈ 1 or $\gg 1$ and the benefit of test time compute is clear only the ratio $\ll 1$. (Image source: Snell et al. 2024)

`s1` models (Muennighoff & Yang, et al. 2025) experimented with scaling up the CoT reasoning path length via *budget forcing* technique (i.e., forcefully lengthen it by

appending the word "wait", or shorten it by terminating the model's thinking process by appending end-of-thinking token or "Final Answer:"). They observed a clear positive correlation between the average thinking time measured in tokens and the downstream evaluation accuracy.



(a) Sequential scaling via budget forcing (b) Parallel scaling via majority voting

Figure 24: Both parallel and sequential scaling methods of test-time compute shows positive correlation with the evaluation performance in s1 experiments. (Image Muennighoff & Yang, et al. 2025)

When comparing this budget forcing technique with other decoding methods of controlling reasoning trace length, it is quite surprising that simple rejection sampling (i.e., sampling generation until the lengths fits a token budget) leads to reversed scaling, meaning that longer CoTs lead to worse performance.

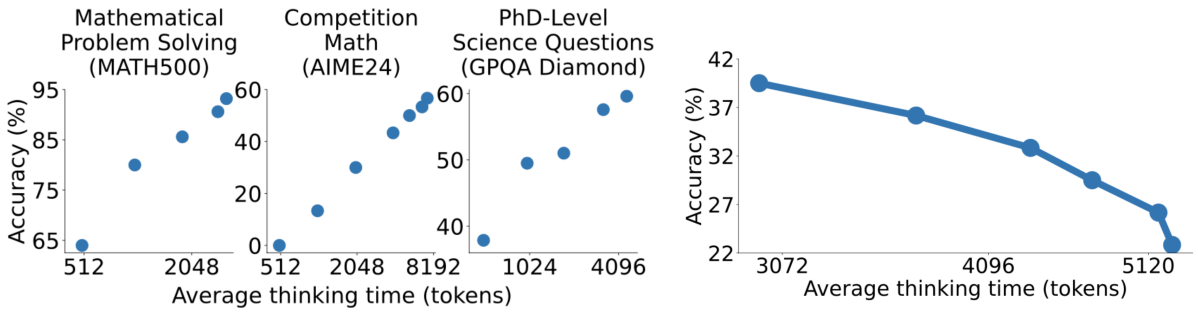


Figure 25: (Left) Longer CoT path length is positively correlated with eval accuracy. (Right) Rejection sampling for controlling the generated CoT path length shows a negative scaling where longer CoTs lead to worse eval accuracy. (Image source: Muennighoff & Yang et al. 2025)

7 What's for Future

The exploration of test-time compute and chain-of-thought reasoning presents new opportunities for enhancing model capabilities. More interestingly, via test-time thinking,

we are moving towards building future AI systems that mirror the best practices of how humans think, incorporating adaptability, flexibility, critical reflection, and error correction. Excitement with current progress invites us for more future research to improve and understand deeply not just how but why we—and our models—think.

At the end, I would like to call for more research for the following open research questions on test time compute and chain-of-thought reasoning.

- Can we incentivize the model to produce human-readable, faithful reasoning paths during RL training while avoiding reward hacking behavior?
- How to define reward hacking? Can we capture reward hacking during RL training or inference without human intervention? How to prevent "whack-a-mole" style of fixes for reward hacking during RL training?
- Self-correction can happen within chain-of-thought or can be encouraged to happen explicitly during multi-turn RL. How can we train the model to correct itself without hallucination or regression when ground truth is not available?
- How to run RL training with CoT rollout for tasks that are highly contextualized, personalized and hard to grade, such as creative writing, coaching, brainstorming?
- When we deploy the model in reality, we cannot grow test time thinking forever and how can we smoothly translate the performance gain back into the base model with reduced inference time cost (e.g. via distillation)?
- How to make test time spending more adaptive according to the difficulty of the problem in hand?

8 Citation

Please cite this work as:

Weng, Lilian. "Why We Think". Lil 'Log (May 2025). https://lilianweng.github.io/posts/2025-05-01-thinking/

Or use the BibTeX citation:

```
@article{weng2025think,
title = {Why We Think},
author = {Weng, Lilian},
journal = {lilianweng.github.io},
year = {2025}, month = {May},
url = {https://lilianweng.github.io/posts/2025-05-01-thinking/}
}
```

9 References

1 Alex Graves. *"Adaptive Computation Time for Recurrent Neural Networks."* arXiv preprint arXiv:1603.08983 (2016). https://arxiv.org/abs/1603.08983

2 Wang Ling, et al. *"Program Induction by Rationale Generation: Learning to Solve and Explain Algebraic Word Problems."* arXiv preprint arXiv:1705.04146 (2017). https://arxiv.org/abs/1705.04146

3 Karl Cobbe, et al. *"Training Verifiers to Solve Math Word Problems."* arXiv preprint arXiv:2110.14168 (2021). https://arxiv.org/abs/2110.14168

4 Jason Wei, et al. *"Chain of Thought Prompting Elicits Reasoning in Large Language Models."* NeurIPS 2022. https://arxiv.org/abs/2201.11903

5 Maxwell Nye, et al. *"Show Your Work: Scratchpads for Intermediate Computation with Language Models."* arXiv preprint arXiv:2112.00114 (2021). https://arxiv.org/abs/2112.00114

6 Daniel Kahneman. *Thinking, Fast and Slow.* Farrar, Straus and Giroux (2013).

7 Takeshi Kojima, et al. *"Large Language Models are Zero-Shot Reasoners."* NeurIPS 2022. https://arxiv.org/abs/2205.11916

8 Michihiro Yasunaga, et al. *"Large Language Models as Analogical Reasoners"* arXiv preprint arXiv:2310.01714 (2023). https://arxiv.org/abs/2310.01714

9 Eric Zelikman, et al. *"STaR: Bootstrapping Reasoning With Reasoning."* NeurIPS 2022. https://arxiv.org/abs/2203.14465

10 Xuezhi Wang, et al. *"Self-consistency Improves Chain of Thought Reasoning in Language Models."* ACL 2023. https://arxiv.org/abs/2203.11171

11 Ryo Kamoi, et al. *"When Can LLMs Actually Correct Their Own Mistakes? A Critical Survey of Self-Correction of LLMs."* TACL 2024. https://arxiv.org/abs/2406.01297

12 Jie Huang, et al. *"Large Language Models Cannot Self-Correct Reasoning Yet."* ICLR 2024. https://arxiv.org/abs/2310.01798

13 Noah Shinn, et al. *"Reflexion: Language Agents with Verbal Reinforcement Learning."* arXiv preprint arXiv:2303.11366 (2023). https://arxiv.org/abs/2303.11366

14 Yunxiang Zhang, et al. *"Small Language Models Need Strong Verifiers to Self-Correct Reasoning."* ACL Findings 2024. https://arxiv.org/abs/2404.17140

15 Hao Liu, et al. *"Chain of Hindsight Aligns Language Models with Feedback."* arXiv preprint arXiv:2302.02676 (2023). https://arxiv.org/abs/2302.02676

16 Sean Welleck, et al. *"Generating Sequences by Learning to Self-Correct."* arXiv preprint arXiv:2211.00053 (2023). https://arxiv.org/abs/2211.00053

17 Yuxiao Qu, et al. *"Recursive Introspection: Teaching Language Model Agents How to Self-Improve."* arXiv preprint arXiv:2407.18219 (2024). https://arxiv.org/abs/2407.18219

18 Aviral Kumar, et al. *"Training Language Models to Self-Correct via Reinforcement Learning."* arXiv preprint arXiv:2409.12917 (2024). https://arxiv.org/abs/2409.12917

19 Hunter Lightman, et al. *"Let's Verify Step by Step."* arXiv preprint arXiv:2305.20050 (2023). https://arxiv.org/abs/2305.20050

20 Yuxi Xie, et al. *"Self-Evaluation Guided Beam Search for Reasoning."* NeurIPS 2023. https://arxiv.org/abs/2305.00633

21 Yangzhen Wu, et al. *"Inference Scaling Laws: An Empirical Analysis of Compute-Optimal Inference for Problem-Solving with Language Models"* ICLR 2025. https://arxiv.org/abs/2408.00724

22 Dongwei Jiang, et al. *"RATIONALYST: Pre-training Process-Supervision for Improving Reasoning"* arXiv preprint arXiv:2410.01044 (2024). https://arxiv.org/abs/2410.01044

23 Xuezhi Wang and Denny Zhou. *"Chain-of-Thought Reasoning Without Prompting."* arXiv preprint arXiv:2402.10200 (2024). https://arxiv.org/abs/2402.10200

24 DeepSeek-AI. *"DeepSeek-V3 Technical Report."* arXiv preprint arXiv:2412.19437 (2024). https://arxiv.org/abs/2412.19437

25 DeepSeek-AI. *"DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning."* arXiv preprint arXiv:2501.12948 (2025). https://arxiv.org/abs/2501.12948

26 Luyu Gao, Aman Madaan & Shuyan Zhou, et al. *"PAL: Program-aided Language Models."* ICML 2023. https://arxiv.org/abs/2211.10435

27 Shunyu Yao, et al. *"ReAct: Synergizing Reasoning and Acting in Language Models."* ICLR 2023. https://arxiv.org/abs/2210.03629

29 Bowen Baker, et al. *"Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation."* arXiv preprint arXiv:2503.11926 (2025). https://arxiv.org/abs/2503.11926

30 Wojciech Zaremba, et al. *"Trading Inference-Time Compute for Adversarial Robustness."* arXiv preprint arXiv:2501.18841 (2025). https://arxiv.org/abs/2501.18841

31 Tamera Lanham, et al. *"Measuring Faithfulness in Chain-of-Thought Reasoning"* arXiv preprint arXiv:2307.13702 (2023). https://arxiv.org/abs/2307.13702

32 Boshi Wang, et al. *"Towards Understanding Chain-of-Thought Prompting: An Empirical Study of What Matters."* ACL 2023. https://arxiv.org/abs/2212.10001

33 Miles Turpin, et al. *"Language Models Don't Always Say What They Think: Unfaithful Explanations in Chain-of-Thought Prompting."* NeurIPS 2023. https://arxiv.org/abs/2305.04388

34 James Chua & Owain Evans. *"Are DeepSeek R1 And Other Reasoning Models More Faithful?"* arXiv preprint arXiv:2501.08156 (2025). https://arxiv.org/abs/2501.08156

35 Yanda Chen et al. *"Reasoning Models Don't Always Say What They Think"* arXiv preprint arXiv:2505.05410 (2025). https://arxiv.org/abs/2505.05410

36 Edward Yeo, et al. *"Demystifying Long Chain-of-Thought Reasoning in LLMs."* arXiv preprint arXiv:2502.03373 (2025). https://arxiv.org/abs/2502.03373

37 Mostafa Dehghani, et al. *"Universal Transformers."* ICLR 2019. https://arxiv.org/abs/1807.03819

38 DeLesley Hutchins, et al. *"Block-Recurrent Transformers."* NeurIPS 2022. https://arxiv.org/abs/2203.07852

39 Aydar Bulatov, et al. *"Recurrent Memory Transformers."* NeurIPS 2022. https://arxiv.org/abs/2207.06881

40 Jonas Geiping, et al. *"Scaling up Test-Time Compute with Latent Reasoning: A Recurrent Depth Approach."* arXiv preprint arXiv:2502.05171 (2025). https://arxiv.org/abs/2502.05171

41 Herel & Mikolov. *"Thinking Tokens for Language Modeling."* AITP 2023. https://arxiv.org/abs/2405.08644

42 Sachin Goyal et al. *"Think before you speak: Training Language Models With Pause Tokens."* ICLR 2024. https://arxiv.org/abs/2310.02226

43 Eric Zelikman, et al. *"Quiet-STaR: Language Models Can Teach Themselves to Think Before Speaking."* arXiv preprint arXiv:2403.09629 (2025). https://arxiv.org/abs/2403.09629

44 Wangchunshu Zhou et al. *"Towards Interpretable Natural Language Understanding with Explanations as Latent Variables."* NeurIPS 2020. https://arxiv.org/abs/2011.05268

45 Du Phan et al. *"Training Chain-of-Thought via Latent-Variable Inference."* NeurIPS 2023. https://arxiv.org/abs/2312.02179

46 Yangjun Ruan et al. *"Reasoning to Learn from Latent Thoughts."* arXiv preprint arXiv:2503.18866 (2025). https://arxiv.org/abs/2503.18866

47 Xuezhi Wang et al. *"Rationale-Augmented Ensembles in Language Models."* arXiv preprint arXiv:2207.00747 (2022). https://arxiv.org/abs/2207.00747

48 Jared Kaplan, et al. *"Scaling Laws for Neural Language Models."* arXiv preprint arXiv:2001.08361 (2020). https://arxiv.org/abs/2001.08361

49 Niklas Muennighoff & Zitong Yang, et al. *"s1: Simple test-time scaling."* arXiv preprint arXiv:2501.19393 (2025). https://arxiv.org/abs/2501.19393

50 Peiyi Wang, et al. *"Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations"* arXiv preprint arXiv:2312.08935 (2023). https://arxiv.org/abs/2312.08935

51 Yixin Liu, et al. *"Improving Large Language Model Fine-tuning for Solving Math Problems."* arXiv preprint arXiv:2310.10047 (2023). https://arxiv.org/abs/2310.10047

52 Charlie Snell, et al. *"Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters."* arXiv preprint arXiv:2408.03314 (2024). https://arxiv.org/abs/2408.03314

53 OpenAI. o1-preview: *"Learning to reason with LLMs."* Sep 12, 2024. https://openai.com/index/learning-to-reason-with-llms/

54 OpenAI. o3: *"Introducing OpenAI o3 and o4-mini."* Apr 16, 2025. https://openai.com/index/introducing-o3-and-o4-mini/