FROM $f(x)$ AND $g(x)$ TO $f(g(x))$: LLMS LEARN NEW SKILLS IN RL BY COMPOSING OLD ONES

Anonymous authors
Paper under double-blind review

ABSTRACT

Does reinforcement learning (RL) teach large language models (LLMs) genuinely new skills, or does it merely activate existing ones? This question lies at the core of ongoing debates about the role of RL in LLM post-training. On one side, strong empirical results can be achieved with RL even without preceding supervised fine-tuning; on the other, critics argue that RL contributes little beyond reweighting existing reasoning strategies. This work provides concrete evidence that LLMs can acquire genuinely new skills during RL by composing existing ones, mirroring one of the central mechanisms by which humans acquire new cognitive skills (Anderson, 1982). To mitigate data contamination and other confounding factors, and to allow precise control over task complexity, we develop a synthetic framework for our investigation. Specifically, we define a skill as the ability to infer the output of a string transformation function $f(x)$ given x. When an LLM has already learned f and g prior to RL, our experiments reveal that RL enables it to learn unseen compositions of them $h(x) = g(f(x))$. Further, this compositional ability generalizes to more difficult problems such as compositions of > 2 functions unseen during RL training. Our experiments provide surprising evidence that this compositional ability, acquired on the source task, transfers to a different target task. This transfer occurs even though the model has never trained on any compositional problems in the target task, and the only requirement is that the model has acquired the target task's atomic skills before its RL training on the source. Our qualitative analysis shows that RL fundamentally changes the reasoning behaviors of the models. In contrast, none of the findings is observed in next-token prediction training with the same data. Our systematic experiments provide fresh insights into the learning behaviors of widely-used post-training approaches for LLMs. They suggest the value of building base models with the necessary basic skills, followed by RL with appropriate incentivization to acquire more advanced skills that generalize better to complex and out-of-domain problems.

1 INTRODUCTION

Reinforcement learning (RL) has achieved broad success in improving large language models (LLMs) on a variety of tasks especially reasoning (OpenAI, 2024; DeepMind, 2025), even directly building upon the base model without any preceding supervised fine-tuning (DeepSeek-AI et al., 2025). Despite the profound success, recent work finds the exploration of RL is impeded by the entropy collapse phenomenon (Cui et al., 2025b; Liu et al., 2025a; Yu et al., 2025b), and the performance gaps between base and RL-trained models diminish as the number of samples (k) increases in pass@k evaluations (Yue et al., 2025). In addition, some argue that the "aha moments" in RL training (OpenAI, 2024; DeepSeek-AI et al., 2025) are not emergent but merely the result of amplifying existing cognitive behaviors present in base models (Gandhi et al., 2025; Liu et al., 2025c; Zhao et al., 2025), which casts shadow on whether LLMs learn new skills during RL training (Wu et al., 2025b). Such observations diverge from established RL findings that predate LLMs, where models were trained from scratch and learned new skills (Silver et al., 2016; 2017; OpenAI et al., 2019). The fact that LLMs are pretrained on vast data prior to RL may contribute to these divergences and call for further investigation into the following important research questions: **(1) Does RL teach new skills to LLMs? (2) If so, how to incentivize it? (3) Are the skills generalizable?**

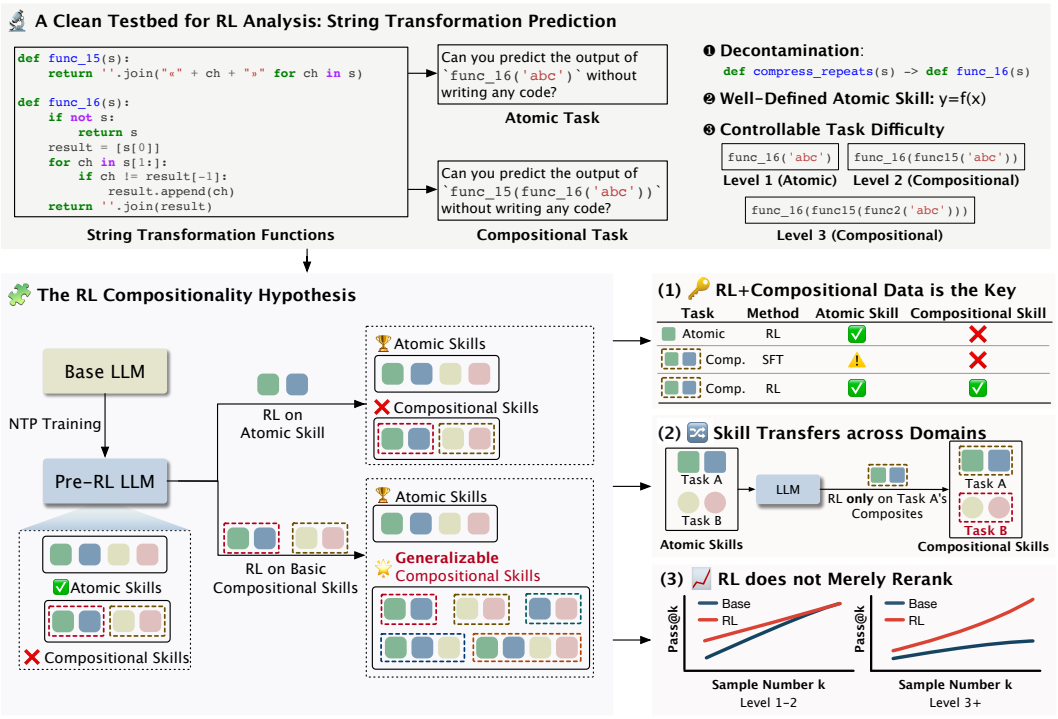

Figure 1: **An overview of our research framework and key findings. (Top)** We introduce a clean string transformation testbed to scientifically analyze RL's capabilities. **(Bottom-Left)** Our central RL Compositionality Hypothesis posits that training on simple composites with RL unlocks generalizable compositional skills. **(Bottom-Right)** Our experiments validate this, showing that: (1) compositional data combined with RL is the key ingredient for learning this new skill; (2) the learned skill transfers across domains; and (3) RL significantly improves difficult problems where the base model fails, while only reranking on problems it solves well.

Answering these questions will advance our understanding of LLM learning behaviors and inform the high-stakes trade-off in resource allocation between pretraining and post-training.

We provide concrete evidence that LLMs indeed learn new skills in RL, by composing and generalizing existing skills to solve more complex problems; For such learning to happen, there should be proper incentivization in RL. Our investigation is grounded in the cognitive skill acquisition process by humans, inspired by Anderson (1982), which argues that humans learn new skills by composing and then internalizing existing ones. Unlike prior works (Gandhi et al., 2025; Yue et al., 2025), we choose to construct a controlled synthetic framework that facilitates:

- **Decontaminated evaluation**: We design a string transformation prediction task with unique functions assigned meaningless identifiers (e.g., `func_16`) to prevent inference from function names. During RL, function definitions are hidden. The tasks will then be unsolvable without going through our atomic skills acquisition training. This setup enables us to investigate the RQs controlling for confounders.
- **Well-defined atomic and compositional skills**: We define atomic skills as single, non-decomposable transformations, and compositional skills as their nested combinations. For example, given input string x, `func_16(`x`)` represents an atomic skill, while `func_15(func_16(`x`))` requires compositional reasoning.
- **Controllable difficulty**: As each skill is instantiated as a Python function, we control difficulty of composition through the depth of nesting. As shown in Fig. 1, the model must perform deductive reasoning to give the output string after a given transformation, e.g., a Level-1 difficulty problem `func_16(`x`)` and a Level-2 one `func_16(func_15(`x`))`. Here the difficulty level is determined by the number of atomic functions composed.

With our framework and a two-stage training protocol that separates atomic from compositional skill acquisition, we conduct experiments with Llama-3.1-8B-Instruct (Dubey et al., 2024) and answer the RQs as follows:

- **RL teaches new compositional skills.** RL on Level-2 problems, receiving only correctness-based outcome rewards without reasoning demonstrations, substantially improves generalization on more difficult problems: performance on unseen Level-3 tasks improves from near-zero to 30%, and Level-4 to 15%. This generalization does not occur in a baseline trained with rejection fine-tuning (RFT) on the same Level-2 problems. This shows that RL enables the acquisition of compositional skills.
- **Both RL and compositional incentives are essential for skill acquisition.** In contrast to the substantial accuracy improvements from RL on Level-2 compositional problems, RFT on the same data and RL on Level-1 atomic problems both yield little improvements on problems higher than Level-2 (e.g., less than 1% improvement at Level-3). This may explain why Sun et al. (2025) conclude that RL does not promote compositional generalization, as their training includes no explicit incentive for composition.
- **The learning achieved by RL generalizes to held-out evaluation, more difficult problems, and even a different task.** All findings above are based on held-out evaluation of compositional problems consisting of atomic skills (functions) unseen in RL training. And as aforementioned, models RL-trained on Level 2 problems show non-trivial gains on problems up to Level 4. For cross-task transfer, compositional RL on the string task boosts accuracy on the unseen Level-3 Countdown problems to 35% for a model with the prerequisite Countdown atomic skills.

Our findings challenge the recent view that current RL with verifiable rewards (RLVR) (Lambert et al., 2024) merely utilizes reasoning patterns in base models rather than learning new reasoning abilities (Yue et al., 2025; Wu et al., 2025b). This view is based on the observation that the pass@k performance gap between RL-trained and base models narrows as k increases (Yue et al., 2025). We conjecture that this observation arises, at least in part, from evaluating and RL training on tasks where base models already achieve high pass@k, possibly due to pretraining on similar tasks that is beyond the control of most academic researchers; thus RL has little incentive to learn a skill that the base model already has. To confirm this conjecture, our experiments show that RL substantially improves pass@k on challenging compositional problems where base model's pass@k is near zero (See Fig. 5). This reveals what we term the "reranking illusion," namely aggregate metrics on mixed-difficulty benchmarks can mask genuine skill acquisition by conflating capabilities of different types. Our qualitative analysis confirms that models fundamentally change their reasoning behaviors after RL training. As shown in Fig. 6, compositional errors, i.e., ignoring composition and misunderstanding function relationships, drop substantially, while failures shift primarily to atomic prediction errors (55%). This behavioral transformation indicates genuine acquisition of compositional skills.

Our findings have important implications for LLM development and highlight RL's critical role in post-training, particularly its potential for easy-to-hard generalization and cross-task transfer. They call for closer coordination between base model development and post-training strategy from a skill acquisition perspective.

2 Background

The Recent Pessimistic View on Whether RL Teaches New Skills to LLMs. RL in LLMs builds on a model pretrained on vast data. While supervised warm-starts are a common technique in traditional RL (Silver et al., 2016; Vinyals et al., 2019; De La Cruz Jr et al., 2019; Silva & Gombolay, 2021), the large-scale and general-purpose nature of LLM creates a different scenario. On one hand, this strong prior enables base LLM to sample reasonable rollouts and thus perform RL directly without any preceding supervised fine-tuning (DeepSeek-AI et al., 2025; Pan et al., 2025; Zeng et al., 2025); on the other hand, it becomes difficult to distinguish genuine skill acquisition from activation of existing capabilities during RL training.

Recent work tries to investigate this but uses loose definitions of "skill", often relying on proxies such as the continually increasing frequency of certain reasoning patterns (Gandhi et al., 2025; Zhao et al., 2025; Liu et al., 2025c) or the diminishing gaps between the pass@k accuracy of models before and after RL, as shown in the bottom right chart in Fig. 1 (Yue et al., 2025; Liu et al., 2025b; Wu et al., 2025b; He et al., 2025; Wen et al., 2025; Zhu et al., 2025). Although these studies show that RL activates behaviors already present in the base model, they did not directly prove that no new skill is learned during the process. Moreover, the pass@k results can be misinterpreted for

many reasons: (1) The causal relation between performance and each skill remains unclear, thus it is not guaranteed that everything learned can be translated into improvements in pass@k accuracy on downstream tasks. (2) The evaluation tasks only provide an obscure overall view, lacking fine-grained analysis on problems of different difficulty levels or domains. (3) The result is confounded by the fact that the model may remain limited new skills to learn or lack the incentive to learn new skills if it already perform decently well before RL, which is possible if models perform RL on the same or similar data seen in next-token prediction (NTP) training (Wu et al., 2025d; Shao et al., 2025; Wang et al., 2025; Cui et al., 2025a; Yu et al., 2025b; Liu et al., 2025b; Wu et al., 2025c) Together, these highlight the urgent need for a deeper analysis of tasks through a clean framework, in which the skills are clearly defined and contribute to the performance causally, and evaluated in a finer granularity.

Compositional Learning as a Testbed Grounded in Cognitive Skill Acquisition in Humans. Although the pessimistic conclusions about RL in LLMs from prior works are debatable, they at least indicate that the success of RL depends on strong base models. This motivates our study of skill composition, where RL learns new abilities by leveraging those already acquired by the base model. Compositional reasoning provides an ideal framework for investigating skill acquisition because it naturally separates atomic knowledge, which mirrors how humans learn cognitive skills (Anderson, 1982). Notably, it is established in cognitive science that both composed skills and the meta-ability to learn composition are non-trivial new skills (Anderson, 1982; Lake et al., 2016). For clarity, we refer to learning new skills as the former throughout this paper. Learning compositional skills helps the model to generalize to more challenging problems and new domains beyond training data, which we will show later. In the field of AI, compositional reasoning has been widely studied before LLMs and has been considered a necessary property of generalization. (Fodor & Pylyshyn, 1988; Lake et al., 2016; Andreas et al., 2015). More recently, Yin et al. (2025) achieved compositional improvements through in-context learning rather than RL, while Sun et al. (2025) found that directly RL in atomic skills fails in compositional generalization. Comparing the two works, we conjecture that an explicit incentive to composition is necessary.

3 Research Framework

In this work, we define "new skills" as novel reasoning strategies that enable models to solve previously unsolvable problems through systematic combination of existing capabilities. We address three critical research questions: **(1) Does RL teach new skills to LLMs? (2) If so, how to incentivize it? (3) Are the learned skills generalizable?**

Hypothesis 1 (The RL Compositionality Hypothesis)**.** *Once a model has acquired the necessary atomic, non-decomposable skills for a task through NTP training, RL with proper incentivization can teach the model to learn new skills by composing atomic skills into more complex capabilities.*

3.1 Task Design: Deductive Reasoning on String Transformation Prediction

To test our hypothesis while avoiding confounders from data contamination and unclear skill boundaries, we design a controlled synthetic task with the following properties: (1) Atomic skills are well defined so that models can learn the fundamental skills separately before RL. Each string transformation function has clear, deterministic behaviors that can be learned independently. (2) Task difficulty can be controlled by adjusting the compositional complexity of the atomic skills, allowing us to test generalization across complexity levels. (3) RL and evaluation tasks do not appear in the LLM pretraining corpus, ensuring that improvements stem from learning rather than memorization.

Task Definition. Specifically, our task involves deductive reasoning on string transformations. Given an input string x and a composition of deterministic transformation functions such as $f(\cdot)$ and $g(\cdot)$, models must predict the output string after applying the specified transformation (e.g., $y = f(g(x))$). We construct 25 unique string transformation functions as atomic skill spanning various computational patterns including character manipulation, reordering, filtering, and structural modifications (see Appendix §D for complete specifications). To mitigate potential contamination, we assign meaningless identifiers to string functions as shown in Fig. 1, so that it is impossible to infer the functionality with function names only.

Difficulty Level. We control compositional complexity through **Difficulty Levels** corresponding to nesting depth, with Level n involving n-function composition. For instance, Level 1 involves single function application (e.g., `func_16(x)` as shown in Fig. 1), while Level 2 involves two-function composition (e.g., `func_16(func_15(x))`). The controlled difficulty provides a fine-grained inspection of model performance, rather than a vague overall number as adopted in prior work (Yue et al., 2025; Liu et al., 2025b; Wu et al., 2025b).

3.2 Training and Evaluation Protocol

Training consists of two stages to separate atomic skill acquisition from compositional skill learning, simulating realistic post-training pipelines.

Stage 1 Training: Atomic Skills Acquisition via RFT. Models learn "atomic skills" in this stage, i.e., receiving explicit function definitions alongside input strings and training via rejection fine-tuning Dong et al. (2023) on their own correct reasoning trajectories. This ensures models internalize each transformation function's behavior before attempting composition. Crucially, this is the only stage where models observe function implementations. An example can be found in Fig. 7.

Stage 2 Training: Compositional Skill Training via Either RFT or RL. In this stage, models see only function names and compositions, such as `func_2(func_16(x))`, with function definitions hidden. See Fig. 8 for examples. This forces reliance on internalized atomic knowledge while learning systematic composition. We compare two approaches: (1) **Composition via online RL** provides models with binary rewards based on output correctness and updates through Group Relative Preference Optimization (GRPO) Shao et al. (2024), testing whether RL is necessary for the acquisition of compositional skills. (2) **Composition via offline RFT** trains models with NTP on correct reasoning trajectories for compositional problems, serving as a baseline to examine whether exposure to compositional examples alone enables composition.

We use Llama-3.1-8B-Instruct, which is identified as a cleaner testbed for RL by recent work (Shao et al., 2025; Agarwal et al., 2025; Wu et al., 2025c), to further minimize the effect of data contamination besides our string tasks. For more details, please refer to Appendix A.

Held-out, Easy-to-Hard, and Cross-Task Evaluation. We assess generalization using rigorous held-out evaluation. In Stage 1, models are trained on all 25 atomic functions (Appendix D). In Stage 2, the functions are partitioned into two disjoint sets: the model trains only on compositions from one set, while the other is held out for constructing evaluation problems. We test model generalization across various difficulty levels, and adopt Countdown (Gandhi et al., 2024; Pan et al., 2025) as testbed for task transfer.

4 RL as a Pathway to Generalizable Skill Acquisition

4.1 LLMs Acquire New Compositional Skills during RL

Our first experiment directly test our RL Compositionality Hypothesis (Hypothesis 1). To do so, we start from an identical Stage 1 base model and apply three different Stage 2 training configurations, allowing us to isolate the impact of incentivizing composition during RL: (1) **RL Level 1**, trained only on atomic tasks; (2) **RL Level 2**, trained only on two-level compositions; and (3) **RL Level 1+2**, trained on a uniform mix. We then evaluate their ability to generalize to held-out tasks from Level 1 up to Level 6, testing whether they can solve problems with unseen function compositions and higher nesting levels than seen in RL training.

As shown in blue curves in Fig. 2, training on Level 1 alone leads to high accuracy on Level 1, peaking at around 90%, but fails to generalize. Its accuracy on Level 2 task remains below 25%, and on Level 3 through 6, it is consistently near *zero*. This demonstrates that learning only the atomic skills through RL is insufficient for learning effective composition.

In contrast, incorporating compositional tasks into RL training yields transformative results. Both the RL Level 2 and RL Level 1+2 models demonstrate strong performance to generalize to problems with nesting depths exceeding their training data. On Level 3, their accuracy improves from 5% to around 30%, and from 1% to 15% on Level 4, which are all significant improvements over the RL Level 1 model. And this trend continues on even Level 5, indicating both models learn a

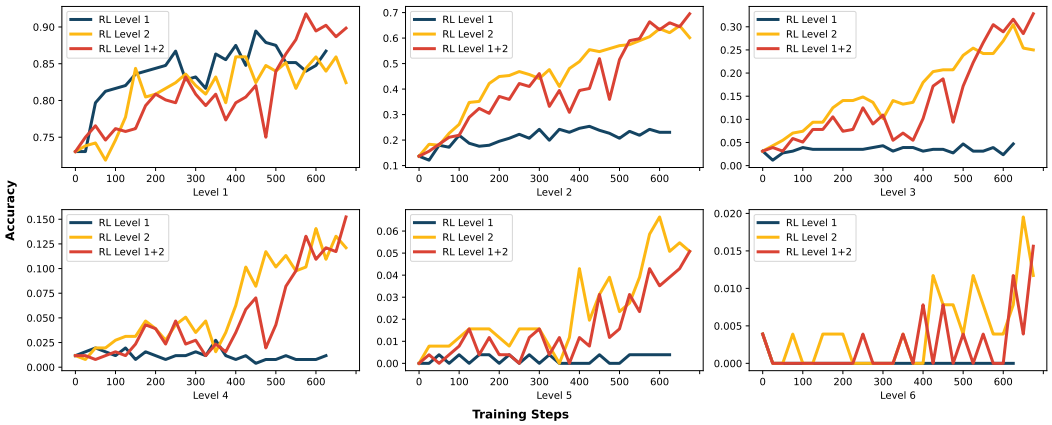

Figure 2: **Test Accuracy on held-out tasks vs. RL training steps, each related to one held-out task difficulty level.** The dark blue curve indicates that training on atomic skills alone (RL Level 1) yields nearly no compositional ability on held-out functions. In contrast, including Level 2 data in RL unlocks strong generalization to more complex problems (Levels 3-6).

generalizable principle of compositional reasoning rather than merely memorizing solutions. This validates our hypothesis that RL can teach genuinely new skills, but only when the training objective explicitly incentivizes their use. These results provide us with evidence to answer RQ1:

TAKEAWAY 1

RL on compositional data teaches new skills that generalize to unseen compositions of known atomic skills.

4.2 COMPOSITIONAL DATA IS THE INCENTIVE FOR RL TO TEACH COMPOSITIONAL SKILLS

Our previous experiment shows that compositional data is necessary for RL to teach new compositional skills, but **can a supervised method, such as RFT, achieve the same results as RL when given the exact same compositional (Level 2) data?** To address this question, we train a model with iterative RFT on the same Level 2 problems and conduct a head-to-head comparison against the RL Level 2 model from §4.1, with both having started from the identical Stage 1 base model.

The results in Fig. 3 show a significant difference in performance from Fig. 2. The RFT model's accuracy is significantly worse than RL across all compositional levels and has only marginal improvement over the first iteration. For example, on Level 3 it never surpasses 2.6%. In contrast, the RL Level 2 model achieves 64% on Level 2 and 27% on Level 3, significantly outperforming the RFT model. Surprisingly, the RFT model attains only 15% accuracy on Level-2 problems. This indicates that RFT fails to generalize even to held-out compositional problems of the same difficulty as its training data, let alone higher difficulties. These results provide the evidence to answer RQ2:

TAKEAWAY 2

RFT, even with on compositional data, is suboptimal for learning compositional skills; RL, in addition to compositional training data, is another important factor in learning generalizable compositional skills.

4.3 COMPOSITIONAL SKILLS LEARNED IN RL ARE TRANSFERABLE, BUT ATOMIC SKILLS ARE PREREQUISITES

While our experiments demonstrate that RL can teach generalizable compositional skills *within* a task, collecting compositional RL data for every new domain is impractical. We therefore test the transferability of the learned compositional skill. Specifically, we conjecture that RL enables models to compose atomic skills on Task B after learning composition on Task A, if the model has already acquired the necessary atomic skills for Task B.

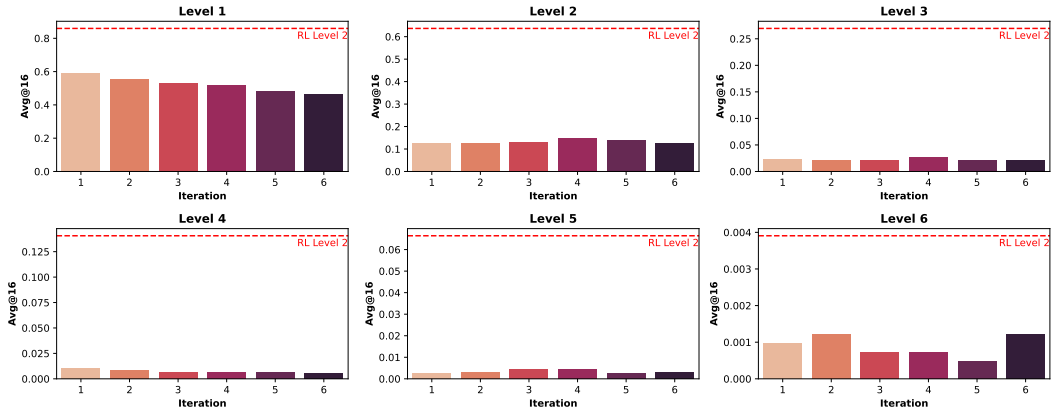

Figure 3: **RL vs. RFT on Compositional Tasks.** RL (red dashed line) achieves substantially higher accuracy across all levels, while iterative RFT fails to learn a generalizable skill.

Table 1: Model configurations for the task transfer experiment.

	Stage 1		Stage 2	
Model Configuration	**String Atomic RFT**	**Countdown Atomic RFT**	**String Atomic RL**	**String Comp. RL**
String-Base + RL L1+2	✓	×	×	✓
Multi-Base	✓	✓	×	×
Multi-Base + RL L1	✓	✓	✓	×
Multi-Base + RL L1+2	✓	✓	×	✓

Experimental Setup. We test this conjecture on the Countdown task, where a model must construct a mathematical expression from a given set of integers to reach a target number (see §E for examples). In Countdown, a Level ℓ task requires the model to construct a mathematical expression using ℓ given integers to reach a target number. The minimum level for Countdown is Level 2. We compare four models to test our hypothesis, as detailed in Tab. 1. These configurations allow us to compare a "atomic-skill-only" baseline (Multi-Base) against models with either transferred atomic RL (Multi-Base + RL L1) or transferred compositional RL (Multi-Base + RL L1+2), as well as a control model from §4.1 that has the compositional skill but lacks the necessary atomic knowledge of Countdown (String-Base + RL L1+2). Note that *none* of the models are trained on Countdown with RL in Stage 2, and are only trained on our string task.

We evaluate these models on unseen, more challenging Countdown problems (Levels 3-5). We report the Avg@32, the average accuracy across 32 responses sampled at temperature 1.0.

Results. The results in Fig. 4 provide clear evidence supporting our hypothesis. The String-Base + RL L1+2 model fails completely. The Multi-Base model achieves reasonable accuracy of approximately 17% at Level 3 but still struggles at higher levels. Multi-Base + RL L1 shows marginal improvement over Multi-Base, increasing accuracy to around 20% at Level 3, with the advantage diminishing on more complex problems. The Multi-Base + RL L1+2 model achieves surprisingly strong performance. It achieves a 35% accuracy at Level 3, outperforming the Multi-Base baseline by more than 18% accuracy. This advantage persists at higher complexities, reaching approximately 6% at Level 4, where other models largely fail and achieve near-zero accuracy. The results show that the compositional skill learned from string transformation transfers to countdown, acting as a *meta-skill* that enhances the use of the target task's atomic knowledge. Finally, the comparison between Multi-base + RL and String-Base + RL L1+2 confirms our fundamental assumption that task-specific atomic skills are prerequisites for compositional skills to be effective.

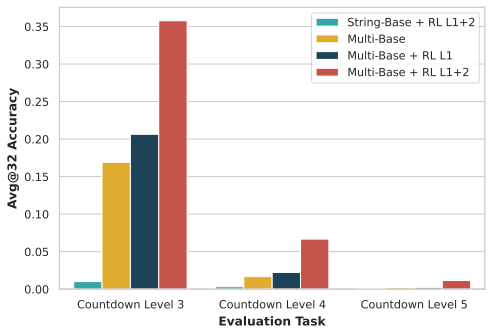

Figure 4: **Avg@32 Accuracy on the Countdown Task**. Atomic skills are a prerequisite for task transfer, and that compositional RL (Multi-Base + RL L1+2) on the unrelated string task offers a significant performance improvement on Countdown. Note that *none* of the models are trained with RL on Countdown.

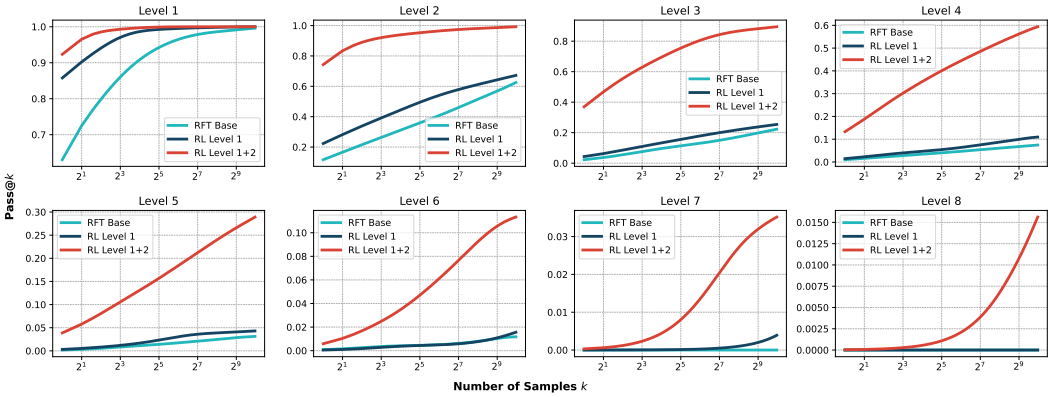

Figure 5: **Pass@k performance across varying difficulty levels**. On easy problems (Levels 1-2), the performance gap shrinks with more samples, consistent with the *reranking* narrative. On hard problems (Levels 3-8), the gap widens substantially, suggesting new skill acquisition.

These results may explain recent findings on generalizable RL improvements. For example, Logic-RL (Xie et al., 2025) reports performance gains on mathematical problems after training on logic puzzles, and Guru (Cheng et al., 2025) shows that domains with greater pre-training exposure benefit more from cross-task generalization. We suggest that LLMs have already acquired essential atomic skills through large-scale pre-training, particularly in mathematics and coding. Thus, incentivizing compositional skills through RL in one task helps combine task-specific skills more effectively across domains. In contrast, domains with less pre-training exposure may lack sufficient atomic skills, limiting compositional skill transfer to downstream tasks. With this finding, we answer RQ3:

TAKEAWAY 3

Compositional skills learned through RL are transferable to a different task where the model possesses the atomic skills.

4.4 RL EXPANDING PERFORMANCE LIMITS IS NOT A FALSE PROMISE

Our findings strongly suggest that RL can teach compositional skills that are novel to the base model. This directly challenges recent arguments that RL merely "reranks" model responses, distilling pass@k performance of the base model into pass@1 (Yue et al., 2025; Wu et al., 2025a). This conclusion is drawn based on a shrinking pass@k performance gap between base and RL-tuned models as k increases. However, we argue this conclusion may stem from two issues: (1) evaluating on mixed-skill benchmarks, therefore an improvement in a specific skill, like composition, can be masked in pass@k if other required skills remain a bottleneck, and (2) using RL training that does not properly incentivize the new skill in the first place.

Our controlled framework allows us to dissect both issues. By isolating the compositional skill at varying difficulty levels, we can reliably assess skill acquisition (addressing issue 1), and by comparing different RL training setups (§4.1), we can test the effect of proper incentivization (addressing issue 2). We compare pass@1000 performance at each difficulty level of our test set, selecting $k = 1000$ as a sufficiently large and practically meaningful budget. Larger budgets would become impractical, as any reasonable model could theoretically achieve pass@$\infty = 1$.

The results are presented in Fig. 5. Both RL Level 1 and RL Level 1+2 models are trained from RFT base model using RL in Stage 2. The RL Level 1 model, which is *not* incentivized properly to learn composition, exhibits a similar trend to the RFT base across almost all levels. On easier problems (Levels 1 and 2) where the RFT base model already shows solving potential evidenced by high pass@k, the performance gaps between RL Level 1+2 model and the RFT model shrink as k increases, aligning with the trends observed in Yue et al. (2025); Wu et al. (2025b). However, a completely different trend is observed on more challenging compositional problems (Levels 3-6). The RL Level 1+2 model's performance substantially outperforms the RFT base with an increasing gap as k grows. For example, at Level 5, the performance gap over the RFT base grows from 4% at pass@1 to approximately 25% at pass@1024. This divergence is clear evidence of new skill acquisition. The results suggest that the pessimistic observation of "RL does not push performance

limits" in prior work may be explained by the lack of incentive for RL to learn new skills, as the base model already achieves high pass@k performance.

TAKEAWAY 4

The prior conclusion that RLVR only utilizes base models' reasoning patterns without learning new abilities is likely an artifact of evaluating and RL training on tasks that base models already achieve high pass@k; thus RL has little incentive to learn a new skill.

4.5 BEHAVIORAL ANALYSIS: RL TRANSFORMS FAILURE MODES

While our results show that training with compositional data unlocks promising generalization, a fundamental question remains: **do models trained under different setups exhibit different behaviors, or do they simply differ in capability while showing similar failure modes?** To investigate this, we analyze the failure modes of different models on Level 3 problems of our string task.

We use Gemini-2.5-Pro to classify responses into five categories: (1) **Correct**, (2) **Ignores Composition** (e.g., analyzing only a single function), (3) **Incomplete Trace** (recognizes composition but terminates early), (4) **Incorrect Composition** (e.g., misinterprets nesting), and (5) **Atomic Error** (errors in atomic functions prediction without the above). Categories 2-4 indicate difficulties with handling compositional problems. And while still incorrect, category 5 represents appropriate compositional behavior, as the error is not due to a lack of compositional skill.

We compare four models: **RFT Base** (after Stage 1 training), **RFT Level 2** (after Stage 2 training on Level 2 problems with RFT), **RL Level 1**, and **RL Level 2**, all from previous sections. The latter three models are all trained from the RFT Base.

Fig. 6 reveals substantial similarities in the failure patterns of RFT Base, RFT Level 2, and RL Level 1 models. Their failures are dominated by ignoring the composition entirely (all >50%) and misunderstanding the compositional structure (all >35%).

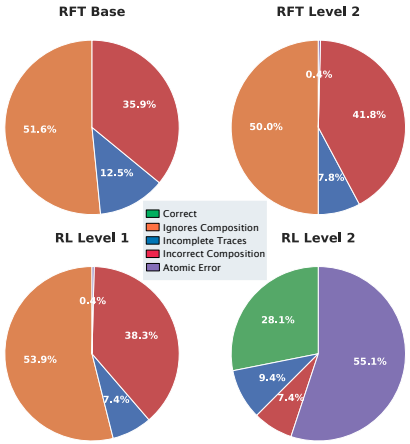

Figure 6: Distribution of failure modes on Level 3 string tasks.

In contrast, the RL Level 2 model demonstrates fundamentally different behaviors. It completely eliminates "Ignores Composition" errors and correctly solves 28.1% of the problems. Crucially, its primary failure mode becomes "Atomic Error." This shows that compositional RL not only improves accuracy but teaches models to parse and execute compositional plans, shifting failures from high-level misunderstandings to lower-level execution errors. See §F for examples of different model responses.

TAKEAWAY 5

Rather than merely improving accuracy, RL on compositional problems fundamentally transforms the model's behavior, enabling it to correctly understand and handle compositions.

5 CONCLUSION

The debate over whether RL can teach LLMs new skills has been clouded by experiments on benchmarks where LLMs already perform well, using coarse-grained metrics that obscure the learning of new capabilities. By stepping back to a cleaner, more controllable experimental environment, our findings provide a clear and optimistic answer: RL can teach genuinely new and powerful skills when the training task properly incentivize composition. Our results show that the compositional skills are learnable through RL and generalize across difficulty levels and different tasks. Our findings suggest that the pessimistic conclusion that RL does not learn new skills may stem from inappropriate evaluation setups rather than fundamental constraints of RL itself.

REFERENCES

Shivam Agarwal, Zimin Zhang, Lifan Yuan, Jiawei Han, and Hao Peng. The unreasonable effectiveness of entropy minimization in llm reasoning. *ArXiv*, abs/2505.15134, 2025. URL https://api.semanticscholar.org/CorpusID:278782149.

John R. Anderson. Acquisition of cognitive skill. *Psychological Review*, 89, 1982.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 39–48, 2015.

Zhoujun Cheng, Shibo Hao, Tianyang Liu, Fan Zhou, Yutao Xie, Feng Yao, Yuexin Bian, Yonghao Zhuang, Nilabjo Dey, Yuheng Zha, Yi Gu, Kun Zhou, Yuqi Wang, Yuan Li, Richard Fan, Jianshu She, Chengqian Gao, Abulhair Saparov, Haonan Li, Taylor W. Killian, Mikhail Yurochkin, Zhengzhong Liu, Eric P. Xing, and Zhiting Hu. Revisiting reinforcement learning for LLM reasoning from A cross-domain perspective. *CoRR*, abs/2506.14965, 2025. doi: 10.48550/ARXIV.2506.14965. URL https://doi.org/10.48550/arXiv.2506.14965.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, et al. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*, 2025a.

Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, Yuxin Zuo, Haozhan Li, Yuchen Fan, Huayu Chen, Weize Chen, Zhiyuan Liu, Hao Peng, Lei Bai, Wanli Ouyang, Yu Cheng, Bowen Zhou, and Ning Ding. The entropy mechanism of reinforcement learning for reasoning language models. *CoRR*, abs/2505.22617, 2025b. doi: 10.48550/ARXIV.2505.22617. URL https://doi.org/10.48550/arXiv.2505.22617.

Gabriel V De La Cruz Jr, Yunshu Du, and Matthew E Taylor. Pre-training with non-expert human demonstration for deep reinforcement learning. *The Knowledge Engineering Review*, 34:e10, 2019.

Google DeepMind. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *ArXiv*, abs/2507.06261, 2025.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Jun-Mei Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiaoling Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bing-Li Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dong-Li Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Jiong Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, M. Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, Ruiqi Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shao-Kang Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wen-Xia Yu, Wentao Zhang, Wangding Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xi aokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyu Jin, Xi-Cheng Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yi Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yu-Jing Zou, Yujia He, Yunfan Xiong, Yu-Wei Luo, Yu mei You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yao Li, Yi Zheng, Yuchen Zhu, Yunxiang Ma, Ying Tang, Yukun Zha, Yuting Yan, Zehui Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhen guo Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zi-An Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang.

Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Nature*, 645, 2025.

Hanze Dong, Wei Xiong, Deepanshu Goyal, Rui Pan, Shizhe Diao, Jipeng Zhang, Kashun Shum, and T. Zhang. Raft: Reward ranked finetuning for generative foundation model alignment. *ArXiv*, abs/2304.06767, 2023.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony S. Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aur'elien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cris tian Cantón Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab A. AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriele Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guanglong Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Ju-Qing Jia, Kalyan Vasuden Alwala, K. Upasani, Kate Plawiak, Keqian Li, Ken-591 neth Heafield, Kevin R. Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuen ley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melissa Hall Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Niko lay Bashlykov, Nikolay Bogoychev, Niladri S. Chatterji, Olivier Duchenne, Onur cCelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasić, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ron nie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sa hana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Chandra Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Vir ginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whit ney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yiqian Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zhengxu Yan, Zhengxing Chen, Zoe Papakipos, Aaditya K. Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adi Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Po-Yao (Bernie) Huang, Beth Loyd, Beto de Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Shang-Wen Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco Guzm'an,

Frank J. Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory G. Sizov, Guangyi Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Han Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kaixing(Kai) Wu, U KamHou, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, A Lavender, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthias Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Mun ish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navy ata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pe dro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollár, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shiva Shankar, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Kumar Gupta, Sung-Bae Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Andrei Poenaru, Vlad T. Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xia Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. The llama 3 herd of models. *ArXiv*, abs/2407.21783, 2024.

Jerry A. Fodor and Zenon W. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28:3–71, 1988.

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of search (sos): Learning to search in language. *ArXiv*, abs/2404.03683, 2024.

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, nathan lile, and Noah D. Goodman. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective stars. *COLM*, 2025.

Andre He, Daniel Fried, and Sean Welleck. Rewarding the unlikely: Lifting grpo beyond distribution sharpening. *arXiv preprint arXiv:2506.02355*, 2025.

Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *CoRR*, abs/1604.00289, 2016.

Nathan Lambert, Jacob Daniel Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James Validad Miranda, Alisa Liu, Nouha Dziri, Xinxi Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Christopher Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hanna Hajishirzi. Tülu 3: Pushing frontiers in open language model post-training. *ArXiv*, abs/2411.15124, 2024.

Mingjie Liu, Shizhe Diao, Ximing Lu, Jian Hu, Xin Dong, Yejin Choi, Jan Kautz, and Yi Dong. Prorl: Prolonged reinforcement learning expands reasoning boundaries in large language models. *CoRR*, abs/2505.24864, 2025a. doi: 10.48550/ARXIV.2505.24864. URL https://doi.org/10.48550/arXiv.2505.24864.

Mingjie Liu, Shizhe Diao, Ximing Lu, Jian Hu, Xin Dong, Yejin Choi, Jan Kautz, and Yi Dong. Prorl: Prolonged reinforcement learning expands reasoning boundaries in large language models. *NeurIPS*, 2025b.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. *COLM*, 2025c.

OpenAI. Openai o1 system card. *ArXiv*, 2024.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand. *CoRR*, abs/1910.07113, 2019. URL http://arxiv.org/abs/1910.07113.

Jiayi Pan, Junjie Zhang, Xingyao Wang, Lifan Yuan, Hao Peng, and Alane Suhr. Tinyzero. https://github.com/Jiayi-Pan/TinyZero, 2025. Accessed: 2025-01-24.

Rulin Shao, Shuyue Stella Li, Rui Xin, Scott Geng, Yiping Wang, Sewoong Oh, Simon Shaolei Du, Nathan Lambert, Sewon Min, Ranjay Krishna, Yulia Tsvetkov, Hanna Hajishirzi, Pang Wei Koh, and Luke S. Zettlemoyer. Spurious rewards: Rethinking training signals in rlvr. *ArXiv*, abs/2506.10947, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Jun-Mei Song, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *ArXiv*, abs/2402.03300, 2024.

Andrew Silva and Matthew Gombolay. Encoding human domain knowledge to warm start reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pp. 5042–5050, 2021.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016. doi: 10.1038/NATURE16961. URL https://doi.org/10.1038/nature16961.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815.

Yiyou Sun, Shawn Hu, Georgia Zhou, Ken Zheng, Hanna Hajishirzi, Nouha Dziri, and Dawn Xiaodong Song. Omega: Can llms reason outside the box in math? evaluating exploratory, compositional, and transformative generalization. *ArXiv*, abs/2506.18880, 2025.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019. doi: 10.1038/S41586-019-1724-Z. URL https://doi.org/10.1038/s41586-019-1724-z.

Yiping Wang, Qing Yang, Zhiyuan Zeng, Liliang Ren, Liyuan Liu, Baolin Peng, Hao Cheng, Xuehai He, Kuan Wang, Jianfeng Gao, et al. Reinforcement learning for reasoning in large language models with one training example. *arXiv preprint arXiv:2504.20571*, 2025.

Xumeng Wen, Zihan Liu, Shun Zheng, Zhijian Xu, Shengyu Ye, Zhirong Wu, Xiao Liang, Yang Wang, Junjie Li, Ziming Miao, et al. Reinforcement learning with verifiable rewards implicitly incentivizes correct reasoning in base llms. *arXiv preprint arXiv:2506.14245*, 2025.

Fang Wu, Weihao Xuan, Ximing Lu, Zaïd Harchaoui, and Yejin Choi. The invisible leash: Why RLVR may not escape its origin. *CoRR*, abs/2507.14843, 2025a. doi: 10.48550/ARXIV.2507.14843. URL https://doi.org/10.48550/arXiv.2507.14843.

Fang Wu, Weihao Xuan, Ximing Lu, zaid. harchaoui, and Yejin Choi. The invisible leash: Why rlvr may not escape its origin. *ArXiv*, abs/2507.14843, 2025b.

Haoze Wu, Cheng Wang, Wenshuo Zhao, and Junxian He. Mirage or method? how model-task alignment induces divergent rl conclusions. 2025c. URL https://api.semanticscholar.org/CorpusID:280985268.

Mingqi Wu, Zhihao Zhang, Qiaole Dong, Zhiheng Xi, Jun Zhao, Senjie Jin, Xiaoran Fan, Yuhao Zhou, Huijie Lv, Ming Zhang, et al. Reasoning or memorization? unreliable results of reinforcement learning due to data contamination. *arXiv preprint arXiv:2507.10532*, 2025d.

Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. Logic-rl: Unleashing LLM reasoning with rule-based reinforcement learning. *CoRR*, abs/2502.14768, 2025. doi: 10.48550/ARXIV.2502.14768. URL https://doi.org/10.48550/arXiv.2502.14768.

Fangcong Yin, Zeyu Leo Liu, Liu Leqi, Xi Ye, and Greg Durrett. Learning composable chains-of-thought. *ArXiv*, abs/2505.22635, 2025.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025a.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Weinan Dai, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: an open-source LLM reinforcement learning system at scale. *CoRR*, abs/2503.14476, 2025b. doi: 10.48550/ARXIV.2503.14476. URL https://doi.org/10.48550/arXiv.2503.14476.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *ArXiv*, 2025.

Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun Ma, and Junxian He. Simplerl-zoo: Investigating and taming zero reinforcement learning for open base models in the wild. *CoRR*, abs/2503.18892, 2025. doi: 10.48550/ARXIV.2503.18892. URL https://doi.org/10.48550/arXiv.2503.18892.

Rosie Zhao, Alexandru Meterez, Sham M. Kakade, Cengiz Pehlevan, Samy Jelassi, and Eran Malach. Echo chamber: RL post-training amplifies behaviors learned in pretraining. *CoRR*, abs/2504.07912, 2025. doi: 10.48550/ARXIV.2504.07912. URL https://doi.org/10.48550/arXiv.2504.07912.

Xinyu Zhu, Mengzhou Xia, Zhepei Wei, Wei-Lin Chen, Danqi Chen, and Yu Meng. The surprising effectiveness of negative reinforcement in llm reasoning. *arXiv preprint arXiv:2506.01347*, 2025.

A Training Details

A.1 Stage 1

All models in our experiments, except for the Multi-Base variants in §4.3, start from the same Stage 1 base model. The process for creating this model is detailed below.

Data Generation. We first generate 50k Level-1 problems by randomly sampling a function and a string input (length 3 to 10). For each problem, we collect 10 responses from the `Llama-3.1-8B-Instruct` model (temperature 1.0, max length 8192). To focus training on problems the model finds non-trivial, we discard any problems where the base model achieves 100% accuracy. We then collect all remaining correct responses to form the SFT dataset, resulting in around 116k training instances.

Rejection Fine-Tuning. We fine-tune the model for 2 epochs with a learning rate of 2×10^{-5} and a global batch size of 128. Other hyperparameters follow the default settings provided by the veRL framework[1].

A.2 Stage 2

Data Generation. The problems used in Stage 2 are created using a similar process as in Stage 1, with the critical differences being that function definitions are hidden and problems can involve composition. For Level-2 problems, we randomly select and compose two functions. Example prompts for Stage 1 vs. Stage 2 can be found in Appendix C. For the Level 1 only and Level 2 only dataset, we create 50k problems. For the Level 1+2 mixed configuration, we create 25k problems for each level and combine them.

Reinforcement Learning. For RL experiments, we use DAPO (Yu et al., 2025a) as the optimization algorithm. We enforce a strictly on-policy setup by setting both the training batch size and mini-batch size to 16. For each prompt, we generate 16 rollouts using a sampling temperature of 1.0 and a maximum response length of 8192. During training, we filter out any problems for which all rollouts are correct or all are incorrect. We use a learning rate of 1×10^{-6} and set the coefficients for both KL divergence and entropy loss to 0.

Rejection Fine-Tuning. For the iterative RFT baseline, we use a learning rate of 2×10^{-5} and a batch size of 128. The process is iterative: the model from the previous iteration is used to generate a new set of rollouts. These rollouts are then filtered for correctness to construct a new SFT dataset, following the same procedure used in Stage 1. The model is then trained on this new dataset to produce the model for the next iteration.

B Evaluation Details

String Transformation Prediction Task. For the string transformation prediction task, the evaluation set consists of 256 randomly generated problems for each level from 1 to 8, resulting in a total of 2048 test problems. During evaluation, we generate responses using a sampling temperature of 1.0 and a maximum response length of 8192 tokens.

Countdown Task. The evaluation set for the Countdown task consists of 128 problems for each level. The evaluation setup is the same as evaluating on the string transformation prediction task.

C Example Prompts for Stage 1 and Stage 2

Here we present the example prompts that we use in Stage 1 and Stage 2 (Fig. 7 and Fig. 8). The biggest difference is that in Stage 2, we do not provide the model with the function definition, and the model must rely on the atomic skills that are learned during Stage 1 to solve.

[1] `https://github.com/volcengine/verl`

```
You are given a code:

def func_16(s):
    """Remove adjacent duplicate characters (compress repeats)."""
    if not s:
        return s
    result = [s[0]]
    for ch in s[1:]:
        if ch != result[-1]:
            result.append(ch)
    return ''.join(result)

def main_solution(x):
    return func_16(x)

Can you predict the output of `main_solution("tihess")` without writing any code? Please reason and put
 your final answer in the following json format: {"output": <your output>}, where <your output> should
be the final string.
```

Figure 7: Example prompt for Stage 1 training. Note that here the function definition is given, in contrast to Stage 2 training and evaluation.

Example prompt for Stage 2 Level 1 training

```
You are given a code:

def main_solution(x):
    return func_16(x)

Can you predict the output of `main_solution("tiheass")` without writing any code? Please reason and
put your final answer in the following json format: {"output": <your output>}, where <your output>
should be the final string.
```

Example prompt for Stage 2 Level 2 training

```
You are given a code:

def main_solution(x):
    return func_2(func_16(x), 3)

Can you predict the output of `main_solution("tiheass")` without writing any code? Please reason and
put your final answer in the following json format: {"output": <your output>}, where <your output>
should be the final string.
```

Figure 8: Example prompts for evaluation and Stage 2 training.

D A Complete List of String Transformation Functions

Here we provide the full list of string functions we use. Note that we replace the function names with meaningless identifiers such as `func_1` in our experiment.

```python
def deterministic_shuffle(s):
    """Reorder characters using a fixed multiplier permutation."""
    L = len(s)
    if L == 0:
        return s
    multiplier = 3
    while gcd(multiplier, L) != 1:
        multiplier += 2
    return ''.join(s[(i * multiplier) % L] for i in range(L))


def repeat_str(s, n):
    """Repeat the string s exactly n times."""
    return s * n


def remove_vowels(s):
    """Remove vowels from the string."""
    vowels = 'aeiouAEIOU'
    return ''.join(ch for ch in s if ch not in vowels)


def sort_chars(s):
    """Sort the characters in the string."""
    return ''.join(sorted(s))


def reverse_words(s):
    """Reverse the order of words in the string."""
```

```
    words = s.split()
    return ' '.join(reversed(words))


def add_prefix(s, pre):
    """Add a fixed prefix to the string."""
    return pre + s


def add_suffix(s, suf):
    """Add a fixed suffix to the string."""
    return s + suf


def interlace_str(s1, s2):
    """Interlace two strings character by character (iterative)."""
    result = []
    len1, len2 = len(s1), len(s2)
    for i in range(max(len1, len2)):
        if i < len1:
            result.append(s1[i])
        if i < len2:
            result.append(s2[i])
    return ''.join(result)

def rotate_str(s, n):
    """Rotate the string s by n positions using slicing."""
    if not s:
        return s
    n = n % len(s)
    return s[n:] + s[:n]


def mirror_str(s):
    """Append the reversed string to the original."""
    return s + s[::-1]


def alternate_case(s):
    """Alternate the case of characters (even-index lower, odd-index
    upper)."""
    return ''.join(ch.lower() if i % 2 == 0 else ch.upper() for i, ch in
     enumerate(s))


def shift_chars(s, shift):
    """
    Shift alphabetical characters by a fixed amount (wrapping around).
    Non-letters remain unchanged.
    """

    def shift_char(ch):
        if 'a' <= ch <= 'z':
            return chr((ord(ch) - ord('a') + shift) % 26 + ord('a'))
        elif 'A' <= ch <= 'Z':
            return chr((ord(ch) - ord('A') + shift) % 26 + ord('A'))
        return ch

    return ''.join(shift_char(ch) for ch in s)


def vowel_to_number(s):
    """Replace vowels with numbers: a/A->1, e/E->2, i/I->3, o/O->4, u/U
    ->5."""
```

```
    mapping = {'a': '1', 'e': '2', 'i': '3', 'o': '4', 'u': '5', 'A': '1
    ', 'E': '2', 'I': '3', 'O': '4', 'U': '5'}
    return ''.join(mapping.get(ch, ch) for ch in s)


def insert_separator(s, sep):
    """Insert a fixed separator between every two characters."""
    return sep.join(s)


def duplicate_every_char(s):
    """Duplicate every character in the string."""
    return ''.join(ch * 2 for ch in s)


def fancy_brackets(s):
    """Enclose each character in fancy brackets."""
    return ''.join("<<" + ch + ">>" for ch in s)


def compress_repeats(s):
    """Remove adjacent duplicate characters (compress repeats)."""
    if not s:
        return s
    result = [s[0]]
    for ch in s[1:]:
        if ch != result[-1]:
            result.append(ch)
    return ''.join(result)


def recursive_reverse(s):
    """Recursively reverse the string."""
    if s == "":
        return s
    return recursive_reverse(s[1:]) + s[0]


def loop_concat(s, n):
    """Concatenate s with itself n times using a loop."""
    result = ""
    for _ in range(n):
        result += s
    return result

def while_rotate(s, n):
    """Rotate the string using a while loop (n times)."""
    count = 0
    while count < n and s:
        s = s[1:] + s[0]
        count += 1
    return s


def recursive_interlace(s1, s2):
    """Recursively interlace two strings character by character."""
    if not s1 or not s2:
        return s1 + s2
    return s1[0] + s2[0] + recursive_interlace(s1[1:], s2[1:])


def loop_filter_nonalpha(s):
    """Remove non-alphabetic characters using an explicit loop."""
    result = ""
    for ch in s:
```

```
        if ch.isalpha():
            result += ch
    return result


def verify_even_length(s):
    """
    Verification operator: if the length of s is even, return s;
    otherwise remove the last character.
    """
    return s if len(s) % 2 == 0 else s[:-1]


def backchain_add_digit(s, depth):
    """
    Backtracking operator: deterministically transform s so it contains
    a digit.
    Applies a fixed sequence of transformations recursively.
    """

    def has_digit(t):
        return any(ch.isdigit() for ch in t)

    transformations = [
        lambda t: t + "1",
        lambda t: "2" + t,
        lambda t: t.replace("a", "3"),
        lambda t: t[::-1],
    ]

    def helper(t, d):
        if has_digit(t):
            return t
        if d == 0:
            return None
        for trans in transformations:
            new_t = trans(t)
            res = helper(new_t, d - 1)
            if res is not None:
                return res
        return None

    result = helper(s, depth)
    return result if result is not None else s


def backchain_palindrome(s, depth):
    """
    Back chaining: try to transform s into a palindrome.
    If s is not already a palindrome and depth permits, append its
    reverse and try again.
    """
    if s == s[::-1]:
        return s
    if depth <= 0:
        return s
    new_s = s + s[::-1]
    return backchain_palindrome(new_s, depth - 1)

# [note: the string concatenation, i.e., '+' is also considered as a
function]
```

Listing 1: All functions

E EXAMPLE FOR COUNTDOWN TASK

```
Level 2 Example
Using the numbers [32, 42], create an equation that equals 74. You can use basic arithmetic operations
(+, -, *, /) and each number can only be used once. Show your work in <think> </think> tags. And return
 the final answer in <answer> </answer> tags, for example <answer> (1 + 2) / 3 * 4 </answer>.

Level 3 Example
Using the numbers [95, 14, 18], create an equation that equals 99. You can use basic arithmetic
operations (+, -, *, /) and each number can only be used once. Show your work in <think> </think> tags.
 And return the final answer in <answer> </answer> tags, for example <answer> (1 + 2) / 3 * 4 </answer
>.
```

Figure 9: Example prompts for Countdown task.

F MODEL RESPONSE EXAMPLES

The following sections contain the complete, unfolded responses from each model for the case studies.

F.1 FULL RESPONSES TO THE LEVEL 2 PROBLEM

RFT Model Response

To predict the output of ‘main_solution(”nar”)‘, we need to understand how the function ‘func_7‘ interlaces two input strings. *[Authors’ Note: It’s not wrong, but it misses another function func_2.]*

Here’s a step-by-step breakdown:

1. ‘func_7(s1, s2)‘ is called with ‘’vptqj’‘ as the first string ‘s1‘ and ‘”nar”‘ as the second string ‘s2‘.
2. The function iterates over the maximum length between the two input strings. In this case, the maximum length is 4, which is the length of ‘”nar”‘.
3. The function appends characters from both strings at each index ‘i‘ to the ‘result‘ list. If an index is out of range for a particular string, it simply skips appending a character from that string.
4. So, the ‘result‘ list will contain characters from both strings, interleaved.

Here’s the interleaved sequence:

- Index 0: ‘’v’‘ (from ‘’vptqj’‘) and ‘’n’‘ (from ‘”nar”‘), so ‘result = [’v’, ’n’]‘
- Index 1: ‘’p’‘ (from ‘’vptqj’‘) and ‘’a’‘ (from ‘”nar”‘), so ‘result = [’v’, ’n’, ’p’, ’a’]‘
- Index 2: ‘’t’‘ (from ‘’vptqj’‘) and ‘’r’‘ (from ‘”nar”‘), so ‘result = [’v’, ’n’, ’p’, ’a’, ’t’, ’r’]‘
- Index 3: ‘’q’‘ (from ‘’vptqj’‘) and ‘’ ’‘ (from ‘”nar”‘), so ‘result = [’v’, ’n’, ’p’, ’a’, ’t’, ’r’, ’q’,”]

Finally, we pass this ‘result‘ list to the ‘join()‘ function to form the final interleaved string.

The final output of ‘main_solution(”nar”)‘ will be: ‘”vnpartqj”‘. *[Authors’ Note: End too early, there’s still one function that hasn’t been analyzed.]*

So, the JSON output should be:

```
{"output": "vnpartqj"}
```

RL Level 1 Model Response

To predict the output of ‘main_solution(”nar”)‘, we need to understand how the function ‘func_7‘ interlaces two strings character by character.

Here’s a step-by-step breakdown:

1. ‘func_7‘ takes two strings ‘s1‘ and ‘s2‘ as input.
2. It creates an empty list ‘result‘ to store the interlaced characters.
3. It calculates the lengths of ‘s1‘ and ‘s2‘ and stores them in ‘len1‘ and ‘len2‘, respectively.
4. It loops from ‘0‘ to the maximum of ‘len1‘ and ‘len2‘ using a ‘for‘ loop.
5. Inside the loop, for each iteration ‘i‘, it checks if ‘i‘ is less than ‘len1‘ and ‘len2‘. If it is, it appends the ‘i-th‘ character of ‘s1‘ and ‘s2‘ to the ‘result‘ list, if it exists.
6. After the loop, it returns the ‘result‘ list as a string by joining all its elements using ‘”.join(result)‘.

Now, let’s analyze the ‘main_solution‘ function:

‘def main_solution(x): return (func_7(’vptqj’, x) + func_2(’xbh’))‘

Here, ‘s1‘ is ‘’vptqj’‘ and ‘s2‘ is ‘x‘, which is ‘”nar”‘ in this case.

Now, let’s interlace the two strings character by character:

‘i |s1 |s2 |result‘
–|—-|—-|———
0 |’v’ |’n’ |[’v’, ’n’]
1 |’p’ |’a’ |[’v’, ’n’, ’p’, ’a’]
2 |’t’ |’r’ |[’v’, ’n’, ’p’, ’a’, ’t’, ’r’]
3 |’q’ ||[’v’, ’n’, ’p’, ’a’, ’t’, ’r’, ’q’]
4 |’j’ ||[’v’, ’n’, ’p’, ’a’, ’t’, ’r’, ’q’, ’j’]
5 |||[’v’, ’n’, ’p’, ’a’, ’t’, ’r’, ’q’, ’j’]
6 |||[’v’, ’n’, ’p’, ’a’, ’t’, ’r’, ’q’, ’j’]

The loop iterates up to the maximum of the lengths of ‘s1‘ and ‘s2‘, which is 5 in this case. After the loop, ‘func_7‘ returns the ‘result‘ list as a string by joining all its elements.

The ‘result‘ list is ‘[’v’, ’n’, ’p’, ’a’, ’t’, ’r’, ’q’, ’j’]‘.

The ‘func_2‘ function is used to remove vowels from a string. It takes a string ‘s‘ as input and returns a new string with all vowels removed.

In the ‘main_solution‘ function, ‘func_2‘ is used to remove vowels from the string ‘’xbh’‘, but the output of ‘func_7‘ is used as the input for ‘func_2‘ *[Authors’ Note: the model mistakes the relationship between the two functions]*. However, in the ‘main_solution‘ function, ‘func_2‘ is not actually used with the interlaced string but with the string ‘’xbh’‘.

The ‘main_solution‘ function returns the result of ‘func_7(’vptqj’, ’nar’) + func_2(’xbh’)‘.

Since ‘func_2‘ is not used with the interlaced string, the output of ‘main_solution(”nar”)‘ will be the interlaced string ‘[’v’, ’n’, ’p’, ’a’, ’t’, ’r’, ’q’, ’j’]‘ as a string, which is:

‘vnpatrqj‘

So, the output of ‘main_solution(”nar”)‘ is:

```
{"output": "vnpatrqj"}
```

RL Level 2 Model Response

To predict the output of ‘main_solution(”nar”)‘, we need to understand what the function ‘func_7‘ does and how it is used in ‘main_solution‘.

‘func_7‘ interlaces two strings character by character. It takes two strings ‘s1‘ and ‘s2‘ as input and returns a new string where characters from ‘s1‘ and ‘s2‘ are alternately appended.

In ‘main_solution‘, ‘func_7‘ is called with ‘’vptqj’‘ as the first string and the input ‘x‘ as the second string. The result of ‘func_7‘ is then concatenated with the result of ‘func_2(’xbh’)‘.

Now, let’s consider what happens when ‘main_solution‘ is called with ‘”nar”‘ as the input.

First, we need to call ‘func_7‘ with ‘’vptqj’‘ and ‘”nar”‘ as the two strings.

The characters in ‘’vptqj’‘ are ‘’v’, ’p’, ’t’, ’q’, ’j’‘.

The characters in ‘”nar”‘ are ‘’n’, ’a’, ’r’‘.

Now, let’s interlace these characters using ‘func_7‘.

The maximum length of the two strings is 5. So, ‘func_7‘ will iterate up to 5 times.

Here’s what happens during each iteration:

1. ‘i = 0‘: ‘s1[i] = ’v’‘ and ‘s2[i] = ’n’‘. The result list is ‘[’v’, ’n’]‘.
2. ‘i = 1‘: ‘s1[i] = ’p’‘ and ‘s2[i] = ’a’‘. The result list is ‘[’v’, ’n’, ’p’, ’a’]‘.
3. ‘i = 2‘: ‘s1[i] = ’t’‘ and ‘s2[i] = ’r’‘. The result list is ‘[’v’, ’n’, ’p’, ’a’, ’t’, ’r’]‘.
4. ‘i = 3‘: ‘s1[i] = ’q’‘ and ‘s2[i] = ”‘. The result list is ‘[’v’, ’n’, ’p’, ’a’, ’t’, ’r’, ’q’]‘.
5. ‘i = 4‘: ‘s1[i] = ’j’‘ and ‘s2[i] = ”‘. The result list is ‘[’v’, ’n’, ’p’, ’a’, ’t’, ’r’, ’q’, ’j’]‘.

The final result list after iterating over all characters from both strings is ‘[’v’, ’n’, ’p’, ’a’, ’t’, ’r’, ’q’, ’j’]‘.

Now, we need to find the characters in ‘’xbh’‘ and remove them from ‘func_2‘. The characters in ‘’xbh’‘ are ‘’x’, ’b’, ’h’‘.

Now, let’s consider what ‘func_2‘ does. This function removes all vowels from a string.

The input string to ‘func_2‘ is ‘’xbh’‘. The vowels in ‘’xbh’‘ are ‘”‘ (there are no vowels in ‘’xbh’‘).

However, we need to remove all vowels from ‘’xbh’‘. Since there are no vowels in ‘’xbh’‘, the string remains the same.

The function ‘func_2‘ will return ‘’xbh’‘ as is.

Now, let’s consider what happens when we concatenate the result from ‘func_7(’vptqj’, ”nar”)‘ with ‘func_2(’xbh’)‘. As explained earlier, ‘func_7(’vptqj’, ”nar”)‘ will return ‘’vnpatrqj’‘.

Now, since ‘func_2(’xbh’)‘ just returns ‘’xbh’‘, the final output of ‘main_solution(”nar”)‘ will be the concatenation of ‘’vnpatrqj’‘ and ‘’xbh’‘.

The concatenation of ‘’vnpatrqj’‘ and ‘’xbh’‘ is ‘’vnpatrqjxbh’‘.

So, the final output of ‘main_solution(”nar”)‘ is ‘’vnpatrqjxbh’‘.

Here’s the JSON output:

```
{"output": "vnpatrqjxbh"}
```

F.2 Full Responses to the Level 3 Problem

RFT Model Response

To predict the output of ‘main_solution(”htoek”)‘, let’s analyze the code.

1. The ‘main_solution‘ function calls ‘func_14‘ with the input string ‘x‘.
2. ‘func_14‘ takes a string ‘s‘ and returns a new string where each character’s case is alternated based on its index.
- Even-index characters (i.e., those at 0, 2, 4, etc. indices) are converted to lowercase.
- Odd-index characters (i.e., those at 1, 3, 5, etc. indices) are converted to uppercase.
3. The input string ‘x‘ to ‘main_solution‘ is ‘”htoek”‘.

Now, let’s apply the logic of ‘func_14‘ to the input string ‘”htoek”‘:

- Index 0 is even, so ’h’ is converted to lowercase: ’h’.
- Index 1 is odd, so ’t’ is converted to uppercase: ’T’.
- Index 2 is even, so ’o’ is converted to lowercase: ’o’.
- Index 3 is odd, so ’e’ is converted to uppercase: ’E’.
- Index 4 is even, so ’k’ is converted to lowercase: ’k’.

The resulting string is ‘’hToEk’‘.

Therefore, the output of ‘main_solution(”htoek”)‘ is: *[Authors’ Note: Ends too early, there’s still one function that hasn’t been analyzed]*

```
{"output": "hToEk"}
```

RL Level 1 Model Response

To predict the output of ‘main_solution(”htoek”)‘, we need to understand what the ‘func_10‘ function does and how it is used in ‘main_solution‘.

‘func_10‘ takes a string ‘s‘ and returns a new string where:

* Even-indexed characters are converted to lowercase
* Odd-indexed characters are converted to uppercase

‘main_solution‘ calls ‘func_10‘ with the string ‘x‘ after applying the ‘func_2‘ function, which removes all vowels from ‘x‘. *[Authors’ Note: the model fails to correctly understand and parse the nesting function]*

Now, let’s apply these steps:

1. ‘func_2‘ removes vowels from ‘”htoek”‘, resulting in ‘”htk”‘.
2. ‘func_10‘ is applied to ‘”htk”‘.

- ‘h‘ is at index 0 (even), so it becomes ‘h‘ (lowercase).
- ‘t‘ is at index 1 (odd), so it becomes ‘T‘ (uppercase).
- ‘o‘ is at index 2 (even), so it becomes ‘o‘ (lowercase).
- ‘e‘ is at index 3 (odd), so it becomes ‘E‘ (uppercase).
- ‘k‘ is at index 4 (even), so it becomes ‘k‘ (lowercase).

The resulting string is ‘”hToEk”‘.

However, the question asks us to predict the output without writing any code, so we might have mistakenly included the characters from the original string ‘”htoek”‘ instead of the output of ‘func_2‘.

To correctly predict the output, we should first apply ‘func_2‘ to ‘”htoek”‘. The vowels in ‘”htoek”‘ are ”o” and ”e”. Removing these vowels leaves us with ‘”htk”‘.

Now, applying ‘func_10‘ to ‘”htk”‘:
- ‘h‘ is at index 0 (even), so it becomes ‘h‘ (lowercase).
- ‘t‘ is at index 1 (odd), so it becomes ‘T‘ (uppercase).
- ‘k‘ is at index 2 (even), so it becomes ‘k‘ (lowercase).

The resulting string is ‘”hTk”‘.

This is the output of ‘func_10(”htk”)‘, which is what ‘main_solution‘ returns.

Therefore, the output of ‘main_solution(”htoek”)‘ is ‘”hTk”‘.

The final answer is:

```
{"output": "hTk"}
```

RL Level 2 Model Response

To predict the output of ‘main_solution(”htoek”)‘, let’s analyze the code.

1. The ‘func_2‘ function removes all vowels from a given string ‘s‘.
2. The ‘func_14‘ function takes a string ‘s‘ and returns a new string where every character in ‘s‘ is duplicated.
3. The ‘func_10‘ function iterates over the characters in the input string ‘s‘. It makes the characters at even indices lower case and the characters at odd indices upper case.

Now, let’s see what happens when we call ‘main_solution(”htoek”)‘.

First, we need to find the output of ‘func_2(”htoek”)‘.

The string ”htoek” contains the characters ”h”, ”t”, ”o”, ”e”, ”k”. All characters that are vowels (i.e., ”o” and ”e”) will be removed from the string.

After removing vowels, the string ‘func_2(”htoek”)‘ will contain only the characters ”h”, ”t”, ”k”.

Now, we need to find the output of ‘func_14(”htoek”)‘.

The characters in ”htoek” are ”h”, ”t”, ”o”, ”e”, ”k”.

Applying ‘func_14‘ to each character:
- ”h” becomes ”hh”

- ”t” becomes ”tt”
- ”o” becomes ”oo”
- ”e” becomes ”ee”
- ”k” becomes ”kk”

The string ‘func_14(”htoek”)‘ will be ”hhttooeekk”.

Now, we need to find the output of ‘func_10(”hhttooeekk”)‘.

We will iterate over the characters in ”hhttooeekk” along with their indices.

The characters in ”hhttooeekk” and their indices are:
- ”h” at index 0
- ”h” at index 1
- ”t” at index 2
- ”t” at index 3
- ”o” at index 4
- ”o” at index 5
- ”e” at index 6
- ”e” at index 7
- ”k” at index 8
- ”k” at index 9

Now, let’s apply the function ‘func_10‘ to ”hhttooeekk”.

- The character at index 0 (”h”) will become lower case because the index is even. So, ”h” becomes ”h”.
- The character at index 1 (”h”) will become upper case because the index is odd. So, ”h” becomes ”H”.
- The character at index 2 (”t”) will become lower case because the index is even. So, ”t” becomes ”t”.
- The character at index 3 (”t”) will become upper case because the index is odd. So, ”t” becomes ”T”.
- The character at index 4 (”o”) will become lower case because the index is even. So, ”o” becomes ”o”.
- The character at index 5 (”o”) will become upper case because the index is odd. So, ”o” becomes ”O”.
- The character at index 6 (”e”) will become lower case because the index is even. So, ”e” becomes ”e”.
- The character at index 7 (”e”) will become upper case because the index is odd. So, ”e” becomes ”E”.
- The character at index 8 (”k”) will become lower case because the index is even. So, ”k” becomes ”k”.
- The character at index 9 (”k”) will become upper case because the index is odd. So, ”k” becomes ”K”.

The final string ‘func_10(”hhttooeekk”)‘ will be ”hHtToOeEkK”.

Now, let’s call ‘main_solution(”htoek”)‘.

Since ‘main_solution(x)‘ calls ‘func_10(func_14(func_2(x)))‘, we need to find the output of ‘func_14(func_2(”htoek”))‘ first.

We already found that ‘func_2(”htoek”)‘ will contain only the characters ”h”, ”t”, ”k”.

Now, let’s apply ‘func_14‘ to ”htk”.

The characters in ”htk” are ”h”, ”t”, ”k”.

Applying ‘func_14‘ to each character:
- ”h” becomes ”hh”
- ”t” becomes ”tt”
- ”k” becomes ”kk”

The string ‘func_14(”htk”)‘ will be ”hhttkk”.

Now, we need to find the output of ‘func_10(”hhttkk”)‘.

The characters in ”hhttkk” and their indices are:
- ”h” at index 0
- ”h” at index 1
- ”t” at index 2
- ”t” at index 3
- ”k” at index 4
- ”k” at index 5

Now, let’s apply the function ‘func_10‘ to ”hhttkk”.

- The character at index 0 (”h”) will become lower case because the index is even. So, ”h” becomes ”h”.
- The character at index 1 (”h”) will become upper case because the index is odd. So, ”h” becomes ”H”.
- The character at index 2 (”t”) will become lower case because the index is even. So, ”t” becomes ”t”.
- The character at index 3 (”t”) will become upper case because the index is odd. So, ”t” becomes ”T”.
- The character at index 4 (”k”) will become lower case because the index is even. So, ”k” becomes ”k”.
- The character at index 5 (”k”) will become upper case because the index is odd. So, ”k” becomes ”K”.

The string ‘func_10(”hhttkk”)‘ will be ”hHtTkK”.

Now, we need to call ‘main_solution(”htoek”)‘.

As ‘main_solution(x)‘ calls ‘func_10(func_14(func_2(x)))‘, the output of ‘main_solution(”htoek”)‘ will be the output of ‘func_10(func_14(func_2(”htoek”)))‘.

We already found that ‘func_14(”hhttooeekk”)‘ will be ”hHtToOeEkK”.

Now, we need to find the output of ‘func_10(”hHtToOeEkK”)‘.

We already calculated this in the previous step. The output of ‘func_10(”hHtToOeEkK”)‘ will be ”hHtToOeEkK”.

However, we need to recall that ‘func_2‘ removes vowels from the string. So, to find the output of ‘func_10(func_14(func_2(”htoek”)])‘, we need to first find the output of ‘func_14(func_2(”htoek”))‘.

We already calculated this earlier. The output of ‘func_14(func_2(”htoek”))‘ will be ”hhttkk”.

Now, we need to find the output of ‘func_10(”hhttkk”)‘.

We already calculated this earlier. The output of ‘func_10(”hhttkk”)‘ will be ”hHtTkK”.

The ‘main_solution‘ function calls ‘func_10(func_14(func_2(x)))‘. To find the final output of ‘main_solution(”htoek”)‘, we need to find the output of ‘func_14(func_2(”htoek”))‘.

The string ”htoek” has characters ”h”, ”t”, ”o”, ”e”, ”k”. The vowels in this string are ”o” and ”e”.

If we remove these vowels from ”htoek”, we are left with ”htk”.

Now, let’s apply ‘func_14‘ to ”htk”.

The characters in ”htk” are ”h”, ”t”, ”k”.

Applying ‘func_14‘ to each character:
- ”h” becomes ”hh”
- ”t” becomes ”tt”
- ”k” becomes ”kk”

The string ‘func_14(”htk”)‘ will be ”hhttkk”.

Now, let’s apply ‘func_10‘ to ”hhttkk”.

We already calculated this earlier. The output of ‘func_10(”hhttkk”)‘ will be ”hHtTkK”.

The ‘main_solution‘ function calls ‘func_10(func_14(func_2(x)))‘. To find the final output of ‘main_solution(”htoek”)‘, we need to find the output of ‘func_14(func_2(”htoek”))‘.

The characters in ”htoek” are ”h”, ”t”, ”o”, ”e”, ”k”. The vowels in this string are ”o” and ”e”.

If we remove these vowels from ”htoek”, we are left with ”htk”.

Now, let’s apply ‘func_14‘ to ”htk”.

We already calculated this earlier. The output of ‘func_14(”htk”)‘ will be ”hhttkk”.

Now, we can apply ‘func_10‘ to ”hhttkk”.

We already calculated this earlier. The output of ‘func_10(”hhttkk”)‘ will be ”hHtTkK”.

So, the output of ‘main_solution(”htoek”)‘ will be ”hHtTkK”.

The final answer is:

```
{"output": "hHtTkK"}
```