CIS 6200: Emerging Theory of LLMs **Fall 2026**

Topic 4A: Computing with Chain of Thought

Last updated: October 8, 2026 *Surbhi Goel*

Acknowledgements: *These notes draw on Nye et al. [3], Wei et al. [4], and Kojima et al. [5] for the evidence that chains of thought help; on Li, Liu, Zhou, and Ma [6] and Merrill and Sabharwal [7] for the computational view; on Sanford, Hsu, and Telgarsky [8] for the communication argument; on Cabannes et al. [9] and Liu et al. [10] for state updates and parallel computation; on Goyal et al. [13] and London and Kanade [11] for pause tokens; and on Hao et al. [14] and Zhu et al. [15] for continuous thoughts. The exposition also draws on Hsu's lectures [1] and Merrill's dissertation [2]. A supplement collects further results.*

AI Disclosure: *These notes were drafted and revised with AI assistance. If you find an error, please let me know.*

When asked to solve a problem, a language model can write intermediate steps before giving its answer. For example, to compute 67×3, it might first write $60 \times 3 = 180$ and $7 \times 3 = 21$, then add the two results. This is *chain-of-thought* generation. Each written result becomes part of the input for later predictions. Why can this help a model solve a problem that it struggles to answer directly? We will study how writing and reading these intermediate results changes the computation a Transformer can perform, and how observing the steps can help a learner discover that computation.

1 Historical Context

Nye et al. compared two ways of training a language model on tasks such as adding long integers [3]. One model was trained to write only the answer. The other was trained to write a *scratchpad* first: the intermediate results of the computation, such as the digits and carries of long addition, followed by the answer (Figure 1). Trained on numbers with up to 8 digits and tested on numbers with 9 and 10 digits, models that answered directly failed completely, while models with a scratchpad improved steadily with model size.

Wei et al. showed that large pretrained models can also produce such steps through *in-context learning* [4]. They put a few worked examples in the prompt, each showing its steps before its answer. The model then continued with the steps for a new problem, without updating its weights. With eight such examples, PaLM, a model with 540 billion parameters, solved 56.9% of the grade-school math word problems in GSM8K, against 17.9% when the examples showed only the answers (Figure 2). The gain appeared only in the larger models in these experiments; the models with fewer than ten billion parameters wrote fluent but illogical chains and did worse than with answers alone. Kojima et al. needed no worked examples at all: starting the answer with "Let's think step by step" raised the accuracy of the InstructGPT model text-davinci-002 from 17.7% to 78.7% on the arithmetic benchmark MultiArith and from 10.4% to 40.7% on GSM8K [5].

```
Input:
2 9 + 5 7

Target:
<scratch>
2 9 + 5 7 ,  C: 0
2 + 5 , 6 C: 1  # added 9 + 7 = 6 carry 1
, 8 6 C: 0  # added 2 + 5 + 1 = 8 carry 0
0 8 6
</scratch>
8 6
```

Figure 1: An addition scratchpad, reproduced from Nye et al. [3, Figure 2]. Each line records the digits still to be added, the partial sum, and the carry after `C:`. The model is trained to generate the scratchpad followed by the answer. Comments beginning with `#` explain the steps and are not training targets.

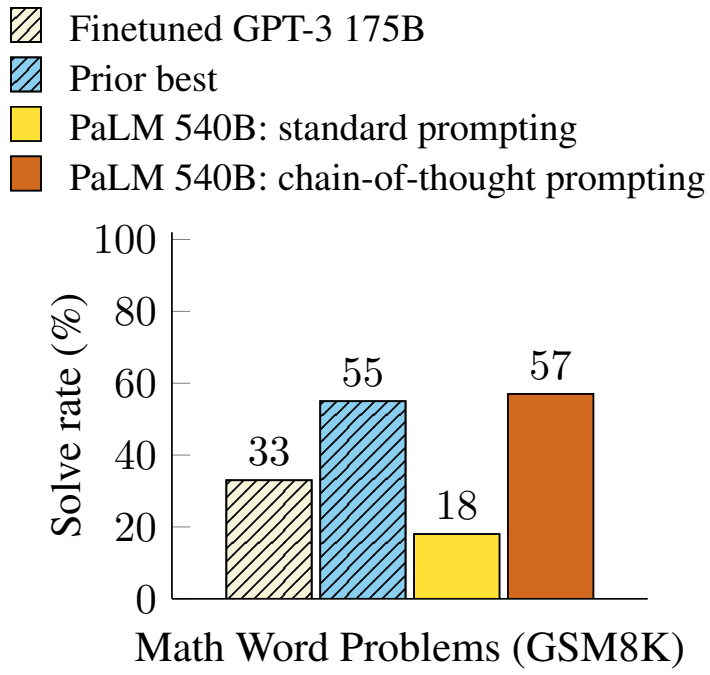


Figure 2: PaLM with 540 billion parameters on GSM8K, reproduced from Wei et al. [4, Figure 2]. The same pretrained model is given examples with answers only (standard prompting) or examples with intermediate steps (chain-of-thought prompting). The latter raises accuracy from 17.9% to 56.9%. No model parameters are updated.

Why do the steps help? In the scratchpad training examples, each intermediate result is a label that tells the learner what to compute. At inference time, a generated result is available for the model to read when computing the next one. These are two different benefits: supervision of the steps and computation using the steps. We begin with the computational question: *What can a Transformer do by writing intermediate results and reading them back?*

2 Generating a Chain of Thought

Let $x = (x_1, \ldots, x_n)$ be the input, a string of tokens from a finite vocabulary $\mathcal{V}$. A decoder-only Transformer with fixed parameters θ produces a distribution for each next token,

$$q_\theta(\cdot \mid x, y_1, \ldots, y_{t-1}), \qquad t = 1, \ldots, N+1. \tag{1}$$

We can sample from this distribution or choose a token by another decoding rule. The selected token is appended to the prefix, embedded like an input token, and read by later positions. For simplicity, we call $y_1, \ldots, y_N$ the *chain of thought* and take y_{N+1} to be a one-token *answer*.

What computation does this add? Suppose the Transformer has L layers, with causal attention as in Topic 3A. At each new position, the representation passes through these layers. An attention head can read the input and the earlier generated positions. Their keys and values have already been computed and can be reused from the KV cache. After the last layer, the model chooses a token and feeds its embedding into the first layer at the next position (Figure 3). The same weights are used again, now with the new token available.

This feedback makes a result of one pass available when the next pass forms its queries. The token carries at most $\log_2 |\mathcal{V}|$ bits, but it need not summarize the entire computation: earlier tokens and their cached representations remain available for later attention heads to read.

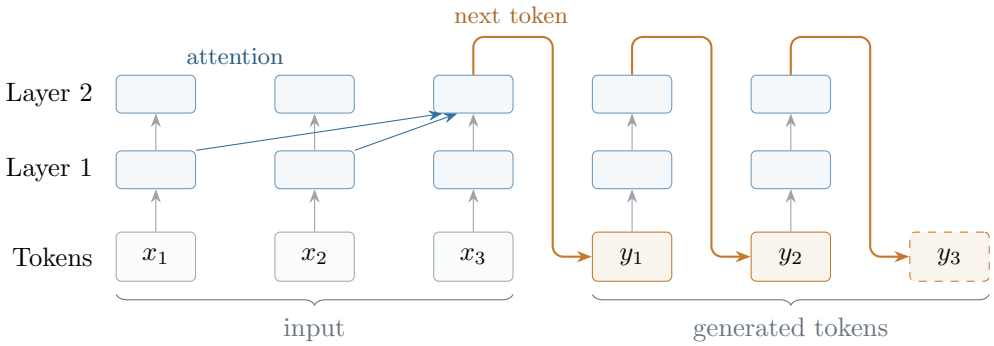


Figure 3: Generation unrolled over positions, for a two-layer Transformer. Attention lets each layer read earlier positions (blue, shown for one position). The output of the last layer at one position reaches the next position only through the generated token (orange).

The last layer at position $n + t - 1$ helps determine the input at position $n + t$. Thus N intermediate tokens and an answer allow a path through $L(N + 1)$ layers of dependent computation, using the same L-layer model. Each pass can read a long history, so a token counts one additional pass rather than a fixed number of elementary operations.

3 What Does a Chain of Thought Add?

A model of fixed depth gets another forward pass each time it writes a token. How much does this change what it can compute? We will use INDEX to understand what a single intermediate result adds, then ask what becomes possible when the chain can grow with the input.

3.1 One Intermediate Result

Recall INDEX from Topic 3B. Alice has s bits, and Bob wants the bit at position r. If Alice speaks before learning r, her message must contain enough information to recover every bit. If Bob can speak first, he names r and Alice replies with just the requested bit.

Our Transformer version puts a marker before each bit: $\star$ before the requested bit and $\cdot$ before the others. A final `read` token asks for the answer. A one-layer model faces the same obstruction as Alice: its query is fixed before attention has found the marker. The communication argument from Topic 3B shows that such a model cannot always answer correctly unless its size or numerical precision grows sufficiently with the number of bits.

Allow one generated token before the answer. The first pass finds $\star$ and writes the position immediately after it. The next pass uses this address to read the bit. For this construction, suppose the vocabulary contains a token $\langle j \rangle$ naming each position j. Figure 4 shows the two passes.

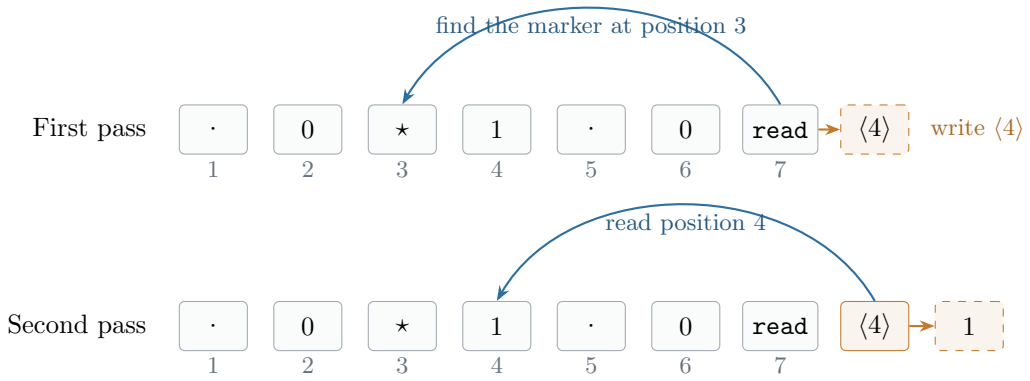


Figure 4: INDEX with bits $(0, 1, 0)$ and a request for the second bit. The first pass writes the position to read; the second uses that position to retrieve the answer. Orange boxes contain generated tokens.

How can the same layer perform both lookups? The current token selects the query. At `read`, the head looks for $\star$ and returns its position; the readout adds one and writes the corresponding name. At $\langle j \rangle$, it looks for position j and returns the bit there. These are the marker and position lookups we already know how to implement. The supplement gives the weights and precision needed to make both predictions exact.

The written address lets the next query depend on what the first pass found. A second layer can create the same dependence within one pass; generation creates it by running the same layer again. The one-token construction does require a vocabulary large enough to name positions. This is an encoding cost, and the supplement studies what happens when the vocabulary is small.

3.2 Chain Length and Computation

The address in INDEX tells the next pass what to read. In a longer computation, a generated token can also hold the state needed for the next update. The addition scratchpad does this with a carry. More generally, a rule of the form

$$s_t = \delta(s_{t-1}, x_t)$$

can be executed by reading the next input symbol, applying the same rule, and writing the new state. A fixed number of layers suffices; the number of generated states grows with the input. Explicit constructions implement every finite-state rule this way [6, 7, 9].

An algorithm may also need results from much earlier in its execution. The chain provides places to store them. If a memory location is updated several times, each update can be appended to the history. To recover its current value, attention reads the most recent write to that location, just as in the flip-flop task of Topic 3C. The entire memory need not fit into the latest token.

This is the idea behind Merrill and Sabharwal's general computation result [7]. A token records a machine's state, the symbol it writes, and its movement to the next memory location. Attention recovers the location and its most recent contents; the transition rule determines the next token. Their construction simulates a T-step Turing-machine computation with $O(T)$ generated tokens. The model's weights encode the rule, and its depth stays fixed.

Consequently, *polynomially many generated tokens allow a fixed-depth Transformer to carry out any polynomial-time decision algorithm.* Conversely, each forward pass can be simulated on an ordinary computer, so polynomially many passes on a polynomial-length sequence still take polynomial time. This gives an exact characterization in their model [2].[1]

What if the chain is short? There is a useful limit to this picture. With a fixed vocabulary of size V, a chain of length T has at most V^T possible token sequences. If $T = c \log n$, this is only polynomial in n:

$$V^{c \log n} = n^{c \log V}.$$

Imagine writing down all these candidate chains in advance. For each one, check whether the model would produce its proposed next token at every supplied prefix. All these checks can run in parallel, because the prefixes have already been specified. Under deterministic generation, exactly one complete chain is consistent with the model; retain the answer predicted after that chain. Thus a logarithmic chain can be replaced by polynomially much parallel work within a bounded number of computational stages [6, 16]. This replacement may use far more resources than the original model; it does not remove the benefit of one token to the one-layer INDEX construction. Long chains are interesting because this enumeration becomes prohibitively large, while executing the chain still follows one step at a time.

4 Comparing Ways to Add Computation

INDEX contains a dependence: the result of one lookup determines the query for another. Other computations have many parts that can be performed at the same time. To compare ways of adding computation, we should ask how many dependent rounds they provide, how much work can happen

[1] The model uses a fixed vocabulary, deterministic generation, averaging hard attention, masked pre-normalization, and logarithmic precision. The supplement states these assumptions and the corresponding results. One token is a whole forward pass, not one elementary machine operation; the simulation need not preserve running time in both directions.

within a round, and what information becomes available to the next round. Table 1 separates these questions.

Change	Where computation is added	What later computation can use
Generated token	Another pass at one new position	The selected token embedding, together with the earlier history
Extra layer	Another update at every position	The preceding layer's representations at all accessible positions
Pause positions	More positions within the existing layers	Hidden representations computed at those positions; their token identities stay fixed
Continuous thought	Another pass at one new position	A computed vector fed directly into the first layer, together with the earlier history

Table 1: Different additions to a Transformer's computation. A loop repeats a block of layers, sharing its weights. Both discrete and continuous generation retain earlier cached representations.

4.1 Depth and Parallel Computation

In INDEX, a second layer can use the address found by the first layer. A generated address token makes that address available to another pass of the same layer. Both create the dependence we need. Extra depth, however, updates representations at all positions, so many dependent lookups can advance together.

Parity illustrates the difference between a sequential algorithm and a parallel one. A chain can write the running parity after each input bit, taking n updates. Another algorithm combines adjacent pairs of bits, then pairs of these results, and so on. After $\lceil \log_2 n \rceil$ parallel rounds, one result contains the parity of the whole input. The same principle works for finite-state updates by composing transition tables for adjacent segments, as in Topic 3B [10]. This compares two algorithms; it does not prove that every chain needs n tokens. Parity also has the count-based solution from Topic 3C.

A *looped Transformer* repeats the same block of layers over all positions. It adds parallel rounds while sharing weights across rounds; the number of repetitions can be chosen at inference time. Merrill and Sabharwal implement finite-state computation with logarithmically many repetitions [16]. Their broader results separate logarithmic depth from logarithmically many generated tokens, under standard complexity assumptions. The supplement gives the precise models. One round over all positions can involve much more work than producing one token, so these round counts are not a comparison at equal computational cost.

What happens when we train these models? Merrill and Sabharwal study sequences of moves on five objects, with a label giving the cumulative arrangement after each move [16]. Figure 5 compares increasing the depth with increasing the width of an ordinary Transformer. Greater depth supports longer sequences in the tested range. This does not tell us whether the model learned the parallel composition algorithm, but it shows that training can make use of additional layers.

There is also evidence for learning with shared blocks. Xu and Sato train looped Transformers to evaluate arithmetic expressions modulo 3 [12]. Figure 6 shows how the number of loops affects the expression sizes that the models can handle.

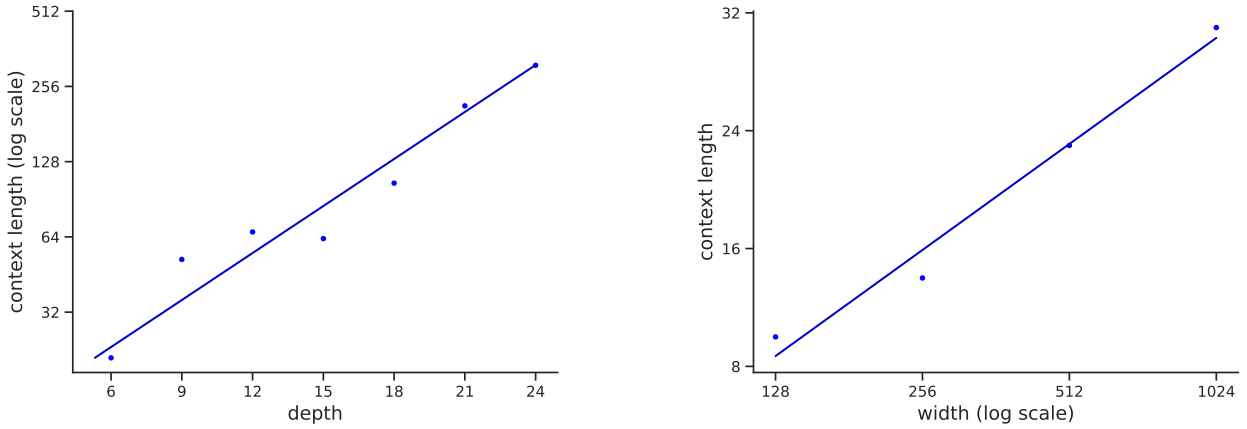


Figure 5: State tracking with trained Transformers, reproduced from Merrill and Sabharwal [16, Figure 2]. Inputs are even permutations of five objects, and training uses intermediate state labels and a sequence-length curriculum. "Context length" is the largest position with at least 95% validation accuracy, taking the best result across seeds. The left panel fixes width at 512; the right fixes depth at 6. These are ordinary Transformers with separate layers, rather than repeated applications of a shared block. Notice that the panels use different logarithmic axes.

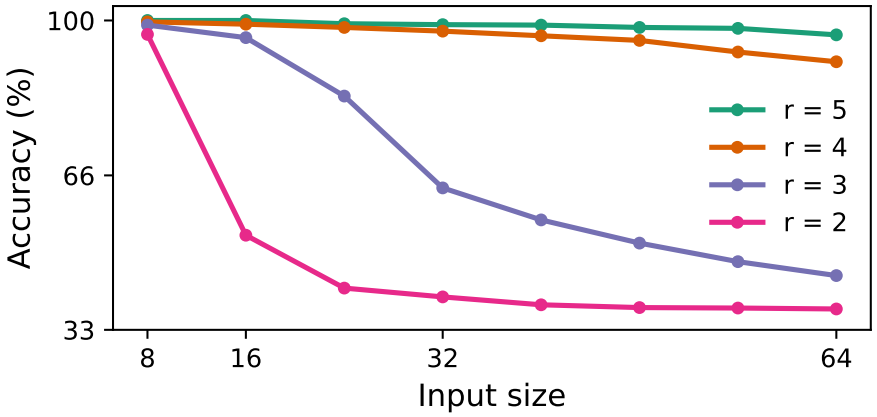


Figure 6: Arithmetic with trained looped Transformers, reproduced from Xu and Sato [12, Figure 6, top panel]. Each curve fixes the loop count r. Training uses a curriculum in expression size; more loops support larger expressions in the tested range. The horizontal axis counts expression-expansion operations, not serialized tokens. This plot concerns loops, not continuous thoughts, and does not establish generalization to lengths beyond the training distribution.

4.2 Pause Tokens and Workspace

A *pause token* is a fixed token inserted before the answer, giving the model additional positions to use [13]. These positions all contain the same visible token, but can have different hidden representations. Since the token identities are known in advance, they can be processed in one forward pass over the longer sequence. The number of layers has not changed.

Return to the count-based parity algorithm. Let $c = \sum_{i=1}^{n} x_i$. In Topic 3C, attention recovered the count and an MLP converted it to its parity. The MLP needed many bends to alternate between zero and one as the count increased. We can instead distribute simple tests across pause positions. Assign position j the threshold test

$$b_j = \mathbf{1}\{c \geq j\}, \qquad j = 1, \ldots, n.$$

Each position reads the same input count; its position specifies a different threshold. A later layer combines the results:

$$\sum_{j=1}^{n} (-1)^{j+1} b_j = \sum_{j=1}^{c} (-1)^{j+1} = c \bmod 2.$$

Exactly the first c tests succeed, so the alternating sum is zero for even c and one for odd c. This is the threshold computation used by London and Kanade [11, Section 5 and Appendix E]. Attention can supply the fraction of ones, each pause's MLP can compare it with its assigned threshold, and a later head can aggregate the signed indicators. The positional thresholds and numerical precision must distinguish the possible counts.

For input $1, 0, 1, 1$, for example, the visible tokens and hidden tests are

pause position	1	2	3	4	
visible token	`pause`	`pause`	`pause`	`pause`	$1 - 1 + 1 - 0 = 1.$
b_j	1	1	1	0	

The extra positions provide space for simultaneous computations that would otherwise have to be organized elsewhere in the model. They do not supply additional sequential updates of a running parity state.[2]

Can training find the threshold computation? London and Kanade train two-layer models with labels for the intermediate tests. For the pause model, each pause receives the label b_j defined above. Figure 7 shows that this model learns parity at lengths where the other tested models fail. With only final-answer supervision, however, all of their tested conditions remained at chance. The useful computation exists, but these experiments needed additional guidance to learn it.

The contrast with INDEX is instructive. In one layer, a pause position's query is still determined before attention has read the marker. No computed address is fed back into the first layer, so adding pauses alone does not remove that obstruction. Extra workspace helps when the existing layers can use it.

[2] This construction uses n pauses; it does not show that they are necessary for parity or that this number is minimal. London and Kanade's strict expressivity separation is a separate result in a more restricted precision model. The supplement discusses that result and its assumptions.

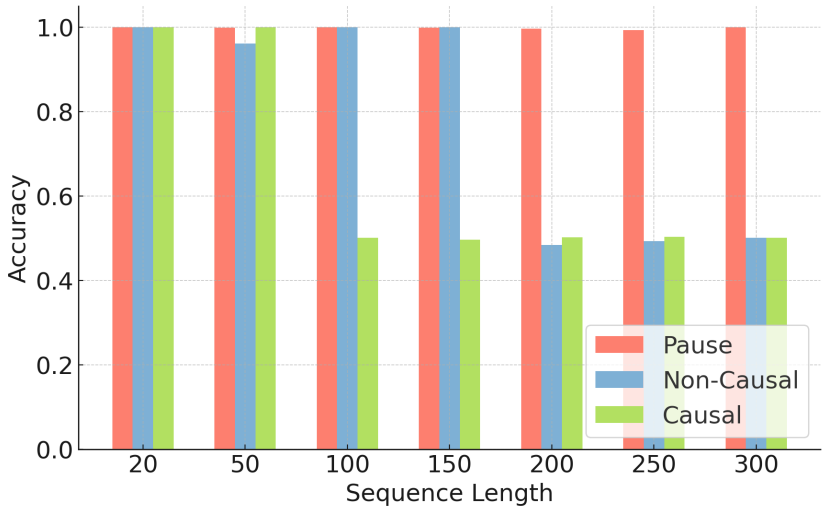


Figure 7: Learning parity with pauses, reproduced from London and Kanade [11, Figure 2]. All models have two layers, four heads, and width 32. All plotted conditions receive intermediate hints: threshold labels for the pause and noncausal models, and prefix-parity labels for the causal model. Models are trained separately at each length and tested on fresh examples of that length. Bars average three seeds. Thus the plot concerns learning at each length, rather than extrapolation to longer inputs.

4.3 Continuous Thoughts and the Feedback Vector

The discrete INDEX construction first computes an address, then chooses a vocabulary token naming it. The next pass embeds that token to recover the address. What if we pass the computed address directly? This is the change made by continuous-thought methods such as Coconut: the last hidden vector becomes the input at the next position [14].

Figure 8 shows this change in the generation process. The model still produces a sequence of positions; during a continuous step, it passes a vector to the next position without first choosing a vocabulary token.

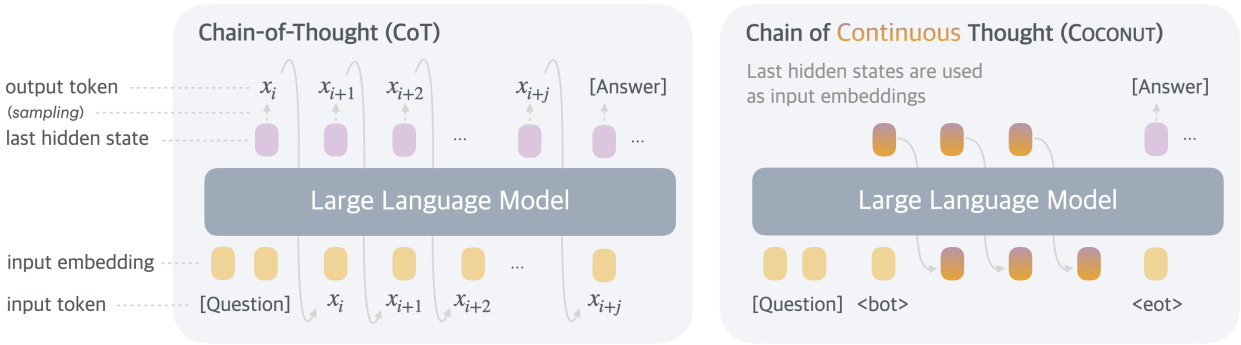


Figure 8: Discrete and continuous generation, reproduced from Hao et al. [14, Figure 1]. On the left, the final hidden state determines a token, which is embedded at the next position. On the right, that hidden state is used directly as the next input embedding. The model returns to token generation to produce its answer.

In our INDEX example, the marker is at position 3, so the address we need is 4. The two ways to pass that result are

$$\begin{aligned} \text{discrete:} \quad & \text{computed address} \longrightarrow \langle 4 \rangle \longrightarrow \text{embedding of } \langle 4 \rangle, \\ \text{continuous:} \quad & \text{computed address} \longrightarrow \text{vector containing the address } 4. \end{aligned}$$

In both cases, the next pass forms a query for position 4 and reads the bit there. For continuous feedback, we design the layer's output to contain the address features used by the next query. The supplement shows how to do this with the same one-attention-layer organization; we do not need a vocabulary token for every possible address.

This isolates the benefit of the richer feedback. With all the address names available, the discrete construction already uses just one intermediate token. In this one-layer setting, a fixed small vocabulary provides only a fixed number of next-pass queries. A vector can carry the numerical address, provided it has enough precision. Naming one of n positions still costs about $\log_2 n$ bits. More generally, a d-coordinate vector with p bits per coordinate can carry up to dp bits through this connection, compared with $\log_2 |\mathcal{V}|$ for one token. These are capacities of the feedback connection; discrete chains also retain earlier hidden representations in their cache.

The same idea extends beyond a single address. A vector can hold several intermediate results in separate coordinates. Xu and Sato construct models that store the partial results of a computation this way and update all results whose inputs are ready in each round [12]. The number of rounds then follows the longest chain of dependencies, while the required vector width and parameter count grow with the amount of computation represented. Zhu et al.'s reached-set construction is another instance: one vector holds many reached vertices of a graph and expands the whole set at once [15]. The supplement develops that example. These results explain how continuous feedback can preserve and update a structured intermediate result.

A continuous thought still appends one position. It differs from a loop, which revises the representations at all positions. A wide feedback vector can itself support parallel work, but that work costs width. Comparisons therefore have to account for the rounds, the available positions, and the size and precision of their vectors.

How are the continuous states learned? Coconut begins with supervised chains of thought. Its curriculum gradually replaces early written steps with continuous positions, while the model is still trained to predict the remaining steps and the answer (Figure 9) [14]. There is no target vector telling each continuous thought what coordinates to store; the training loss is backpropagated through those thoughts.

Zhu et al. examine what a trained continuous thought represents on a reachability task [15]. The input specifies graph edges, a root, and two candidate vertices, exactly one of which is reachable from the root. They train small models from scratch on problems requiring three or four hops, using a staged procedure based on supervised solution paths. Figure 10 shows both their test accuracy and an analysis of the second continuous thought. The thought has higher inner product with several reached vertices, especially those on the current frontier. This supports the idea that one vector can retain several possibilities, although it does not establish that the model implements the exact reached-set construction.

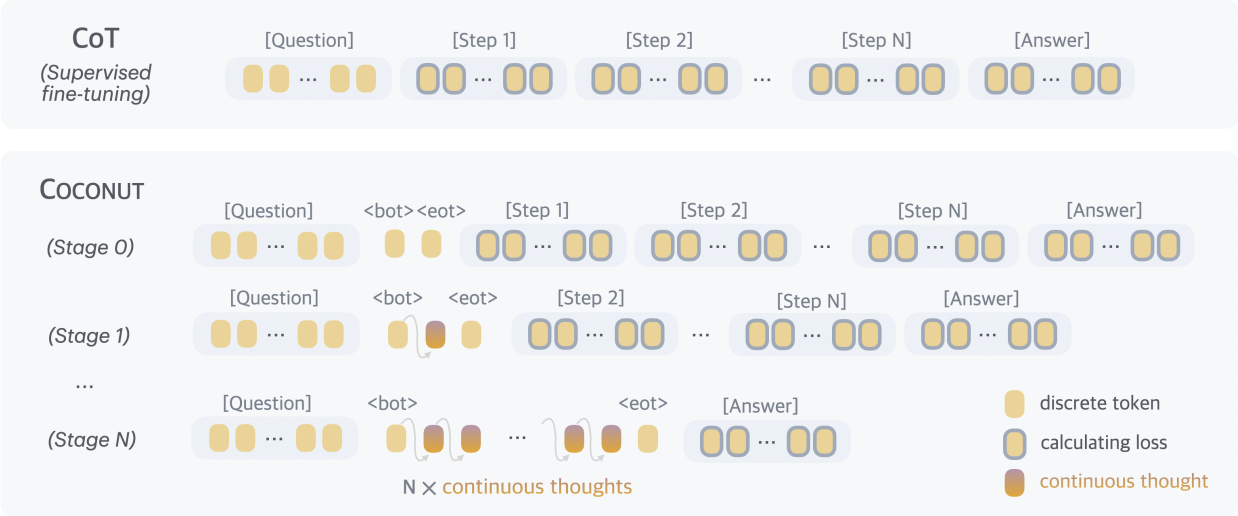
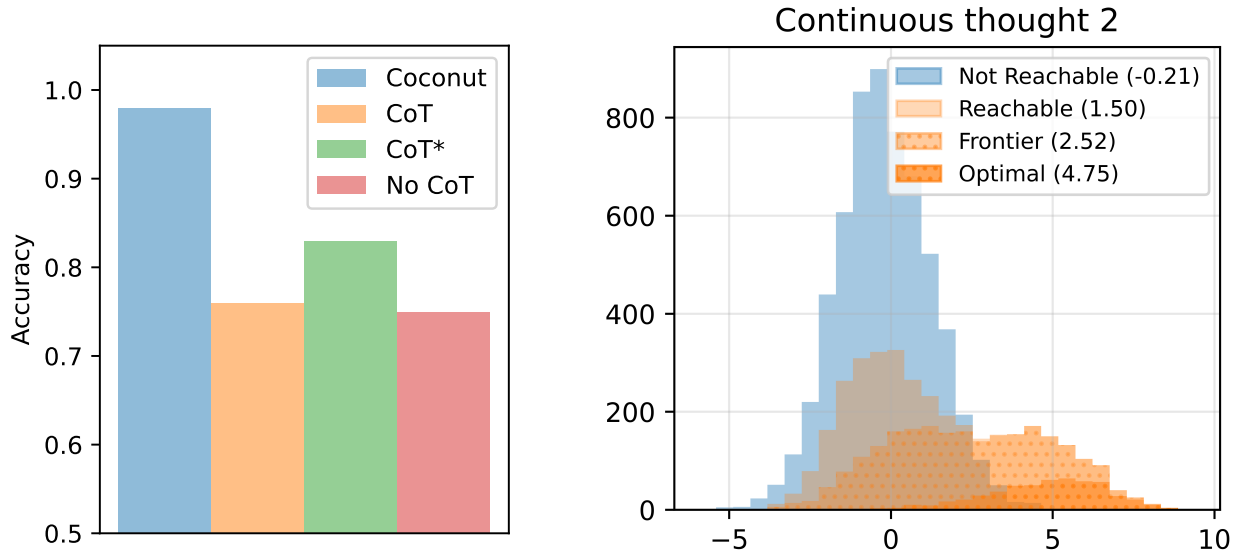


Figure 9: Coconut's training curriculum, reproduced from Hao et al. [14, Figure 2]. At first, the targets include all written steps. Successive stages replace more of those steps with continuous thoughts. The outlined tokens show where the prediction loss is applied. This procedure uses step supervision to learn a computation that will eventually run without writing those steps.



(a) Test accuracy

(b) Second continuous thought

Figure 10: Trained continuous thoughts, reproduced from Zhu et al. [15, Figures 4 and 6]. (a) Coconut, CoT, and No CoT have two layers; CoT* has twelve. The accuracy axis starts at 0.5, the chance level. Coconut uses staged training with path labels. (b) The horizontal axis is the inner product between the thought and a vertex embedding; the vertical axis counts vertices. Here "Not reachable" means not reached within two hops, not globally unreachable. The groups overlap: Optimal $\subseteq$ Frontier $\subseteq$ Reachable. Optimal denotes frontier vertices on the supervised path; legend numbers give group means. These similarities describe a weighted representation, not probabilities or an exact uniform indicator of the reached set.

5 Computation and Supervision

The constructions supply weights that implement a rule. How would a model learn that rule? Consider the state update $s_t = \delta(s_{t-1}, x_t)$. If a training example includes the intermediate states, it gives labels for the local predictions

$$(s_{t-1}, x_t) \longmapsto s_t.$$

For addition, these labels reveal which carry and digit follow each pair of digits and incoming carry. Without the intermediate states, the learner sees only the final answer. With them, it can fit the individual transitions that occur in the traces. At inference time, it must produce those states itself.

Intermediate states therefore have two roles: they give supervision during training and store results during generation. Li et al. compare these roles on the task of composing a sequence of permutations [6]. They train models to predict only the answer, to predict intermediate compositions at the input positions as additional training targets, or to generate those compositions before answering. The generated chains perform best, especially for shallow models. Both of the latter treatments receive intermediate labels; only the chain feeds the predicted results back for later computation.[3]

The next lecture separates two questions about supervision. Can intermediate labels reduce the number of independently sampled training inputs needed to learn a rule? Can they make finding a rule that fits the data computationally easier? We will compare training from answers with training from complete chains, while allowing both learners to generate their own intermediate steps at test time. This keeps the extra computation available at inference fixed and isolates what the training labels contribute.

Further reading. The supplement gives the precise models and constructions, the vocabulary-size bound for INDEX, and the general results relating chain length, depth, and parallel computation. It also discusses learning to use pause tokens and the limits of the empirical evidence.

References

[1] D. Hsu. *Chain-of-Thought.* COMS 6998-7 lecture slides, Columbia University, 2025. Lecture slides.

[2] W. Merrill. *A Theory of the Computational Power and Limitations of Language Modeling Architectures.* PhD dissertation, New York University, 2025. Sections 4.1–4.2. Dissertation.

[3] M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, C. Sutton, and A. Odena. *Show Your Work: Scratchpads for Intermediate Computation with Language Models.* arXiv:2112.00114, 2021. Full paper.

[4] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.* NeurIPS, 2022. Full paper.

[5] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. *Large Language Models are Zero-Shot Reasoners.* NeurIPS, 2022. Full paper.

[6] Z. Li, H. Liu, D. Zhou, and T. Ma. *Chain of Thought Empowers Transformers to Solve Inherently Serial Problems.* ICLR, 2024. Theorem numbers refer to arXiv version 4. Full paper.

[3] The paper evaluates direct prediction by accuracy per permutation entry, and chains by whether every next token is correct on the supplied prefixes. The latter implies correct greedy generation when the whole chain passes, but is not a direct measurement of final-answer accuracy after free generation.

[7] W. Merrill and A. Sabharwal. *The Expressive Power of Transformers with Chain of Thought.* ICLR, 2024. Full paper.

[8] C. Sanford, D. Hsu, and M. Telgarsky. *One-Layer Transformers Fail to Solve the Induction Heads Task.* arXiv:2408.14332, 2024. Full paper.

[9] V. Cabannes, C. Arnal, W. Bouaziz, A. Yang, F. Charton, and J. Kempe. *Iteration Head: A Mechanistic Study of Chain-of-Thought.* NeurIPS, 2024. Full paper.

[10] B. Liu, J. T. Ash, S. Goel, A. Krishnamurthy, and C. Zhang. *Transformers Learn Shortcuts to Automata.* ICLR, 2023. Full paper.

[11] C. London and V. Kanade. *Pause Tokens Strictly Increase the Expressivity of Constant-Depth Transformers.* NeurIPS, 2025. Full paper.

[12] K. Xu and I. Sato. *A Formal Comparison Between Chain of Thought and Latent Thought.* ICML, 2026. Full paper.

[13] S. Goyal, Z. Ji, A. S. Rawat, A. K. Menon, S. Kumar, and V. Nagarajan. *Think before you speak: Training Language Models With Pause Tokens.* ICLR, 2024. Full paper.

[14] S. Hao, S. Sukhbaatar, D. Su, X. Li, Z. Hu, J. Weston, and Y. Tian. *Training Large Language Models to Reason in a Continuous Latent Space.* COLM, 2025. Full paper.

[15] H. Zhu, S. Hao, Z. Hu, J. Jiao, S. Russell, and Y. Tian. *Reasoning by Superposition: A Theoretical Perspective on Chain of Continuous Thought.* NeurIPS, 2025. Full paper.

[16] W. Merrill and A. Sabharwal. *A Little Depth Goes a Long Way: The Expressive Power of Log-Depth Transformers.* NeurIPS, 2025. Full paper.