COMPUTATIONAL REASONING MACHINE

Anonymous authors
Paper under double-blind review

ABSTRACT

Contemporary work demonstrates that reasoning strategies can structure transformer inference, showing improved task-performance over pure streams of tokens. These frameworks enable *computational reasoning*: The reasoning processes feature causal dependency structures, procedural abstractions, and task concurrency. However, prior work implementing reasoning strategies requires ad-hoc programmatic wrappers, inflexible and blind to the problem at hand.

All of these concepts have analogies to computational primitives in computer systems (e.g. programming languages, operating systems, and computer architecture). We demonstrate that, instead of requiring a model to reason solely in natural language in a stream of tokens, we can augment the model's vocabulary with computational primitives: variables with scopes, functions, and deferred execution. Doing so enables a model to manage its context (reducing the number of tokens attended to), to abstract and reuse its own reasoning process (reducing the number of generated tokens), and to multitask (increasing the decoding throughput) natively during the course of its decoding process.

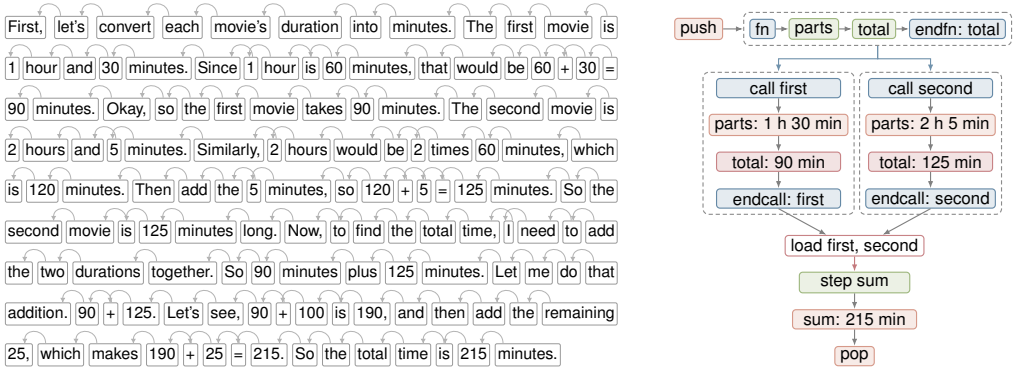
We augment the transformer architecture to produce what we term a *Computational Reasoning Machine* to elicit computational reasoning naturally in transformers. Specifically, we modify self-attention to reflect the machine's causal dependency structures, reuse procedural abstractions, reveal task concurrency opportunities, and reject invalid executions of the machine. Our results show a reduction in mean attended tokens per token (39.0-60.8%) and an increase in available decoding parallelism (1.1-1.28x), at the cost of increased KV cache usage (2.6-40%) and a 1.9-17.7pp drop in accuracy. RL narrows the accuracy drop to 1.2-6.7pp.

1 INTRODUCTION

Chain-of-Thought (Wei et al., 2022) has enabled transformers to solve a variety of challenge reasoning tasks in domains such as math, science, and language reasoning (Wei et al., 2022; Jaech et al., 2024; Chua & Evans, 2025), and recent work as shown scaling inference-time compute is effective at further increasing models' reasoning capabilities (Snell et al., 2025; Brown et al., 2024). However, self-attention (Vaswani et al., 2017) makes this *token-stream reasoning* expensive, since compute scales quadratically with context length.

To address the computational bottleneck, contemporary work demonstrates that reasoning strategies can structure token-stream reasoning to improve task-performance (Packer et al., 2023; Chen et al., 2023a; Gao et al., 2023b; Zhang et al., 2025; Snell et al., 2025; Wang et al., 2023; Yao et al., 2023; Venkatraman et al., 2025; Shinn et al., 2023; Khot et al., 2023; Prasad et al., 2024; Samanta et al., 2026; Bamberger et al., 2026; Ning et al., 2024; Zhuge et al., 2024; Qian et al., 2025; Wang et al., 2025b; Yang et al., 2025; Dang et al., 2025). Some reasoning strategies reduce context size through careful context management (Packer et al., 2023; Chen et al., 2023a; Gao et al., 2023b; Zhang et al., 2025). Some maintain information isolation by making independent inference passes to increase accuracy through voting or critique (Snell et al., 2025; Wang et al., 2023; Yao et al., 2023; Venkatraman et al., 2025; Shinn et al., 2023; Prasad et al., 2024) or to reduce latency (Ning et al., 2024). Others structure the reasoning with planning to specify ahead of time the steps to perform to improve accuracy (Prasad et al., 2024) or latency (Ning et al., 2024). Multi-agent frameworks utilize separate instances of models to manage information flow and enable parallel computation (Zhuge et al., 2024; Qian et al., 2025; Wang et al., 2025b; Yang et al., 2025; Dang et al., 2025).

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



(a) Token-stream reasoning

(b) Computational reasoning.

Figure 1: Token-stream reasoning versus the Computational Reasoning Machine: token-stream reasoning requires all tokens to attend to all previous tokens, encoding no reasoning structure. The Computational Reasoning Machine generates computational primitives that encode causal dependencies, specify and reuse procedural abstractions, and identify parallelism opportunities.

Computational Reasoning. These reasoning strategies enable *computational reasoning*: They encode causal dependency structures, implement procedural abstractions, and orchestrate task concurrency. Causal dependency structures enable reasoning steps to depend on a subset of prior reasoning steps; enabling information isolation and more efficient computation. Procedural abstractions specify ahead of time, a template of reasoning steps to perform and specialize them to the runtime inputs when executed; this enables forward-looking planning and reuse of the planning, reducing duplicate or wasted computation. Task concurrency reduces latency by identifying independent tasks to execute in parallel; this enables inference-time compute scaling without increasing latency. Computational reasoning structures token-stream reasoning to increase computational efficiency and throughput over standard autoregressive inference.

Prior approaches implement reasoning strategies with programs that orchestrate transformer inference. These implementations are specialized to execute specific reasoning algorithms. This limits how the model combines causal dependencies, procedural abstractions, and task concurrency during inference.

Computational Reasoning Machine. We demonstrate that we can augment the model's vocabulary with computational primitives that correspond to computational reasoning so the model can control its reasoning strategy. Specifically, we teach the model to reason with our computational reasoning language, CRM ASSEMBLY, with primitives enabling variables with scopes (causal dependency structures), functions (procedural abstraction), and deferred execution (task concurrency). The primitives construct a computational reasoning graph, as shown in Figure 1 encoding computational reasoning, enabling the model to manage dependencies, to abstract and reuse its own reasoning process, and to multitask natively during the course of its decoding process – the model programs its own computational reasoning strategies. We term this augmented transformer architecture a *Computational Reasoning Machine*.

We implement the Computational Reasoning Machine by modifying self-attention to reflect the machine's causal dependency structures, to reuse planning with procedural abstractions, and to reveal task concurrency opportunities. We perform supervised finetuning on a dataset of CRM ASSEMBLY traces annotated by GPT-6-sol (OpenAI, 2026) to teach the model how to use CRM ASSEMBLY primitives, then use preference optimize with BoNBon alignment (Gui et al., 2024) to teach the model how to use CRM ASSEMBLY primitives effectively. Our evaluation on reasoning benchmarks demonstrate that the Computational Reasoning Machine reduces the mean attended tokens per token by and increases the available decoding parallelism by at the cost of KV cache usage and accuracy, that we show is recoverable with RL.

Contributions. We present the following contributions in this work:

- We present the *Computational Reasoning Machine*, a reasoning system that natively programs its inference with computational reasoning: its reasoning processes encode causal dependency structures, use procedural abstractions, and offer task concurrency opportunities.

- We define CRM ASSEMBLY, a reasoning language for specifying computational reasoning, providing primitives for variables with scopes, functions, and deferred execution during reasoning.
- We implement the Computational Reasoning Machine in a transformer by modifying self-attention to reflect the machine's causal dependency structure, to reuse planning with procedural abstraction, and to reveal task concurrency opportunities. We train the model to program its inference on-the-fly with CRM ASSEMBLY.
- We evaluate the Computational Reasoning Machine on reasoning benchmarks and show that it reduces the mean attended tokens per token by 39.0-60.8% and increases available parallelism by 1.1-1.28x, compared to token-stream reasoning. These gains come at the cost of increased KV cache usage (2.6-40%), and a 1.9-17.7pp drop in accuracy. RL narrows the accuracy drop to 1.2-6.7pp.

Implications. The Computational Reasoning Machine reframes reasoning from a stream of natural language tokens to a . The machine controls its own inference, devising and writing its own reasoning strategies, enabling performance gains without hand-crafted reasoning algorithms. Our work shifts the focus from designing ad-hoc reasoning strategies to training machines to program their own inference effectively.

2 COMPUTATIONAL REASONING BY EXAMPLE

The Computational Reasoning Machine generates computational primitives as it reasons: variables with scopes encode causal dependency structures, functions provide procedural abstraction, and deferred execution enables task concurrency opportunities. This section presents an example token-stream reasoning trace, explains why it cannot encode computational reasoning, and shows how the Computational Reasoning Machine performs computational reasoning natively.

2.1 COMPUTATIONAL REASONING & THE LIMITATIONS OF TOKEN-STREAM REASONING

Standard causal self-attention exposes each generated token to the entire preceding token stream, even when the reasoning steps have fewer dependencies. Figure 2a presents a GSM8K math problem (Cobbe et al., 2021); Figure 2b shows a token-stream reasoning trace, and Figure 2c shows a computational reasoning graph for the same problem that encodes causal dependency structure, procedural abstraction, and task concurrency opportunities for this problem. Each node represents an action during the reasoning process, with edges identifying dependencies (and each dashed box representing dependencies to or from every node in the box) between actions. For example, loading `parts` depends on the value produced by the `parts` step.

Causal Dependency Structures. The steps for converting the first movie's length to minutes are independent of those for the second movie, while the sum depends on both converted lengths and only those lengths. Encoding this causal dependency structure during inference would reduce the number of tokens attended to by each generated token. However, the self-attention mechanism of token-stream reasoning forces all tokens to attend to all preceding tokens, violating the desired dependency structure. The computational reasoning graph encodes this structure explicitly: the two movie conversions have no edges between them, while the final total node descends from both.

Procedural Abstraction. In the trace, the model first emits tokens that plan how to compute the total minutes. This planning is abstract from the specifics of the first movie and the second movie. The model then rederives the steps to perform again. This is wasteful recomputation because token-stream reasoning cannot ensure adherence and reuse of planned steps. The computational reasoning graph defines a sequence of steps for identifying the parts of the movie's length then summing them; this is generic to any movie. The graph then applies the function first movie and the second movie with function calls to execute same template of steps.

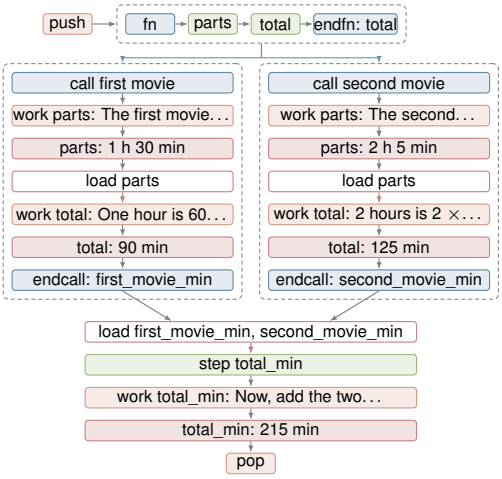
Task Concurrency. Because the two movie conversions do not depend on each other, they should take place concurrently to increase throughput. This parallelism relationship arises directly by encoding the causal dependency structure. Token-stream reasoning cannot automatically identify task concurrency opportunities because it captures only a fully dependent causal structure. Identifying conditionally independent nodes identifies concurrency opportunities: the conversions of the two movies are conditionally independent given the function and can take place concurrently.

```
Max plans to watch two movies this weekend. The
first movie is 1 hour and 30 minutes long while the
second movie is 2 hours and 5 minutes long. How
many minutes will it take Max to watch the two
movies?
```

(a) Example prompt.

```
I need to add the durations of both movies together
and convert everything into minutes. First, let's
convert each movie's duration into minutes. The
first movie is 1 hour and 30 minutes. Since 1 hour
is 60 minutes, that would be 60 + 30 = 90 minutes.
Okay, so the first movie takes 90 minutes. The
second movie is 2 hours and 5 minutes. 2 hours
would be 2 times 60 minutes, which is 120 minutes.
Then add the 5 minutes, so 120 + 5 = 125 minutes.
So the second movie is 125 minutes long. Now, to
find the total time, I need to add the two
durations together. So 90 minutes plus 125 minutes.
So the total time is 215 minutes.
```

(b) Token-stream reasoning trace.



(c) Computational reasoning graph.

```
1  <push/>
2    <fn id="movie_minutes" task="Convert a movie's time to minutes" params="movie"></fn>
3      <step id="parts" task="Identify the hours and minutes in the movie's time"></step>
4      <step id="total" task="Convert the hours to minutes and add the minutes"></step>
5    <endfn ret="total"></endfn>
6    <call fn="movie_minutes" args="movie:'first movie'" as="first_movie_min"></call>
7      <work id="parts"></work> The first movie is 1 hour and 30 minutes.
8      <ret value="1 hour and 30 minutes"></ret>
9      <load ids="parts:val"></load>
10     <work id="total"></work> One hour is 60 minutes, plus 30 minutes makes 90 minutes.
11     <ret value="The first movie is 90 minutes long."></ret>
12   <endcall/>
13   <call fn="movie_minutes" args="movie:'second movie'" as="second_movie_min"></call>
14     <work id="parts"></work> The second movie is 2 hours and 5 minutes.
15     <ret value="2 hours and 5 minutes"></ret>
16     <load ids="parts:val"></load>
17     <work id="total"></work> 2 hours is 2 x 60 = 120 minutes, plus 5 minutes gives 125 minutes.
18     <ret value="the second movie is 125 minutes long"></ret>
19   <endcall/>
20   <load ids="first_movie_min:val,second_movie_min:val"></load>
21   <step id="total_min" task="Add the two movie lengths"></step>
22   <work id="total_min"></work> Now, add the two times together: 90 minutes + 125 minutes.
23   90 + 125. 90 + 100 is 190, then add 25 more, which makes 215. So, total time is 215 minutes.
24   <ret value="Max will take 215 minutes to watch the two movies."></ret>
25 <pop id="total_min"></pop>
```

(d) Computational Reasoning Machine execution trace with CRM ASSEMBLY primitives.

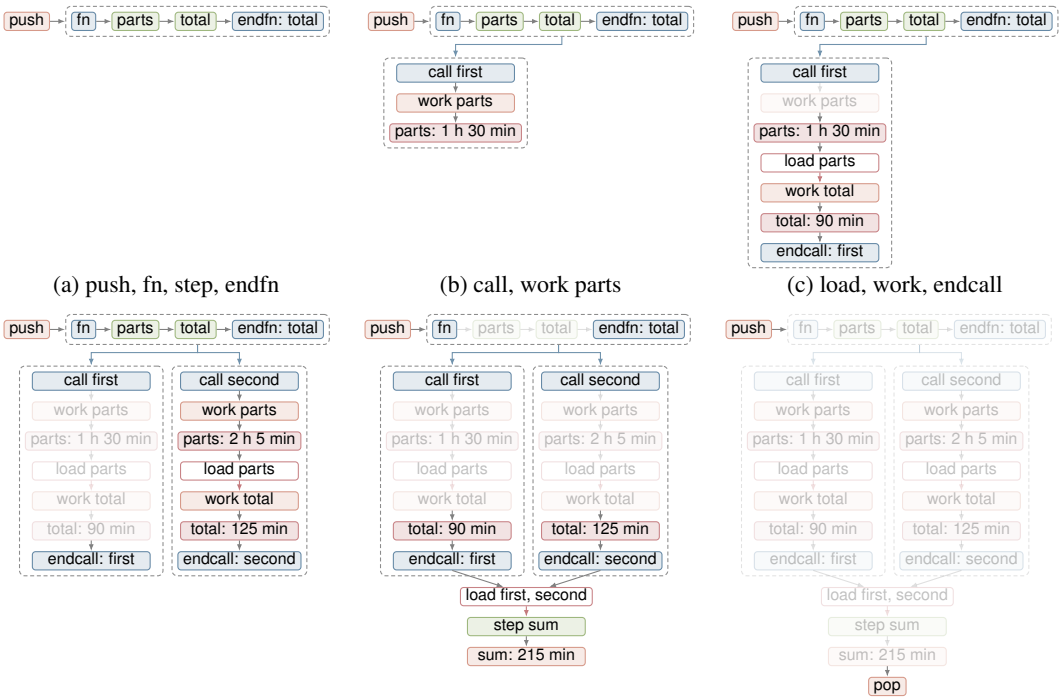
Figure 2: Token-stream and computational reasoning for the same GSM8K problem.

2.2 COMPUTATIONAL REASONING MACHINE

Figure 2d shows an example Computational Reasoning Machine trace that explicitly encodes computational reasoning using our reasoning language CRM ASSEMBLY. The machine generates CRM ASSEMBLY primitives then executes the generated primitives to construct the computational reasoning graph (Figure 2c) and a partial view over the graph to inform the next execution step, as shown in Figure 3. We implement the machine using a transformer by encoding CRM ASSEMBLY primitives as tokens and using attention masking to construct partial views.

The trace begins with `<push>`, which opens a scope for its variables (Line 1); each step in CRM ASSEMBLY are named and stored to variables. The machine then defines the `movie_min` function with parameter `movie` and declares two steps: `parts` extracts a movie's hours and minutes, and `total` converts them to minutes. The `<endfn>` primitive names `total` as the function's result (Lines 2 to 5).

The machine calls `movie_min` for the two movies, naming the results `first_movie_min` and `second_movie_min` (Lines 6 and 13). In the first call, `<work>` associates the scratchwork and `<ret>` value with the declared `parts` step (Line 7). The machine then loads that value when it works on `total` (Line 9). The second call follows the same steps for the other movie. Finally, the machine loads only the return values from both calls, not the scratchwork, and declares then works on `total_minutes`, before closing the scope with `<pop>`, exporting that variable to the outer scope (Lines 20 to 25).



(a) push, fn, step, endfn (b) call, work parts (c) load, work, endcall

(d) call, load, work, endcall (e) load, step, work (f) pop

Figure 3: Graph transitions of the computational reasoning graph encoded by the Computational Reasoning Machine trace in Figure 2d. Grayed nodes are hidden from view: a call body is hidden after `endcall`, the function defined steps are visible only during a call, `load` reveals referenced values, and `pop` hides the scope interior. The Computational Reasoning Machine samples the next primitive conditioned on the current graph view.

Causal Dependency Structure. The first call creates a `parts` value, then loads it into the view for `total` while hiding the `parts` scratchwork (Figures 3b and 3c). The second call adds a separate branch while the view hides the first call's body (Figure 3d). The final load brings only the two call results into view for the sum, and `<pop>` hides the scope interior (Figures 3e and 3f).

Procedural Abstraction. The machine declares the `parts` and `total` steps as computation to perform once in the function plan (Figure 3a). Both calls keep these declarations in view while adding their own work and values (Figures 3b to 3d). The second call reuses the plan (Figure 3d).

Task Concurrency. The second call's view includes the shared function plan but not the first call's body, and neither branch depends on the other (Figure 3d). It is only the the final load brings both call results into view for the sum (Figure 3e). This dependency structure identifies the calls as a parallelism opportunity that synchronizes at the load.

Performance. We implement the Computational Reasoning Machine by adding CRM ASSEMBLY primitives to a base transformer model (Qwen3-4B-Instruct-2507 (Qwen Team, 2025)) as special tokens and using attention masks to selectively construct views according to the primitives. See Appendix B for a diagram of the example trace attention mask. We train the model on on CRM ASSEMBLY traces annotated by GPT-6-Sol (OpenAI, 2026).

Figure 4: Performance on example problem.

	CoT	CRM (Ours)	Δ
Attended Tokens/Out Tokens	515.0	**280.8**	−45.5%
Theoretical Parallelism	1.00	**1.43**	+43.3%
Resident KV Tokens/Out Token	**515.0**	740.1	+43.7%

Figure 4 compares the machine with a token-stream reasoning baseline trained by the same method on the example problem using $k = 3$ samples. Theoretical parallelism is the ratio of sequential decoding steps to the minimum steps permitted by the dependency graph (Jin et al., 2025). The machine reduces mean attended tokens per output token by 45.5% and enables $1.43\times$ theoretical parallelism. Mean resident KV tokens per output token increase by 43.7% because distinct visible prefixes require separate cache entries.

3 COMPUTATIONAL REASONING MACHINE

The Computational Reasoning Machine models a distribution over evolving computation graphs and a view over the graph. It uses CRM ASSEMBLY, a language with computational primitives that encode causal dependency structures through variables with scopes, define and reuse procedural abstractions using functions, and identify task concurrency opportunities through deferred execution. We implement the using a transformer, by augmenting its vocabulary with the primitives and modifying the attention masks to reflect the graph view.

3.1 EXECUTION MODEL

The Computational Reasoning Machine models the computation graph $G_t = (V_t, E_t)$ at execution step t, where each node represents a *computation event*. Let $\mathcal{V}$ be the set of possible computation events, each a distinct action in deriving an answer (posing a subproblem, working it out, stating or using a result, beginning or completing a line of reasoning, or defining or applying a reusable plan) so $V_t \subseteq \mathcal{V}$ and $v_t \in \mathcal{V} \setminus V_{t-1}$ is the event added at step t. An edge $u \to v$ indicates that event v depends on event u. The machine maintains the full graph G_t and a view $\widetilde{G}_t \subseteq G_t$ containing the events and dependencies active at step t. It updates both states as $(G_t, \widetilde{G}_t) = \mathcal{U}(G_{t-1}, \widetilde{G}_{t-1}, v_t)$. The Computational Reasoning Machine models the joint distribution over these states as follows:

$$p_{\mathrm{CRM}}(G_{1:T}, \widetilde{G}_{1:T} \mid G_0, \widetilde{G}_0) = \prod_{t=1}^{T} \Big[p_{\mathrm{CRM}}(v_t \mid G_{t-1}, \widetilde{G}_{t-1})\, \mathbb{1}\{(G_t, \widetilde{G}_t) = \mathcal{U}(G_{t-1}, \widetilde{G}_{t-1}, v_t)\} \Big].$$

Generatively, at each execution step the machine samples an event and updates the graph and its view:

$$v_t \sim p_{\mathrm{CRM}}(\cdot \mid G_{t-1}, \widetilde{G}_{t-1}), \quad (G_t, \widetilde{G}_t) = \mathcal{U}(G_{t-1}, \widetilde{G}_{t-1}, v_t).$$

The machine encodes computation events with primitives whose operational semantics define updates to the graph and its view. We next define CRM ASSEMBLY, our computational reasoning language that defines these primitives and semantics.

3.2 THE CRM ASSEMBLY LANGUAGE

CRM ASSEMBLY is an imperative language, with primitives for variables with scopes, functions, and deferred execution. A program encodes a computation graph, and executing the program constructs the graph and a view over the graph. Every primitive adds a new node to the graph, and specifies a modification to the graph and the view. Figure 5 shows the abstract syntax. Appendix A gives the formal operational semantics.

Names	$n, f \in N$	Values v
Types	$d \in \{\mathsf{val}, \mathsf{prov}\}$	
Steps	$D ::= \mathsf{step}\ n(v) \mid D; D \mid \epsilon$	
Call body	$C ::= \mathsf{load}\ n(d) \mid \mathsf{work}\ n(v)$	
	$\mid \mathsf{value}\ n(v) \mid C; C \mid \epsilon$	
Primitives	$P ::= \mathsf{step}\ n(v) \mid \mathsf{load}\ n(d)$	
	$\mid \mathsf{work}\ n(v) \mid \mathsf{value}\ n(v)$	
	$\mid \mathsf{push} \mid \mathsf{pop}\ n$	
	$\mid \mathsf{fn}\ f(v, n, D, n)$	
	$\mid \mathsf{call}\ f(n_a, n, C)$	
	$\mid P; P \mid \epsilon$	

Figure 5: Abstract syntax of CRM ASSEMBLY.

Variables with Scopes. A step $n(v)$ primitive adds a *step* node with name n and task description v into the graph and the view. The work $n(v)$ primitive adds a *work* node with name n scratchwork v to the graph and the view with an edge from the *step* node with name n to it. The value $n(v)$ primitive adds a *value* node with name n result v and an edge from the work node with name n, and it hides the work and value nodes from the view. The push primitive opens a new scope. It adds a *push* node into the graph and view, and all subsequently added nodes have an edge from the push node. The pop n primitive closes the current scope by removing all child nodes of the most recent push node from the view; it also stores the name n in a *pop* node. A load $n(d)$ primitive adds a *load* node to the graph and view. With $d = \mathsf{val}$, it adds an edge from the value node named n to all subsequent nodes until the end of the current scope. With $d = \mathsf{prov}$, it adds an edge from the work node named n and from all the parent edges of that node to all subsequent nodes until the end of the current scope.

Functions. A fn $f(v, n_p, D, n_r)$ primitive adds a *function* node with name f, task description v, parameter n_p, and return step name n_r into the graph and the view; it also stores the current view into the function node. Its step declarations D add step nodes into the graph with an edge from the function node to each, but not into the view.

A call $f(n_a, n, C)$ primitive adds a *call* node to the graph, storing the current view, with edges from the function node and from the value node with name n_a, then executes the body C, a sequence of load, work, and value primitives. Taking the stored view in function node f, it adds the function step nodes and the call node into view, then aliases the parameter n_p to the work and value nodes with name n_a. Every work node in the call body has an edge from the call node. When the call ends, the call node gains an edge from the work and value node with name n_r, the machine restores the stored view of the call node, and n aliases the value node with name n_r.

Deferred Execution. A step $n(v)$ primitive adds only a step node to the graph, and the matching work node enters the graph when a later work $n(v)$ primitive executes. Other primitives may add nodes in between, so a program can declare several steps up front and work on them in any order. Because work, value, and call body nodes stay outside the view of later primitives until a load h primitive names them, a work or call primitive can execute concurrently with the primitives that follow it, and the two synchronize at the load h or pop n primitive that references their names.

3.3 Implementation

Thus far, we have described the mechanics of the CRM ASSEMBLY language using abstract syntax to discuss the computational model that underlies the Computational Reasoning Machine. We next describe we implement the machine with a transformer.

Concrete Syntax & Encoding. The abstract syntax used in Figure 5 represents primitives as abstract program constructs. The implementation realizes the constructs using a concrete syntax and encodes them as special tokens in the transformer vocabulary. For example, a `step eg_step(example task)` primitive serializes into the concrete syntax `<step id="eg_step" task="example task"></step>`, where underlines show distinct special tokens. Appendix A.2 shows the full encoding list.

Attention Mask. The machine forms each next-token attention mask from the current graph view, exposing prefix tokens that encode visible computation and masking those that encode hidden events. While the machine encodes an event, it also keeps that event's already decoded tokens visible. The transformer samples the next token under this mask; when the event is complete, the machine updates the graph and view before forming the next mask.

KV Cache. As decoding proceeds, changes to the graph view can change which earlier tokens a token in the prefix attends to. For example, x_i may attend to an earlier token x_j in one decoding iteration but not the next. Its cached KV entries were computed under the previous attention pattern and must therefore be recomputed. To avoid recomputing the entire prefix, the machine stores KV entries in a trie and reuses them along paths where both the tokens and their attention patterns match. When a path diverges, the machine computes new entries only where needed.

Training. We construct a CRM ASSEMBLY SFT dataset by annotating CoT reasoning traces from the Nemotron Cascade SFT dataset (Wang et al., 2025a) with CRM ASSEMBLY primitives, using GPT-6-Sol (OpenAI, 2026) as the annotator. We finetune a base transformer on this dataset under the same attention mask that the CRM ASSEMBLY primitives determine at inference time to produce the `CRM` model, then apply BoNBoN alignment (Gui et al., 2024) with LLM-as-a-Judge (Zheng et al., 2023) rankings to produce the `CRM-RL` model. Appendix D describes the dataset construction, the training row layout that enforces the attention mask, and the preference optimization procedure.

4 Evaluation

The Computational Reasoning Machine uses CRM ASSEMBLY primitives to program its own inference to encode causal dependency structures (to reduce tokens to attend to), define and use procedural abstractions (to generate less tokens), and identify task concurrency (to increase decoding throughput). We evaluate whether Computational Reasoning Machine reduces compute cost and identifies parallelism compared to token-stream reasoning on reasoning tasks.

Baselines. We compare against the baseline `CoT`, the model finetuned from the starter model using the original reasoning traces from the seed dataset for each problem in the `CRM` training set. As reference points, we also compare against the starter model, the off-the-shelf reasoning model of the starter model family, and `CoT-RL`, the `CoT` model optimized using the same RL methodology.

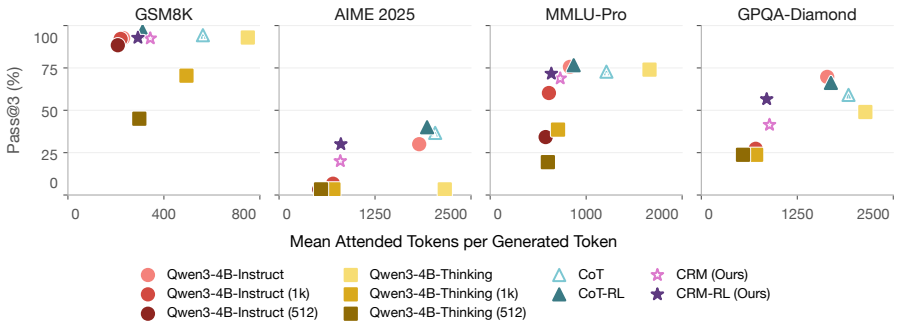


Figure 6: Pass@3 accuracy vs. mean attended tokens per token.

Experimental Setup. We use `Qwen3-4B-Instruct-2507` and `Qwen3-4B-Thinking-2507` (Qwen Team, 2025) as the starter and reference reasoning model, respectively. We finetune and preference optimize with Tunix (Bao et al., 2025) on TPU v5p. Appendix D.4 lists the hyperparameter settings.

Methodology. We evaluate against reasoning benchmarks: GSM8K (Cobbe et al., 2021), AIME 2025 (Zhang & Math-AI, 2025), MMLU-PRO (Wang et al., 2024), and GPQA-Diamond (Rein et al., 2024). Unless specified otherwise, we evaluate all methods using 4096 context length. When scoring for correctness, we strip special tokens, leaving each step's work and value but not its task description. Errored traces and truncated traces are scored as is. Figure 6 shows the pass@3 accuracy to the mean attended tokens per generated tokens; see Appendix E for additional evaluation metrics.

Reduction in attended tokens. `CRM` reduces the number of mean attended tokens per generated tokens across all benchmarks, with 39.0-60.8% decrease from `CoT`. In GSM8K, where responses are shorter, `CRM` is comparable to the `Qwen3-4B-Instruct` baseline, which does not perform reasoning. In benchmarks where longer reasoning is required, such as AIME 2025 and GPQA-Diamond, `CRM` exhibits significant reduction in the number of attended tokens against all baselines. The decrease in mean attended tokens is not fully due to shorter generation lengths (4.8-25.8% decrease from `CoT`); `CRM` and `CRM-RL` use less mean attended tokens than their mean output tokens generated.

Decrease in accuracy. The efficiency gains of `CRM` come at the cost af accuracy. `CRM` produces accuracy drops of 1.9-17.7pp from `CoT`. Some decrease in accuracy is due to `CRM` aborting execution at syntax or runtime errors, leaving truncated traces. See Appendix E.3 for a break down of failures by each model. When evaluated with shorter context lengths at 512 and 1024 tokens, the `Qwen3-4B-Instruct` and `Qwen3-4B-Thinking` baselines match `CRM` in mean attended tokens, but have 0.3-49.4pp lower accuracy than `CRM`. Preference optimization with an accuracy objective recovers the reduced accuracy, as shown by `CRM-RL` having a lessened accuracy drop of 1.2-6.7pp from `CoT`. Appendix E.4 reports how pass@3 accuracy changes across those RL training rounds. `Qwen3-4B-Thinking` does not perform well on AIME 2025 due to truncation.

Increase in memory cost. `CRM` trades reduction in attention cost for increase in memory cost, with 2.6-40% increase in mean resident KV cache tokens per output tokens from `CoT`. `CRM` requires multiple KV-cache entries for the tokens with different prefixes due to attention masking, so it stores more tokens in the KV-cache in benchmarks with greater reduction in attended tokens.

Parallelism Opportunities. Following the methodology from Jin et al. (2025), we compute the mean theoretical parallelism. All the baselines implement standard autoregressive decoding, so there exists no parallelism for those models. `CRM` enables 1.1-1.28x mean theoretical parallelism.

5 Related Work

Reasoning Languages. Chain-of-Thought expresses reasoning in natural language (Wei et al., 2022; Guo et al., 2025; Jaech et al., 2024). Other approaches use latent representations (Ramji et al., 2026), describe internal features and activations (Li et al., 2025; Binder et al., 2024; Plunkett et al., 2025), or teach abstract reasoning templates (Gao et al., 2026; Hong et al., 2024), but do not enable computational reasoning. Code-based reasoning (Gao et al., 2023a; Chen et al., 2023b) could in theory elicit computational reasoning through Turing-complete languages, but these methods do not encourage it. Our language CRM ASSEMBLY explicitly encodes computational reasoning.

External Reasoning Orchestrators. Handcrafted algorithms encode dependencies for accuracy (Snell et al., 2025; Wang et al., 2023; Yao et al., 2023; Venkatraman et al., 2025; Shinn et al., 2023; Khot et al., 2023; Samanta et al., 2026; Bamberger et al., 2026; Prasad et al., 2024) or identify parallelism opportunities (Ning et al., 2024). Some use tools or code interpreters to access information outside of the model's context (Packer et al., 2023; Chen et al., 2023a; Gao et al., 2023b; Zhang et al., 2025); others divide or summarize reasoning across model calls (Lee & Kim, 2023; Sun et al., 2025; Wu et al., 2025; 2026). Agent swarms use separate models to encode causal dependencies and enable task concurrency (Zhuge et al., 2024; Qian et al., 2025; Wang et al., 2025b; Yang et al., 2025; Dang et al., 2025). The Computational Reasoning Machine generalizes these capabilities, performing computational reasoning without handcrafted reasoning algorithms or heuristics.

Self-Programmed Inference. Kontonis et al. (2026) train models to emit a token that masks all reasoning generated since the previous such token, while Jin et al. (2024) train models to select attention spans from handcrafted, task-specific partitions. DA uses model-generated primitives to select one, all, or none of the syntactically partitioned prompt (Ho et al., 2026), but not reasoning within the response. The Computational Reasoning Machine uses richer task-agnostic primitives to group and select its own reasoning semantically without handcrafted partitions. Learned parallel decoding methods enable models to decide when to parallelize with explicit join and fork primitives (Jin et al., 2025; Schroeder et al., 2025; Chen et al., 2025; Pan et al., 2025). Our work identifies parallelism opportunities without explicit concurrency primitives.

Computer-Like Neural Computation. Prior work incorporates computer-system abstractions into neural architectures for algorithmic computation. Prior work couples neural controllers to differentiable read-write memory (Graves et al., 2014; 2016; Kurach et al., 2016; Le et al., 2020) or adds reusable subroutines or compositional operations (Reed & de Freitas, 2016; Neelakantan et al., 2016; Macfarlane et al., 2026). Another approach constructs transformer weights analytically to implement a WebAssembly interpreter (Percepta Research Team, 2026). We integrate computational primitives into the transformer's execution model to enable computational reasoning.

6 Discussion & Conclusion

The Computational Reasoning Machine programs its own inference with CRM ASSEMBLY, reducing attended tokens per generated token and exposing opportunities for parallel decoding compared to token-stream reasoning. Beyond a transformer model, the Computational Reasoning Machine is also a computing system: it interprets and executes programs. With this perspective, both machine learning and computer system design spaces apply to Computational Reasoning Machine.

Training and Inference Design. Future training objectives could teach the Computational Reasoning Machine to use CRM ASSEMBLY primitives more effectively by accounting for runtime, KV-cache usage, and parallelism alongside accuracy (Jin et al., 2025). Constrained decoding could enforce program syntax and prevent some execution failures (Ugare et al., 2024; Reddy et al., 2026).

Programming System Design. CRM ASSEMBLY's imperative semantics synergizes with autoregressive generation, but the model must track open scopes, unfinished steps, and loaded values. A functional semantics could make computations more compositional and declarative, but its evaluation order might diverge from autoregressive generation order. Future designs could add primitives that are complementary but orthogonal to computational reasoning, such as arithmetic operations, control flow, or explicit KV-cache management.

Limitations. We trained and evaluated a 4B-parameter transformer with a 4096-token context; larger models or context sizes may change the nature of the result. Training the Computational Reasoning Machine requires new data, but our data construction procedure is inexpensive in practice (about $0.14/trace using GPT-6-Sol). Our analysis identifies independent decoding steps and estimates an upper bound on possible speedup; we do not decode those steps concurrently. Realizing that speedup requires efficient batching and KV-cache management across dynamic branches.

Conclusion. The Computational Reasoning Machine programs its inference using variables with scopes, functions, and deferred execution to perform computational reasoning, enabling task-performance improvements over token-stream reasoning. Our work reframes inference from a stream of natural-language tokens into a computational substrate for reasoning. This direction calls for training models to program their own reasoning more effectively and scalably.

AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic training data, to implement methods, and to provide feedback on our theoretical framework. Specifically, the CRM ASSEMBLY SFT dataset was produced by using GPT-6-Sol (OpenAI, 2026) to annotate existing reasoning traces with CRM ASSEMBLY primitives; this use is part of the research method and is described in the paper. AI coding assistants were used to help implement the Computational Reasoning Machine and our training and evaluation code. AI tools gave feedback on the design of CRM ASSEMBLY and its semantics, but the formalism was developed by the authors. We have not used generative AI tools to formulate mathematical claims or proofs, propose or refine hypotheses, design experiments or methodology, or interpret results, and translation and qualitative or thematic data analysis are not applicable to this work. Additionally, we used generative AI tools to help create figures, create or edit software code, summarize or analyze existing literature, edit parts of the paper to improve readability, and identify relevant literature.

We have reviewed AI-assisted work. Generated traces were automatically filtered for validity (syntax, in-scope variable references, missing arguments, and empty block scopes) before training. Critical parts of the AI-assisted code were reviewed by 2 authors. All figures were checked by the authors against the definitions and results in the paper. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Zachary Bamberger, Till R Saenger, Gilad Morad, Ofra Amir, Brandon M Stewart, and Amir Feder. State-of-thoughts: Structured action templates for tree-of-thoughts. In *Conference on Language Modeling*, 2026.

Tianshu Bao, Jeff Carpenter, Lin Chai, Haoyu Gao, Yangmu Jiang, Shadi Noghabi, Abheesht Sharma, Sizhi Tan, Lance Wang, Ann Yan, Weiren Yu, et al. Tunix (tune-in-jax), 2025.

Felix J Binder, James Chua, Tomek Korbak, Henry Sleight, John Hughes, Robert Long, Ethan Perez, Miles Turpin, and Owain Evans. Looking inward: Language models can learn about themselves by introspection. *arXiv preprint arXiv:2410.13787*, 2024.

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

Howard Chen, Ramakanth Pasunuru, Jason Weston, and Asli Celikyilmaz. Walking down the memory maze: Beyond context limit through interactive reading. *arXiv Preprint arXiv:2310.05029*, 2023a.

Keyu Chen, Zhifeng Shen, Daohai Yu, Haoqian Wu, Wei Wen, Jianfeng He, Ruizhi Qiao, and Xing Sun. Aspd: Unlocking adaptive serial-parallel decoding by exploring intrinsic parallelism in llms. *arXiv preprint arXiv:2508.08895*, 2025.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023b.

James Chua and Owain Evans. Are deepseek r1 and other reasoning models more faithful? *arXiv preprint arXiv:2501.08156*, 2025.

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

Yufan Dang, Chen Qian, Xueheng Luo, Jingru Fan, Zihao Xie, Ruijie Shi, Weize Chen, Cheng Yang, Xiaoyin Che, Ye Tian, et al. Multi-agent collaboration via evolving orchestration. *Advances in neural information processing systems*, 2025.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, 2023a.

Silin Gao, Antoine Bosselut, Samy Bengio, and Emmanuel Abbe. Abstral: Augmenting llms' reasoning by reinforcing abstract thinking. In *International Conference on Learning Representations*, 2026.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023b.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626), 2016.

Lin Gui, Cristina Gârbacea, and Victor Veitch. Bonbon alignment for large language models and the sweetness of best-of-n sampling. In *Advances in Neural Information Processing Systems*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081), 2025.

Namgyu Ho, Huzama Ahmad, Woosung Koh, Se-Young Yun, Tal Schuster, and Cicero Nogueira dos Santos. Language models can control their own attention. *arXiv preprint arXiv:2609.02737*, 2026.

Ruixin Hong, Hongming Zhang, Xiaoman Pan, Dong Yu, and Changshui Zhang. Abstraction-of-thought makes language models better reasoners. In *Conference on Empirical Methods in Natural Language Processing*, 2024.

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

Tian Jin, Zifei Xu, Sayeh Sharify, Xin Wang, et al. Self-selected attention span for accelerating large language model inference. *arXiv preprint arXiv:2404.09336*, 2024.

Tian Jin, Ellie Y Cheng, Zack Ankner, Nikunj Saunshi, Blake M Elias, Amir Yazdanbakhsh, Jonathan Ragan-Kelley, Suvinay Subramanian, and Michael Carbin. Learning to keep a promise: Scaling language model decoding parallelism with learned asynchronous decoding. In *International Conference on Machine Learning*, 2025.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In *International Conference on Learning Representations*, 2023.

Vasilis Kontonis, Yuchen Zeng, Shivam Garg, Lingjiao Chen, Hao Tang, Ziyan Wang, Ahmed Hassan Awadallah, Eric Horvitz, John Langford, and Dimitris Papailiopoulos. Memento: Teaching llms to manage their context. In *Conference on Language Modeling*, 2026.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *International Conference on Learning Representations*, 2016.

Hung Le, Truyen Tran, and Svetha Venkatesh. Neural stored-program memory. In *International Conference on Learning Representations*, 2020.

Soochan Lee and Gunhee Kim. Recursion of thought: A divide-and-conquer approach to multi-context reasoning with language models. In *Association for Computational Linguistics*, 2023.

Belinda Z Li, Zifan Carl Guo, Vincent Huang, Jacob Steinhardt, and Jacob Andreas. Training language models to explain their own computations. *arXiv preprint arXiv:2511.08579*, 2025.

Matthew Macfarlane, Clem Bonnet, Herke van Hoof, and Levi Lelis. Gradient-based program synthesis with neurally interpreted languages. In *International Conference on Learning Representations*, 2026.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *International Conference on Learning Representations*, 2016.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Prompting llms for efficient parallel generation. In *International Conference on Learning Representations*, 2024.

OpenAI. Introducing GPT-6 Sol and Luna, 2026. URL https://openai.com/index/introducing-gpt-6-sol-and-luna/.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Jiayi Pan, Xiuyu Li, Long Lian, Charlie Snell, Yifei Zhou, Adam Yala, Trevor Darrell, Kurt Keutzer, and Alane Suhr. Learning adaptive parallel reasoning with language models. In *Conference on Language Modeling*, 2025.

Percepta Research Team. Constructing an LLM-Computer. Percepta Blog, 2026. URL https://www.percepta.ai/blog/constructing-llm-computer.

Dillon Plunkett, Adam Morris, Keerthi Reddy, and Jorge Morales. Self-interpretability: Llms can describe complex internal processes that drive their decisions. *arXiv preprint arXiv:2505.17120*, 2025.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. Adapt: As-needed decomposition and planning with language models. In *Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics*, 2024.

Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. Scaling large language model-based multi-agent collaboration. In *International Conference on Learning Representations*, 2025.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Keshav Ramji, Tahira Naseem, and Ramón Fernandez Astudillo. Thinking without words: Efficient latent reasoning with abstract chain-of-thought. *arXiv preprint arXiv:2604.22709*, 2026.

Avinash Reddy, Thayne T Walker, James S Ide, and Amrit Singh Bedi. The hidden cost of structured generation in llms: Draft-conditioned constrained decoding. In *International Conference on Machine Learning*, 2026.

Scott Reed and Nando de Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations*, 2016.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. In *Conference on Language Modeling*, 2024.

Ankur Samanta, Akshayaa Magesh, Ayush Jain, Kavosh Asadi, Youliang Yu, Daniel Jiang, Boris Vidolov, Kaveh Hassani, Paul Sajda, Jalaj Bhandari, et al. Structure enables effective self-localization of errors in llms. *arXiv preprint arXiv:2602.02416*, 2026.

Philip Schroeder, Nathaniel W Morgan, Hongyin Luo, and James Glass. Thread: Thinking deeper with recursive spawning. In *Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

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

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon LLM agent via context-folding. *arXiv Preprint arXiv:2510.11967*, 2025.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Syncode: Llm generation with grammar augmentation. *arXiv preprint arXiv:2403.01632*, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, 2017.

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

Boxin Wang, Chankyu Lee, Nayeon Lee, Sheng-Chieh Lin, Wenliang Dai, Yang Chen, Yangyi Chen, Zhuolin Yang, Zihan Liu, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Nemotron-cascade: Scaling cascaded reinforcement learning for general-purpose reasoning models. 2025a.

Qian Wang, Tianyu Wang, Zhenheng Tang, Qinbin Li, Nuo Chen, Jingsheng Liang, and Bingsheng He. Megaagent: A large-scale autonomous llm-based multi-agent system without predefined sops. In *Association for computational linguistics*, 2025b.

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

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. MMLU-PRO: A more robust and challenging multi-task language understanding benchmark. In *Advances in Neural Information Processing Systems*, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.

Ian Wu, Yuxiao Qu, Amrith Setlur, and Aviral Kumar. Reasoning cache: Continual improvement over long horizons via short-horizon rl. *arXiv Preprint arXiv:2602.03773*, 2026.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, et al. Resum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025.

Yingxuan Yang, Huacan Chai, Shuai Shao, Yuanyi Song, Siyuan Qi, Renting Rui, and Weinan Zhang. Agentnet: Decentralized evolutionary coordination for llm-based multi-agent systems. 2025.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, 2023.

Alex L Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025, 2025.

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

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs. In *International Conference on Machine Learning*, 2024.

A CRM ASSEMBLY SEMANTICS

A.1 SEMANTICS

$$b ::= \mathsf{decl} \mid \mathsf{res}(\widetilde{G}_p, \widetilde{G}_v) \mid \mathsf{clos}(\Gamma, \widetilde{G}, p, D, n_r), \qquad K ::= \epsilon \mid \varphi \cdot K,$$
$$\varphi ::= \mathsf{scope}(\Gamma, \widetilde{G}) \mid \mathsf{work}(n, \widetilde{G}) \mid \mathsf{call}(\Gamma, \widetilde{G}, n, n_r, D, s),$$
$$\langle s, \Gamma, K, \widetilde{G}\rangle \longrightarrow \langle s', \Gamma', K', \widetilde{G}'\rangle$$

$$\textsc{Step}\ \frac{n \notin dom(\Gamma) \qquad \mathsf{call} \notin K}{\langle \mathsf{step}\ n(v); s, \Gamma, K, \widetilde{G}\rangle \to \langle s, \Gamma[n \mapsto \mathsf{decl}], K, \widetilde{G}; \mathsf{step}\ n(v)\rangle}$$

$$\textsc{Work}\ \frac{\Gamma(n) = \mathsf{decl} \qquad \mathsf{call} \notin K}{\langle \mathsf{work}\ n(v); s, \Gamma, K, \widetilde{G}\rangle \to \langle s, \Gamma, \mathsf{work}(n, \widetilde{G}) \cdot K, \widetilde{G}; \mathsf{work}\ n(v)\rangle}$$

$$\textsc{Work-Call}\ \frac{\Gamma(n) = \mathsf{decl} \qquad K = \mathsf{call}(\Gamma_0, \widetilde{G}_0, n', n_r, D, s') \cdot K_0 \qquad \mathsf{step}\ n(v') \in D}{\langle \mathsf{work}\ n(v); s, \Gamma, K, \widetilde{G}\rangle \to \langle s, \Gamma, \mathsf{work}(n, \widetilde{G}) \cdot K, \widetilde{G}; \mathsf{work}\ n(v)\rangle}$$

$$\textsc{Value}\ \frac{}{\langle \mathsf{value}\ n(v); s, \Gamma, \mathsf{work}(n, \widetilde{G}_0) \cdot K, \widetilde{G}\rangle \to \langle s, \Gamma[n \mapsto \mathsf{res}(\widetilde{G}, \mathsf{value}\ n(v))], K, \widetilde{G}_0\rangle}$$

$$\textsc{Load-Val}\ \frac{\Gamma(n) = \mathsf{res}(\widetilde{G}_p, \widetilde{G}_v)}{\langle \mathsf{load}\ n(\mathsf{val}); s, \Gamma, K, \widetilde{G}\rangle \to \langle s, \Gamma, K, (\widetilde{G}; \mathsf{load}\ n(\mathsf{val})) \sqcup \widetilde{G}_v\rangle}$$

$$\textsc{Load-Prov}\ \frac{\Gamma(n) = \mathsf{res}(\widetilde{G}_p, \widetilde{G}_v)}{\langle \mathsf{load}\ n(\mathsf{prov}); s, \Gamma, K, \widetilde{G}\rangle \to \langle s, \Gamma, K, (\widetilde{G}; \mathsf{load}\ n(\mathsf{prov})) \sqcup \widetilde{G}_p\rangle}$$

$$\textsc{Push}\ \frac{\mathsf{call} \notin K}{\langle \mathsf{push}; s, \Gamma, K, \widetilde{G}\rangle \to \langle s, \Gamma, \mathsf{scope}(\Gamma, \widetilde{G}) \cdot K, \widetilde{G}; \mathsf{push}\rangle}$$

$$\textsc{Pop}\ \frac{\Gamma(n) = R = \mathsf{res}(\widetilde{G}_p, \widetilde{G}_v) \qquad n \notin dom(\Gamma_0) \ \vee\ \Gamma_0(n) = \mathsf{decl}}{\langle \mathsf{pop}\ n; s, \Gamma, \mathsf{scope}(\Gamma_0, \widetilde{G}_0) \cdot K, \widetilde{G}\rangle \to \langle s, \Gamma_0[n \mapsto R], K, \widetilde{G}_0; \mathsf{push}; \mathsf{pop}\ n\rangle}$$

$$\textsc{Fn}\ \frac{f \notin dom(\Gamma) \qquad \mathsf{call} \notin K \qquad \mathsf{step}\ n_r(v_r) \in D \qquad \widetilde{G}_f = \widetilde{G}; \mathsf{fn}\ f(v, p, D, n_r)}{\langle \mathsf{fn}\ f(v, p, D, n_r); s, \Gamma, K, \widetilde{G}\rangle \to \langle s, \Gamma[f \mapsto \mathsf{clos}(\Gamma[D], \widetilde{G}_f, p, D, n_r)], K, \widetilde{G}; \mathsf{fn}\ f(v, p, \epsilon, n_r)\rangle}$$

$$\textsc{Call}\ \frac{\Gamma(f) = \mathsf{clos}(\Gamma_f, \widetilde{G}_f, p, D, n_r) \qquad \Gamma(n_a) = R_a \qquad \mathsf{call} \notin K \qquad o = \mathsf{call}\ f(n_a, n, \epsilon)}{\langle \mathsf{call}\ f(n_a, n, C); s, \Gamma, K, \widetilde{G}\rangle \to \langle C, \Gamma_f[p \mapsto R_a], \mathsf{call}(\Gamma, \widetilde{G}; o, n, n_r, D, s) \cdot K, \widetilde{G}_f; o\rangle}$$

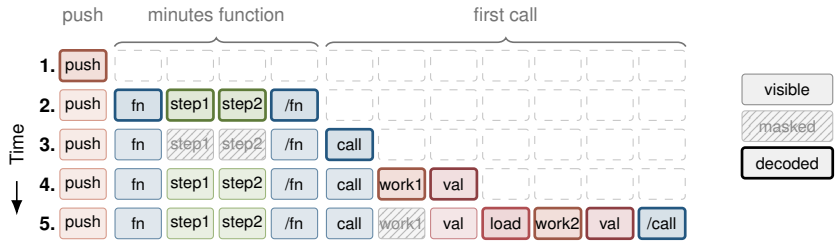
$$\textsc{Return}\ \frac{\Gamma(n_r) = R \qquad n \notin dom(\Gamma_0) \ \vee\ \Gamma_0(n) = \mathsf{decl}}{\langle \epsilon, \Gamma, \mathsf{call}(\Gamma_0, \widetilde{G}_0, n, n_r, D, s) \cdot K, \widetilde{G}\rangle \to \langle s, \Gamma_0[n \mapsto R], K, \widetilde{G}_0\rangle}$$

A.2 CRM ASSEMBLY PRIMITIVE TOKEN ENCODING

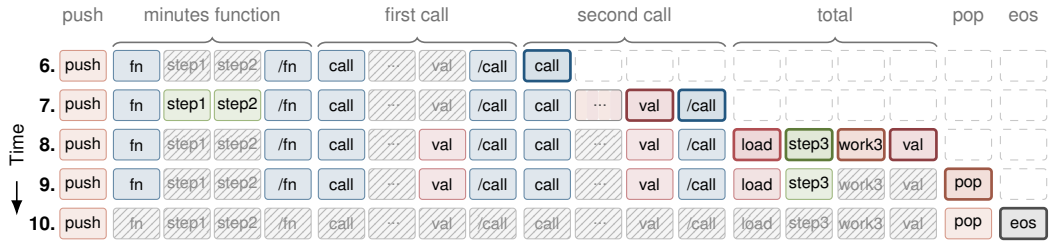
Table 1: Encoding of abstract constructs as tokens. Each underlined string is one special token.

Abstract construct	Special tokens
n(val)	n<:val>
n(prov)	n<:prov>
step $n(v)$	<step id="n" task="v"></step>
work $n(v_w)$	<work id="n"></work>v_w
value $n(v_r)$	<ret value="v_r"></ret>
load h	<load ids="h"></load>
push	<push>
pop n	<pop id="n"></pop>
fn $f(v, n_p, D, n_r)$	<fn id="f" task="v" params="n_p"></fn>D<endfn ret="n_r"></endfn>
call $f(n_a, n, C)$	<call fn="f" args="n_a" as="n"></call>C<endcall>

B Attention Masks of the Example Trace



(a) Attention mask as the machine declares and calls a function.



(b) Attention mask as the machine calls the function again, works, and ends the scope.

Figure 7: The Computational Reasoning Machine generates tokens under the attention mask determined by the CRM ASSEMBLY primitives in the prefix. A `push` opens a scope and an `fn` declares a sequence of prescribed steps; each `call` is a scope for performing those steps; `load` unmasks the values of referenced steps; and `pop` masks the scoped interior.

C Annotation Prompts

C.1 `annotate_step`

```
You are given a reasoning trace. Divide it into consecutive steps and say where each step begins.

You do not reproduce the trace. You return only a list of boundaries, as JSON:

[
  {"task": "write down the expression for the problem", "starts": "The expression is"},
  {"task": "simplify the expression", "starts": "Now simplify"},
  {"task": "state the final answer", "starts": "So the answer"}
]

- "starts" is a short snippet, copied exactly from the trace, marking where that step BEGINS. Five to ten
  words is ideal. It must appear in the trace, and it must be enough to tell that position apart from
  earlier ones. Take it from the start of a LINE where the step begins on one, and from the start of a
  sentence otherwise.
- The first entry's "starts" must be the opening words of the trace.
- List the steps in order. Each "starts" must occur later in the trace than the one before it.
- "task" is your own description of what that step does, phrased as an instruction, in plain text.
- EVERY STEP MUST DO SOMETHING. A step computes a quantity, establishes a fact, examines a case, rules
  a possibility out, carries out a check, or draws a conclusion. If you cannot say what a stretch of
  text works out or decides, it is not a step of its own.
- The "task" has to read as an instruction that would PRODUCE that step's text. If the only honest
  description is "introduce the problem", "restate the question" or "acknowledge the task", then it is
  not a step - merge it into the step that follows.
- Pure framing belongs to the work it leads into: "Okay, let's tackle this" or "Hmm, let me think" is
  not a step by itself, so attach it to the step that follows.
- A CHANGE OF DIRECTION STARTS A NEW STEP - when it actually changes something. "Wait", "Alternatively",
  "Actually", "But hold on", "Let me reconsider" often mark the reasoning abandoning one line and trying
  another; each real turn is its own step, however many there are. Abandoning an approach, overturning a
  result, re-deriving a quantity and trying a second method are all work: they rule something in or out.
- These words are also used as bare hesitation, with the reasoning carrying straight on. That is not a
  change of direction and does not start a step. Decide by what FOLLOWS the word, not by the word
  itself: if the text after it does not do anything different, it belongs to the step already running.
- No step may consist only of a reconsideration. "Wait, no." on its own is not a step - attach it to
  whichever neighbouring step carries the work.
```

```
- Traces that circle back repeatedly therefore get many steps, one per real attempt. Do not gather a long
  run of genuinely different attempts into a single step.
- ONE RESULT PER STEP. A step runs from the start of a piece of work to the result it reaches. As soon
  as a result is reached and the reasoning turns to the next thing, that step has ended. If a stretch of
  text works out two quantities, checks two cases, or tries two approaches, it is two steps.
- Do not begin a step in the middle of a sentence, a code block, a LaTeX display, a table or a list
  item.
- A HEADING BELONGS TO THE SECTION IT INTRODUCES, NEVER TO THE ONE ABOVE IT. When a step's work sits
  under a title, a heading, a section marker such as "#### **2. Key Findings**" or "##### **a. Daily
  Screen Time**", a numbered or lettered item, or a bolded label alone on its line, begin the step AT
  that line - not at the prose underneath it. If a horizontal rule such as "---" sits immediately above
  the heading, begin at the rule. A heading left on the previous step titles work that step does not
  contain, and reads as though that step wrote it.
- BEGIN AT THE OUTERMOST HEADING OF A RUN. When one heading is immediately followed by another, or a
  label such as "**Economy & Work:**" introduces a numbered or bulleted list, begin the step at the
  FIRST line of that run. The outer heading and whatever starts directly beneath it open the same step;
  the remaining items under that heading are steps of their own and start at their own markers.

Return only the JSON array.
```

C.2 annotate_deps

```
You are given a reasoning trace already divided into numbered steps. For each step, give it a short
variable name and record what it depends on. You do not rewrite the trace.

Return JSON, one object per step, in the same order as the input:

[
  {"id": "plan",    "exec": "root",  "data": []},
  {"id": "pens",    "exec": "plan",  "data": []},
  {"id": "books",   "exec": "plan",  "data": []},
  {"id": "bob",     "exec": "plan",  "data": []},
  {"id": "alice",   "exec": "plan",  "data": ["pens", "books"]},
  {"id": "diff",    "exec": "plan",  "data": ["alice", "bob"]}
]

Note the shape: `plan` has five children. `pens`, `books` and `bob` are independent of one another and
could be done in any order. `alice` and `diff` are children of `plan` too — the plan is what called for
them — and they name in "data" the results they combine. Nothing here is a chain.

- "id" is a short lowercase name, unique within the trace, saying what the step produces: `pens`,
  `alice_total`, `check_units`, `sequence_value_at_four`.
- IDS MUST NOT BE ENUMERATED. No `step1`/`step2`, and no family that differs only by a trailing number:
  `test_n2`, `test_n3`, `test_n4` is exactly the pattern to avoid. Name what each one produces instead —
  `iterative_update_n_equals_two`, `verify_sample_input`, `sequence_value_n_equals_four`. A number is
  fine INSIDE a name that says something (`case_two_negative_root`); it is not fine as the only thing
  distinguishing two ids. A model reading such a trace learns the numbering as a counter and invents
  `test_n47` at inference to get past a duplicate-id guard.
- "exec" is the id of the ONE earlier step that CREATED this step's task — the step whose work is the
  reason this step is being done at all. Exactly one id, never a list. The first step's "exec" is "root".
  This records where the step came from. It does NOT give the step any information: if the step needs that
  step's work, "data" has to name it too.
- DO NOT DEFAULT TO THE PREVIOUS STEP. A step written after another is not necessarily caused by it. Ask:
  if I deleted the previous step, would this step still make sense? If yes, its "exec" is something
  earlier — usually whichever step set up the task, planned the approach, or listed what to check.
- INDEPENDENT STEPS ARE SIBLINGS. When several steps each carry out one part of a plan, check one case of
  several, establish separate facts, or try alternative methods, they all take the SAME "exec" — the step
  that set them up — and they do not depend on each other. A step that reads the problem statement rather
  than any earlier working is a sibling of the step that set the task, not a child of whatever happens to
  precede it.
- A chain where every step's "exec" is the one before it is almost always wrong. Real traces branch: they
  set something up, then do several things under it, then combine them.
- "data" lists the ids of earlier steps this step RELIES ON — because it reads their work (numbers,
  expressions, conclusions it picks up and uses) or because it depends on what they settled even though
  it does not quote them. THIS INCLUDES THE "exec" STEP: if a step needs the work of the step it came
  from, list that step in "data" as well. Nothing is supplied to a step unless "data" names it.
- Checks and verifications count. If a step states a result as final, or moves on as though something is
  settled, and an earlier step is what established or confirmed that, list it. A step presenting final
  code relies on the steps that tested the code, even though it does not read their numbers.
- LIST ONLY WHAT IS NECESSARY. A step gets the work of exactly the steps in its "data" and nothing else.
  Ask of each candidate: without this step's work in front of me, could I still carry out this step from
  its task and the problem? If yes, leave it out. Do not list a step merely because it came earlier, or
  because it is the step this one follows from.
- Many steps need nothing: they work from the problem statement, from general knowledge, or from their own
  task alone. "data" is [] for those, and that is the common case, not a mistake.
- Decide "data" from the step's text. If it restates or operates on a value that an earlier step produced,
  that step is a data dependency. If it works everything out from the problem itself, "data" is empty.
- A step that combines, compares or reconciles earlier results lists all of them in "data".
```

```
- Every id used in "exec" or "data" must belong to a step EARLIER in the list. Never refer forwards, and
  never refer to the step itself.
- NO STEP'S WORK MAY GO NOWHERE. Every step except the last must appear in the "data" of at least one
  later step. If a step establishes a fact, checks a case, or computes a value, then whichever later step
  uses or concludes from it must name it. Independent siblings still have to be joined: if several steps
  each work out one part, the step that combines them lists ALL of them.
- Check your answer before returning it: walk the list and confirm each id appears in some later "data".
  A step whose result no later step names is almost always a missed dependency, not wasted work.

Return only the JSON array.
```

C.3 annotate_scoped_modes

```
You are given a problem and a reasoning trace that answers it, already divided into steps in the order
they run, and already divided into BLOCKS. For each step, decide WHICH EARLIER STEPS it needs to be
shown, and HOW MUCH of each.

WHAT A BLOCK DOES TO WHAT A STEP CAN SEE. Some runs of steps are wrapped in a block. While a block is
open its steps are ordinary. When it CLOSES, everything it held goes out of reach — except ONE step,
named when the block opened, which is what the block returns. The input marks this: a block's steps
are indented, and where it closes you will see which step it returned.

So each step lists the steps it MAY NAME, and that list is the whole of what is available to it. A
step that has closed a block behind it can name that block's return step and nothing else of the
block. What you are choosing is which of the nameable steps' WORK arrives, and at what depth.

WHAT A STEP ALREADY SEES, FOR FREE:

Every still-visible step's DECLARATION — its name and the task it was given. So leaving a step out
never hides its existence. What your list decides is whose WORK arrives:

- "val"  — that step's work: the number, expression or conclusion it reached.
- "prov" — its work, AND WHAT THAT STEP COULD SEE when it did the work.

"prov" therefore reaches BACKWARDS THROUGH the step you name, without you naming what lies behind it.
If the step you read had itself read something, you get that too. If it read nothing, "prov" gives
exactly what "val" gives.

NAMING A CLOSED BLOCK'S RETURN STEP IS THE SAME CHOICE, WITH MORE AT STAKE. The two modes are how much
of the collapsed block comes back:

- "val"  — the return step's work alone. The block is a black box that produced a result, and the
           step you are writing uses that result.
- "prov" — the return step's work AND the whole interior of the block as that step saw it: the other
           steps' declarations, and the work of whichever of them the return step had read.

So "prov" onto a closed block un-collapses the derivation, and "val" keeps it collapsed. Ask the same
question you ask anywhere else — does this step USE the result, or does it reason about how the result
was reached? — and answer it knowing that "prov" here can bring back several steps at once. Prefer
"val" unless the step genuinely works from the derivation.

Return JSON, one object per step THAT NEEDS SOMETHING, in input order. A step that works from nothing
is left out. This is the SHAPE, not an example — every id must come from that step's "may name" list:

[
  {"id": "<step id, exactly as given>",
   "deps": {"<an id from that step's may-name list>": "val" | "prov", ...}},
  ...
]

CHOOSING THE STEPS:

- Read the step's work and ask what it is working FROM. A quantity it substitutes, a finding it builds
  on, a draft it edits. Those are its dependencies. Everything else in the trace is not.
- Only from the step's own "may name" list. Naming anything else is not a mistake to be corrected —
  it is unreachable, and the edge would deliver nothing.
- Prefer the shortest list that supports the work.
- If a step works from something two steps back, either name that step directly, or name the step in
  between at "prov" and reach it through that. Name it directly when the step uses that result itself;
  use "prov" when what it needs is the REASONING, not one value.

CHOOSING THE MODE:

- "val" — the step reads a result and carries on with it: substituting a number, combining two
  quantities, restating a finding, continuing from a value.
- "prov" — the step engages with HOW the result came about: re-deriving it, auditing the arithmetic,
  locating a mistake, judging whether a method was valid, or writing something that has to stay
  consistent with choices made upstream.
```

```
ONE QUESTION DECIDES THE MODE. Does the step treat that result as a GIVEN and use it, or does it
reason about how the result was arrived at? Using it is "val". Reasoning about its derivation is
"prov".

EXAMPLE. One trace in the input format, with the answer.

[root] root
    task: A rectangle has perimeter 34 and area 60. How long is its diagonal?
    work: none

[0] setup
    task: Name the sides and write the two conditions
    work: Let the sides be a and b. Perimeter 2(a + b) = 34, so a + b = 17, and ab = 60.
    may name: root

    --- a block opens ---

    [1] quadratic
        task: Turn the two conditions into one equation
        work: a and b are the roots of x^2 - 17x + 60 = 0.
        may name: root, setup

    [2] disc
        task: Take the discriminant
        work: 17^2 - 4*60 = 289 - 240 = 49, and the square root of that is 7.
        may name: root, setup, quadratic

    [3] roots
        task: Solve for the two sides
        work: x = (17 +/- 7) / 2, so the sides are 12 and 5.
        may name: root, setup, quadratic, disc

    --- the block closes, returning `roots` ---

[4] diagonal
    task: Take the diagonal
    work: sqrt(a^2 + b^2) = sqrt(144 + 25) = sqrt(169) = 13.
    may name: root, setup, roots

[5] answer
    task: State the answer
    work: The diagonal has length \boxed{13}.
    may name: root, setup, roots, diagonal

[{"id": "quadratic", "deps": {"setup": "val"}},
 {"id": "disc",      "deps": {"quadratic": "val"}},
 {"id": "roots",     "deps": {"disc": "prov"}},
 {"id": "diagonal",  "deps": {"roots": "val"}},
 {"id": "answer",    "deps": {"diagonal": "val"}}]

Read `diagonal`. It works from "the sides are 12 and 5", which is `roots`'s work — and `roots` is what
the block returned, so it is nameable even though the three steps around it are not. It reads it at
"val": it substitutes 12 and 5 into the Pythagorean sum and never touches the quadratic they came from.
Asking for "prov" there would hand back the equation, the discriminant and the check as well, none of
which `diagonal` uses. It lists neither `setup` nor `root`: the two conditions are not what it
substitutes, the side lengths are.

`roots` reads `disc` at "prov" because it uses the square root in the quadratic formula and needs the
discriminant it came from; `disc` reads `quadratic` at "val" because it takes that equation's
coefficients and computes. And `answer` simply restates 13, so it reads `diagonal` at "val".
```

C.4 annotate_ret

```
You are given a problem and a reasoning trace that answers it, already divided into steps in the order
they run. For each step, write the VALUE it returns.

WHAT A VALUE IS.

Every step ends by stating its result — its VALUE. A step's value is shown to every later step, always.
So a later step that needs a number, an expression, a verdict or a decision from an earlier step reads
it there.

What a value does NOT carry is how the step got there. The working behind a step — the substitutions,
the arithmetic, the reasoning, the false turns — stays private to that step, and a later step reaches
it only by asking for it explicitly. That is why a value has to stand on its own: for most later steps
it is the only thing of this step they will see.
```

```
WRITING A VALUE.

A value RESTATES what the step concluded. It does not summarise the step, does not describe what the
step did, and does not introduce anything the step's work does not contain.

- Take it from the step's own work. If the work ends "so f(1) = -2", the value is "f(1) = -2".
- Keep the terms the step used. If the step wrote its result as a fraction, the value is that fraction,
  not a decimal you converted it to.
- State the result, not the activity. "f(1) = -2" — not "computed f(1)" and not "the inner function was
  evaluated".
- Include what a later step would need to USE it: the quantity and what it is the quantity OF. "-2" on
  its own is not usable; "f(1) = -2" is.
- A step whose result is a judgement returns the judgement: "the subtraction checks out, 18 is right",
  "n = 3 fails, 10 is composite", "the formal register is right for this reply".
- A step that decides something returns the decision, not the deliberation.
- Keep it to one short line. If it takes more than about 40 words, you are summarising the working
  rather than stating the result.
- Never write a value the step's work does not support. If the work reached no result, say what it
  did establish, in one clause — an honest thin value is right, an invented confident one is not.

OUTPUT.

Return JSON: one object per step, in input order, INCLUDING the root step, and every step must appear.
Root carries a "work" as well; no other step does, because every other step's work is already written.
This is the SHAPE, not an example — every id must come from the input:

[
  {"id": "root", "work": "<the reading of the problem>", "value": "<what that reading established>"},
  {"id": "<step id, exactly as given>", "value": "<what this step hands forward>"},
  ...
]

THE ROOT STEP IS A STEP LIKE ANY OTHER, and it is the one whose work you also write. Its declaration
is the problem itself, so it carries no task of its own — what it does is READ the problem. Give it a
"work" as well as a "value":

- "work" — the reading. What the problem gives, what it asks for, and the terms the rest of the trace
  has to work in: the quantities and what they are quantities of, the units, the form the answer has
  to take, a constraint that is easy to miss, notation worth fixing.
- "value" — what that reading established, stated as a later step would use it.

WHAT ROOT'S WORK IS NOT. It does not solve the problem, does not do any arithmetic, and does not
choose a method or a plan of attack. Deciding HOW to proceed is the first step's job and that step is
already written below — root's work must not pre-empt it or repeat it. If your root work contains a
calculation or a strategy, it belongs to a later step and root has been given the wrong thing.

Root's value is not the answer either. The trace has not reached one yet.

EXAMPLES. Two short traces in the input format, with the answer for each.

--- example 1 ---

[root] root
    task: Alice has 3 boxes of 4 pens and 5 books. How many items does she have?
    work: none

[0] pens
    task: Count Alice's pens
    work: Alice has 3 boxes of 4 pens, so 3 x 4 = 12 pens.

[1] books
    task: Count Alice's books
    work: She also has 5 books.

[2] total
    task: Add the two counts
    work: 12 + 5 = 17 items in total.

[{"id": "root",
  "work": "Two quantities are given: pens, as 3 boxes of 4 each, and books, as a flat 5. The question asks how
   many ITEMS she has, so pens and books are counted alike and the answer is one number, a count of things rather
   than a number of boxes.",
  "value": "a single count of Alice's items, from 3 boxes of 4 pens and 5 books"},
 {"id": "pens",  "value": "Alice has 12 pens"},
 {"id": "books", "value": "Alice has 5 books"},
 {"id": "total", "value": "17 items in total"}]

Root's work reads the problem and fixes what is being counted; it does not multiply 3 by 4, which is
`pens`'s job. Note that `pens` returns "Alice has 12 pens" and not "12": the bare number would not
tell a later step what it counted, and `total` reads this value rather than the working behind it.
```

```
--- example 2: a step whose result is a verdict ---

[root] root
    task: A job pays 240 dollars for 8 hours. What is the hourly rate?
    work: none

[0] rate
    task: Find the hourly rate
    work: 240 dollars over 8 hours, so 240 / 8 = 30 dollars an hour.

[1] check
    task: Verify the rate against the stated total
    work: Re-deriving it the other way: at 30 an hour, 8 hours gives 8 x 30 = 240, which matches the
    stated total, so the division was right.

[2] answer
    task: State the answer
    work: The rate is \boxed{30} dollars an hour.

[{"id": "root",
  "work": "The job pays 240 dollars in total for 8 hours of work. What is asked for is an HOURLY rate, so the
   answer is a rate in dollars per hour rather than the total already given, and it should come out as dollars
   per one hour.",
  "value": "an hourly rate in dollars per hour, for 240 dollars paid over 8 hours"},
 {"id": "rate",   "value": "the rate is 30 dollars an hour"},
 {"id": "check",  "value": "the rate of 30 an hour checks out against the 240 total"},
 {"id": "answer", "value": "30 dollars an hour"}]

`check` returns the verdict, not the re-derivation that produced it. A step that wants to see that
re-derivation asks for `check`'s working; every other step reads the verdict.
```

C.5 annotate_load

```
You are given a reasoning trace already broken into numbered steps.
Each step shows its id, its task, the working it wrote, and the value it returned.
The trace is fixed: do not rewrite any of it.

For EVERY step after step 0, say what that step had to be able to read. A step
reads for two separate reasons, and you report them separately:

  DECLARATION — what it needed to CHOOSE AND WRITE its own task, before any of
  its working existed. A task that names a quantity, a candidate, a discrepancy
  or a choice an earlier step found depends on that step, even when the working
  then re-derives everything. Do not assume the task was handed to the model.

  BODY — what it needed to write the working and value it actually wrote, once
  its own task was already visible.

Each read names an earlier step and how much of it is needed:
  `id:val`  that step's returned value alone
  `id:prov` that step's working as well

Output one JSON array and nothing else:
[{"id":"step-id","declaration":["earlier-id:val"],"body":["earlier-id:prov"]}, ...]

Rules:
- A step may only name ids whose number is STRICTLY LESS than its own. Never a
  later step, and never itself. Check each id against the numbers before
  emitting it — a step cannot read a result that does not exist yet.
- Either list may be empty when the step needs nothing for that obligation.
- Include an entry for every step except step 0.
- Report what the trace shows, not what could plausibly be related.

EXAMPLES. Short traces in the input format, with the answer for each.

--- example 1: a task that names an earlier result depends on it ---

step 0 · id=root
  task: A tank holds 240 L. It drains at 8 L/min. How long until it is empty?
  work: none
  value: the minutes taken to empty a 240 L tank draining at 8 L/min

step 1 · id=rate
  task: State the drain rate per minute
  work: The tank loses 8 L every minute, and that rate is constant.
  value: 8 L/min

step 2 · id=divide
```

```
  task: Divide the volume by the rate
  work: 240 / 8 = 30.
  value: 30 minutes

step 3 · id=units
  task: Check that 30 is in minutes and not hours
  work: The rate was per minute, so 240/8 is a count of minutes. 30 minutes.
  value: the answer is 30 minutes

[{"id":"rate","declaration":[],"body":["root:val"]},
 {"id":"divide","declaration":["rate:val"],"body":["root:val","rate:val"]},
 {"id":"units","declaration":["divide:val","rate:val"],"body":["rate:val","divide:val"]}]

`rate` needs nothing to be CHOSEN — reading the rate off the problem is where anyone starts — but its
working quotes the problem, so `root:val` is a body read. `divide` is the opposite case: you cannot
decide to divide by the rate until the rate exists, so `rate:val` is a DECLARATION read as well as a
body one. `units` is the case to study: its task names the number 30 and the unit question, neither of
which exists before `divide` and `rate` ran, so both are declaration reads even though the working
re-states them.

--- example 2: a fresh line of reasoning, and when `prov` is needed ---

step 0 · id=root
  task: Is 91 prime?
  work: none
  value: whether 91 is prime

step 1 · id=small
  task: Try the small primes
  work: 91 is odd, not divisible by 3 (9+1=10), does not end in 0 or 5.
  value: 91 survives 2, 3 and 5

step 2 · id=seven
  task: Try 7
  work: 7 x 13 = 91.
  value: 91 = 7 x 13, so it is composite

step 3 · id=audit
  task: Re-examine the trial division for an arithmetic slip
  work: The divisibility-by-3 test summed 9+1 to 10, which is right, and 7 x 13 is 91. No slip.
  value: the trial division holds

[{"id":"small","declaration":[],"body":["root:val"]},
 {"id":"seven","declaration":["small:val"],"body":["root:val"]},
 {"id":"audit","declaration":["seven:val"],"body":["small:prov","seven:prov"]}]

`seven` declares on `small:val` — trying 7 is the next candidate only because 2, 3 and 5 are gone —
but its working needs nothing from `small`, just the number from the problem. `audit` is where `prov`
earns its place: re-examining the arithmetic means reading the WORKING of the two steps, not their
verdicts, so both are `:prov`. Note what would be wrong: giving `audit` `small:val` and `seven:val`.
Their values say the tests passed, which is exactly what an audit may not take on trust.


This trace has SCOPES, shown as `--- a scope opens` and `--- the scope
closes, returning x`. A scope retires every step inside it when it closes:
after that line, the only name from inside it that anything may read is the
id the scope returned.

--- an example: a scope closes and takes its steps with it ---

step 0 · id=root
  task: Which of 14, 15, 16 has the most divisors?
  work: none
  value: the number among 14, 15, 16 with the most divisors

--- a scope opens

step 1 · id=count14
  task: Count the divisors of 14
  work: 1, 2, 7, 14.
  value: 14 has 4 divisors

step 2 · id=count15
  task: Count the divisors of 15
  work: 1, 3, 5, 15.
  value: 15 has 4 divisors

step 3 · id=counts
  task: Collect the two counts
  work: Both 14 and 15 have 4 divisors.
  value: 14 and 15 both have 4 divisors
```

```
--- the scope closes, returning `counts`

step 4 · id=count16
  task: Count the divisors of 16
  work: 1, 2, 4, 8, 16.
  value: 16 has 5 divisors

step 5 · id=answer
  task: Compare against the earlier counts and name the winner
  work: 16 has 5, the others had 4. So 16.
  value: <U+0008>oxed{16}

[{"id":"count14","declaration":[],"body":["root:val"]},
 {"id":"count15","declaration":[],"body":["root:val"]},
 {"id":"counts","declaration":["count14:val","count15:val"],"body":["count14:val","count15:val"]},
 {"id":"count16","declaration":[],"body":["root:val"]},
 {"id":"answer","declaration":["counts:val","count16:val"],"body":["counts:val","count16:val"]}]

`answer` needs the earlier counts, and after the scope closed the only name that survives it is
`counts` — the id the scope returned. Naming `count14` there would be wrong: that step is out of reach
once its scope has gone, whatever its value was. This is why a step inside a scope that later matters
outside it has to be what the scope returns.
```

C.6 annotate_deferred

```
You are given a problem and a reasoning trace that answers it. Every step of the trace is DECLARED
and then WORKED. Decide, for each step, WHERE ITS DECLARATION IS WRITTEN: in front of its own
working, or earlier. The trace itself is fixed: do not rewrite any of it.

Three kinds of thing appear between the workings:

    <step id="X" task=". . ."></step>      X is DECLARED: a promise that X will be done
    <load ids="a:val,b:prov"></load>   handles are acquired, which is how anything is read
    <work id="X"></work>. . . text. . . <ret value=". . ."></ret>
                                       X is WORKED: the marker, the working, and the value it returned

A `<ret . . .>` always follows its own `<work . . .>` with nothing in between, so a step's working and its
return are written together and neither moves. What moves is a declaration, and it moves to a LINE:
the trace is printed one element per line, the lines are numbered, and a number means "written in
front of this line". Moving a declaration earlier and delaying the working it promises are the same
change to the trace, so there is one line to choose per step, and it is the declaration's.

A NUMBERED LINE IS ONE THE TRACE FIXES: a working, the `<load>` that serves one, or a `<push>` /
`<pop>`. Those stay where they are whatever you answer, which is what makes them safe to point at.
THE DECLARATIONS THEMSELVES CARRY NO NUMBER — a declaration is the thing that moves, so a number
naming one would name a position your own answer decides. Nor do the seed's two lines, the frame of
the trace, or a `<load>` that travels with the declaration or the pop below it.

A DECLARATION SITS JUST IN FRONT OF ITS OWN LINE, so naming a step's `<load . . .>` or `<work . . .>` line
puts a group beside that step's declaration, and the two are written as one run in the trace's own
order. That is how several steps are planned together.

WHAT AN EARLY DECLARATION SAYS. A declaration written before its own working is a commitment made
without that working's result: the trace knew the step was coming before it did the steps in between.
Several declarations at one position say that of all of them at once — the problem determined this
much, and the trace settled it before starting. A declaration in front of its own marker says the
opposite, that what to do next was decided by the result just written.

So the question for each step is whether the trace could have known that step was coming. Most steps
are free to move; what decides the position is whether moving one states something true about the
reasoning.

DECLARE EARLY when the problem determined the step:

- several quantities each needing the same treatment: all of them are known from the start, and doing
  the first tells the trace nothing it did not already know about the second;
- a decomposition the problem's structure dictates: convert the units, apply the rate, total it;
- a step that combines results, or that states the answer, when the trace was always going to end
  that way.

DECLARE AT ITS OWN MARKER when the trace found the step:

- a recheck, a second method or a sanity test that exists because an earlier result looked wrong,
  surprising or too clean;
- a correction of a mistake, and whatever the correction needs;
- a case split whose branches depend on a value computed earlier;
- a step whose task could not have been WRITTEN earlier, because its wording names a number, a
```

```
  candidate or a finding that did not exist yet.

THE CHECK ON A HOIST. Write the step's task out as it would read at the earlier line. If it still
reads as a task — if it does not have to name something not yet computed — the step belongs early. If
writing it there needs the result it is reacting to, it belongs where it is.

A trace is usually some of both. Three parallel computations planned at the start and two checks
declared where they were thought of is two groups; a search whose every step reacts to the last moves
nothing at all; a problem that dictates its own decomposition is one group at the front. All three
are answers this format can state.

READS WRITTEN IN FRONT OF A DECLARATION. Most `<load>` operations sit between a declaration and its
marker: they serve the working, and the working does not move. Where one sits in front of a
DECLARATION instead, that step needed a result in order to be CHOSEN, so its declaration cannot rise
above the working that produced what it names. Those steps are listed under the trace as
`(needs: . . . )`, and such a step's line must be after the `<work id=". . . ">` line of what it names.

Output one JSON object and nothing else:

{
  "groups": [
    {"before":  <the number of the line these declarations are written in front of>,
     "declare": ["<step id>", "<step id>", ...]},
    ...
  ]
}

Rules:
- `before` is one of the numbers printed with the trace, and every declared id names a step of the
  trace, spelled as shown.
- A step you do not mention keeps its declaration where it is. That is the answer whenever the shape
  of the reasoning is unclear, and a trace that planned nothing ahead of itself gets an empty
  `groups`.
- No step is declared in more than one group.
- A step's `before` is EARLIER than its own lines. Its `<load . . . >` and `<work id="X">` lines are the
  position it already has, so naming one of them moves it nowhere.
- A step listed as `(needs: X)` has a `before` after the number of X's `<work id="X">` line.
- Steps planned together take the SAME `before`, and to plan a step alongside one that already stands
  where you want it, name the line that step's own declaration sits in front of.

EXAMPLES. Two short traces in the input format, with the answer for each.

--- example 1: a plan the problem determined, then a check the trace found ---

PROBLEM
Two cars leave together at 55 and 70 mph. After 3 hours, how far apart?

TRACE, as it is written today
     <step id="root" task="Two cars leave together at 55 and 70 mph. After 3 hours, how far apart?"></step>
     <work id="root"></work>Both travel for three hours.<ret value="two distances, then a difference"></ret>
     <step id="slow" task="Distance for the slower car"></step>
  1  <work id="slow"></work>55 * 3 = 165 miles.<ret value="165 miles"></ret>
     <step id="fast" task="Distance for the faster car"></step>
  2  <work id="fast"></work>70 * 3 = 210 miles.<ret value="210 miles"></ret>
     <step id="gap" task="Difference between the two distances"></step>
  3  <load ids="slow:val,fast:val"></load>
  4  <work id="gap"></work>210 - 165 = 45 miles apart.<ret value="45 miles"></ret>
     <step id="recheck" task="Redo it from the speed difference instead"></step>
  5  <load ids="gap:val"></load>
  6  <work id="recheck"></work>That 45 looks small. Other way: 70 - 55 = 15 mph, times 3 = 45.<ret value="45
  miles, confirmed"></ret>
     <step id="answer" task="State the distance"></step>
  7  <load ids="recheck:val"></load>
  8  <work id="answer"></work>They are 45 miles apart.<ret value="45 miles"></ret>

{"groups": [
  {"before": 1, "declare": ["fast", "gap"]},
  {"before": 5, "declare": ["answer"]}
]}

The problem gives two cars and asks for a difference, so `slow`, `fast` and `gap` were settled before
any arithmetic. `slow` is already the first thing declared and stays where it is; `fast` and `gap`
join it in front of line 1, which is `slow`'s own working, so the three declarations are written as
one run and no working comes between them. `recheck` is not part of that plan: the trace redid the calculation
  because 45 looked
small, which is a reaction to `gap`'s value, and "redo it from the speed difference" is a task only
once there is a result to redo. Note that nothing forbids hoisting it — its `<load>` sits after its
declaration, so no `(needs: . . . )` mark applies and the line is a judgement about the reasoning.
`answer` goes in front of line 5, next to `recheck`, rather than into the plan at the front: when the
plan was made the trace did not know it would answer from a confirmed value.
```

```
--- example 2: each step reacts to the last ---

PROBLEM
Find the smallest n > 1 with n, n+2 and n+6 all prime.

TRACE, as it is written today
     <step id="root" task="Find the smallest n > 1 with n, n+2 and n+6 all prime."></step>
     <work id="root"></work>Test candidates in order.<ret value="a search over n"></ret>
     <step id="try3" task="Test n = 3"></step>
  1  <work id="try3"></work>3, 5, 9 - 9 is not prime. No.<ret value="n = 3 fails"></ret>
     <load ids="try3:val"></load>
     <step id="try5" task="Test the next candidate"></step>
  2  <work id="try5"></work>5, 7, 11 - all prime.<ret value="n = 5 works"></ret>
     <load ids="try5:val"></load>
     <step id="smaller" task="Confirm nothing below 5 works"></step>
  3  <work id="smaller"></work>n = 2 gives 4; n = 3 already failed; n = 4 gives 4.<ret value="nothing
   smaller"></ret>
     <step id="answer" task="State the smallest n"></step>
  4  <load ids="smaller:val"></load>
  5  <work id="answer"></work>n = 5.<ret value="5"></ret>

STEPS WHOSE DECLARATION NEEDS A RESULT
  try5 (needs: try3)
  smaller (needs: try5)

{"groups": [
  {"before": 3, "declare": ["answer"]}
]}

A search does not know its own next candidate. "Test the next candidate" is a task only once n = 3 has
failed, and `try5` reads `try3`'s value in front of its declaration, so its mark states the bound
directly: it cannot be declared above line 1, `try3`'s working, and it already stands just after it.
`smaller` is the same case. `answer` is the one step that moves, to line 3, next to `smaller` — by the
time the trace had checked downwards it knew it was about to answer.

This trace has BLOCKS, written as `<push>` . . .  `<pop id="X"></pop>`, with the steps inside a block
indented one level. A block retires every step inside it when it closes: from the pop onwards, the
only name from inside it that anything may read is the id the pop carries.

A declaration and the working it promises sit in the SAME block, so the lines open to a step are the
ones inside its own block. A step inside a block cannot be declared in front of the `<push>` that
opens it or anywhere above that, and a step outside one cannot be declared at any line from that push
up to and including its `<pop>`: a declaration written inside a block is retired when the block
closes, and its promise could never be forced.

--- an example: a block bounds a hoist ---

PROBLEM
Rose bought 4 cakes on Monday and three times that many on Tuesday. How many in all?

TRACE, as it is written today
     <push>
         <step id="root" task="Rose bought 4 cakes on Monday and three times that many on Tuesday. How many in
   all?"></step>
         <work id="root"></work>Two days, then a total.<ret value="a sum over two days"></ret>
         <step id="monday" task="Cakes bought on Monday"></step>
  1      <work id="monday"></work>Monday: 4 cakes.<ret value="4"></ret>
         <step id="tuesday" task="Three times Monday's number"></step>
  2      <load ids="monday:val"></load>
  3      <work id="tuesday"></work>Tuesday: 3 * 4 = 12.<ret value="12"></ret>
         <step id="total" task="Total across the two days"></step>
  4      <load ids="tuesday:val"></load>
  5      <work id="total"></work>4 + 12 = 16.<ret value="16"></ret>
  6      <push>
             <load ids="total:val"></load>
             <step id="recount" task="Re-add the two days separately"></step>
  7          <work id="recount"></work>4 and 12 again: 16. Same.<ret value="16 re-added"></ret>
             <step id="verify" task="Check the total against the re-count"></step>
  8          <load ids="recount:val"></load>
  9          <work id="verify"></work>Both give 16, so 16 stands.<ret value="16 confirmed"></ret>
 10      <pop id="verify"></pop>
         <step id="answer" task="State how many cakes"></step>
 11      <load ids="verify:val"></load>
 12      <work id="answer"></work>She bought \boxed{16} cakes.<ret value="16"></ret>
     <pop id="answer"></pop>

STEPS WHOSE DECLARATION NEEDS A RESULT
  recount (needs: total)

{"groups": [
  {"before": 1, "declare": ["tuesday", "total"]},
```

```
  {"before": 7, "declare": ["verify"]}
]}

The two days and their total are what the problem asks for, so `tuesday` and `total` join `monday` in
front of line 1 and the three are written as one plan. `recount` and `verify` are the trace's reaction
to that total: `recount` stays where it is, which its `(needs: total)` mark also requires, and
`verify` joins it in front of line 7. `answer` sits outside the block, so lines 7 to 10 are closed to
it — a declaration written there would be retired by the pop before the answer could be worked. Line
6 is the `<push>` itself, which is outside, so its choices are lines 1 to 6; taking one of them would
claim the trace planned to answer from a re-count before doing either, so it stays where it is.
```

C.7 annotate_func

```
You are given a problem and a reasoning trace that answers it. The trace is a list of steps. Each step
has an id, the handles it READS, a task, its working, and the value it returns.

Decide whether part of this trace CAN BE REWRITTEN so that one procedure is written once and CALLED
more than once. You are not looking for steps that already match. You are asking whether two or more
runs of steps do the same kind of work, and whether one procedure — written in general terms, with
parameters for what differs — would serve all of them once you are free to restate the steps.

YOU MAY RESTATE THE STEPS. The procedure decides how many steps each run has, what each step reads,
what each step says and what it returns. Two runs the trace wrote at different lengths, or in a
different order, or reading different things, become two calls of one procedure when one plan
describes both — and writing that plan is the work here.

WHAT A DEFINITION HOLDS. A name, a task, and its ROLES — one declaration each, and nothing else. No
working, no values, no reads of its own: every call writes the reads that serve its own bodies, works
each role itself, and ends itself. What the calls of a definition share is exactly its declarations.

A PROCEDURE HAS AT LEAST TWO ROLES. One role is one declaration shared, and every call spends a
header and a terminator to reach it, which costs more than writing that declaration out again. So a
one-role procedure is not a smaller answer here — it is no answer, and a trace whose repeated work
really is a single step has nothing to rewrite. Where the repetition is real, the two roles are
usually already there in the trace: the step that sets the instance up and the step that concludes
it, or the reading of an input and the computation over it.

WHAT A CALL REPLACES IS A CONSECUTIVE RUN OF THE TRACE'S STEPS. Name the run by its step ids, in the
trace's order, with nothing in between left out. The runs of one procedure do not overlap and do not
share a step.

THE RUN'S LAST STEP KEEPS ITS ID AND ITS VALUE. The call exports its result under that id and the
steps after it read it there, so the run's own value is what the last role returns and you do not
retype it — write that role's WORKING so that it reaches the value the trace already shows for that
step. Everything else in the run is private to the call.

NO BLOCK MAY CROSS A RUN'S EDGE. The listing marks where each block opens and closes. A block that
opens inside a run and closes outside it — or the other way round — would put half a scope into the
call, and the call would leave that scope open, so a run straddling one cannot become a call. A block
wholly inside a run, or one that wholly contains it, is fine.

NOTHING OUTSIDE A RUN MAY READ ANY STEP OF IT BUT THE LAST. A call retires its own bodies when it
ends, so if a step after the run reads one of the run's earlier steps, that run cannot become a call.
The reads are shown for every step; check them.

WHAT YOU WRITE

  * The ROLES — the procedure's steps, in the order they run, TWO OR MORE. Each has a NAME, a TASK
    and a READS list, and all three describe the step for ANY call: the name is `line`, not
    `apple_cost`, and the task is "Multiply the quantity by the unit price", not "Cost of the apples".
  * The READS of a role are written ONCE, in the procedure's own names: an earlier role's name, a
    parameter's name, or the id of a step OUTSIDE every run that all the calls read the same way. Use
    `x:val` for the value a step returned and `x:prov` for its working as well. Each call writes this
    same list itself, under its own bindings, which is what makes it one procedure.
  * The PARAMETERS — one name per thing that differs between the calls. A parameter is bound, per
    call, to a STEP that stands BEFORE that call's run. A step every call reads identically needs no
    parameter: name it directly in the role's reads, and each call loads it where it stands.
  * The GROUPS — one per call: the run it replaces, what each parameter is bound to for this call,
    the steps this call WRITES so it has something to bind, and one BODY per role. A body is the
    working that role writes in this call and the value it returns; the LAST role's `value` is taken
    from the trace, so write `""` there.

WHAT VARIES BETWEEN THE CALLS IS A PARAMETER, AND YOU WRITE THE STEP IT BINDS TO. A call binds a
parameter to a step id, so an input the trace never gave a step of its own — a quantity read straight
out of the problem, which is where a varying input usually sits — cannot be an argument until there
is a step holding it. Write that step under `binds`: it stands immediately before the call, and the
call's `args` name it.
```

```
    "binds": [{"id": "n_three", "task": "The input this check uses", "reads": ["root:val"],
               "work": "", "value": "3"}],
    "args": {"n": "n_three"}

A BOUND STEP WRITES NO WORKING. Its whole content is the value it hands the call, so `"work": ""` is
what you put there: working that restates the value — "the answer choice being evaluated is option B"
before `<ret value="B">` — says a second time what the return already says, and pays a body's tokens
per call to do it. Write working only where REACHING the value takes reasoning: "the remaining case
is the one where x is negative, so this call gets -2".

TWO CALLS OF ONE PROCEDURE MUST DIFFER IN THEIR ARGUMENTS. A call's result is fixed by the procedure
and what it is called with, so two calls binding the same steps return the same value — which means
the trace stated one computation twice and there is no procedure to write. Every procedure you
propose therefore has at least one parameter, and no two of its calls bind the same thing. Three
calls checking n = 3, n = 2 and n = 4 differ in n, so n is the parameter and each call writes the
step holding its own value; a header reading `params=""` with the number buried in the body's prose
says nothing about what the procedure was applied to.

THE EXPORT NAME CANNOT CARRY THE DIFFERENCE. `as` is the id the run's last step already had: the
steps after the call read the result by it, and the call itself never sees it. So a procedure whose
instances are told apart only by exporting as `exact` and as `radical` has put the difference in the
one place that cannot hold it. If what varies is the FORM the answer is wanted in — exact against
decimal, factored against expanded, one unit against another — that form is a parameter: bind a step
stating it and name it in `args`.

    "binds": [{"id": "want_decimal", "task": "The form the answer is wanted in", "reads": ["root:val"],
               "work": "This part asks for a decimal approximation.", "value": "a decimal"}],
    "args": {"form": "want_decimal"}

THE WORKING MUST BE TRUE OF THIS PROBLEM. Rewrite what the trace's own steps said into the roles you
laid out — the same quantities, the same arithmetic, the same conclusion. Do not introduce a number
the trace did not have, and do not drop a case it handled. A rewritten run that reaches the run's
value by different reasoning is worse than no rewrite at all.

HOW TO FIND THE TWO ROLES. The repeated stretch is wider than the step that catches your eye, so
start from that step and look outwards:

  * TAKE THE WIDEST RUN THAT STILL LINES UP. Before settling on a run, look at the steps on either
    side of it. If each instance is preceded by the same kind of set-up step, or followed by the same
    kind of check, those belong in the procedure too — the procedure is the whole repeated stretch,
    not its most obviously identical step.
  * SPLIT THE BODY INTO ROLES rather than writing one role that does everything. If each instance
    reads an input, computes with it and states a conclusion, that is three roles, and the definition
    then says what the procedure DOES instead of naming it. A step whose working has two clauses —
    price it, then discount it — is two roles as soon as you write it as two.
  * A ONE-ROLE ANSWER IS NOT A SMALLER ANSWER. If, after looking outwards and after trying to split
    the body, the repeated work is genuinely one step that cannot be stated as two, leave it out of
    your answer entirely.

DO NOT REWRITE THE WHOLE TRACE INTO ONE PROCEDURE. A procedure is a part the trace does more than
once, not the trace itself.

LOOK FOR THESE BEFORE YOU ANSWER, in this order:

  * A MULTI-STEP ROUTINE CARRIED OUT SEVERAL TIMES — each item read, converted and checked; each
    case set up, computed and concluded. This is the best answer and the one to look for first.
  * A LOOP THE TRACE UNROLLED — the same update carried out once per iteration, usually several
    steps per turn of it.
  * THE SAME COMPUTATION AT DIFFERENT LENGTHS — one line priced in a step and another in two. Give
    both the same plan and split or merge the working to fit it.
  * THE SAME TREATMENT APPLIED TO SEVERAL ITEMS — each of ten technologies assessed, each of four
    quantities converted. The item is the parameter; where the trace read it straight out of the
    problem and gave it no step, write that step under `binds`. Where the treatment of one item is a
    single step, ask whether stating it as two — what this item is, then what it comes to — is a true
    account of the working. If it is not, this trace has nothing to rewrite.

Answer `{"rewrites": []}` when none of those is there: a trace that computes one thing once, or whose
similar-looking steps do genuinely different work, has nothing to rewrite. But decide that by looking
for the shapes above and not finding them, rather than by finding no two steps already alike.

Output one JSON object and nothing else:

{
  "rewrites": [
    {"id": "<short name for the procedure>",
     "task": "<what it does, written for any call — no specific quantity>",
     "params": ["<parameter name>", ...],
     "steps": [{"id": "<role name>", "task": "<what this step does, for any call>",
                "reads": ["<handle>", ...]}, ...],
     "groups": [
       {"replaces": ["<step id>", ...],
```

```
       "binds": [{"id": "<new step id>", "task": "<what it states>", "reads": ["<handle>", ...],
                  "work": "", "value": "<the argument it hands the call>"}, ...],
       "args": {"<parameter name>": "<step id, of the trace or of this call's binds>"},
       "bodies": [{"work": "<this call's working for role 1>", "value": "<what it returns>"}, ...]},
      ...
    ]}
  ]
}

`steps` holds at least two roles. `replaces` is the run in the trace's order. `bodies` is one per
role, in role order. `args` binds every parameter and nothing else, every step under `binds` is bound
by one, and no two calls of a procedure bind the same things.

EXAMPLE

PROBLEM
A shop sells apples and pears, both at 3 each, with 10% off any line over 5 items. Kim buys 4 apples
and 7 pears. What is the total?

TRACE
  unit          reads: root:val
                task: Read the shared unit price
                work: Both fruits are 3 each.
                value: 3
  apples        reads: root:val
                task: Read how many apples
                work: Kim buys 4 apples.
                value: 4
  apple_cost    reads: apples:val, unit:val
                task: Cost of the apples
                work: 4 apples at 3 each is 12, and 4 is not over 5, so there is no discount.
                value: 12
  pears         reads: root:val
                task: Read how many pears
                work: Kim buys 7 pears.
                value: 7
  pear_gross    reads: pears:val, unit:val
                task: Pears before the discount
                work: 7 pears at 3 each is 21.
                value: 21
  pear_net      reads: pear_gross:val, pears:val
                task: Pears after the discount
                work: 7 is over 5, so take 10% off 21, giving 18.9.
                value: 18.9
  total         reads: apple_cost:val, pear_net:val
                task: Add the two line totals
                work: 12 + 18.9 = 30.9.
                value: 30.9

{"rewrites": [
  {"id": "line_total", "task": "One line at the shared unit price, discounted when the line is large",
   "params": ["qty"],
   "steps": [{"id": "gross", "task": "Multiply the quantity by the shared unit price",
              "reads": ["qty:val", "unit:val"]},
             {"id": "net", "task": "Take 10% off when the quantity is over five",
              "reads": ["gross:val", "qty:val"]}],
   "groups": [
     {"replaces": ["apple_cost"], "args": {"qty": "apples"},
      "bodies": [{"work": "4 apples at 3 each is 12.", "value": "12"},
                 {"work": "4 is not over 5, so no discount applies and 12 stands.", "value": "12"}]},
     {"replaces": ["pear_gross", "pear_net"], "args": {"qty": "pears"},
      "bodies": [{"work": "7 pears at 3 each is 21.", "value": "21"},
                 {"work": "7 is over 5, so take 10% off 21, giving 18.9.", "value": "18.9"}]}]}
]}

The trace priced the apples in ONE step and the pears in TWO, so no group of its own steps matches
another and nothing can be factored here without restating something. One plan describes both lines
all the same: price it, then discount it. The apples' single step had two clauses in its working, so
it is written as two roles, its second body saying what that step's second clause said — and that is
what makes this a procedure rather than a name for one step. Each run's last step keeps its id —
`apple_cost` and `pear_net` — and its value, so `total` reads exactly what it read before.

The definition that gets written states two declarations and which of them it returns, and that is
the whole of it. Each call then writes `qty:val, unit:val`, works `gross`, writes `gross:val, qty:val`
and works `net`. `unit` is read the same way by both calls, so it is named in the role's reads rather
than passed as a parameter, and each call reads it where it stands.
```

C.8 annotate_scopes

```
You are given a problem and a finished reasoning trace that answers it. The trace is already divided
into steps in the order they run, every step already names the earlier steps it READS, and some runs
of steps are CALLS of a procedure. Decide which RUNS OF CONSECUTIVE STEPS should become BLOCKS, and
for each block, WHICH ONE OF ITS STEPS is the result it hands back.

WHAT A BLOCK IS. A block is a scope. While it runs, the steps inside it read what they always would.
The moment it closes, everything it held is put out of reach — except ONE step, named when the block
opened. That step is the block's RETURN. From then on, the only step of the block any later step can
reach is the return.

So a block is the right shape for work whose CONCLUSION is what the rest of the trace needs, where
the working itself was scaffolding — and where that working takes MORE THAN ONE STEP:

- a verification or a sanity check that runs to several steps: afterwards, all that matters is
  whether it held;
- a sub-derivation that produces one quantity: afterwards, all that matters is the quantity;
- an exploration of a case, a candidate or an approach that was ruled in or out: afterwards, all that
  matters is the verdict;
- a false start that was corrected: afterwards, all that matters is the correction.

THE READS ARE FIXED. Each step lists `read later by:` — every later step that reads it. A block may
only retire a step that NO step after the block reads. So for a run to be a block, every step of the
run that is `read later by` a step outside the run must be the block's return, and there can be only
one such step. If two steps of the run are read from outside it, the run is not a block.

THE RULES A BLOCK MUST KEEP:

1. It is a run of AT LEAST TWO consecutive steps. A block of one step hides nothing.
2. Its return is one of its own steps, and it is the only step of the run read from outside the run.
3. Blocks nest or are disjoint. A block inside another may not return a step the outer block needs
   to return.
4. A CALL moves as one piece. The steps marked `call k of f` belong to one call of the procedure `f`:
   a block contains all of a call's steps or none of them, and a block never sits inside a call.
5. No block contains the step marked `(the trace's answer)`. The trace ends by returning that step
   from the outermost scope, which can return only a step of its own scope, never one inside a block.

Most traces have a few natural blocks; some have none. Do not scope a run just because it is
allowed — scope it because its working is scaffolding the rest of the trace does not need.

Answer with a JSON object and nothing else:

{"blocks": [{"start": "<first step id>", "end": "<last step id>", "return": "<step id>"}, ...]}

`{"blocks": []}` when no run is worth scoping.

EXAMPLES. Three short traces in the input format, with the answer for each.

--- example 1: two calls, each inside the sub-derivation it finishes, and a check ---

PROBLEM: A bakery bakes 3 trays of 12 muffins in the morning and sells 30, then bakes 2 more trays in the
  afternoon and sells 20. How many muffins are left at closing?

[0] tray
    task: Read how many muffins a tray holds
    reads: root:val
    work: Each tray holds 12 muffins.
    value: 12
    read later by: morning_left__baked (val), afternoon_left__baked (val)

[1] morning_trays
    task: Read the morning's trays
    reads: root:val
    work: The morning bakes 3 trays.
    value: 3
    read later by: morning_left__baked (val)

[2] morning_left__baked   (call 1 of session, role `baked`)
    task: Multiply the trays by the muffins per tray
    reads: morning_trays:val, tray:val
    work: 3 trays of 12 is 36 muffins.
    value: 36
    read later by: morning_left (val)

[3] morning_left   (call 1 of session, role `left`)
    task: Subtract the muffins sold
    reads: morning_left__baked:val, root:val
    work: 36 - 30 = 6 left after the morning.
    value: 6
    read later by: total_left (val)
```

```
[4] afternoon_trays
    task: Read the afternoon's trays
    reads: root:val
    work: The afternoon bakes 2 trays.
    value: 2
    read later by: afternoon_left__baked (val)

[5] afternoon_left__baked   (call 2 of session, role `baked`)
    task: Multiply the trays by the muffins per tray
    reads: afternoon_trays:val, tray:val
    work: 2 trays of 12 is 24 muffins.
    value: 24
    read later by: afternoon_left (val)

[6] afternoon_left   (call 2 of session, role `left`)
    task: Subtract the muffins sold
    reads: afternoon_left__baked:val, root:val
    work: 24 - 20 = 4 left after the afternoon.
    value: 4
    read later by: total_left (val)

[7] total_left
    task: Add what is left from both sessions
    reads: morning_left:val, afternoon_left:val
    work: 6 + 4 = 10.
    value: 10
    read later by: verify (val), answer (val)

[8] recheck
    task: Recount the day's baking and sales in one go
    reads: root:val
    work: 36 + 24 = 60 baked and 30 + 20 = 50 sold over the day.
    value: 60 baked, 50 sold
    read later by: verify (val)

[9] verify
    task: Check the total against the day's count
    reads: recheck:val, total_left:val
    work: 60 - 50 = 10, the same as 6 + 4.
    value: 10 checks out
    read later by: answer (val)

[10] answer   (the trace's answer)
    task: State the answer
    reads: total_left:val, verify:val
    work: \\boxed{10} muffins are left.
    value: \\boxed{10}
    read later by: nothing

{"blocks": [{"start": "morning_trays",   "end": "morning_left",   "return": "morning_left"},
            {"start": "afternoon_trays", "end": "afternoon_left", "return": "afternoon_left"},
            {"start": "recheck",         "end": "verify",         "return": "verify"}]}

Each session is a sub-derivation that produces one number. `morning_trays` and `morning_left__baked`
are read only inside `morning_trays` .. `morning_left`, and the rest of the trace reads `morning_left`
alone, so the run is a block returning `morning_left`. It holds call 1 whole. The afternoon is the same
shape. The check is scaffolding too: `recheck` is read only by `verify`, and `answer` reads only the
verdict, so the pair is a block returning `verify`.

What would be refused, and why:

- `tray` .. `morning_left`: `afternoon_left__baked` reads `tray` after that block has closed.
- `morning_left__baked` .. `morning_left`: that run is call 1 itself, and a block never sits inside a
  call.
- `morning_left` .. `afternoon_trays`: it takes the second step of call 1 without the first.
- `total_left` .. `verify`: `answer` reads `total_left`, and the block could return only one of
  `total_left` and `verify`.

--- example 2: a block nested inside a block ---

PROBLEM: A rectangle has perimeter 34 and area 60. How long is its diagonal?

[0] setup
    task: Name the sides and write the two conditions
    reads: root:val
    work: Let the sides be a and b; then a + b = 17 and ab = 60.
    value: a + b = 17, ab = 60
    read later by: quadratic (val), squares (val)

[1] quadratic
    task: Turn the two conditions into one equation
```

```
    reads: setup:val
    work: a and b are the roots of x^2 - 17x + 60 = 0.
    value: x^2 - 17x + 60 = 0
    read later by: disc (val), roots (val)

[2] disc
    task: Take the discriminant
    reads: quadratic:val
    work: 17^2 - 4*60 = 289 - 240 = 49.
    value: 49
    read later by: discchk (val)

[3] discchk
    task: Confirm the discriminant is a perfect square
    reads: disc:val
    work: 49 = 7^2 exactly, so its square root is 7.
    value: 7
    read later by: roots (val)

[4] roots
    task: Solve for the two sides
    reads: quadratic:val, discchk:val
    work: x = (17 +/- 7) / 2, so the sides are 12 and 5.
    value: 12 and 5
    read later by: squares (val)

[5] squares
    task: Add the squares of the sides
    reads: setup:val, roots:val
    work: a^2 + b^2 = 144 + 25 = 169.
    value: 169
    read later by: answer (val)

[6] answer   (the trace's answer)
    task: Take the root and state the answer
    reads: squares:val
    work: sqrt(169) = 13, so the diagonal is \\boxed{13}.
    value: \\boxed{13}
    read later by: nothing

{"blocks": [{"start": "quadratic", "end": "roots",   "return": "roots"},
            {"start": "disc",      "end": "discchk", "return": "discchk"}]}

`quadratic` .. `roots` turns the two conditions into the two sides. Every step of it except `roots` is
read only inside the run, and `squares` reads `roots`, so the run is a block returning `roots`. Inside
it, `disc` .. `discchk` establishes that the discriminant's root is exactly 7. `disc` is read only by
`discchk`, and `roots` reads `discchk` after the inner block has closed but while the outer one is
still open, so the inner block returns `discchk` and closes first.

What would be refused, and why:

- `setup` .. `roots`: `squares` reads `setup` after that block has closed.
- `disc` .. `roots` as the inner block: it would return `roots`, the step the outer block returns, and
  a block's return must sit in the block's own scope rather than in a block nested inside it.

--- example 3: nothing to scope ---

PROBLEM: Tom has 5 red marbles and 7 blue ones and gives away 4. How many does he have left?

[0] red
    task: Read the red marbles
    reads: root:val
    work: Tom has 5 red marbles.
    value: 5
    read later by: total (val), answer (val)

[1] blue
    task: Read the blue marbles
    reads: root:val
    work: Tom has 7 blue marbles.
    value: 7
    read later by: total (val), answer (val)

[2] total
    task: Add the two colours
    reads: red:val, blue:val
    work: 5 + 7 = 12 marbles.
    value: 12
    read later by: answer (val)

[3] answer   (the trace's answer)
```

```
    task: Subtract the marbles given away and state the answer
    reads: red:val, blue:val, total:val
    work: 5 red and 7 blue make 12, and 12 - 4 = 8, so \\boxed{8}.
    value: \\boxed{8}
    read later by: nothing

{"blocks": []}

`answer` reads `red`, `blue` and `total` itself, so every run of two or more steps before it has two
steps read from outside the run: `red` .. `blue` loses `red` to `total`, `blue` .. `total` loses `blue`
to `answer`, and `red` .. `total` loses `red` and `blue` to `answer`. No block is possible, and the
empty list is the answer.
```

D Training Methodology

D.1 Dataset

We construct the CRM ASSEMBLY SFT dataset by using an LLM annotator model to annotate CoT reasoning traces from the Nemotron Cascade SFT dataset (Wang et al., 2025a). We subsample traces within max sequence length $C = 4096$ from the dataset and annotate each trace with CRM ASSEMBLY primitives in a series of annotation passes. The system prompts are in Appendix C. We filter for trace validity (syntax, in-scope variable references, missing arguments, empty block scopes). We used GPT-6-Sol (OpenAI, 2026) as the annotator model to get a final pool of 6372 CRM ASSEMBLY traces.

D.2 Supervised Finetuning

We finetune a base transformer model on the CRM ASSEMBLY SFT dataset to produce the CRM model. The finetuning process must train the model with same attention mask as determined by the CRM ASSEMBLY primitives at inference time. The single forward pass during standard finetuning does not suffice. Given a sequence of tokens S_1 that is visible for some targets S_2 in the trace and then invisible for subsequent targets S_3, a standard forward pass would reuse the KV-cache computed for S_2 while S_1 is visible for S_3. However, this is incorrect as S_3 is not supposed to attend to S_1 according to the semantics of the CRM ASSEMBLY primitives. We construct the training rows and attention masks to ensure the independency fully hold.

The naive implementation is to finetune the model with each distinct computational reasoning graph view as a single training example. This results in slow training as each trace now makes additional forward passes on prefixes that are shared. Instead, for efficient training, the training pipeline breaks the trace into segments that are always attend causally to each other, constructs a trie with each segment as a node, sharing ancestors only when the prefixes match, resulting in distinct paths for each unique prefix. Reading off of the trie, the nodes are laid out with a depth-first preorder traversal of the trie, where there may be multiple copies of a segment if it is an ancestor to segments attending to different prefixes before the segment. Segment copies are not loss targets. The pipeline also constructs the attention mask such that nodes do not attend to non-ancestors in the training row.

This layout results in longer training rows, requiring larger training context window sizes, but the inference context window sizes do not change. We subsample the dataset for $D_{\text{SFT}} = 2000$ traces to SFT on, stratified to keep the same ratio of problem domain as in the original dataset and filtered for problems with max sequence length $K = 8196$ when laid out in training rows.

D.3 Preference Optimization

From the CRM model, we apply preference optimization using BoNBoN alignment (Gui et al., 2024) to improve accuracy. We sample $D_{\text{BonBon}} = 500$ problems from the Nemotron Cascade SFT dataset, excluding the problems in the CRM ASSEMBLY dataset, as the training dataset. The BoNBoN objective requires the selection of the best and worst performing of the G completions. We use LLM-as-a-Judge (Zheng et al., 2023) to rank the G completions given each sample's correctness and grading rubric (Appendix D.5).

D.4 HYPERPARAMETERS

All finetuned models used batch size 8, 3 epochs, learning rate 3×10^{-5} with cosine decay to 3×10^{-6} and 3% warmup, and seed 42. For BoNBoN, we sample $G = 8$ completions, with temperature 0.6, Top-P 0.95, batch 4, seed 42, and 2 rounds. CoT models use 4096 context length for both training and inference, because the training dataset only contained 4096 length training instances. The CRM models use max sequence length 4096 during inference and 8192 during training, due to the custom training row layout. Evaluation used decoding batch size 1.

We use the recommended sampling temperature of 0.7, TopP 0.8, TopK20 for `Qwen3-4B-Instruct` and temperature 0.6, TopP 0.95, TopK20 `Qwen3-4B-Thinking` For the finetuned and preference optimized models, we performed a temperature sweep and selected the best temperature for each (Appendix E.1).

D.5 PREFERENCE OPTIMIZATION JUDGE PROMPT

D.5.1 COT

```
You are ranking {n} attempts at the same maths problem, written by the same model.
Put them in order, BEST FIRST. Rank 1 is the best attempt, rank {n} the worst.
Each attempt is labelled CORRECT or INCORRECT. That has already been checked against the reference
answer — take it as given and do not re-verify the arithmetic.
Each attempt is plain chain-of-thought prose: the model reasons in free text and ends by writing
its final answer in \boxed{}. There is no markup to interpret.
What to weigh, in rough order of importance:
  - whether it got the right answer
  - conciseness — did it get there without wandering, repeating itself, or re-checking what it
    had already settled
  - soundness — does each step follow from what came before, or did it reach its answer through a
    guess, an unjustified leap, or an error it happened to cancel later
  - completeness — does it finish, stating one final answer clearly
Use your own judgement beyond those. If something else about an attempt strikes you as making it
better or worse — a cleaner line of attack, a fragile step that happened to work, a dead end it
recovered from — let it count. The four above are where to start, not the whole of what matters.
You must produce a strict order: every attempt appears exactly once, no ties. When two are genuinely
hard to separate, pick one and say so briefly.
Return one JSON object and nothing else:
{"order": [<attempt numbers, best first, all {n} of them>], "why": "<one or two sentences on what decided the
   top and the bottom>"}
```

D.5.2 CRM

```


Put them in order, BEST FIRST. Rank 1 is the best attempt, rank {n} the worst.

Each attempt is labelled CORRECT or INCORRECT. That has already been checked against the reference
answer — take it as given and do not re-verify the arithmetic.

# 1. The trace format

The trace is written in a scoped annotation language. You need five tags:

- `<step id="X" task="...">` DECLARES a step. `task` is written BEFORE any work
  for that step exists.
- `<work id="X"></work>` opens step X's BODY. The free text that follows, up to
  the next `<ret>`, is the body: scratch work, visible to no later step.
- `<ret value="...">` is step X's RETURN. **This is the only thing that survives
  to later steps.** Everything in the body is discarded.
- `<load ids="X:val,Y:val">` acquires handles. A step can only read the values of
  ids that were loaded before its body opened. Loads only ever ADD.
- `<push>` opens a scope; `<pop id="X">` closes it. A pop RESTORES the acquisition
  state saved at its matching push and re-exports ONLY X's value. Every other
  value produced inside that scope becomes permanently unreachable.

`<fn>` / `<call>` / `<endcall>` are a function facility. `<call>` arguments may
only be step ids, never literals, so a step whose sole purpose is to bind a
constant for a later call legitimately has an empty body.
Two attempts at: "Two cars leave together at 55 and 70 mph. After 3 hours, how far apart?"

ATTEMPT A
<step id="slow" task="Distance for the slower car"></step>
<work id="slow"></work>55 * 3 = 165 miles.<ret value="165 miles"></ret>
<step id="fast" task="Distance for the faster car"></step>
```

```
<work id="fast"></work>70 * 3 = 210 miles.<ret value="210 miles"></ret>
<step id="gap" task="Difference between the two distances"></step>
<load ids="slow:val,fast:val"></load>
<work id="gap"></work>210 - 165 = 45.<ret value="45 miles"></ret>

ATTEMPT B
<step id="setup" task="Understand the problem"></step>
<work id="setup"></work>Two cars, different speeds, same time. We want a gap.<ret value="a distance
  problem"></ret>
<step id="slow" task="Distance for the slower car"></step>
<work id="slow"></work>55 * 3 = 165 miles.<ret value="165 miles"></ret>
<step id="fast" task="Distance for the faster car"></step>
<work id="fast"></work>70 * 3. Let me see, 70 * 3 = 210. Yes, 210 miles.<ret value="210 miles"></ret>
<step id="gap" task="Difference between the two distances"></step>
<load ids="slow:val,fast:val"></load>
<work id="gap"></work>210 - 165 = 45.<ret value="45 miles"></ret>

Both reach 45. A is the better trace, and the reasons are structural, not stylistic:

- `setup` in B is NOT NECESSARY. Nothing ever loads `setup:val` — no later `<load ids="...">`
  names it — so its return reaches no other step and the step does no work for the trace. Whether
  its text sounds sensible is beside the point; look at what is loaded.
- `fast` in B is not CONCISE. The body wanders ("Let me see... Yes") where A's states the
  arithmetic once. Bodies are discarded at the `<ret>`, so length there buys nothing.
- Both are COMPLETE in the same way: every declared step's `<ret>` delivers what its `task`
  promised, and `gap` loads exactly the two values it needs.

A trace can also fail completeness while looking finished — a `<step>` whose `task` promises a
result and whose `<ret value="...">` hands back a restatement, a plan, or nothing usable. And a
`<push>` ... `<pop id="X">` that discards a value a later step then tries to load is a real
structural error, not a stylistic one.

What to weigh, in rough order of importance:

  - whether it got the right answer
  - conciseness — did it get there without wandering, and without steps that do no work
  - necessity — is every step's return actually loaded by a later step, or does it dangle
  - completeness — does each step's `<ret>` deliver what its `task` promised, and does the trace
    finish

Use your own judgement beyond those. If something else about an attempt strikes you as making it
better or worse — a cleaner decomposition, a fragile step that happened to work, a dead end it
recovered from, a scope used well — let it count. The four above are where to start, not the whole
of what matters.

You must produce a strict order: every attempt appears exactly once, no ties. When two are genuinely
hard to separate, pick one and say so briefly.

Return one JSON object and nothing else:

{"order": [<attempt numbers, best first, all {n} of them>], "why": "<one or two sentences on what decided the
  top and the bottom>"}
```

E Additional Evaluation Results

E.1 Temperature Sweep

For each model, we sweep across temperature $T = \{0.25, 0.5, 0.75, 1.0\}$ with TopP 0.95, TopK 20, using a $N = \min(\text{benchmark size}, 50)$ subsampled of problems from each benchmark, and select the temperature with the highest macro avereage of pass@k scores for $k = 3$.

Table 2: Temperature sweep for `CoT`.

T	GSM8K	AIME 2025	MMLU-Pro	GPQA-Diamond	Overall
0.25	**94.0**	**36.7**	**76.0**	**64.0**	**67.7**
0.5	88.0	30.0	74.0	66.0	64.5
0.75	92.0	36.7	76.0	66.0	67.7
1	96.0	33.3	78.0	62.0	67.3

Table 3: Temperature sweep for `CoT-RL`.

T	GSM8K	AIME 2025	MMLU-Pro	GPQA-Diamond	Overall
0.25	96.0	30.0	76.0	66.0	67.0
0.5	96.0	40.0	80.0	60.0	69.0
0.75	**94.0**	**40.0**	**82.0**	**66.0**	**70.5**
1	96.0	36.7	76.0	66.0	68.7

Table 4: Temperature sweep for `CRM`.

T	GSM8K	AIME 2025	MMLU-Pro	GPQA-Diamond	Overall
0.25	**96.0**	**20.0**	**68.0**	**46.0**	**57.5**
0.5	88.0	16.7	68.0	42.0	53.7
0.75	92.0	20.0	62.0	32.0	51.5
1	92.0	10.0	62.0	40.0	51.0

Table 5: Temperature sweep for `CRM-RL`.

T	GSM8K	AIME 2025	MMLU-Pro	GPQA-Diamond	Overall
0.25	**96.0**	**30.0**	**76.0**	**60.0**	**65.5**
0.5	90.0	16.7	74.0	64.0	61.2
0.75	94.0	23.3	70.0	60.0	61.8
1	96.0	20.0	74.0	56.0	61.5

E.2 Additional Metrics

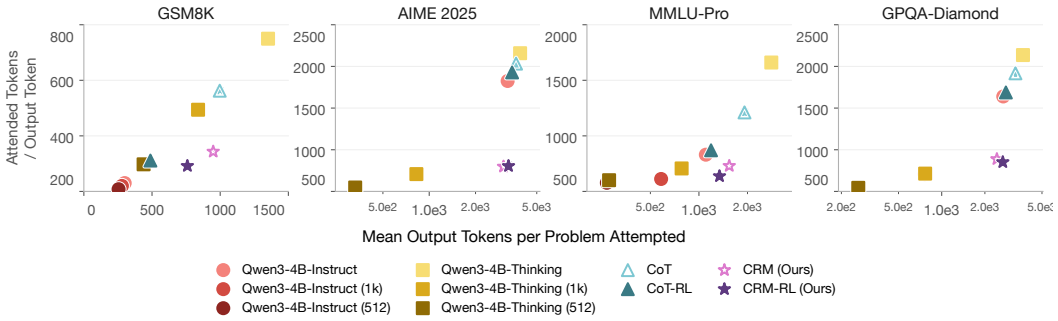


Figure 8: Mean attended tokens per output token vs. mean output tokens per problem.

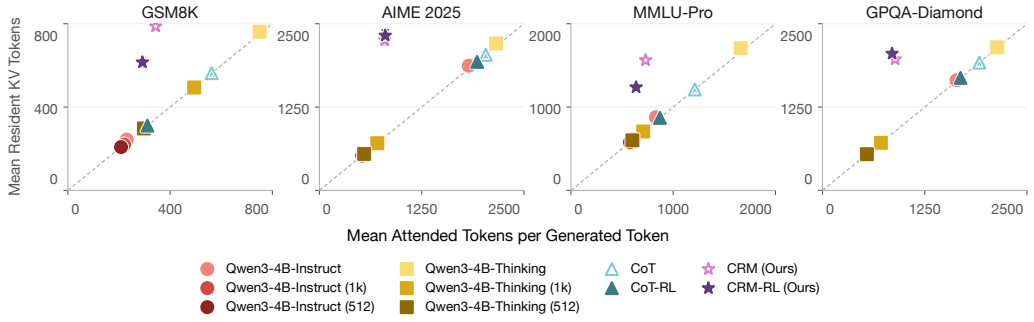


Figure 9: Mean KV tokens in resident per generate token to mean attended tokens per generated token. Fully causal attention models sit on the $y = x$ line.

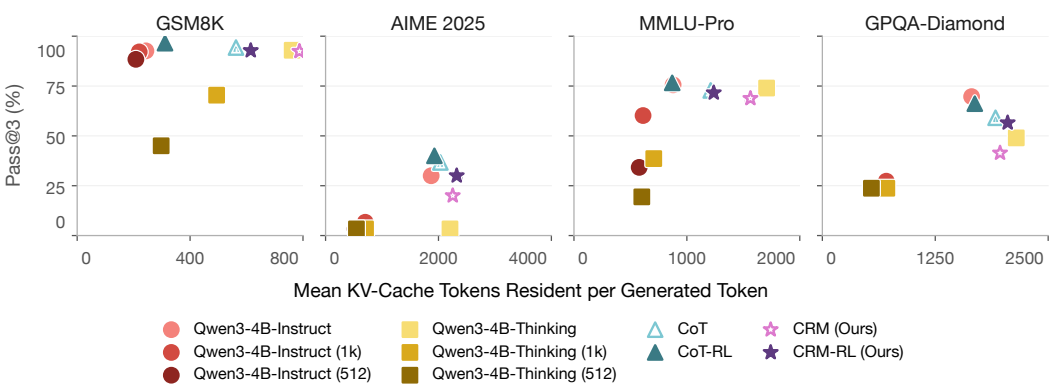


Figure 10: Pass@3 accuracy vs. mean KV-cache tokens resident per generated token.

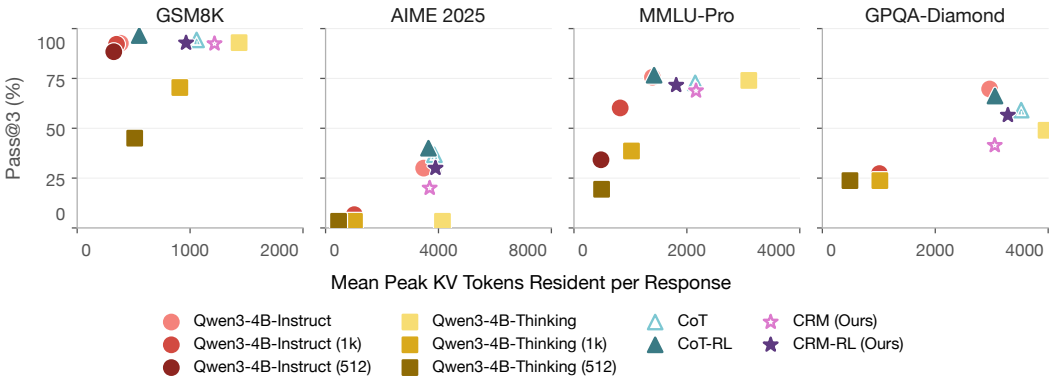


Figure 11: Pass@3 accuracy vs. mean peak KV tokens resident per response.

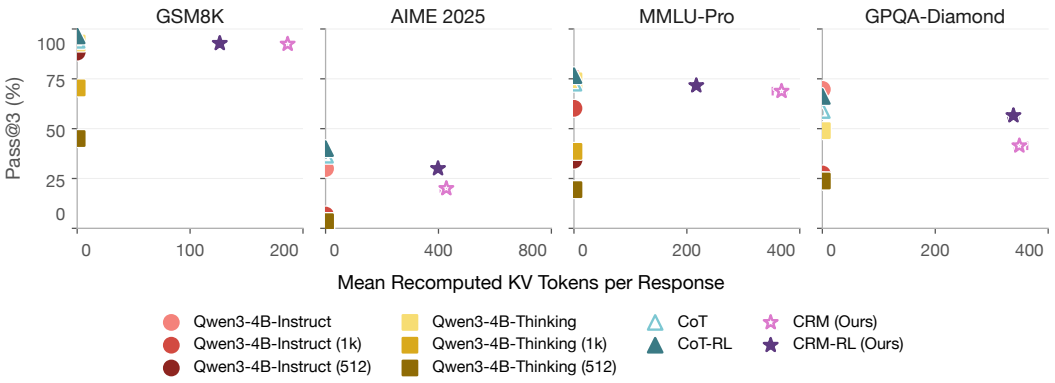


Figure 12: Pass@3 accuracy vs. mean recomputed KV tokens per response.

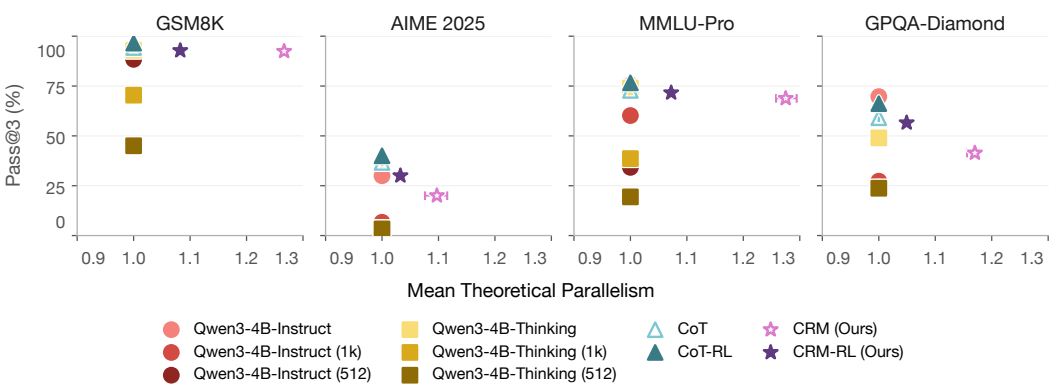


Figure 13: Pass@3 accuracy vs. mean theoretical parallelism.

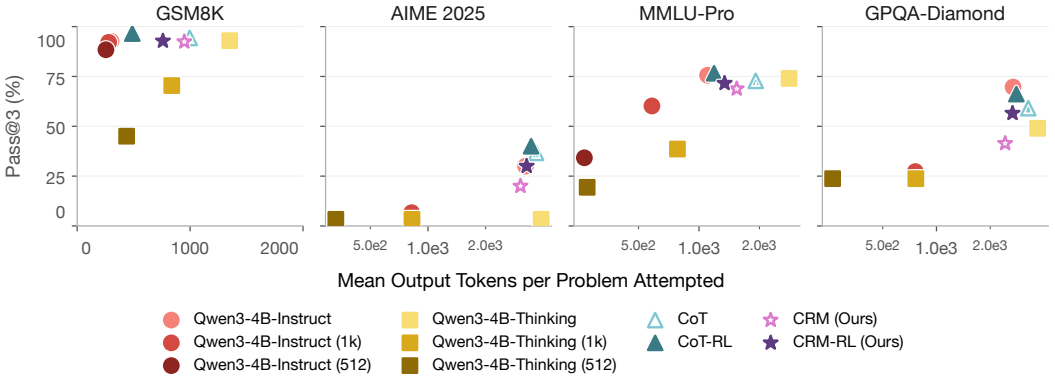


Figure 14: Pass@3 accuracy vs. mean output tokens per problem.

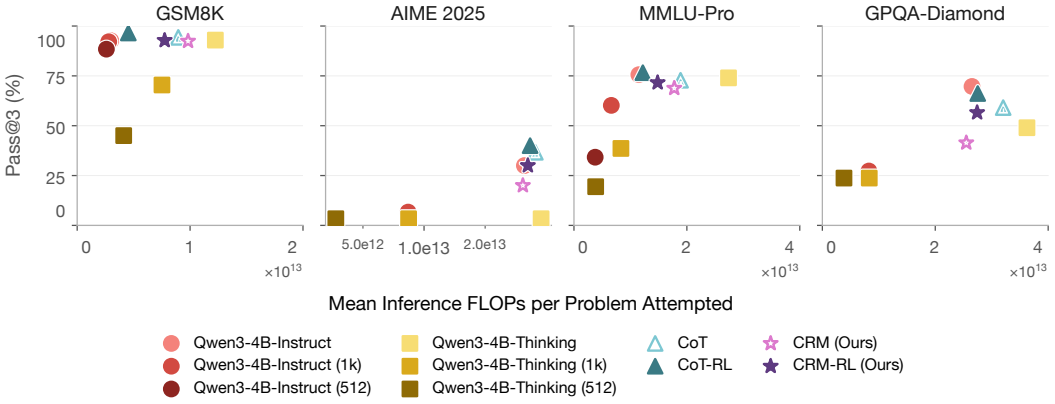


Figure 15: Pass@3 accuracy vs. mean inference FLOPs per problem.

Table 6: Accuracy (%) by model and benchmark.

	GSM8K		AIME 2025		MMLU-Pro		GPQA-Diamond	
	Pass@1	Pass@3	Pass@1	Pass@3	Pass@1	Pass@3	Pass@1	Pass@3
Qwen3-4B-Instruct	89.2	92.6	28.9	30	65.7	75.6	56.6	69.7
Qwen3-4B-Instruct (1k)	88.4	92.1	6.67	6.67	50.3	60.2	25.9	27.3
Qwen3-4B-Instruct (512)	83	88.4	2.22	3.33	22.5	34.2	23.7	23.7
Qwen3-4B-Thinking	87.5	92.9	1.11	3.33	62.1	74	39.9	49
Qwen3-4B-Thinking (1k)	50.9	70.4	3.33	3.33	21.2	38.6	23.7	23.7
Qwen3-4B-Thinking (512)	21.9	45	1.11	3.33	11.3	19.4	23.7	23.7
CoT	90.9	94.3	25.6	36.7	62.1	72.8	47.6	59.1
CoT-RL	94.5	96.5	27.8	40	66.1	76.6	52.5	66.2
CRM (Ours)	80.7	92.4	12.2	20	52.5	68.8	24.6	41.4
CRM-RL (Ours)	85.9	92.8	17.8	30	59.5	71.6	38.7	56.6

Table 7: KV-cache metrics by model and benchmark.

	GSM8K		AIME 2025		MMLU-Pro		GPQA-Diamond	
	Peak KV	KV/Tok.	Peak KV	KV/Tok.	Peak KV	KV/Tok.	Peak KV	KV/Tok.
Qwen3-4B-Instruct	382	243	3,476	1,871	1,390	878	2,963	1,652
Qwen3-4B-Instruct (1k)	348	220	1,019	703	819	612	1,012	708
Qwen3-4B-Instruct (512)	324	208	461	523	479	578	489	536
Qwen3-4B-Thinking	1,434	762	4,141	2,204	3,097	1,706	3,963	2,147
Qwen3-4B-Thinking (1k)	909	494	1,024	708	1,019	707	1,019	714
Qwen3-4B-Thinking (512)	509	298	461	547	489	601	489	543
CoT	1,058	563	3,857	2,034	2,148	1,211	3,523	1,917
CoT-RL	549	311	3,645	1,928	1,420	870	3,058	1,688
CRM (Ours)	1,216	787	3,689	2,253	2,162	1,563	3,053	1,967
CRM-RL (Ours)	964	615	3,895	2,323	1,811	1,238	3,286	2,051

Table 8: Recomputed KV-cache tokens by model and benchmark.

	GSM8K	AIME 2025	MMLU-Pro	GPQA-Diamond
CRM (Ours)	186	428	368	349
CRM-RL (Ours)	126	398	217	338

Table 9: Inference compute metrics by model and benchmark.

	GSM8K		AIME 2025		MMLU-Pro		GPQA-Diamond	
	Gen. Tok.	Attn./Tok.	Gen. Tok.	Attn./Tok.	Gen. Tok.	Attn./Tok.	Gen. Tok.	Attn./Tok.
Qwen3-4B-Instruct	300	230	3,234	1,825	1,104	830	2,686	1,639
Qwen3-4B-Instruct (1k)	279	220	822	703	583	612	761	708
Qwen3-4B-Instruct (512)	255	208	332	523	269	578	264	536
Qwen3-4B-Thinking	1,351	750	3,897	2,159	2,810	1,660	3,686	2,136
Qwen3-4B-Thinking (1k)	838	494	825	708	781	707	767	714
Qwen3-4B-Thinking (512)	438	298	330	547	277	601	262	543
CoT	997	563	3,668	2,034	1,917	1,211	3,266	1,917
CoT-RL	488	311	3,457	1,928	1,189	870	2,801	1,688
CRM (Ours)	949	343	3,040	798	1,542	730	2,425	888
CRM-RL (Ours)	759	292	3,276	805	1,341	637	2,666	851

Table 10: Inference FLOPs by model and benchmark.

	GSM8K	AIME 2025	MMLU-Pro	GPQA-Diamond
Qwen3-4B-Instruct	2.98×10^{12}	3.12×10^{13}	1.15×10^{13}	2.65×10^{13}
Qwen3-4B-Instruct (1k)	2.79×10^{12}	8.35×10^{12}	6.62×10^{12}	8.25×10^{12}
Qwen3-4B-Instruct (512)	2.59×10^{12}	3.68×10^{12}	3.76×10^{12}	3.83×10^{12}
Qwen3-4B-Thinking	1.22×10^{13}	3.77×10^{13}	2.74×10^{13}	3.62×10^{13}
Qwen3-4B-Thinking (1k)	7.51×10^{12}	8.39×10^{12}	8.32×10^{12}	8.31×10^{12}
Qwen3-4B-Thinking (512)	4.12×10^{12}	3.68×10^{12}	3.85×10^{12}	3.83×10^{12}
CoT	8.95×10^{12}	3.54×10^{13}	1.89×10^{13}	3.2×10^{13}
CoT-RL	4.53×10^{12}	3.33×10^{13}	1.22×10^{13}	2.75×10^{13}
CRM (Ours)	9.82×10^{12}	3.07×10^{13}	1.78×10^{13}	2.55×10^{13}
CRM-RL (Ours)	7.75×10^{12}	3.25×10^{13}	1.48×10^{13}	2.74×10^{13}

Table 11: Theoretical parallelism by model and benchmark.

	GSM8K	AIME 2025	MMLU-Pro	GPQA-Diamond
Qwen3-4B-Instruct	1	1	1	1
Qwen3-4B-Instruct (1k)	1	1	1	1
Qwen3-4B-Instruct (512)	1	1	1	1
Qwen3-4B-Thinking	1	1	1	1
Qwen3-4B-Thinking (1k)	1	1	1	1
Qwen3-4B-Thinking (512)	1	1	1	1
CoT	1	1	1	1
CoT-RL	1	1	1	1
CRM (Ours)	1.27	1.1	1.28	1.17
CRM-RL (Ours)	1.08	1.03	1.07	1.05

E.3 Failure analysis

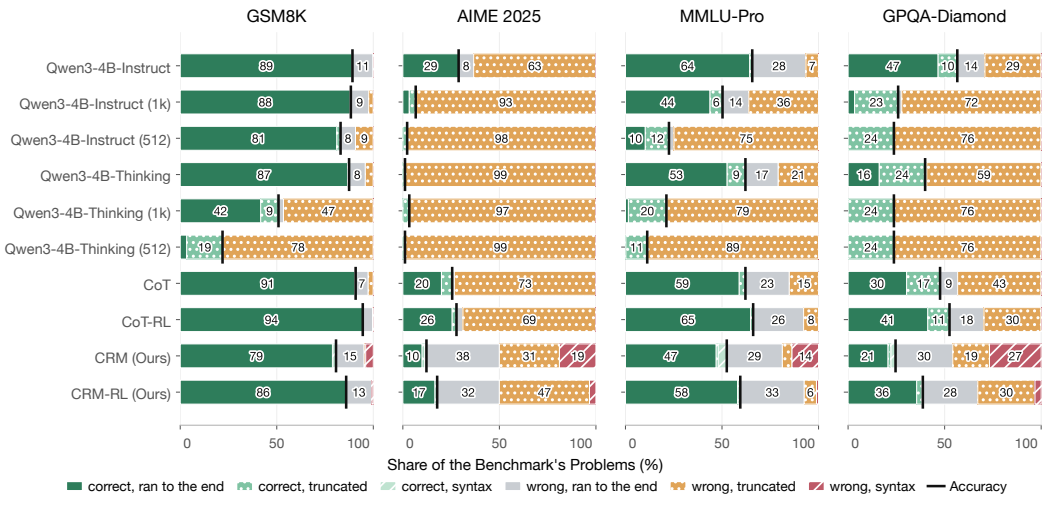


Figure 16: Share of problems by outcome pooling all samples. Each bar splits correct and wrong traces into those that ran to the end, truncated, or failed with a syntax error. The black line marks accuracy.

E.4 Performance across RL Rounds

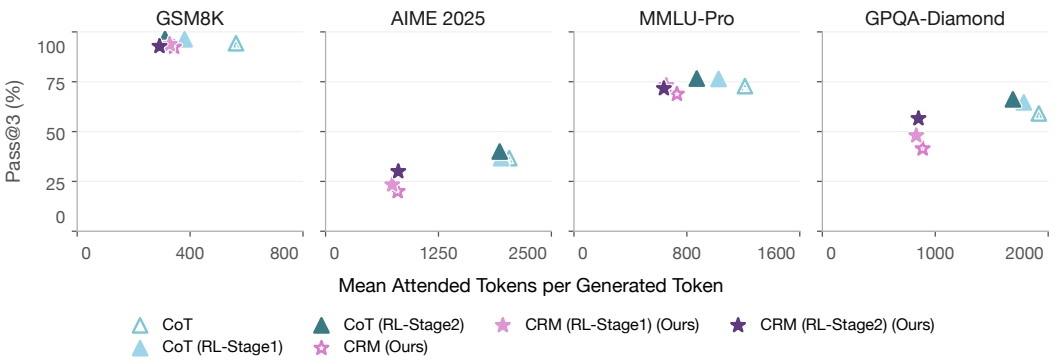


Figure 17: Pass@3 accuracy vs. mean attended tokens per generated token across RL rounds.

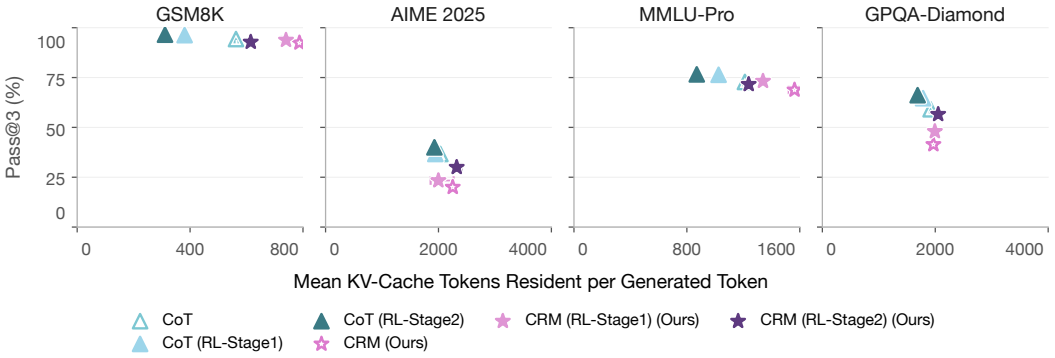


Figure 18: Pass@3 accuracy vs. mean KV-cache tokens resident per generated token across RL rounds.

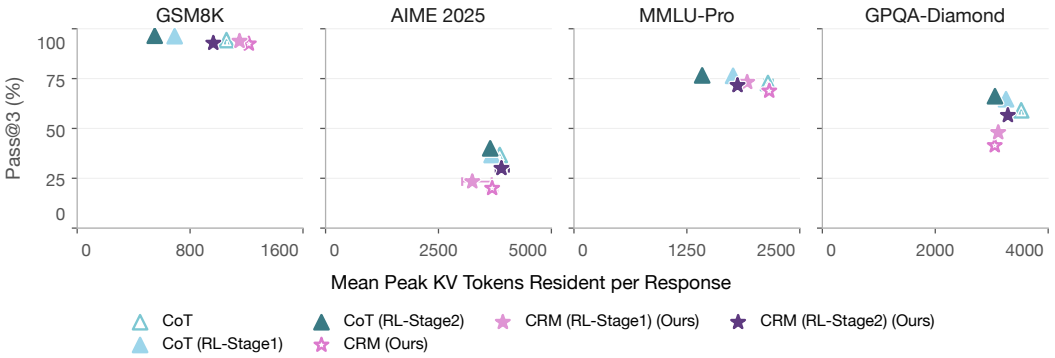


Figure 19: Pass@3 accuracy vs. mean peak KV tokens resident per response across RL rounds.

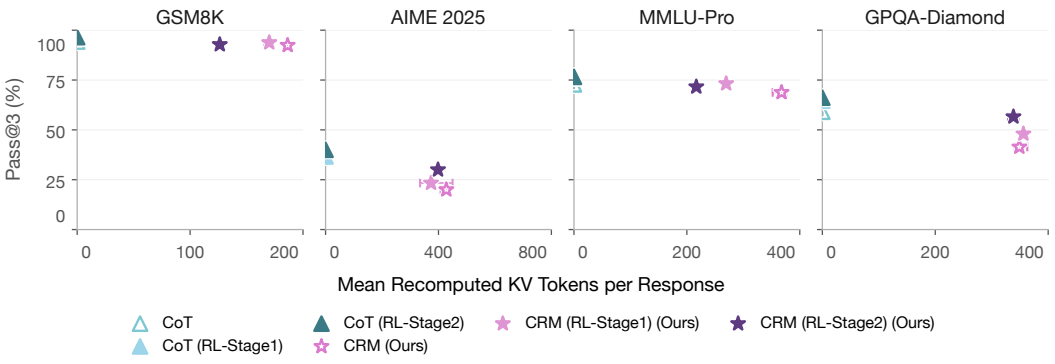


Figure 20: Pass@3 accuracy vs. mean recomputed KV tokens per response across RL rounds.

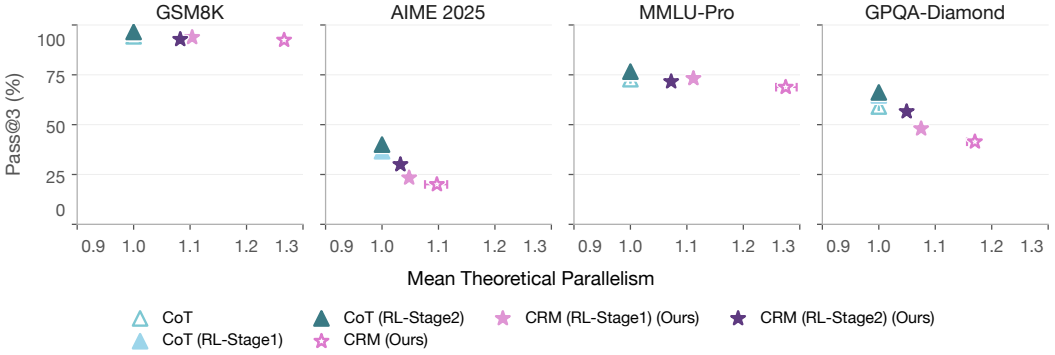


Figure 21: Pass@3 accuracy vs. mean theoretical parallelism across RL rounds.

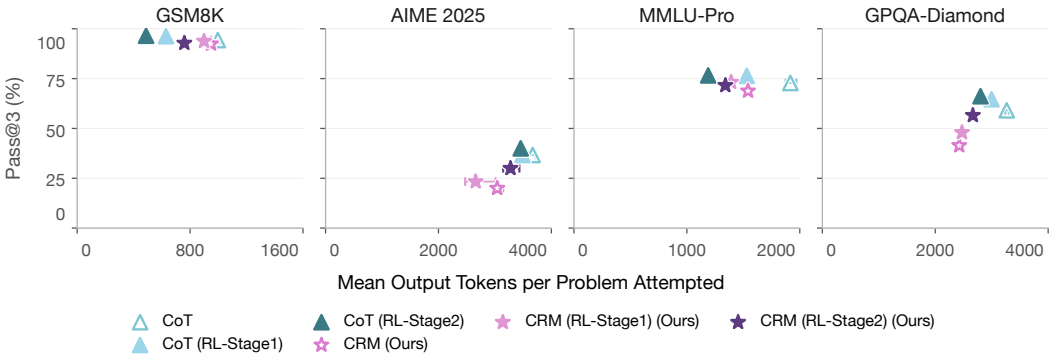


Figure 22: Pass@3 accuracy vs. mean output tokens per problem across RL rounds.

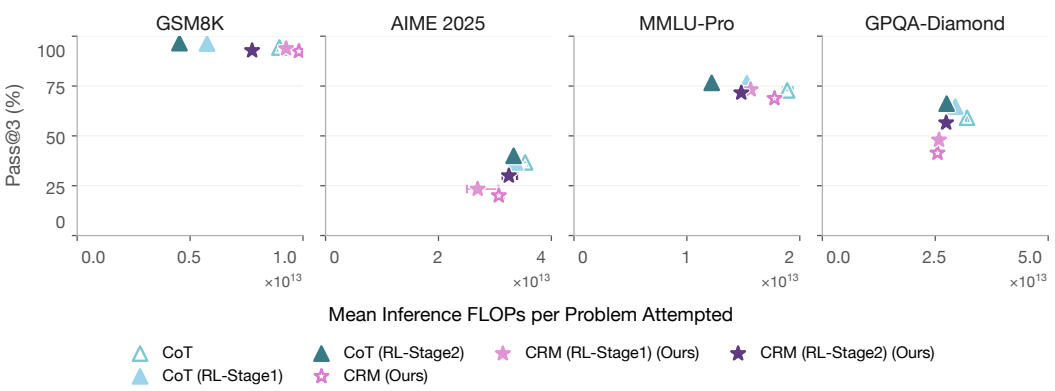


Figure 23: Pass@3 accuracy vs. mean inference FLOPs per problem across RL rounds.

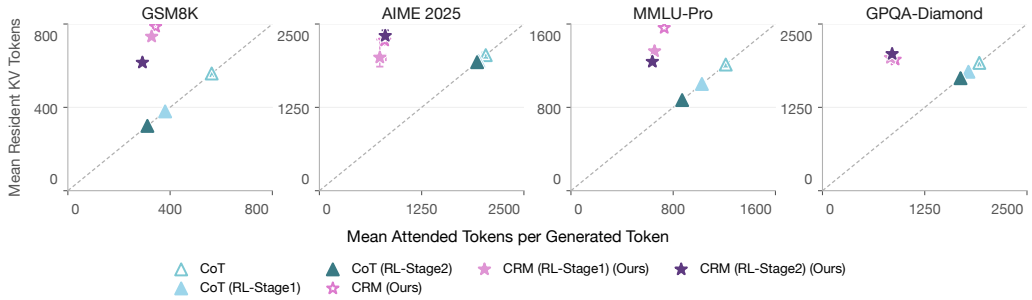


Figure 24: Mean KV tokens resident per generated token vs. mean attended tokens per generated token across RL rounds. Fully causal attention models sit on the $y = x$ line.

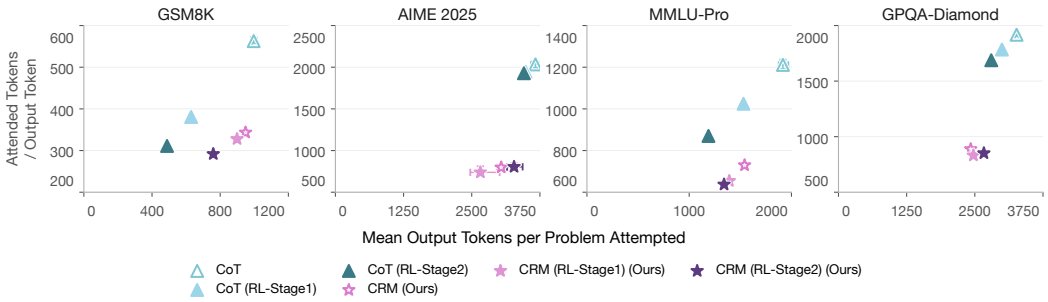


Figure 25: Mean attended tokens per output token vs. mean output tokens per problem across RL rounds.

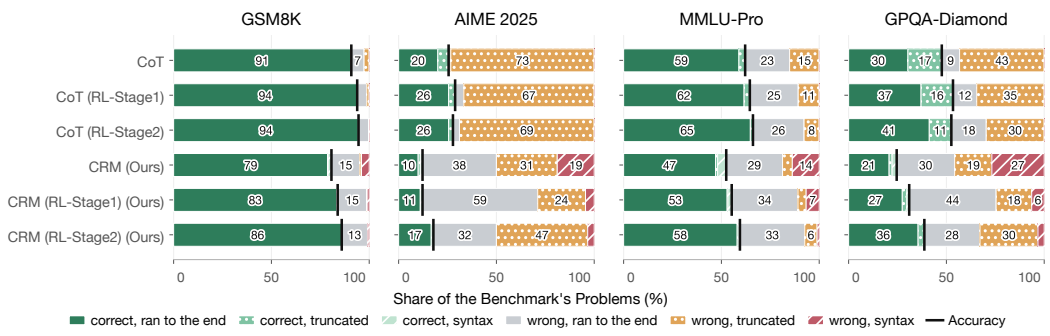


Figure 26: Share of problems by outcome for RL rounds pooling all samples. Each bar splits correct and wrong traces into those that ran to the end, truncated, or failed with a syntax error. The black line marks accuracy.