COMPOSITIONAL SELF-IMPROVEMENT: PROMISES AND PITFALLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Although self-improvement is becoming increasingly important as tasks become more complex, models struggle to generate reliable supervision for harder, out-of-distribution instances. One way to address this challenge is to exploit compositional structure: models often solve simpler subproblems more reliably than the full problem, so their subproblem predictions can be composed into supervision for harder ones. We systematically study when such composed supervision supports self-improvement and when it fails. Our controlled experiments identify three failure modes: unreliable subproblem predictions, errors introduced by composition, and limited supervision coverage. Beyond controlled settings, these findings carry over to more realistic coding and reasoning tasks: composed labels are more accurate than direct self-generated labels and improve performance even on compositions longer than those seen during training, while the failure modes remain visible. Further, although these failure modes often keep compositional self-improvement from fully succeeding on its own, a short SFT warmup with compositional self-improvement can bootstrap reinforcement learning on tasks that require composing skills the model already has, particularly when direct exploration rarely succeeds. Together, these findings characterize when compositional self-improvement can provide sufficiently reliable supervision to expand model capabilities.

1 INTRODUCTION

As models tackle increasingly complex tasks, reliable human supervision can become costly or difficult to obtain. This motivates methods that can improve models with less reliance on human annotations. One approach to *self-improvement* is to learn from self-generated supervision, such as pseudo-labels or model-generated feedback (Lee, 2013; Huang et al., 2023; Lee et al., 2025). However, extending self-improvement beyond the range of problems the model can already solve reliably requires supervision on harder, out-of-distribution instances — precisely where the model's own predictions may be unreliable.

We explore whether compositional structure can bypass this bottleneck. Even when a model cannot solve a complex problem directly, it may still be able to solve its components. This suggests a route to supervision: predictions on these components can be combined into a training target for the full problem. We study this iterative setting as *compositional self-improvement*: the model constructs supervision for harder instances from predictions on simpler components, trains on the resulting targets, and repeats the process as its capabilities expand.

This approach builds on a broader literature using compositional structure to construct training data by recombining existing examples or their constituent parts, usually with the goal of improving generalization to novel compositions (Jia & Liang, 2016; Andreas, 2020; Qiu et al., 2022). More recent work similarly composes existing problems into harder or longer-horizon instances for reasoning evaluation and training (Xiao & Zhao, 2025; Motwani et al., 2025; Lu et al., 2026; Xu et al., 2026). Iterated amplification provides a closer conceptual precedent, constructing supervision for difficult problems from solutions to easier subproblems (Christiano et al., 2018).

Yet prior work leaves open when compositional supervision can sustain genuine self-improvement. Once the component solutions themselves are model-generated, errors can propagate through composition. Moreover, the composition rule can fail even with correct components, while filtering such errors can leave important regions without supervision.

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.

Corresponding to these potential issues, our results reveal three bottlenecks to continued compositional self-improvement: *unreliable component predictions*, *structural errors in composition*, and *limited supervision coverage*. Each bottleneck calls for different remedies. Improving the underlying model reduces errors inherited from the components; repairing or restricting the composition rule addresses errors introduced during composition; and when filtering creates coverage gaps, further progress requires supervision that reaches the missing patterns. Thus, making composed labels more accurate is not by itself sufficient for continued self-improvement.

Our experiments illustrate both the promise and these pitfalls. In controlled experiments on three algorithmic tasks, composed supervision progressively extends reliable prediction beyond the model's initial range, and each bottleneck appears clearly: weaker starting models gain less, invalid composition rules cause improvement to break down, and filtering restores progress but can leave coverage gaps. Beyond these tasks, in more natural settings such as coding and multi-hop reasoning, composed supervision still outperforms direct self-improvement, although some failure modes remain visible. Finally, we show that even imperfect composed supervision can provide a useful warm start for reinforcement learning.

Our contributions are as follows:

1. We develop a framework for analyzing compositional self-improvement, relating the reliability of composed supervision to component errors, composition errors, and supervision coverage.
2. We identify three bottlenecks to continued improvement—*unreliable component predictions*, *structural errors in composition*, and *limited supervision coverage*—and use controlled experiments to systematically study their effects and the limitations of filtering.
3. We show that ***composed supervision can expand the solved regime*** on algorithmic tasks, outperform direct self-improvement in program generation and multi-hop reasoning, and provide a useful warm start for reinforcement learning on compositional tasks.

2 Related Work

Compositional generalization. Compositional generalization concerns handling novel combinations of familiar components (Lake & Baroni, 2018; Keysers et al., 2020; Kim & Linzen, 2020). Prior work studies the roles of architectural biases, training scale, and data coverage (Bahdanau et al., 2019; Redhardt et al., 2025). Least-to-most and decomposed prompting use simpler subproblems to improve predictions at inference time (Zhou et al., 2023; Khot et al., 2023). We examine whether training on composed targets improves direct prediction, including generalization to longer compositions and to cases not directly covered by the composed supervision.

Synthetic data generation using compositionality. Compositional data augmentation constructs new training examples by recombining existing examples or their constituent parts (Jia & Liang, 2016; Andreas, 2020; Akyürek et al., 2021). Qiu et al. (2022) further show that grammar-generated training data can improve predictions beyond the generator's coverage. Recent work similarly composes existing problems into harder instances for curriculum reinforcement learning (Motwani et al., 2025; Xu et al., 2026). A closely related use of compositional structure appears in iterated amplification, which constructs supervised targets for harder problems through decomposition and model-generated subanswers (Christiano et al., 2018). Our work studies a related question in the setting of self-improvement: when predictions on simpler components can provide reliable supervision for harder problems, and what limits the resulting capability expansion.

Self-improvement. Models can improve by training on self-generated solutions or feedback (Zelikman et al., 2022; Huang et al., 2023; Singh et al., 2024). Iterative self-training can extend performance to harder algorithmic instances, although errors in self-generated labels can propagate across rounds (Lee et al., 2025). System 2 distillation trains a model on answers produced by its own multi-step inference procedures to improve direct prediction (Yu et al., 2024). Our study examines how component errors, composition errors, and supervision coverage constrain repeated training on composed predictions. Complementing work on learning compositions of familiar operations through RL (Yuan et al., 2025), we also test composed supervision as an RL warm start.

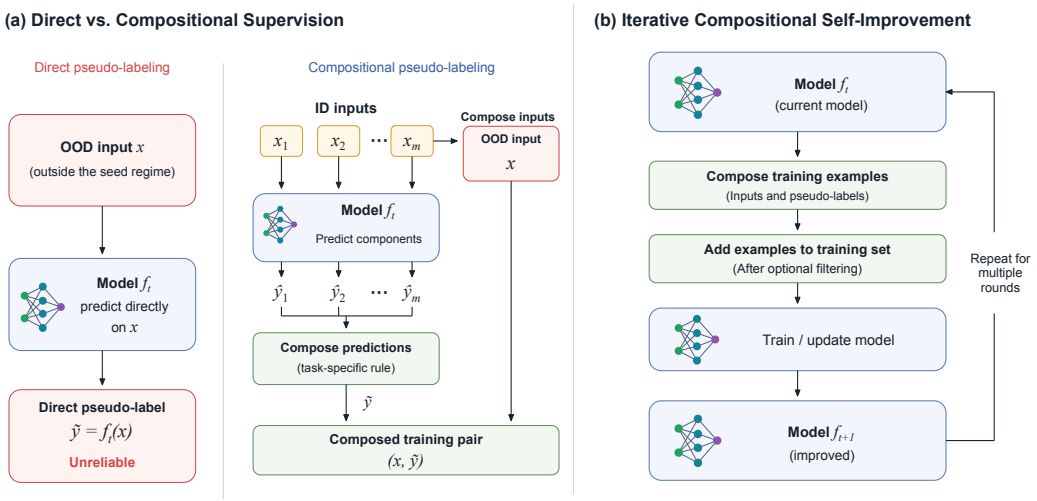


Figure 1: Overview of compositional self-improvement. Direct pseudo-labeling can be unreliable on harder instances. Compositional self-improvement constructs targets by combining predictions on simpler subproblems, filters the resulting examples, and trains the model to solve larger problems directly. Updated predictions provide supervision for subsequent rounds.

3 Understanding Compositional Self-Improvement

We first formalize compositional self-improvement and its training procedure. We then use controlled algorithmic tasks to understand when it works and why it can fail. These tasks let us separately vary component reliability, composition validity, and supervision coverage. After showing that composed supervision can expand the solved regime, we examine how each of these factors can limit further improvement. We connect these findings to theoretical results in Section B.

3.1 Compositional Self-Improvement

Let $f^\star : \mathcal{X} \to \mathcal{Y}$ be the unknown target function and $\hat{f}_0$ an initial model trained on simpler problems, which we call the *seed regime*. For the controlled tasks below, a task-specific level $\ell : \mathcal{X} \to \mathbb{N}$ indexes problem scale, such as string length or the number of rooms. This level describes the task, not the model's accuracy. Compositional self-improvement uses predictions on simpler components to construct supervision for higher-level problems (Figure 1).

Algorithm 1 Compositional self-improvement

Require: initial model $\hat{f}_0$, schedule $\mathcal{S}$, rule A, filter G, rounds T

for $t = 1, \dots, T$ **do**
  $\widetilde{D}_t \leftarrow \emptyset$
  for all (x, π) selected by $\mathcal{S}$ at round t **do**
    obtain $\hat{y}_1, \dots, \hat{y}_m$ by following π with $\hat{f}_{t-1}$
    $\tilde{y} \leftarrow A(\pi, \hat{y}_1, \dots, \hat{y}_m)$
    add $(x, \tilde{y})$ to $\widetilde{D}_t$ if accepted by G
  end for
  train $\hat{f}_t$ on $\widetilde{D}_t$ with task-specific replay
end for
return $\hat{f}_T$

For a parent problem x, a supplied composition plan π specifies its subproblems $x_1, \dots, x_m$ and their order or dependencies. A composition rule A combines their answers and is valid on a composition when

$$A\big(\pi, f^\star(x_1), \dots, f^\star(x_m)\big) = f^\star(x). \tag{1}$$

For dependent plans, this condition refers to execution with correct intermediate answers.

Parent problems can be constructed from available components or paired with supplied decompositions. At round t, the model follows π to obtain predictions $\hat{y}_1, \dots, \hat{y}_m$, using earlier predictions in later queries when required. The candidate target is $\tilde{y} = A(\pi, \hat{y}_1, \dots, \hat{y}_m)$, where aggregation may be performed by a fixed program or a model. An optional filter G retains examples based on their structure or predicted answers, without consulting reference answers for the parent problems.

Table 1: Composition examples for the controlled tasks. Grid routes are joined by the moves through the corridor between rooms, addition concatenates block sums, and max run length keeps the longer run. Filters reject carries across blocks and runs across the join. Blue and purple mark the components; orange marks parts added by the rule.

Task	Component 1	Component 2	Composed parent → label	Filter

Grid navigation	`#######`
	`#.....#`
	`#...#.#`
	`#.....#`
	`#.#..G#`
	`#S...##`
	`#######`
	→ RRURR
		`#######`
		`#....G#`
		`#S....#`
		`###...#`
		`##....#`
		`#.....#`
		`#######`
		→ URRRR
			`#################`
			`#.....###....G#`
			`#...#.#.......#`
			`#.....#.###...#`
			`#.#.....##....#`
			`#S...####.....#`
			`#################`
			→ RRURRRRUURRURRRR
				No filter

Addition	$11 + 22 = 33$	$33 + 44 = 77$	$1133 + 2244 = 3377$	Accept
	$11 + 22 = 33$	$60 + 45 = 105$	$1160 + 2245 = 33105$	Reject
Max run length	`3111` → `1|3`	`022227` → `2|4`	`3111022227` → `2|4`	Accept
(symbol | length of the longest run)	`01111` → `1|4`	`11200` → `1|2`	`0111111200` → `1|4`	Reject

The model trains on retained examples and repeats with its updated predictions (Algorithm 1).

3.2 Experimental Setup

We first study compositional self-improvement on three algorithmic tasks where the composition procedure and sources of error can be controlled. These tasks let us test whether composed supervision can expand the solved regime and separately examine component accuracy, composition errors, and supervision coverage.

Tasks. We consider the tasks of grid navigation, addition, and max run length, in which the model identifies the longest run of repeated symbols in a string and its length (Table 1). For each task, the difficulty level corresponds to the number of rooms, the maximum addend length, and the string length, respectively, with seed ranges of one room, 3–7 digits, and lengths 6–10. The self-improvement schedules extend these ranges, and a round can cover more than one level. The composition rules have properties useful for our analysis. Grid navigation admits an exact composition of component routes. Addition and max run length use simple composition rules that can fail at boundaries unless invalid cases are filtered. Full task definitions appear in Section A.1.

Models and training. All three tasks use 14.2M-parameter character-level Transformers. For each task, we first train a model on the seed regime and then construct supervision for progressively harder instances from predictions on simpler components, with seed-regime replay. We evaluate held-out instances across both the seed and expanded difficulty ranges after each round. For all three tasks, we additionally compare against direct self-improvement, which uses the model's prediction on the full harder instance as its pseudo-label under the same expansion schedule (Section C.1). Full training details appear in Section A.2.

3.3 Compositional Self-Improvement Expands the Solved Regime

Figure 2 shows that **compositional self-improvement expands the solved regime in all three tasks**, while direct self-improvement under the same schedules yields little or no improvement beyond the initial range on addition and grid navigation (Section C.1). Predictions on simpler instances can thus supervise instances beyond the model's reliable range. At the supervision level, our analysis makes this advantage precise: composed labels are more accurate than direct self-labels whenever the combined component and composition errors fall below the model's direct error on the same problems, a simple sufficient condition formalized in Equation (6).

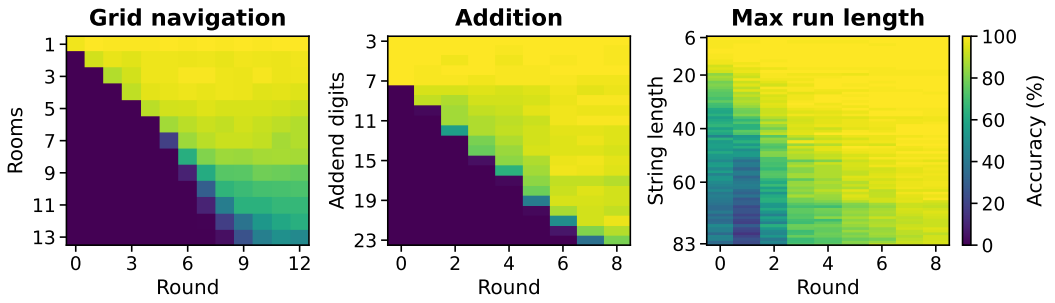


Figure 2: Compositional self-improvement expands the range of instances solved by the model. Each heatmap shows accuracy across self-improvement rounds. From left to right: grid navigation by the number of rooms, addition by addend digit length, and max run length by string length. In all three tasks, reliable prediction extends beyond the initial model's reliable range.

3.4 Failure Modes of Compositional Self-Improvement

While Section 3.3 shows that compositional self-improvement yields consistent gains, these gains are based on three conditions: reliable component predictions, valid composition, and supervision that covers the harder instances. We illustrate how the gains are limited when each condition fails: weaker starting models plateau early in grid navigation, invalid compositions stall progress in addition unless filtered, and in max run length, composed labels never contain runs longer than those of their components, and accuracy on such runs falls sharply.

Failure mode I: Low component accuracy. Compositional self-improvement relies on accurate component predictions to construct reliable supervision for larger instances. We examine this failure mode in grid navigation by varying seed-regime accuracy and applying the same twelve-round schedule to each starting model (Section A.2). Weaker starting models plateau earlier, while stronger models sustain larger gains (Figure 3). Models with initial one-room accuracies of 79.9% and 99.4% reach final mean accuracies of 19.4% and 80.9% on 2–13 rooms, respectively. **Reliable component predictions are therefore crucial for gains by compositional self-improvement**. Our analysis captures this directly: under a valid composition rule, component errors pass directly into the composed label (Proposition B.2).

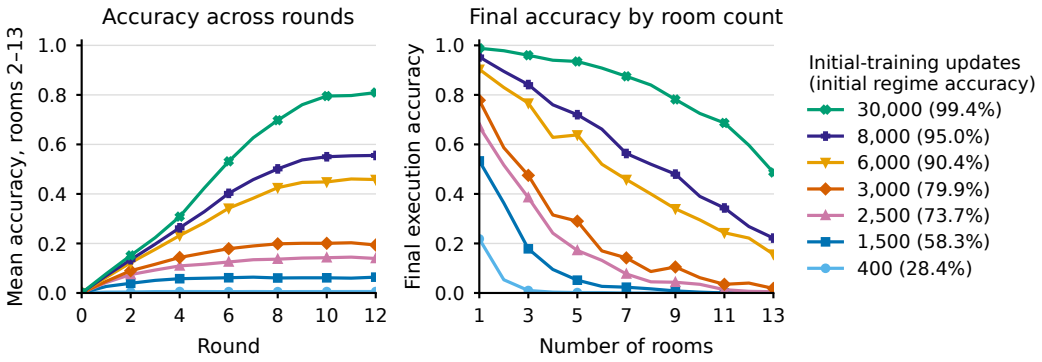


Figure 3: Weaker starting models can limit compositional self-improvement. Left: mean accuracy on 2–13 rooms across rounds. Right: accuracy by room count after the final round. The legend gives initial-training updates and one-room accuracy. Curves average three seeds.

Failure mode II: Structural errors from invalid composition rules. In practice, composition rules designed for self-improvement may be imperfect, introducing errors even when all component predictions are correct. We illustrate this with addition: concatenating blockwise sums can produce incorrect labels when a carry crosses the block boundary. When such failure conditions are known, we can filter out invalid compositions. This removes incorrect labels but also every parent with a carry

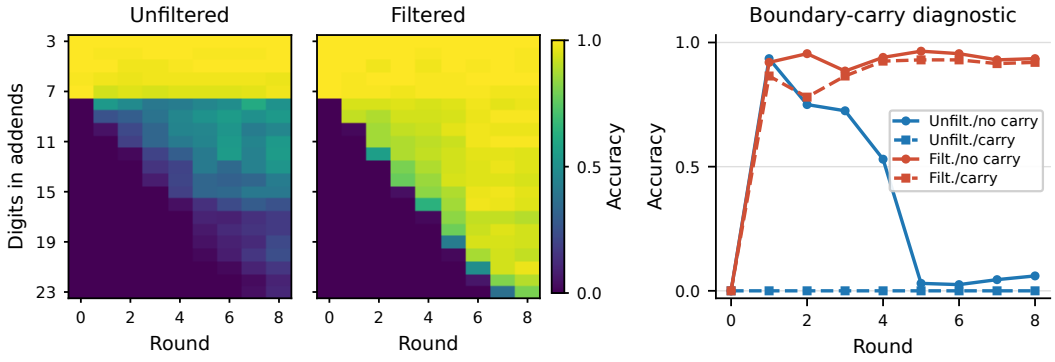


Figure 4: Structural errors undermine self-improvement, while filtering restores gains in addition, including on test sums with boundary carries. Left and middle: accuracy by addend length across rounds without and with filtering. Right: accuracy at each round's newest lengths (up to 22–23 digits), on sums with and without a carry across the block boundary.

across its own block boundary. Figure 4 shows that unfiltered composition degrades after initial gains, while filtering restores self-improvement. The filtered model still solves test sums with boundary carries (Figure 4, right), likely because carries at those positions occur within training blocks.

Failure mode III: Limited supervision coverage. Even when composed labels are correct, they may cover only some of the output patterns that arise in harder instances. In max run length, for example, if we build ten-symbol strings only from pairs of five-symbol strings, the rule can correctly supervise runs of at most five symbols: concatenating `01111` and `11200` requires `1|6`, but the rule produces `1|4`. Filtering removes such errors but leaves longer runs without correct supervision. Thus, although self-improvement achieves high overall max run length accuracy (Figure 2), this aggregate success conceals failure on runs beyond the coverage of composed supervision: accuracy on longer boundary-crossing runs falls sharply, with or without filtering (Figure 5). Our analysis formalizes this limitation: if composition stays within a restricted output space, valid composition cannot directly supervise outputs outside it (Proposition B.4). Unlike the boundary carries in addition, these runs require outputs that no composed label contains, and the model does not extrapolate to them.

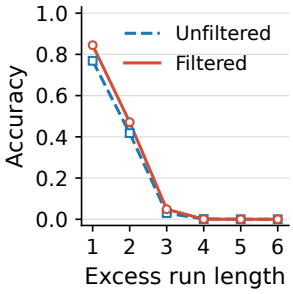


Figure 5: Boundary-crossing accuracy versus excess longest-run length over a same-length reference (Section A.1.3).

4 Compositional Self-Improvement Beyond Controlled Tasks

Next, we analyze compositional self-improvement on two real-world tasks with natural compositional structure: program generation and multi-hop reasoning. In both, the compositional and direct methods label the same parent problems under the same schedule (Table 2).

4.1 Program Generation

Many programs are assembled from smaller programs, making program generation a natural target for composition. We use Bash pipelines, where a pipe passes the output of one command to the next. A multi-step pipeline can therefore be composed from commands predicted for its steps.

Setup. We build pipelines from 119 command variants in TLDR[1] and NL2Bash (Lin et al., 2018). Both methods start from Qwen3.5-2B trained on all 119 individual commands, rehearse them every round, and run three rounds on two-, three-, and four-command pipelines. Compositional self-improvement joins its predicted commands for the individual steps with pipes, while direct

[1] `https://github.com/tldr-pages/tldr`

Table 2: Composition examples for program generation and multi-hop reasoning. Colors follow Table 1. Neither filter uses reference answers.

Task	Component 1	Component 2	Composed parent → label	Filter
Program generation	Output the first 5 lines → `head -5`	Sort in descending order → `sort -r`	Step 1: output the first 5 lines. Step 2: sort in descending order. → `head -5 | sort -r`	Parses as a pipeline
Multi-hop reasoning	Who issued the Royal Proclamation of 1763? → King George III	What year did King George III die? → 1820	What year did the person who issued the Royal Proclamation of 1763 die? → 1820	Answer format; answer found in the passages

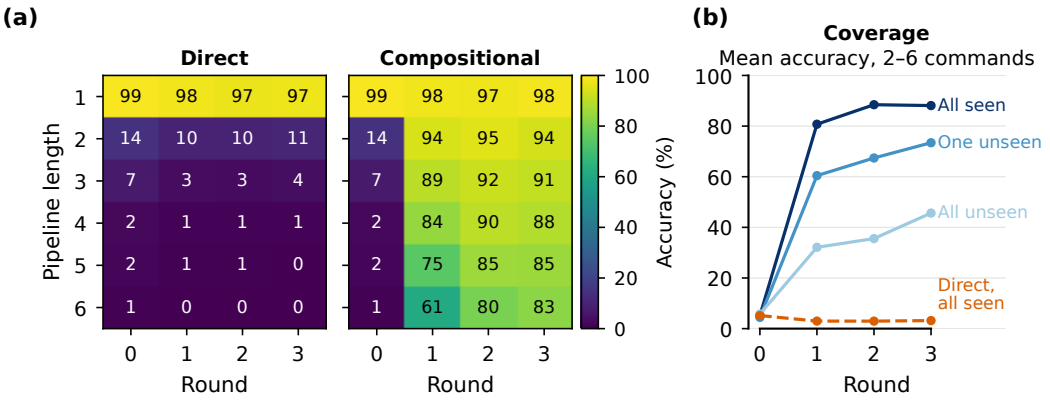


Figure 6: Compositional self-improvement improves program generation, with less transfer to commands absent from training compositions. The initial model is trained on all individual commands; a command is *seen* if it also appears in training compositions. (a) Accuracy of direct and compositional self-improvement on test pipelines of seen commands; length 1 tests single commands. (b) Mean accuracy on 2- to 6-command test pipelines whose commands are all seen, include one unseen command, or are all unseen. Round 0 is the initial model; round r trains on pipelines of length $r + 1$.

self-improvement predicts the full pipeline; both keep a label only if it parses. Beyond comparing the two methods, we examine how coverage affects compositional self-improvement. For this, we exclude a fixed set of 30 commands from training compositions in every round. Note that the model still learns each of them individually during initial training and rehearses them throughout. We evaluate direct generation by execution on one- to six-command test pipelines with no, one, or only held-out commands, averaging four seeds. Details are in Section A.1.4.

Results. Figure 6(a) shows that **composed supervision improves program generation**, including on five- and six-command pipelines, longer than any used for training, while direct self-improvement makes little progress. However, **gains decrease for commands absent from training compositions** (Figure 6(b)), even though the model writes each of them correctly on its own. After three rounds, mean accuracy on 2- to 6-command pipelines is 88% when all commands are seen, 73% with one unseen command, and 46% when all are unseen. This is an analogue of failure mode III: composed labels contain only pipelines of seen commands, which restricts the output space they supervise. Further analyses appear in Sections C.3, C.4 and C.6.

4.2 Multi-hop reasoning

Reasoning often chains simple steps, each using the last one's result. We study this with multi-hop question answering, using the decompositions annotated in MuSiQue. Unlike in program generation, later steps depend on earlier predictions: the model answers the sub-questions in order, substituting each answer into the next, and its final answer becomes the label.

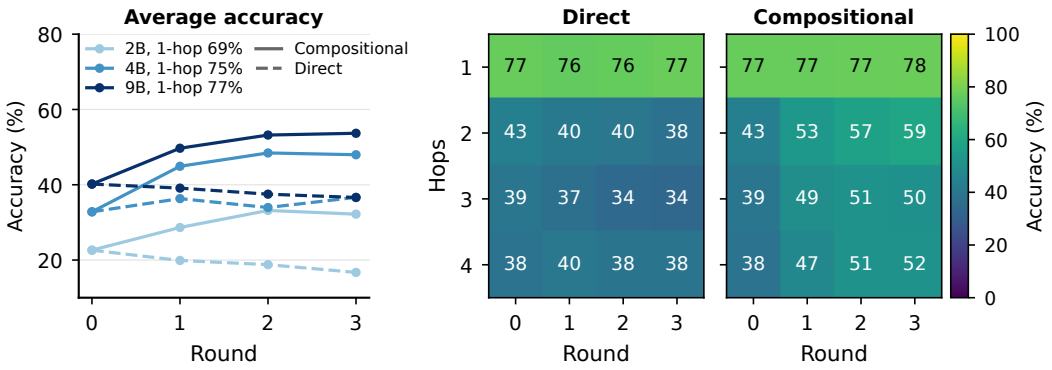


Figure 7: Composed supervision improves multi-hop reasoning on MuSiQue. Left: mean accuracy on 2- to 4-hop questions for all sizes, labeled with each initial model's single-hop accuracy. Middle and right: exact-match accuracy of Qwen3.5-9B by hop count under direct and compositional self-improvement. Round 0 is the initial model, trained on single-hop questions; rounds 1–3 train on 2-, 3-, and 4-hop questions.

Setup. We use MuSiQue (Trivedi et al., 2022), which provides 2- to 4-hop questions with annotated decompositions, and train Qwen3.5-2B, 4B, and 9B to examine how the gains depend on model size. Each model is first trained on single-hop questions with gold answers and then runs three rounds on 2-, 3-, and 4-hop questions, respectively. The compositional method follows the annotated decomposition, while direct self-improvement answers the original question in one step. Filters check the answer format and whether the answer occurs in the passages or the question, without reference answers; the compositional method applies the same checks to its intermediate answers. At evaluation, the model answers each question directly from the provided passages, without decompositions. Results average three seeds; details are in Section A.1.5.

Results. Figure 7 shows that **composed supervision improves multi-hop reasoning** at all three model sizes, while direct self-improvement makes little progress. The gains extend beyond the hop counts used for training: 4-hop accuracy rises during the first two rounds, which train only on 2- and 3-hop questions. **Component reliability limits the gains**, consistent with failure mode I. A composed label is correct only if its sub-answers are, and first-round composed-label accuracy rises from 45% for 2B to 65% for 9B, with most errors arising at the final step, where the model answers from its own intermediate answers. The 2B model, whose components are least reliable, gains least. Additional analyses of label errors appear in Section C.6.

5 Compositional Supervision as an RL Warm Start

Although the failure modes above limit compositional self-improvement on its own, it can still give RL a useful starting point. Yuan et al. (2025) show that RL with outcome rewards can teach a model to compose existing skills, but learning can be extremely inefficient when correct samples are rare. We show that a short SFT warmup stage on composed targets, even imperfect ones, lets RL take off.

Setup. We use string transformations from Yuan et al. (2025) and Python programs composed from type-compatible functions in MBPP+ and HumanEval+ (Austin et al., 2021; Chen et al., 2021; Liu et al., 2023). We train a 14.2M-parameter Transformer on one- and two-operation string chains and Qwen3.5-2B on single-function Python problems, and then train and evaluate on five-operation string chains and three-function Python programs (Table 3). The compositional warmup stage briefly fine-tunes the initial model on composed data before RL; direct RL starts from the initial model. We use GRPO (Shao et al., 2024) with binary outcome rewards. We compare test accuracy against token usage, counting warmup label generation, warmup target tokens, and all RL samples. Additional configurations and details are in Sections A.2 and C.9.

Table 3: Composition examples for the RL warm-start tasks, showing two functions of longer chains.

Task	Component 1	Component 2	Composed parent → label	Filter
String transformations	`func_20(kqz)` → `k-q-z`	`func_11(k-q-z)` → `-q-zk`	`func_11(func_20(kqz))` → `func_20=k-q-z | func_11=-q-zk => -q-zk`	Each step nonempty, length ≤ 96
Typed Python	Last digit → p_1: `return x % 10`	Integer to binary string → p_2: `return bin(x)[2:]`	Step 1: last digit. Step 2: integer to binary string. → `return p2(p1(x))`	Parses as one function

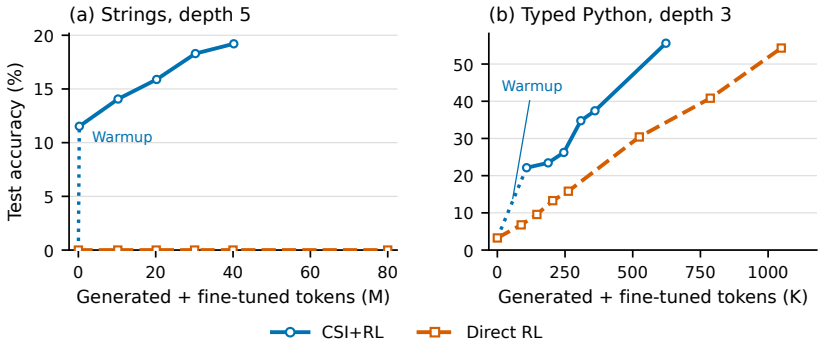


Figure 8: A compositional SFT warmup improves subsequent RL on five-operation string transformations and three-function Python programs. CSI+RL first fine-tunes on labels from compositional self-improvement (CSI), then runs RL, using half the token budget of direct RL. Curves show mean test accuracy over three seeds; dotted segments mark the SFT warmup. Token counts include label generation, RL samples, and warmup targets.

Results. Figure 8 shows that **composed supervision provides an effective warm start for RL**. On five-operation string chains, direct RL never samples a correct composition and receives no reward, even with twice the generation budget, whereas RL after a short compositional SFT warmup improves steadily. On typed Python, where direct RL also makes progress, RL after the warmup reaches 55.6% with half the generation budget, while direct RL needs the full budget to reach 54.3%. Here composition plays a different role from Sections 3 and 4: the composed targets need not be the final training signal, only move the model to a regime where correct compositions are sampled often enough for outcome rewards to guide learning. Nor must they be fully correct; only 77% of the string warmup labels are. More results appear in Sections C.6 and C.9.

6 Conclusion

We studied the promises and pitfalls of self-improvement that exploits the compositional structure of tasks, and our analysis yields three practical lessons. First, teach atomic skills well: composed labels inherit every component error, so capabilities cannot expand on shaky foundations. Second, find where the composition rule breaks. Composition rules used for data augmentation are often only approximately valid (Jia & Liang, 2016; Andreas, 2020; Akyürek et al., 2021), so deciding which composed examples to filter out matters as much as generating them. Third, check which outputs the composed labels can reach. A rule may confine its labels to a restricted output space; filtering cannot widen it, and more rounds do not help. Reaching new outputs requires new components or new rules. Additionally, composed pseudo-labels make a strong warm start for RL on compositional tasks such as coding, potentially enabling RL training on long-horizon tasks. More broadly, our results show that the structure of a task can itself be a source of supervision. Beyond compositionality, tasks and data carry many other kinds of structure, and we hope future work will let models improve themselves by exploiting them as well.

AI USE STATEMENT

We used AI coding and writing assistants (Claude Code, Codex, and Claude) throughout this project. They assisted with implementing experiment, evaluation, and plotting code; designing and constructing the task datasets; developing and checking the theoretical analysis; providing feedback on the methodology and experiments; and drafting and editing the manuscript.

REFERENCES

Ekin Akyürek, Afra Feyza Akyürek, and Jacob Andreas. Learning to recombine and resample data for compositional generalization. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=PS3IMnScugk.

Jacob Andreas. Good-enough compositional data augmentation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7556–7566, 2020. doi: 10.18653/v1/2020.acl-main.676. URL https://aclanthology.org/2020.acl-main.676/.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Dzmitry Bahdanau, Shikhar Murty, Michael Noukhovitch, Thien Huu Nguyen, Harm de Vries, and Aaron Courville. Systematic generalization: What is required and can it be learned? In *International Conference on Learning Representations*, 2019.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Paul Christiano, Buck Shlegeris, and Dario Amodei. Supervising strong learners by amplifying weak experts, 2018. URL https://arxiv.org/abs/1810.08575.

Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. doi: 10.1080/01621459.1963.10500830.

Jiaxin Huang, Shixiang Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

Robin Jia and Percy Liang. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12–22, 2016. doi: 10.18653/v1/P16-1002. URL https://aclanthology.org/P16-1002/.

Daniel Keysers et al. Measuring compositional generalization: A comprehensive method on realistic data. In *International Conference on Learning Representations*, 2020.

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. URL https://arxiv.org/abs/2210.02406.

Najoung Kim and Tal Linzen. COGS: A compositional generalization challenge based on semantic interpretation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, 2020.

Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proceedings of the 35th International Conference on Machine Learning*, 2018.

Dong-Hyun Lee. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In *Workshop on Challenges in Representation Learning*, 2013.

Nayoung Lee, Ziyang Cai, Avi Schwarzschild, Kangwook Lee, and Dimitris Papailiopoulos. Self-improving transformers overcome easy-to-hard and length generalization challenges. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. NL2Bash: A corpus and semantic parser for natural language interface to the Linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 2018. European Language Resources Association (ELRA). URL https://aclanthology.org/L18-1491/.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Yi Lu, Jianing Wang, Linsen Guo, Wei He, Hongyin Tang, Tao Gui, Xuanjing Huang, Xuezhi Cao, Wei Wang, and Xunliang Cai. R-Horizon: How far can your large reasoning model really go in breadth and depth? In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2510.08189.

Sumeet Ramesh Motwani, Alesia Ivanova, Ziyang Cai, Philip Torr, Riashat Islam, Shital Shah, Christian Schroeder de Witt, and Charles London. h1: Bootstrapping LLMs to reason over longer horizons via reinforcement learning, 2025. URL https://arxiv.org/abs/2510.07312.

Linlu Qiu, Peter Shaw, Panupong Pasupat, Pawel Nowak, Tal Linzen, Fei Sha, and Kristina Toutanova. Improving compositional generalization with latent structure and data augmentation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4341–4362. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.naacl-main.323. URL https://aclanthology.org/2022.naacl-main.323/.

Florian Redhardt, Yassir Akram, and Simon Schug. Scaling can lead to compositional generalization. In *Advances in Neural Information Processing Systems*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Avi Singh, John D. Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J. Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron Parisi, Abhishek Kumar, Alex Alemi, Alex Rizkowsky, Azade Nova, Ben Adlam, Bernd Bohnet, Gamaleldin Elsayed, Hanie Sedghi, Igor Mordatch, Isabelle Simpson, Izzeddin Gur, Jasper Snoek, Jeffrey Pennington, Jiri Hron, Kathleen Kenealy, Kevin Swersky, Kshiteej Mahajan, Laura Culp, Lechao Xiao, Maxwell L. Bileschi, Noah Constant, Roman Novak, Rosanne Liu, Tris Warkentin, Yundi Qian, Yamini Bansal, Ethan Dyer, Behnam Neyshabur, Jascha Sohl-Dickstein, and Noah Fiedel. Beyond human data: Scaling self-training for problem-solving with language models. *Transactions on Machine Learning Research*, 2024. URL https://arxiv.org/abs/2312.06585.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. MuSiQue: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022. doi: 10.1162/tacl_a_00475. URL https://aclanthology.org/2022.tacl-1.31/.

Xisheng Xiao and Hanlin Zhao. From a and B to A+B: Can large language models solve compositional math problems? In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 13057–13078. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.660. URL https://aclanthology.org/2025.emnlp-main.660/.

Xin Xu, Clive Bai, Kai Yang, Tianhao Chen, Yangkun Chen, Weijie Liu, Hao Chen, Yang Wang, Saiyong Yang, and Can Yang. Composition-RL: Compose your verifiable prompts for reinforcement learning of large language models, 2026. URL https://arxiv.org/abs/2602.12036.

Ping Yu, Jing Xu, Jason Weston, and Ilia Kulikov. Distilling system 2 into system 1, 2024. URL https://arxiv.org/abs/2407.06023.

Lifan Yuan, Weize Chen, Yuchen Zhang, Ganqu Cui, Hanbin Wang, Ziming You, Ning Ding, Zhiyuan Liu, Maosong Sun, and Hao Peng. From $f(x)$ and $g(x)$ to $f(g(x))$: LLMs learn new skills in RL by composing old ones, 2025.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems*, 2022.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2205.10625.

APPENDIX

Section A defines the tasks and gives the experimental setups, Section B provides the theoretical analysis, and Section C reports additional results and analyses.

A EXPERIMENTAL DETAILS

A.1 TASK DEFINITIONS

For each task, we give the task itself (input, target, and how correctness is scored), the composition rule that builds a parent's label from the model's answers to its components, the filter, and the data. Grid navigation, program generation, string transformations, and typed Python compose exactly; addition and max run length need a filter to exclude parents on which the rule fails. Tables 1 to 3 show an example of each rule, and Figures 9 to 13 show the full model inputs and saved outputs for one training label per task.

A.1.1 GRID NAVIGATION

Task. An instance is a chain of k rooms drawn as one text grid (`#` wall, `.` free, `S` start, `G` goal), and the model outputs a route over $\{\texttt{U}, \texttt{D}, \texttt{L}, \texttt{R}\}$. Many routes reach the goal, so a route is correct if executing it from the start reaches the goal within a step cap set by the grid size. Lee et al. (2025) self-improve on maze shortest paths given as adjacency lists; the room structure is introduced here and released as MODULARGRID.

Composition. Adjacent rooms are joined by a corridor of three columns, open only at the left room's exit row, across the rows between the exit and the entry, and at the right room's entry row. Every route must therefore leave each room by its exit and enter the next by its entry, and the corridor actions b_i between rooms i and $i+1$ are fixed by these two rows: `RR`, then `U` or `D` once per row between them, then `RR`. The shortest route of a chain is the concatenation of its rooms' shortest routes and the corridor actions,

$$f^{\star}_{\text{grid}}(x_1 \circ \cdots \circ x_k) = f^{\star}_{\text{grid}}(x_1) \| b_1 \| \cdots \| b_{k-1} \| f^{\star}_{\text{grid}}(x_k).$$

Filter. None: the corridors make the rule valid on every chain.

Data. Each room has a 5×5 interior with wall probability 0.28 and a shortest entry-to-exit path of 4–12 steps. Initial training uses $200{,}000$ one-room grids. Each round builds $15{,}000$ new chains per room count, for every count from 2 to the current maximum, from the replay rooms. The test set contains 200 chains per room count from 1 to 13, generated before any training data. Test chains never appear in training, and their rooms are excluded from the composed pools, although many of them occur among the initial-training rooms, which come from the same generator.

A.1.2 ADDITION

Task. The input is a pair of decimal integers (A, B), and the target is their sum $A + B$.

Composition. Both addends are split into aligned blocks, $A = a_1 \| \cdots \| a_k$ and $B = b_1 \| \cdots \| b_k$, where block i has d_i digits and $\|$ denotes concatenation. Each component is one block sum $s_i = a_i + b_i$, and the label concatenates the leading sum s_1, which may overflow its width, with the lower sums padded to their widths.

Filter. The label is correct exactly when no carry crosses a block boundary, that is, $s_i < 10^{d_i}$ for every non-leading block $i = 2, \ldots, k$. The filter accepts only such parents; without it, a boundary carry gives a wrong label.

Data. Initial training uses addends of 3–7 digits (Table 4). Each round samples $10{,}000$ new composed problems per addend length, for every length from 8 to the current maximum, and $5{,}000$ new replay problems per seed length, labeled by the current model. The test set contains 100 problems per length from 3 to 23, sampled uniformly and excluded from every training pool.

A.1.3 MAX RUN LENGTH

Task. The input is a string x over the digits 0–9. A run is a maximal block of equal symbols, and the target is the symbol and length of the longest run, written `symbol|length`; among runs of equal length, the leftmost counts.

Composition. A parent is the concatenation xy of two components, and its label is the answer of the component with the longer run, taking x on ties because every run of x precedes every run of y:

$$f^{\star}_{\text{run}}(xy) = \begin{cases} f^{\star}_{\text{run}}(x), & L(x) \geq L(y), \\ f^{\star}_{\text{run}}(y), & L(y) > L(x), \end{cases}$$

where L is the length of the longest run.

Filter. The rule fails when a run continues across the boundary, since the merged run can be longer than both component runs. The filter $G(x, y) = \mathbf{1}\{x_{|x|} \neq y_1\}$ accepts a parent only if the last symbol of x differs from the first symbol of y.

Data. Initial training uses $50{,}000$ strings per length from 6 to 10, which are replayed with true labels in every round. Each round draws $2{,}000$ new composed strings per length, for every length from 12 to the current maximum. The test set contains 100 strings per length from 6 to 83, drawn before any training data and excluded from it. For the boundary sweep of Figure 5, which tests inputs the filter excludes from training, we take a same-length reference string whose longest run has length r, build an input whose longest run has length $r + \Delta$ and crosses the midpoint, and evaluate the saved models of rounds 0–7 on their current length ranges.

Grid navigation

Component 1 input	Component 2 input	Parent input

`Q:`
`#######`
`#S....#`
`##.#..#`
`#..##.#`
`#....G#`
`#.#.###`
`#######`
`A:`
	`Q:`
	`#######`
	`#.....#`
	`#.....#`
	`#....G#`
	`##...##`
	`#S.##.#`
	`#######`
	`A:`
		`Q:`
		`###############`
		`#S....###.....#`
		`##.#..###.....#`
		`#..##.###....G#`
		`#.......##...##`
		`#.#.###....##.#`
		`###############`
		`A:`

Component 1 output: `RRRDRDD`
Component 2 output: `RUURRR`
Composed target: `RRRDRDD RRDRR RUURRR` (corridor actions in the middle)

Addition

Component 1 input	Component 2 input	Parent input
`Q: 6133 + 8894 = ?` `A:`	`Q: 34355 + 50024 = ?` `A:`	`Q: 613334355 + 889450024 = ?` `A:`

Component outputs `15027` and `84379`; composed target `1502784379`.

Max run length

Component 1 input	Component 2 input	Parent input
`Q: runlen(493036) = ?` `A:`	`Q: runlen(887476) = ?` `A:`	`Q: runlen(493036887476) = ?` `A:`

Composed target `8|2`, accepted because the boundary symbols `6` and `8` differ.

Figure 9: Saved round-1 labels for the algorithmic tasks, with the prompts the model sees. The composed targets and all inputs are transcribed from the saved label files. Grid child routes are recovered by splitting the saved label at the corridor actions; addition component outputs are the model's saved labels for the same component prompts; max run length component outputs were not saved. All three labels are correct.

A.1.4 PROGRAM GENERATION

Task. The input is an instruction listing k steps, each a natural-language description of one text-processing command, and the target is a one-line Bash pipeline that reads standard input and applies the steps in order. A program is correct if its output matches that of the reference pipeline on all 24 test inputs of its parent.

Composition. A component is one step, whose answer is one command, and the label joins the component answers with pipes,

$$A(\hat{y}_1, \ldots, \hat{y}_k) = \hat{y}_1 \mid \hat{y}_2 \mid \cdots \mid \hat{y}_k.$$

A pipe passes each command's output to the next command's input, so the label is correct whenever every component command is correct (Figure 10 shows the full prompts).

Filter. Structural only: the label must parse as a pipeline.

Data. Commands come from TLDR[2] (common and Linux pages) and NL2Bash (Lin et al., 2018), restricted to 21 standard text utilities. Of 143 extracted single-command examples, we keep the 119 that run deterministically on 24 probe inputs, produce at least 4 distinct outputs, and do not act as the identity (71 from TLDR, 48 from NL2Bash). Within each utility, a hash ranking holds out 30 commands from training compositions, so each held-out command has a seen command of the same utility; all 119 are still trained and rehearsed as single commands. Pipelines are split into training, validation, and test partitions (70, 15, and 15%) by a hash of their unordered command set, so no command set appears in two partitions. Training rounds use 2,447, 4,096, and 4,096 parents of length two, three, and four; the two-command pool has only 2,447 pipelines. The 24 inputs of a parent come from six seeded generators (logs, CSV, words, key–value pairs, numbers, and tab-separated text) and are shared by all parents of a test set and length. Every checkpoint is evaluated on the 119 single commands and on three pipeline test sets, split equally between two and three commands: 800 pipelines of seen commands, 800 with one held-out command, and 600 with only held-out commands. Longer pipelines use a separate test set of seen commands, with all 568 eligible two-command pipelines of the test partition and 1,000 pipelines at each length from three to six. No test pipeline equals a training composition, although 83–99% of the three- to six-command test pipelines contain a two-command training composition as a contiguous part.

A.1.5 MULTI-HOP REASONING

Task. The input is a MuSiQue question q with its released paragraphs P, and the target is a short answer, scored by exact match under the official answer aliases. At evaluation, the model answers each question directly from its paragraphs, without a decomposition.

Composition. The released decomposition splits q into sub-questions $(q_1, \ldots, q_k)$, $k \in \{2, 3, 4\}$, where q_j may refer to an earlier answer through a placeholder $\#i$. The model answers the sub-questions in order, seeing its earlier answers and with each placeholder filled by the corresponding answer. For the last step, it answers the original question, with the filled last sub-question given as the final reasoning step (Figure 11):

$$\hat{a}_j = h\big(q_j[\#i \mapsto \hat{a}_i],\, P,\, \hat{a}_{<j}\big)\ \ (j < k), \qquad A(x) = \hat{a}_k = h\big(q,\, P,\, \hat{a}_{<k},\, q_k[\#i \mapsto \hat{a}_i]\big),$$

where h is the current model. Later queries therefore depend on the model's earlier answers, and the rule is valid when the released decomposition leads to the released answer.

Filter. Every answer $\hat{a}_j$ must be non-empty, short, free of unresolved placeholders, and present in P or q (Section A.2).

Data. Parents come from the 19,938 answerable MuSiQue training questions, ordered by a fixed hash. We take the first 512 eligible questions of each hop count, excluding the parents of the seed questions and 22 questions that match a development question or sub-question; 14,019, 4,269, and 1,144 questions were eligible. The three-hop parents contain 437 `3hop1` and 75 `3hop2` questions,

[2] https://github.com/tldr-pages/tldr

Shared instruction block
Use only standard text utilities (such as sort, uniq, grep, cut, tr, head, tail, sed, awk, wc, rev, tac, nl, paste, fold, fmt, cat, column). The text arrives on standard input; do not name any file. Reply with the command on one line and nothing else.

Component 1: model input
Write one Bash command that reads standard input and does the following.

Task: Output the first 5 lines of a file
Selected output: `head -5`

Component 2: model input
Write one Bash command that reads standard input and does the following.

Task: Sort a file in descending order
Selected output: `sort -r`

Parent: model input
Write one Bash pipeline that reads standard input and applies the following steps in order, each step on the output of the previous one, joined with `|`.

Step 1: Output the first 5 lines of a file
Step 2: Sort a file in descending order
Composed target: `head -5 | sort -r`

Figure 10: Saved Bash composition example `label2_00059` from round 1. The shared instruction block shown above is inserted after the opening sentence in each prompt; the remaining prompt text and selected outputs are transcribed unchanged. Execution inputs are passed to the program at evaluation and are not part of the prompt. Line wrapping is for display.

and the four-hop parents 274 `4hop1`, 63 `4hop2`, and 175 `4hop3` questions. The seed set contains 1,024 single-hop sub-questions with gold-filled references and gold answers, drawn from 484 training questions. We evaluate on all 2,417 answerable development questions (1,252 two-hop, 760 three-hop, and 405 four-hop); the single-hop row uses 512 deduplicated development steps.

A.1.6 String transformations

Task. Following the controlled setting of Yuan et al. (2025), the input is a source string x_0 and an ordered chain of functions $(f_1, \ldots, f_D)$ from a library of 21 string functions, and the target is the trace $x_i = f_i(x_{i-1})$ for $i = 1, \ldots, D$, ending with the final answer x_D. The library removes operators of Yuan et al. (2025) that duplicate another operator on these inputs or leave most inputs unchanged. The standard configurations use all 21 functions with $D = 4$ and $D = 5$; the length-preserving configuration uses the 11 functions that keep string length unchanged, with $D = 4$. A parent is kept only if every intermediate value is nonempty and at most 96 characters and every function changes the final answer.

Composition. A component applies one function to the model's previous answer, and the label concatenates the step answers,

$$\hat{x}_i = h(f_i, \hat{x}_{i-1}), \qquad A(x) = (\hat{x}_1, \ldots, \hat{x}_D).$$

As in multi-hop reasoning, each query takes the model's own earlier answer, and the rule is valid on every parent (Figure 12). A controller supplies the parent question, the current function, and the model's earlier values at each step; it never executes a function or supplies a correct intermediate value, and the same controller is used for both methods in training and evaluation.

Filter. Structural only: each value must parse and have at most 96 characters.

Data. The initial model is trained on 200,000 chains of one function and 200,000 chains of two functions, covering all 21 functions and all 441 ordered pairs, with lowercase sources of 4–8 characters. Each parent has one fresh lowercase source of 4–10 characters and functions drawn with replacement. Parents are assigned to 2,048 training, 128 development, and 512 test parents by a hash of their multiset of functions, so every reordering of the same functions falls in one split; multisets used in earlier screening are excluded.

Component 1 input (released first sub-question)

```
Answer the question using the passages. Reply with the answer only: a name
or short phrase.

[1] Henry Wolf (engraver): Henry Wolf (1852–1916) was a French-born wood
engraver who lived and worked in the United States during his most
influential work period and until his death.

[2] Jeanette Wohl: Jeanette Wohl (October 16, 1783, Frankfurt am Main –
November 27, 1861, Paris) was a longtime friend and correspondent of ...

[... paragraphs 3–20 ...]

Question: Michael Schwerner >> place of death
Answer:
```

Component 1 output: `Mississippi`

Component 2 input (final step)

```
Answer the question using the passages. Reply with the answer only: a name
or short phrase.

[1] Henry Wolf (engraver): ...
[... paragraphs 2–20 ...]

Earlier model predictions (may be incorrect):
#1: Michael Schwerner >> place of death -> Mississippi
Final reasoning step: when did Mississippi become a right to work state

Question: When did the state where Michael Schwerner died become a right to
work state?
Answer:
```

Component 2 output: `1954`

Parent input

```
Answer the question using the passages. Reply with the answer only: a name
or short phrase.

[1] Henry Wolf (engraver): ...
[... paragraphs 2–20 ...]

Question: When did the state where Michael Schwerner died become a right to
work state?
Answer:
```

Composed target: `1954`

Figure 11: Saved multi-hop reasoning label `2hop__333835_77060` from the 2B model in round 1. The prompts are rendered from the prompt template with the saved sub-questions and answers; the 20 paragraphs (10,973 characters) are truncated for display. The parent input is also the prompt of direct self-improvement. The label is correct.

A.1.7 TYPED PYTHON

Task. The input is the natural-language specification of three type-compatible unary functions (g_1, g_2, g_3), and the target is the body of `solve(x)` that applies them in order. A program is correct if it passes all 24 test inputs of its parent.

Composition. A component asks for a program $\hat{p}_i$ for one specification, and the label places the component programs in a fixed wrapper (Figure 13),

$$A(\hat{p}_1, \hat{p}_2, \hat{p}_3) = \big[\texttt{solve(x) : return}\ \hat{p}_3(\hat{p}_2(\hat{p}_1(\texttt{x})))\big],$$

so the label is correct whenever every component program is correct on its inputs.

Filter. Structural only: the label must parse as one Python function.

Data. Functions come from MBPP+ and HumanEval+ (Austin et al., 2021; Chen et al., 2021; Liu et al., 2023): 189 unary functions over integers, strings, and lists with at least 48 extended test inputs and at least 4 distinct outputs. The initial model is Qwen3.5-2B after 128 fine-tuning updates on its

	Model input	Saved output
Component 1	`Q: func_2(bnqi) = ?`	`func_2=binq => binq`
Component 2	`Q: func_15(binq) = ?`	`func_15=elqt => elqt`
Component 3	`Q: func_15(elqt) = ?`	`func_15=hotw => hotw`
Component 4	`Q: func_4(hotw) = ?`	`func_4=hOtW => hOtW`
Component 5	`Q: func_13(hOtW) = ?`	`func_13=WhOt => WhOt`

Parent input

```
Q: func_13(func_4(func_15(func_15(func_2(bnqi))))) = ?
A:
```

Composed target

```
func_2=binq | func_15=elqt | func_15=hotw | func_4=hOtW | func_13=WhOt => WhOt
```

Figure 12: Saved compositional warmup label for strings of depth five. Each input is written as `Q: ... = ?` followed by a new line and `A:`; each component input applies one function to the model's previous output. The parent input is also the prompt of direct RL. All text is transcribed from the saved files, and the label is correct.

own single-function programs that pass 12 acceptance inputs, and we use the 106 functions it then solves reliably. Parents are chains with matching input and output types and no repeated function, assigned to 2,048 training, 128 development, and 512 test parents by a hash of their function multiset. The 24 inputs of a parent are extended test inputs of its first function on which the whole chain runs with the expected types; the first 12 form the RL reward, and all 24 are used for evaluation.

Shared instruction block

```
Complete the body of the Python function solve(x). The response already
starts with `def solve(x):` and indentation; continue with its body without
repeating the header. You may import standard-library modules and define
helper functions inside solve. Apply the following operations in order,
passing each output as the next input. Return the final value. Inputs satisfy
the listed preconditions at every step. Output only the Python continuation,
with correct indentation and no Markdown fences.
```

Component 1 input

```
Step 1:
"""
Write a function to remove lowercase substrings
from a given string.
assert remove_lowercase("PYTHon")==('PYTH')
"""

Input preconditions:
assert isinstance(str1, str), "invalid inputs"
```

Component 1 output

```
def solve(x):
    if not isinstance(x, str):
        raise ValueError('Input must be a string')
    return ''.join((c for c in x if c.isupper()))
```

Component 2 input

```
Step 1:
"""
Write a python function to convert the given
string to lower case.
assert is_lower("InValid") == "invalid"
"""
```

Component 2 output

```
def solve(x):
    return x.lower()
```

Component 3 input

```
Step 1:
"""
Write a python function to split a string into
characters.
assert split('python') == ['p','y','t','h','o','n']
"""
```

Component 3 output

```
def solve(x):
    if not isinstance(x, str):
        raise ValueError('invalid inputs')
    return list(x)
```

Parent input

The instruction block followed by the three step blocks, numbered Steps 1–3.

Composed target

```
def solve(x):
    def _component_0(_value):

        def solve(x):
            if not isinstance(x, str):
                raise ValueError('Input must be a string')
            return ''.join((c for c in x if c.isupper()))
        return solve(_value)
    def _component_1(_value):

        def solve(x):
            return x.lower()
        return solve(_value)
    def _component_2(_value):

        def solve(x):
            if not isinstance(x, str):
                raise ValueError('invalid inputs')
            return list(x)
        return solve(_value)
    value = x
    value = _component_0(value)
    value = _component_1(value)
    value = _component_2(value)
    return value
```

Figure 13: Saved compositional warmup label for typed Python (MBPP+ functions 478, 477, and 565). The response of every prompt begins with `def solve(x):`, which the model continues; the component preconditions of Steps 2 and 3 are omitted for space. All text is transcribed from the saved files, and the composed program passes all 24 test inputs of its parent.

A.2 TRAINING SETUP

This subsection gives the training setup of every experiment. In all of them, every round runs to completion and no checkpoint is selected on validation data.

Algorithmic tasks (Section 3). All three tasks use a 14.2M-parameter Transformer with six layers, hidden size 384, six attention heads, and no positional encoding, trained with AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.99$, $\epsilon = 10^{-12}$) in bfloat16. Labels are generated with greedy decoding. Table 4 lists the seed training, round schedule, composed and replay data, and filter of each task. For the component-accuracy study of grid navigation (Figure 3), we train three initial models with different seeds (batch size 128) and run the grid schedule of Table 4 from their checkpoints after 400, 1,500, 2,500, 3,000, 6,000, 8,000, and 30,000 updates.

Table 4: Training setup for the algorithmic tasks.

	Grid navigation	Addition	Max run length
Model	14.2M-parameter character-level Transformer		
Learning rate	5×10^{-4}	5×10^{-4}	5×10^{-4}
Weight decay	0.1	0.1	0.1
Batch size	256	256	256
Initial training	200,000 one-room grids, 30,000 updates	3–7 digits, 50,000 per length, 10,000 updates	Lengths 6–10, 50,000 per length, 15,000 updates
Rounds	12	8	8
Range	1 to 13 rooms, $+1$ per round	7 to 23 digits, $+2$ per round	Length 10 to 83: 20 after round 1, then $+9$ per round
Composed examples	15,000 per room count up to the maximum, redrawn each round	10,000 per length up to the maximum, redrawn each round	2,000 per length up to the maximum, redrawn each round
Seed-regime replay	5,000	25,000	250,000
Filter	None	No boundary carry	Boundary symbols differ
Training per round	9,000 updates	3,000 updates	3,000 updates
Test set	200 per room count, 1–13	100 per length, 3–23	100 per length, 6–83

Program generation and multi-hop reasoning (Section 4). Both experiments fine-tune Qwen3.5 models with LoRA (Table 5). Initial training uses components with reference answers: single commands for Bash and single-hop questions for MuSiQue. Each round labels its parents with the model from the preceding round, continues training from that model, and mixes the new labels with rehearsal of the initial examples. Labels are never executed or checked against reference answers; only the filters of Section A.1 are applied.

Table 5: Hyperparameters for the experiments in Section 4.

	Program generation	Multi-hop reasoning
Model	Qwen3.5-2B; LoRA rank 16, $\alpha = 32$	Qwen3.5-2B, 4B, 9B; LoRA rank 16 on attention and feed-forward projections
Initial training	All 119 commands, 96 updates, learning rate 10^{-4}	1,024 single-hop questions, 100 updates, batch size 16, learning rate 2×10^{-4}
Rounds	3 (pipelines of 2, 3, 4 commands)	3 (2-, 3-, 4-hop questions)
Parents per round	2,447, 4,096, 4,096	512
Training per round	192 AdamW updates, learning rate 10^{-4}, batch size 16, gradient clipping 1.0	64 AdamW updates, learning rate 10^{-4}, batch size 8
Rehearsal	25% of each batch	2 of 8 examples per batch
Seeds	4	3

Table 6: Hyperparameters for the RL experiments in Section 5.

	Strings	Typed Python
Initial model and compositional warmup		
Initial model	14.2M character Transformer	Qwen3.5-2B, LoRA rank 16, $\alpha = 32$
Initial training	Chains of 1–2 functions	128 updates on single functions
Warmup SFT	24 updates, learning rate 10^{-4}	24 updates, learning rate 10^{-5}
Target tokens per update	8,192	4,096
GRPO		
Optimizer, learning rate	AdamW, 10^{-5}	
Samples per group	16	8
Groups with varied rewards per update	16	4
Sampling temperature	1.0	
Maximum generation length	96 characters per step	4,096 tokens
Clip ratio, KL coefficient	0.2, 0	

RL warm start (Section 5). The compositional warmup labels the first 512 training parents, in a fixed hash order, with the initial model and fine-tunes on them (Section A.1); RL samples prompts from all 2,048 training parents. Both methods then run GRPO (Shao et al., 2024) with the configuration released by Yuan et al. (2025) (Table 6). The reward is 1 for a correct final answer and 0 otherwise; for Python, it uses 12 of the 24 inputs of each parent, and evaluation uses all 24 with greedy decoding. Generated tokens include warmup label generation and every sampled RL token, including groups discarded for uniform reward, and the learning curves also charge the target tokens of the warmup fine-tuning; prompt tokens and evaluation are excluded. On strings, the warmup-then-RL runs generate 40M tokens and direct RL 80M, with checkpoints every 10M tokens up to 40M and at 80M. On Python, they generate 524,288 and 1,048,576 tokens, with checkpoints every 262,144 tokens; continuations restore the optimizer and random-number state without new labels, so label costs are charged once. The reported checkpoints are fixed in advance, and three seeds share the initial model and data.

B Theoretical Analysis

When can predictions on simpler problems provide useful supervision for harder ones? We separate three requirements: reliable component predictions, valid composition, and coverage of the problems that need supervision. We first bound composed-label error, then characterize restrictions on valid supervision. A supplementary learning result gives a sufficient condition for improvement on the population receiving that supervision.

B.1 Task levels and composed supervision

Task levels. Let $f^\star : \mathcal{X} \to \mathcal{Y}$ be the target function. We use finite input, answer, and probability spaces with exact-match loss. A task-specific level $\ell : \mathcal{X} \to \mathbb{N}$ indexes problem scale. Write

$$\mathcal{X}_k = \{x : \ell(x) = k\}, \qquad \mathcal{X}_{<k} = \{x : \ell(x) < k\}. \tag{2}$$

Levels describe problem scale, without assuming that lower-level problems are solved or that accuracy decreases with level. Seeds may cover only part of a level, and rounds may span or revisit levels.

Fix a level k, the current model $\hat{f}$, and an available source pool $S \subseteq \mathcal{X}_{<k}$. We consider supplied plans whose component queries remain in S in both the actual execution and an execution with correct intermediate answers. For fixed-query plans this requires $\ell(x_i) < k = \ell(x)$ for each component. Separate auxiliary tasks can be included in the aggregation procedure rather than assigned the same level function. This lower-level representation is a setting for analysis; the label-reliability bound also applies without a level ordering.

Candidate and training examples. Let D_k be a specified distribution of candidate inputs at level k. It may be induced by sampling and composing components, or by selecting parent problems and assigning supplied plans. A parent without an available plan is rejected. Let $\Pr_k$ denote probability over this candidate-generation procedure and its labeling randomness, conditional on the current model and history. Its parent marginal is D_k. The event $G = 1$ means that an example is retained for training, and we assume $q_k := \Pr_k(G = 1) > 0$.

The generator D_k need not cover all of $\mathcal{X}_k$. Since a level does not specify how its inputs are weighted, errors below refer to this generator. Other evaluation distributions need separate arguments.

Comparison with correct components. Let $\widetilde{Y}$ be the actual composed label. Labels on candidates without a plan can be defined arbitrarily because these are always rejected. For the same candidate, let $\widetilde{Y}^\circ$ be the result of rerunning the supplied plan with correct component answers. For dependent plans, recompute subsequent queries from the corrected answers. We consider executions with a finite call budget and couple them using the same initial state, query and update rules, and non-component random choices indexed by execution step. The aggregation procedure is unchanged, including any learned aggregator. For fixed queries and deterministic assembly,

$$\widetilde{Y}^\circ = A\big(\pi, f^\star(X_1), \dots, f^\star(X_m)\big).$$

Let E_{comp} be the event that at least one component answer in the actual execution is incorrect on the query actually issued. If this event does not occur, the actual and corrected executions coincide. Define

$$\begin{aligned} \rho_k &= \Pr_k[\widetilde{Y} \neq f^\star(X) \mid G = 1], \\ \epsilon_k &= \Pr_k[E_{\mathrm{comp}} \mid G = 1], \\ \xi_k &= \Pr_k[\widetilde{Y}^\circ \neq f^\star(X) \mid G = 1]. \end{aligned} \tag{3}$$

Here ρ_k is composed-label error, ϵ_k is the probability of any component error, and ξ_k is composition error with correct components. All three use the original acceptance decision, so they refer to the same training population. For learned aggregation, ξ_k includes its errors given correct components.

B.2 Component reliability and composition validity

Correcting the components changes the composed label only when a component prediction was wrong. This gives a bound that allows correlated mistakes and prediction-dependent filtering.

Proposition B.1 (Reliability at a task level)**.** *Without independence between component mistakes,*

$$|\rho_k - \xi_k| \le \epsilon_k, \tag{4}$$

and hence

$$\rho_k \le \min\{1, \epsilon_k + \xi_k\}. \tag{5}$$

If each execution uses at most m component calls and $\epsilon_{k,i}$ is the probability, conditional on acceptance, that call i occurs and is incorrect, then $\epsilon_k \le \sum_{i=1}^{m} \epsilon_{k,i}$.

Proof. The actual and corrected labels coincide whenever no component prediction is incorrect. Thus

$$\left|\mathbf{1}_{\{\widetilde{Y} \neq f^\star(X)\}} - \mathbf{1}_{\{\widetilde{Y}^\circ \neq f^\star(X)\}}\right| \le \mathbf{1}_{E_{\mathrm{comp}}}.$$

Taking expectations conditional on $G = 1$ gives Equation (4) and then Equation (5). A conditional union bound gives the last claim. For dependent plans, coincidence outside E_{comp} follows by induction through the coupled executions. □

When component mistakes are rare, composed-label error is close to the error that remains with correct components. This separates unreliable component predictions from an invalid composition procedure and motivates checking both on the examples that survive filtering.

When composition improves the supervision. Compare composition with the current model's direct predictions on the same accepted parent population. Write

$$\rho_k^{\mathrm{direct}} = \Pr_k[\hat{f}(X) \neq f^\star(X) \mid G = 1].$$

A sufficient condition for composed labels to be more accurate is

$$\epsilon_k + \xi_k < \rho_k^{\text{direct}}. \tag{6}$$

Thus composition can improve supervision when its components and assembly are more reliable than direct prediction on the accepted parents. The comparison must use the same parents: separately filtered evaluations do not establish this advantage. Likewise, ϵ_k concerns the components actually used, rather than average accuracy on lower levels.

Failure I: Component errors. A valid rule can propagate component mistakes into its labels. Some mistakes may be irrelevant or cancel; when none is masked, composed-label error equals the probability of a component error.

Proposition B.2 (Component errors without masking)**.** *For fixed-query deterministic assembly, suppose the rule is valid for every accepted candidate. If every incorrect component-answer vector in an accepted execution changes the assembled answer relative to the correct vector, then*

$$\rho_k = \epsilon_k. \tag{7}$$

Proof. Validity makes the parent label correct when all components are correct. The no-masking condition makes it incorrect whenever some component is incorrect. The two error events therefore agree on accepted examples. □

Under this condition, composition cannot compensate for weak component predictions. Improving label quality requires reducing the frequency of component mistakes among the accepted compositions, even when the rule itself is exact.

Failure II: Composition errors. With perfect component predictions, Proposition B.1 gives

$$\epsilon_k = 0 \quad \Longrightarrow \quad \rho_k = \xi_k. \tag{8}$$

Even perfect components leave the composition error ξ_k on this accepted population; more generally, $\rho_k \geq \xi_k - \epsilon_k$. A *structurally sound* filter guarantees that the execution with correct components answers the accepted parent correctly, giving $\xi_k = 0$. Checks based on uncertain predictions need not provide this guarantee. These bounds concern supervision; a student may correct some teacher errors.

B.3 Coverage within a task level

Filtering can remove invalid compositions, but how much supervision can remain? To answer this, consider deterministic supplied procedures with fixed or dependent component queries from S. Define

$$\mathcal{V}_k(S) = \left\{ x \in \mathcal{X}_k : \begin{array}{l} \text{some allowed plan using components in } S \\ \text{returns } f^\star(x) \text{ with correct component answers} \end{array} \right\}. \tag{9}$$

The set $\mathcal{V}_k(S)$ describes which parents admit valid supervision from the available procedures. Membership does not depend on the model's predictions or a particular filter. A rejected plan may have a valid alternative, and the generator need not find every valid plan.

Proposition B.3 (Validity and retention)**.** *Let $c_k = D_k(\mathcal{V}_k(S))$ be the probability mass that admits a valid supplied composition. For any actual acceptance procedure,*

$$q_k(1 - \xi_k) \leq c_k. \tag{10}$$

In particular, a structurally sound filter must satisfy $q_k \leq c_k$. If $q_k > c_k$, then $\xi_k \geq 1 - c_k/q_k$.

Proof. An accepted example with a correct oracle-component answer supplies a valid allowed plan for its parent. Therefore

$$q_k(1 - \xi_k) = \Pr_k[G = 1, \widetilde{Y}^\circ = f^\star(X)] \leq \Pr_k[X \in \mathcal{V}_k(S)] = c_k.$$

The two consequences follow directly. □

The available valid compositions limit how much supervision a sound filter can retain. To exceed this limit, compositional self-improvement must add source components or change its composition procedures; relaxing the filter alone introduces composition error. This concerns validity with correct components, since component mistakes can occasionally cancel a rule error.

Output restrictions. A closed family of answers can restrict $\mathcal{V}_k(S)$ even when composition produces inputs at larger levels. Let $C \subseteq \mathcal{Y}$ contain $f^\star(S)$. For fixed-query deterministic assembly, suppose

$$A(\pi, y_1, \ldots, y_m) \in C \quad \text{whenever } y_1, \ldots, y_m \in C \text{ and } \pi \text{ is allowed.} \tag{11}$$

For dependent procedures, the corresponding requirement is that the complete oracle-component execution preserves C; the fixed-query condition alone does not establish this.

Proposition B.4 (Output-coverage obstruction at a level)**.** *Under the preceding closure condition,*

$$\mathcal{V}_k(S) \subseteq \{x \in \mathcal{X}_k : f^\star(x) \in C\}. \tag{12}$$

For $U_{k,C} = \{x \in \mathcal{X}_k : f^\star(x) \notin C\}$,

$$\Pr_k[X \in U_{k,C} \mid G = 1] \leq \xi_k. \tag{13}$$

Consequently, structurally sound acceptance provides no training examples from $U_{k,C}$.

Proof. An oracle-component execution uses answers from C and returns an answer in C. It cannot be correct on a parent whose answer lies outside C. This proves the set inclusion and the event inclusion underlying Equation (13). Conditioning on the original acceptance event proves the inequality. □

Composition can therefore increase input complexity while leaving some required answers outside the reach of valid supervision. Supervising those answers requires new source answers or a composition rule that can produce them; a better filter alone cannot resolve the restriction.

Persistence. Let S_t be the union of the original source pool S_0 and all parents retained through round t. Suppose $f^\star(S_0) \subseteq C$, every round's oracle-component composition preserves the same C, and new parents use only components from S_t and pass a structurally sound filter. Then

$$f^\star(S_t) \subseteq C \qquad \text{for every } t. \tag{14}$$

Indeed, source answers lie in C; closure puts every oracle-composed answer in C; and structural soundness makes this the retained parent's true answer. Induction gives the claim. The levels of retained inputs may keep increasing, but additional rounds preserve the same answer restriction unless new source inputs or composition operations are introduced.

What missing coverage does not imply. The preceding results restrict which labels composition can supply. A student can still generalize to patterns absent from those labels. To obtain a learning limitation, we need competing target behaviors that the learner cannot distinguish.

Proposition B.5 (Unresolved behavior within a level)**.** *Suppose two possible targets* f_0, f_1 *satisfy the same supplied structural information and induce identical distributions for everything observed by the learner, including source labels, the initial model, and composed training examples. Let* $b_k = \Pr_{X \sim D_k}[f_0(X) \neq f_1(X)]$. *For any possibly randomized learning algorithm,*

$$\max_{j \in \{0,1\}} \mathbb{E}_j \Pr_{X \sim D_k}[\hat{f}(X) \neq f_j(X)] \geq b_k/2. \tag{15}$$

The same learning rule is used under both targets. Its private randomness and the fresh evaluation input are independent of the training observations and of each other.

Proof. The learner has the same output distribution under both targets. For any fixed predictor, the sum of its two errors is at least b_k, since it cannot match both targets where they disagree. Taking expectations proves the bound. □

When two target behaviors produce the same training evidence, the learner has no basis for choosing between them on the inputs where they disagree. A stronger accuracy guarantee therefore requires observations or assumptions that distinguish the alternatives. Missing coverage alone does not establish this ambiguity for a particular task or model.

B.4 Learning from composed labels: a supplementary guarantee

We next ask when label quality leads to a more accurate student. The following standard finite-class bound concerns the accepted population at level k. Define its clean and pseudo-label risks by

$$R_k^+(h) = \Pr_k[h(X) \neq f^\star(X) \mid G = 1], \qquad L_k(h) = \Pr_k[h(X) \neq \widetilde{Y} \mid G = 1].$$

For every predictor h,

$$|R_k^+(h) - L_k(h)| \leq \rho_k, \tag{16}$$

and also

$$|R_k^+(h) - \rho_k| \leq L_k(h). \tag{17}$$

The first inequality follows because true-label and pseudo-label losses coincide when the pseudo-label is correct. The second follows because the student and pseudo-labeler have the same correctness whenever their answers agree. In particular,

$$R_k^+(h) \geq \xi_k - \epsilon_k - L_k(h). \tag{18}$$

Under Proposition B.2, the analogous bound is

$$R_k^+(h) \geq \epsilon_k - L_k(h). \tag{19}$$

A student that closely matches the teacher on this population inherits most of its label error. Correcting that error requires departing from some teacher predictions. Fitting a finite training set alone does not establish a small population imitation loss $L_k(h)$.

Theorem B.6 (Learning on supervised examples at a level)**.** *Fix a nonempty finite class $\mathcal{H}$ containing $f^\star$ before sampling. Draw $n \geq 1$ independent examples from the conditional law of $(X, \widetilde{Y})$ given $G = 1$. Suppose $\hat{f}_{\text{new}} \in \mathcal{H}$ minimizes empirical exact-match pseudo-label error up to an additive $\tau \geq 0$. For $\delta \in (0, 1)$, with probability at least $1 - \delta$,*

$$R_k^+(\hat{f}_{\text{new}}) \leq 2\rho_k + 2\sqrt{\frac{\log(2|\mathcal{H}|/\delta)}{2n}} + \tau. \tag{20}$$

Proof. Let $u_n = \sqrt{\log(2|\mathcal{H}|/\delta)/(2n)}$. Hoeffding's inequality (Hoeffding, 1963) and a union bound give uniform deviation at most u_n between empirical and population pseudo-label risks. On this event, approximate minimization and $f^\star \in \mathcal{H}$ imply $L_k(\hat{f}_{\text{new}}) \leq L_k(f^\star) + 2u_n + \tau = \rho_k + 2u_n + \tau$. Equation (16) proves the result. □

Under these learning assumptions, accurate composed labels can train a student that is accurate on the accepted parents. More independent examples and better optimization tighten the guarantee, but neither removes its dependence on label quality.

A sufficient condition on the supervised population. Combining the theorem with Proposition B.1 certifies improvement over the current model on the same accepted parent population whenever

$$2(\epsilon_k + \xi_k) + 2\sqrt{\frac{\log(2|\mathcal{H}|/\delta)}{2n}} + \tau < R_k^+(\hat{f}). \tag{21}$$

The condition separates label error, sampling error, and optimization error. It is sufficient for improvement on accepted parents, without pointwise noise assumptions or independence among component mistakes. It does not guarantee improvement on excluded problems.

Evaluation across the level. To evaluate a predictor on the pre-filter level-k population, write $R_k(h) = \Pr_{X \sim D_k}[h(X) \neq f^\star(X)]$. When $0 < q_k < 1$, the law of total probability gives

$$R_k(h) = q_k R_k^+(h) + (1 - q_k) \Pr_k[h(X) \neq f^\star(X) \mid G = 0]. \tag{22}$$

For $q_k = 1$, $R_k(h) = R_k^+(h)$. The theorem controls error on accepted examples; error on rejected examples remains unconstrained. Bounding the latter by one gives the conservative full-population bound

$$R_k(\hat{f}_{\text{new}}) \leq q_k \min\{1, 2\rho_k + 2u_n + \tau\} + (1 - q_k),$$

where u_n is defined in the proof above. Thus high retention limits the mass left uncontrolled under D_k. It says nothing about problems that D_k omits or about a different evaluation law.

C Additional Experimental Results

C.1 Direct Self-Improvement and Filtering

Figure 2 shows compositional self-improvement alone. This subsection compares it with direct self-improvement on the three algorithmic tasks and, for addition and max run length, with composition without the filter. It then shows the Bash and RL warm-start comparisons of Sections 4.1 and 5 alike.

Setup. For each task, all runs share the initial model, parent inputs, round schedule, and seed-regime replay of Table 4; the runs differ only in how new labels are produced. Direct self-improvement labels each parent with the current model's greedy prediction on the full parent. Unfiltered composition keeps every composed label. Filtered composition applies the filters of Table 1, which reject addition pairs with a carry across a block boundary and max run length pairs with equal boundary symbols. Grid navigation has no filter.

Results. Figure 14 shows that **direct self-improvement does not extend the solved range on grid navigation or addition**: after the last round, its accuracy is 0% at every room count above one and every addend length above seven, while compositional self-improvement reaches 80.9% mean accuracy on 2–13 rooms. On max run length, the initial model already reaches 59.8% mean accuracy on lengths 11–83; direct self-improvement raises this to 75.9%, and unfiltered and filtered composition reach 97.0% and 97.6%. **The filter has the largest effect on addition**, where unfiltered composition reaches 33.5% mean accuracy on 8–23-digit addends and 11.0% at 23 digits, compared with 93.7% and 80.0% with the filter.

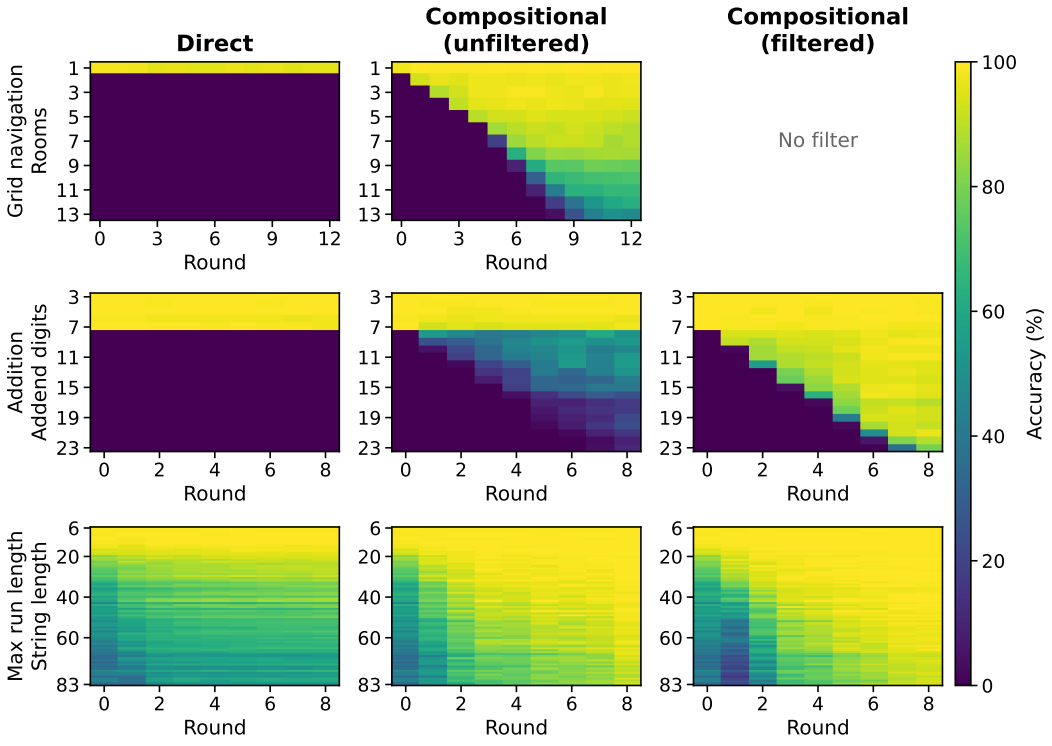


Figure 14: Accuracy by instance size across self-improvement rounds for the algorithmic tasks of Section 3.3, by label source. Runs in a row share the initial model, parents, and schedule. Grid navigation has no filter. The compositional panel for grid navigation and the filtered panels for addition and max run length are the runs in Figure 2; the addition panels also appear in Figure 4.

Bash and RL warm start. Neither experiment filters labels by correctness. The structural check (Tables 2 and 3) rejected none of the $42{,}556$ composed Bash labels and none of the 512 warmup labels

for strings of depth five or typed Python, so the runs in Figures 6 and 8 are unfiltered composition. Figure 6(a) shows the Bash runs next to direct self-improvement, and Figure 8 shows the RL runs next to direct RL. After three rounds, compositional self-improvement reaches 83–94% accuracy on Bash pipelines of two to six commands, while direct self-improvement stays at or below 11%. On strings of depth five, RL after the compositional warmup reaches 19.2% accuracy at 40M tokens, and direct RL remains at 0% through 80M tokens.

C.2 Length-Filtered Direct Self-Improvement on Addition

Lee et al. (2025) show that direct self-improvement can extend addition to long operands, whereas our direct baseline fails beyond the initial training range (Section C.1). Their method adds one digit per round and discards labels more than two tokens shorter than the longest prediction for the same length; they also find that forward addition, as in our setting, requires such filtering while reversed addition does not. We test whether this curriculum and length filter let direct self-improvement match compositional self-improvement.

Setup. All runs use the training setup of Table 4 and start from the same two initial models, retrained on addends of 3–7 digits so that they are accurate at eight digits, as the length filter requires. We compare compositional self-improvement, adding two digits per round for eight rounds, with length-filtered direct self-improvement on the same curriculum and on the one-digit curriculum of Lee et al. (2025) for 16 rounds. All runs end at 23 digits.

Results. Figure 15 shows that **compositional self-improvement reaches 23 digits in eight rounds, while length-filtered direct self-improvement adds at most one digit**. Compositional self-improvement reaches at least 97% accuracy at every length. Direct self-improvement on the one-digit curriculum solves nine digits in the first round and no longer lengths after 16 rounds, with about twice the GPU time.

Direct self-improvement extends only to neighboring lengths the model already generalizes to. The model labels nine-digit parents correctly because it generalizes from eight digits, but from ten digits on its labels are mostly wrong, and the length filter cannot detect wrong sums of the correct length. Direct self-improvement therefore depends on how far the model generalizes beyond its training range, which is small here and hard to know in advance. Compositional self-improvement instead builds labels from components the model already solves: concatenating block sums of solved lengths gives correct labels for longer addends, however far the model generalizes on its own.

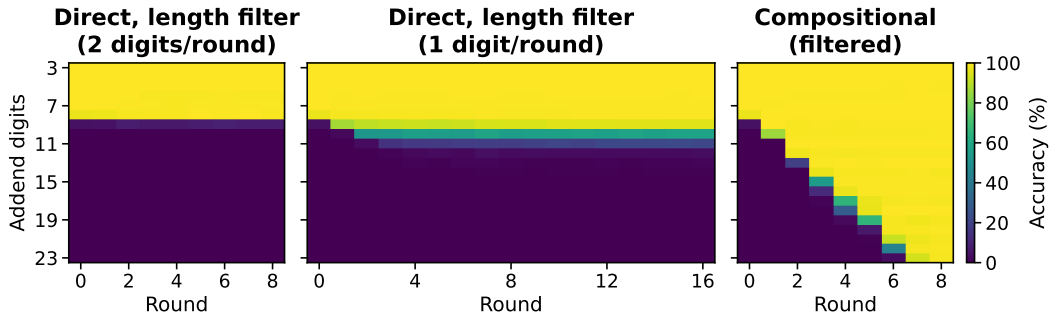


Figure 15: Addition accuracy by addend length across self-improvement rounds, averaged over two retrained initial models. All runs share these initial models and end at 23 digits; panel widths are proportional to the number of rounds. Length filter: labels shorter than the longest prediction for the same length minus two tokens are discarded (Lee et al., 2025). Compositional runs use the boundary-carry filter of Table 1.

C.3 Pipeline Length in Bash

Figure 6 shows that training on pipelines of two to four commands improves accuracy on pipelines of up to six commands. This subsection examines that improvement by length and how the number of commands in the model's outputs changes across rounds.

Setup. We use the test pipelines of seen commands from Section A.1.4: 568 two-command pipelines and 1,000 pipelines at each length from three to six. For each greedy output, we count its commands and record whether the same checkpoint writes each requested command correctly when asked for it alone. Panel (c) of Figure 16 uses this record. For example, the test pipeline `head -5 | sort -r` enters panel (c) only if the model answers `head -5` and `sort -r` when asked for each step separately; it then counts as correct only if the model's one-line answer to the full two-step instruction is also correct, which fails if it swaps the two steps or drops the `-r` flag.

Results. Figure 16(a) shows that **later rounds mainly help on the longest pipelines**. From round 1 to round 3, accuracy increases by 10.3 points at length five and 21.4 points at length six, compared with at most 4.1 points at shorter lengths. Direct self-improvement stays near zero beyond two commands. Figure 16(b) explains part of the early gain: the initial model answers about 70% of multi-step instructions with a single command, and one round of training on two-command pipelines teaches the model to produce the correct number of commands at most lengths. Figure 16(c) shows that at lengths four to six, accuracy also rises on pipelines for which the model writes every step correctly as a single command, so later rounds improve how the model combines these commands.

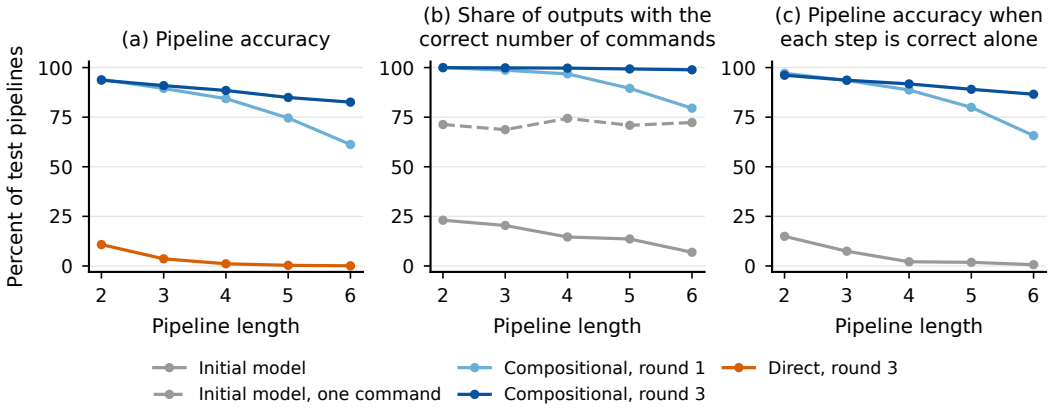


Figure 16: Bash outputs on test pipelines of seen commands, by pipeline length. (a) Accuracy. (b) Share of outputs with the correct number of commands, regardless of whether the output is right, and the initial model's share of single-command outputs. (c) Accuracy on pipelines whose every step the same checkpoint writes correctly alone (see Setup). Panels (b) and (c) pool the four seeds.

C.4 Transfer to Held-Out Commands in Bash

Training compositions in Section 4.1 use only 89 of the 119 commands; we call the other 30 held-out commands. Figure 6 shows compositional self-improvement on test pipelines with no, one, or only held-out commands. This subsection describes these test sets and adds direct self-improvement on each of them.

Setup. All 119 commands are trained as single commands at initialization and rehearsed in every round (Section A.1.4). We evaluate every checkpoint on the single-command test and on pipelines of two to six commands in three test sets: no held-out command (Section C.3), one held-out command (400 pipelines per length for lengths two and three and 1,000 for four to six), and only held-out commands (300 and 1,000 per length).

Results. Figure 17 shows that **composition transfers to commands that never appear in training compositions**, although less fully than to seen commands. Accuracy decreases as more commands in a pipeline are held out and as pipelines get longer, while the model still writes each held-out command correctly on its own, so the gap arises when held-out commands are composed. The third round narrows the gap, most visibly for pipelines built only from held-out commands. Direct self-improvement stays near zero on all three test sets.

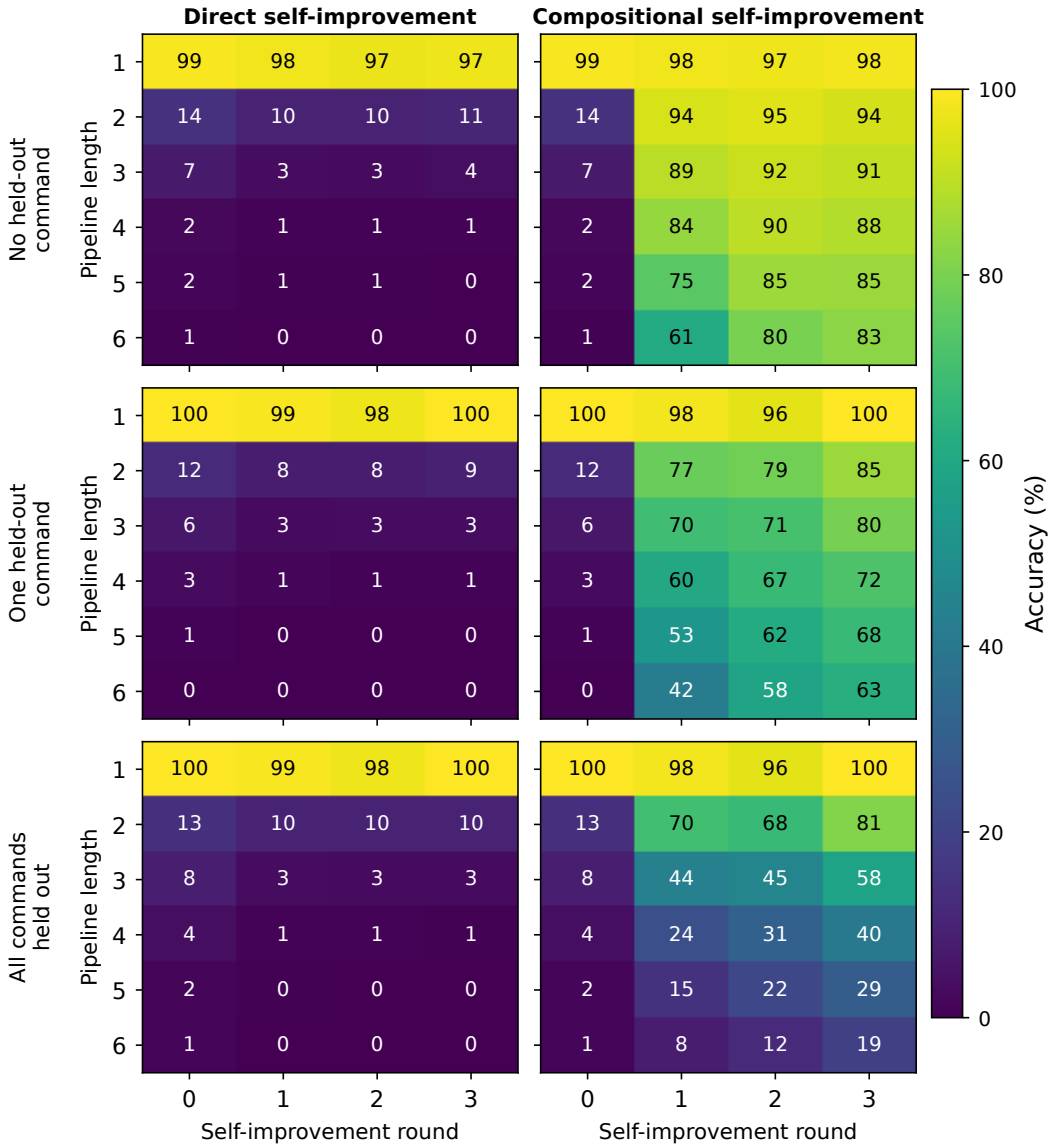


Figure 17: Bash accuracy by pipeline length and self-improvement round for test pipelines with no, one, or all commands held out. The same checkpoints are evaluated on three test sets, which differ in how many of their commands were excluded from training pipelines. Length 1 is the single-command test on the 89 seen commands (top) or the 30 held-out commands (middle and bottom). Cells average four seeds; round 0 is the shared initial model.

C.5 Multi-Hop Reasoning at All Model Sizes

Figure 7 shows per-hop accuracy for the 9B model only. Figure 18 shows it for all three model sizes.

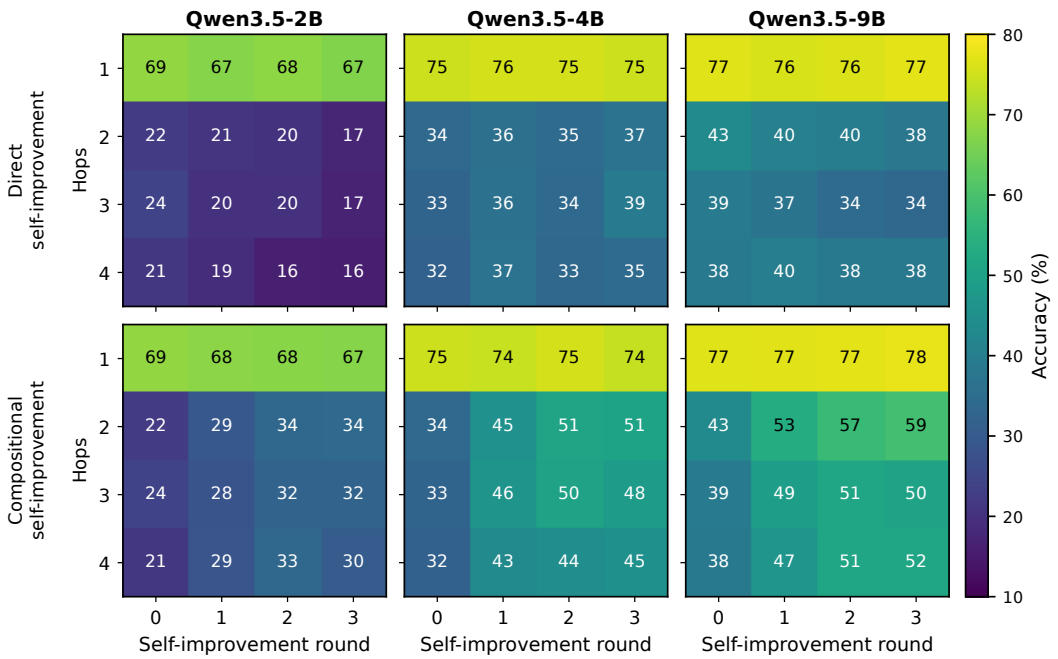


Figure 18: MuSiQue results for all three model sizes. Cells show direct-answer exact-match accuracy (%), averaged over three seeds. Top: direct self-improvement. Bottom: compositional self-improvement. The 9B panels repeat Figure 7. After three rounds, compositional self-improvement exceeds direct self-improvement on 2- to 4-hop questions by 15.5, 11.3, and 17.1 points on average for 2B, 4B, and 9B, respectively.

C.6 Decomposition of Composed-Label Error

Compositional self-improvement helps when composed labels are more accurate than the model's direct predictions on the same parents. This subsection first compares the two label sources and then asks where the errors of composed labels come from, splitting them into the two terms of Proposition B.1: errors inherited from the components and errors introduced by the composition rule.

Setup. We audit the saved labels of each round against reference answers after training; the audit never affects which labels are kept. Following Equation (3), all rates are computed on the accepted labels of one round:

- *wrong labels* (ρ): the share of labels that differ from the reference answer;
- *summed component errors* ($\hat{\Sigma} = \sum_i \epsilon_i$): the share of labels whose i-th component answer is wrong (ϵ_i), summed over components;
- *rule errors* (ξ): the share of parents whose label is wrong even with correct component answers.

In words, Proposition B.1 says that wrong labels cannot exceed summed component errors plus rule errors; this holds in every measured round. A component answer is the child route in grid navigation, recovered by splitting the label at the corridor actions (on six rounds of the seed-accuracy study, where child routes were also stored, the recovery matches them); each command in Bash, run on the reference output of the preceding commands (in rounds 2 and 3, the first component is the predicted multi-command prefix); each function answer in strings and typed Python; and each step answer in MuSiQue, scored against the released step answer without aliases. For addition and max run length, component answers were not logged, so only ρ and ξ are available: ξ is recorded during label generation for addition and measured on three regenerated candidate pools for max run length.

Findings.

- **Composed labels are far more accurate than direct labels.** A composed label only requires the model to solve components it already handles, while a direct label requires it to solve the harder parent. In Bash, 91% of composed labels are correct in the first round, compared with 8% of direct labels, and in MuSiQue at every model size and round (Table 8).
- **With a valid rule, every wrong label traces back to a wrong component.** No label is wrong when all of its components are correct, and almost every wrong component produces a wrong label (Table 7). Label quality is therefore set by the least reliable component, such as the multi-command prefix in Bash or the last functions of a string chain.
- **In MuSiQue, errors arise mostly at the final step.** There the model answers the original question from its own earlier answers, and this step has the highest error at every size and round (Table 8).
- **Without a filter, addition labels are wrong almost exactly when the rule is.** The share of wrong labels matches the share of parents with a boundary carry, so nearly all label errors come from the rule rather than the components. The filter removes these parents, after which few labels are wrong.
- **The filter keeps max run length labels within the coverage of the seed set.** A filtered label never contains a run longer than those in the seed set, as Equation (14) requires, so inputs with longer runs never receive composed supervision; Figure 5 evaluates such inputs.
- **In the grid study, label quality follows the accuracy of the initial model.** Every wrong label has a wrong child route, so weaker initial models produce mostly wrong labels (Table 9).

Table 7: Where the errors of composed labels come from (%), measured on the kept labels of one round. Wrong labels: share of labels that are wrong. Summed component errors: share of labels with a wrong answer in each component, summed over components. Rule errors: share of parents whose label would be wrong even if every component answer were correct. The last two columns count labels that are wrong although every component is correct, and labels that are correct although some component is wrong. Symbols refer to Proposition B.1. Addition and max run length logged no component answers, so only wrong labels and rule errors are available. Grid navigation rows cover all labels of rounds 1 and 12 (2 and 2–13 rooms), while Figure 19 reports only the newest room count.

Task	Labels	Kept	Wrong labels (ρ)	Summed component errors ($\hat{\Sigma}$)	Rule errors (ξ)	Wrong with correct components	Correct with a wrong component
Grid navigation	2 rooms	100	0.9	0.9	0.0	0.0	0.0
	2–13 rooms	100	3.5	3.5	0.0	0.0	0.0
Bash	length 2	100	8.8	9.1	0.0	0.0	0.1
	length 4	100	30.3	32.3	0.0	0.0	0.3
Strings	depth 5	100	23.0	26.8	0.0	0.0	0.0
Typed Python	depth 3	100	5.9	7.4	0.0	0.0	1.6
Addition, unfiltered	up to 9 digits	100	60.9	–	60.8	–	–
	up to 23 digits	100	83.5	–	83.3	–	–
Addition, filtered	all rounds	100	0.1–2.4	–	0.0	–	–
Max run length, unfiltered	length ≤ 20	100	5.9	–	5.9	–	–
	length ≤ 83	100	6.0	–	3.1	–	–
Max run length, filtered	all rounds	90.0	0.2–1.2	–	0.0	–	–

Table 8: Training labels in the MuSiQue experiment (%), averaged over three seeds. Label accuracy is scored with the official answer aliases. Step error gives, for each step of the composed labels in order, the share whose answer differs from the released step answer (scored without aliases); the last value is the final step. The last two rows give the share of wrong composed labels whose first wrong step is the final step, and the share of composed labels that are correct although some step answer differs from the released one. Label acceptance never uses reference answers.

Model	Quantity	Round 1 (2 hops)	Round 2 (3 hops)	Round 3 (4 hops)
2B	Compositional labels correct	44.8	42.3	49.9
	Direct labels correct	28.9	21.6	24.3
	Step error, step 1 to final	12.9, 58.3	15.9, 27.2, 61.8	19.7, 32.6, 33.8, 55.9
	Wrong labels, first error at final step	83.4	60.7	39.4
	Correct labels with a wrong step	6.8	15.9	30.8
4B	Compositional labels correct	60.5	63.1	65.6
	Direct labels correct	44.5	42.0	44.6
	Step error, step 1 to final	11.1, 43.2	12.7, 21.6, 42.5	13.3, 27.3, 29.0, 43.8
	Wrong labels, first error at final step	84.9	62.5	47.7
	Correct labels with a wrong step	8.2	19.6	36.3
9B	Compositional labels correct	65.2	66.5	70.5
	Direct labels correct	53.2	44.7	46.1
	Step error, step 1 to final	8.9, 39.3	11.5, 19.5, 40.0	13.2, 24.9, 28.2, 39.2
	Wrong labels, first error at final step	88.1	63.4	47.0
	Correct labels with a wrong step	8.9	18.6	36.8

Table 9: Composed-label error in the grid seed-accuracy study (Figure 3; %), averaged over three seeds. One-room accuracy is the initial model's test accuracy. Mean child error is $\hat{\Sigma}/2$ on the 2-room labels of the first round. In every cell, all wrong labels have at least one wrong child route and the rule error is 0.

Initial-training updates	One-room accuracy	Mean child error	ρ, 2 rooms	ρ, 13 rooms
400	28.4	76.3	94.3	99.4
1,500	58.3	44.1	68.3	93.7
2,500	73.7	29.1	49.7	86.0
3,000	79.9	21.4	38.1	77.1
6,000	90.4	9.2	17.5	48.5
8,000	95.0	5.3	10.3	32.2
30,000	99.4	0.6	1.2	4.7

C.7 Decomposed Inference with the Initial Model

Compositional self-improvement uses the decomposition only to build training labels; the trained model answers full problems in one pass. A natural alternative is to skip training and supply the decomposition at test time: the initial model answers each subproblem, and the composition rule combines its answers. This subsection compares the two.

Setup. Each test instance is split into subproblems of the seed regime, which the initial model answers greedily; the rule then combines the answers (Section A.1). Grid chains are split into single rooms, Bash pipelines into single commands, string chains and Python problems into single functions, and MuSiQue questions into their released sub-questions. Addends are split into blocks of 3 to 7 digits in which no block starts with 0, and max run length strings into $\lceil L/10 \rceil$ blocks of near-equal length. The rule is applied to every test instance, including those on which it is invalid: 23.3% of 8–23-digit test sums have no carry across a block boundary, and 66.2% of 11–83-symbol test strings have no run crossing a block boundary. Test sets and metrics are those of the main experiments, and so are the initial models, except that the initial model for max run length is retrained with its recorded settings; answering directly, it reaches 98–100% on the seed lengths (recorded: 99–100%) and 54.2% on lengths 11–83 (recorded: 59.8%). The "Initial, direct" columns report the recorded models.

Findings.

- **Where the rule holds on every instance, decomposed inference is more accurate than the final compositional model.** The initial model already answers most components correctly, for example 118 of the 119 single commands, so composing its answers at test time works well (Tables 10 and 11). The exception is 4-hop MuSiQue with the 4B and 9B models, where four sub-answers must all be correct.
- **Where the rule fails on part of the test set, the final compositional model is more accurate.** Decomposed inference inherits every failure of the rule: in addition, it answers correctly almost exactly the sums without a carry across a block boundary, and in max run length, it errs mainly on strings with a run that continues across a block boundary.
- **The two approaches need different information at test time.** Decomposed inference requires the reference decomposition and one model call per subproblem. Compositional self-improvement uses the decomposition only to build training labels, answers in one pass, and, with a filter, learns to handle instances on which the rule fails.

Table 10: Accuracy (%) of the initial model with direct inference and with decomposed inference, and of the final compositional model with direct inference. Decomposed inference receives the decomposition of each test instance, which direct inference does not. Rows with a size range average the accuracy per size. Bash values average four seeds for the final model; the initial model is shared. MuSiQue values average three seeds and use the 2,417 development questions of Figure 7.

Task	Test instances	Initial, direct	Initial, decomposed	Final round, direct
Grid navigation	2–13 rooms	0.0	88.6	80.9
	13 rooms	0.0	76.5	47.0
Addition	8–23 digits	0.0	23.2	93.7
	23 digits	0.0	4.0	80.0
Max run length	11–83 symbols	59.8	87.9	97.6
	83 symbols	37.0	81.0	94.0
Bash	2–6 commands	5.2	94.9	88.1
	6 commands	0.6	92.9	82.6
MuSiQue 2B	2 hops	22.4	50.5	33.8
	3 hops	24.0	48.9	32.4
	4 hops	21.4	36.8	30.4
MuSiQue 4B	2 hops	34.0	60.5	50.5
	3 hops	32.7	58.6	48.3
	4 hops	31.8	44.3	45.1
MuSiQue 9B	2 hops	43.1	63.4	59.2
	3 hops	39.2	55.7	50.3
	4 hops	38.3	43.9	51.5

Table 11: Test accuracy (%, 512 parents) in the RL warm-start experiments of Section 5 and Section C.9. The initial model is the shared initial model; the warmup and RL columns average three seeds, after the warmup and at the last checkpoint (40M tokens for strings, 524K for Python).

Task	Initial, direct	Initial, decomposed	After warmup	After warmup and RL
Strings, depth 5	0.0	77.3	11.5	19.2
Strings, depth 4	0.2	85.4	38.7	50.1
Length-preserving, depth 4	0.0	90.0	47.5	61.7
Typed Python	3.3	92.2	22.1	55.6

C.8 Pseudo-Label Accuracy by Round

This subsection reports the accuracy of the training labels in every round of the main experiments, audited against reference answers after training.

Setup. For the algorithmic tasks, a label is audited if its parent has the size added in that round; seed-regime replay is excluded. Grid routes are scored by execution, and addition and max run length labels by exact match. Bash labels are scored on all 24 inputs of their parent, and the RL warmup labels by the final answer (strings) or all 24 test inputs (typed Python). MuSiQue label accuracy by round is in Table 8.

Results. Composed labels remain more accurate than direct labels in every round. In grid navigation, 99.1% of composed labels are correct in round 1 and 93.8% in round 12, while every direct label is wrong (Figure 19). With the filter, max run length and addition labels stay above 96% in every round. Without it, max run length label accuracy decreases from 94.5% to 89.5% and addition label accuracy from 38.0% to 3.2%, since the share of boundary carries grows with the number of blocks. Direct max run length labels decrease from 87.1% to 54.9%, and direct addition labels are wrong in every round. In Bash (Table 12), composed label accuracy decreases from 90.6% in round 1 to 80.6% in round 3, and direct label accuracy from 7.9% to 0.5%. The RL warmup labels are correct for 77.0% of parents for strings of depth five and 94.1% for typed Python.

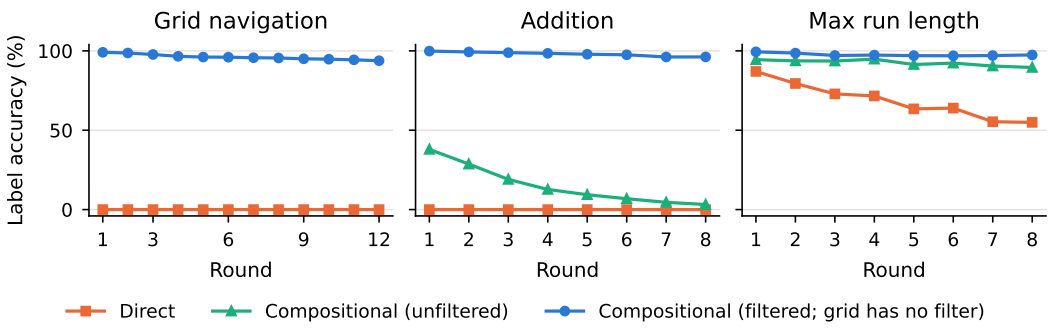


Figure 19: Accuracy of the training labels for the instance size added in each round, audited against reference answers after training. Grid navigation has no filter.

Table 12: Accuracy (%) of training labels. (a) Bash, by self-improvement round, averaged over four seeds. (b) The 512 labels of the single compositional warmup in each RL configuration.

(a) Bash

Label source	Round 1	Round 2	Round 3
Compositional	90.6	83.6	80.6
Direct	7.9	2.0	0.5

(b) RL warmup

Configuration	Warmup labels
Strings, depth 5	77.0
Strings, depth 4	87.5
Length-preserving, depth 4	87.7
Typed Python	94.1

C.9 RL on Other String Configurations

Figure 8 shows strings of depth five and typed Python. This subsection repeats the RL comparison on two string configurations of depth four: all 21 functions, and the 11 length-preserving functions.

Setup. Length-preserving strings use only the 11 functions whose output has the same length as their input, such as rotations, character shifts, and case changes (Section A.1). Both configurations follow the protocol of Section A.2. Test accuracy is evaluated after the warmup, at 40M generated tokens, and at 80M for direct RL.

Results. Figure 20 shows **the same pattern as strings of depth five**: the compositional warmup lets the policy obtain reward, RL improves it further, and direct RL stays near zero accuracy even with twice as many generated tokens.

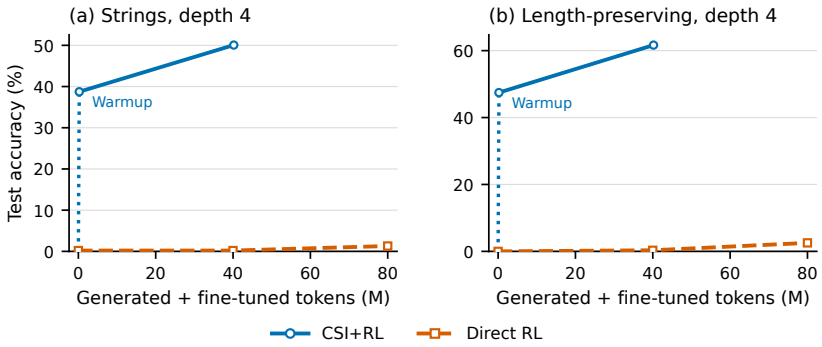


Figure 20: Test accuracy (512 held-out parents) against generated and fine-tuned tokens for the two depth-four string configurations, averaged over three seeds. The dotted segment is the compositional warmup; token accounting follows Figure 8.