FROM EXAMPLES TO SOLUTIONS: A COGNITIVE FRAMEWORK FOR LLM CODE GENERATION

Shashwat Saxena*, **Shreyas Kowshik***, **Vikhyath Kothamasu***
Carnegie Mellon University
{ssaxena2,skowshik,vkothama}@cs.cmu.edu

ABSTRACT

Humans learn complex reasoning tasks, such as coding, by studying worked examples that reveal patterns and edge cases, a cognitive strategy largely underutilized in current AI training paradigms. Humans solve coding problems through a cognitive abstraction of solved examples, using them to base their thought process before writing code—a process that mirrors how a coach demonstrates techniques before an athlete performs them independently. Inspired by this cognitive process, we introduce COACH (**CO**gnitive **A**bstraction **C**onditioning for code **H**elp), a framework where an example generator produces step-by-step solved examples that condition a solution generator, improving pass@1 accuracy by **32%** over vanilla GRPO-based RLVR on the MBPP dataset. The key mechanism is joint training via GRPO, where both models receive the same execution-based reward, incentivizing the example generator to produce examples that genuinely aid solution generation rather than superficial reasoning—mimicking how effective coaching adapts demonstrations to maximize learner success. Additionally, COACH demonstrates improved sample efficiency, achieving comparable performance to the baseline with less than $\frac{2}{5}$th of the training data. For RLVR-based code generation, our results suggest that explicitly modeling intermediate reasoning artifacts through cognitive abstraction—rather than end-to-end training—yields substantial gains. More broadly, this work demonstrates that grounding AI training in human cognitive patterns offers a promising path toward more effective and interpretable reasoning systems.

1 INTRODUCTION

A central goal in building robust AI systems is to bridge the gap between human cognitive processes and machine learning. While Large Language Models (LLMs) have demonstrated strong capabilities in code generation Rozière et al. (2023), they often lack the structured reasoning and generalization abilities that characterize human intelligence Gendron et al. (2024). Cognitive science research suggests that humans benefit from studying prior solutions when approaching novel problems Chi et al. (1989b); Sweller & Cooper (1985a). A well-documented strategy is the use of *worked examples*, where learners study step-by-step solutions to analogous problems before attempting new challenges. This cognitive pattern, known as the "worked-example effect," has been studied in educational psychology and shown to reduce cognitive load while improving problem-solving performance Sweller (1988a); Renkl (1997).

In this work, we ask: *Can we incorporate this human cognitive strategy into AI reasoning to create more effective code generation systems?* We propose COACH (**CO**gnitive **A**bstraction **C**onditioning for code **H**elp), a framework that explicitly models the human practice of learning from worked examples. The name COACH reflects two key inspirations from cognitive science: (1) the process mirrors how a coach demonstrates techniques through concrete examples before an athlete performs independently, and (2) cognitive abstraction is the fundamental mechanism by which humans distill patterns from examples to guide future problem-solving. Our approach introduces an *example generator* that produces step-by-step solved examples, and a *solution generator* that conditions on these examples to produce code. This design instantiates the cognitive science principle that structured

*Equal contribution

intermediate representations aid reasoning. Rather than treating LLM reasoning as an opaque end-to-end process, we decompose it into interpretable stages: first, understanding the problem through a concrete example (cognitive abstraction), then applying that understanding to generate a solution (conditioned execution). We train both models jointly using Reinforcement Learning with Verifiable Rewards (RLVR) (Wen et al., 2025), **?** where execution-based feedback provides a grounded learning signal.

Our experiments demonstrate that incorporating this human-inspired cognitive structure leads to substantial improvements: COACH achieves a **32%** improvement over vanilla RLVR in both pass@1 and pass@16 metrics while also being more sample efficient, requiring less than $\frac{2}{5}^{th}$ of the data for comparable performance. These results suggest that explicitly modeling human cognitive patterns, specifically the use of worked examples as reasoning scaffolds, offers a promising direction for building AI systems that reason more like humans.

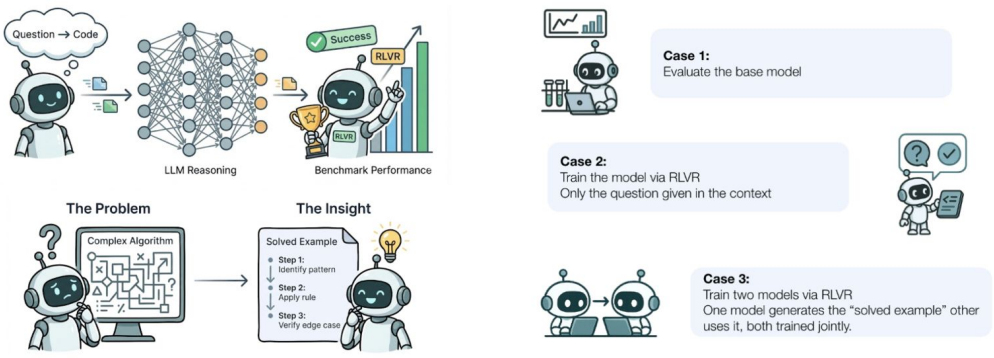


Figure 1: Overview of COACH (COgnitive Abstraction Conditioning for code Help). Left: motivation—RLVR improves benchmark performance, and humans solve complex tasks by studying worked examples that reveal patterns and edge cases. Right: experimental setup comparing (1) the base model, (2) standard RLVR trained only on the question context, and (3) COACH, where one model generates a "solved example" (cognitive abstraction) that the other model uses as additional context (conditioning) during joint training.

Contributions

1. We introduce COACH, a cognitively-inspired framework that incorporates the worked-example effect from human learning into AI reasoning for code generation through explicit cognitive abstraction and conditioning
2. We demonstrate that COACH leads to a **32%** improvement over vanilla RLVR in both pass@1 and pass@16 metrics, with improved sample efficiency, validating the effectiveness of modeling human cognitive patterns
3. We present qualitative analysis showing how COACH's structured intermediate representations (solved examples) help AI systems handle edge cases that end-to-end approaches miss, providing insights into interpretable reasoning

2 RELATED WORK

Human Cognition: Learning from Worked Examples, Analogy, and Feedback. Cognitive science and educational psychology have long shown that humans rarely approach novel problems tabula rasa; instead, they rely on prior solution traces, analogical mappings, and corrective feedback to reduce search and improve transfer (Sweller & Cooper, 1985b; Atkinson et al., 2000; Gentner, 1983). The *worked-example effect* argues that studying step-by-step solutions can outperform unguided problem solving for novices by reducing extraneous cognitive load and supporting schema acquisition (Sweller & Cooper, 1985b; Sweller, 1988b; Atkinson et al., 2000). Beyond passive exposure, learners benefit from *self-explanation*—actively justifying intermediate steps and linking them to

underlying principles—which improves conceptual understanding and generalization (Chi et al., 1989a). A complementary account comes from analogical reasoning, where structural alignment between a solved "base" and a novel "target" guides inference and problem decomposition (Gentner, 1983). These findings motivate cognitively-inspired training signals that provide structured traces, intermediate checks, and opportunities to reflect on mistakes rather than optimizing only final outcomes.

Cognitively Inspired Refinement: Reflection, Textual Gradients, and Neural–Symbolic Grounding. Several recent methods operationalize human-like reflection by turning critiques into learning signals. Reflexion (Shinn et al., 2023) improves agent performance through verbal feedback and episodic memory of failures, while TextGrad (Yuksekgonul et al., 2024) treats natural-language feedback as "pseudo-gradients" that can drive iterative optimization of textual artifacts. In code generation, CodeGrad (Zhang et al., 2025) pushes this further by integrating formal verification signals into the refinement loop, converting multi-step checks and constraints into structured textual updates—a neurosymbolic pathway that connects symbolic validity to neural generation. By conditioning on relevant critiques, the training signal can be enhanced to provide strong information on

Learning Reasoning Abstractions and Verification Artifacts. To address sparse binary rewards and degenerate exploration, recent RLVR work learns intermediate *abstractions* that guide solution construction. RLAD (Qu et al., 2025) introduces a two-player setup in which an abstraction generator proposes high-level strategies or lemmas and a solution generator implements them, encouraging structured exploration and better generalization. In code domains, abstractions can be instantiated as *verification artifacts* such as unit tests. Sol-Ver (Lin et al., 2025) alternates solver and verifier roles within a self-play loop to jointly improve code and test generation, and CURE (Wang et al., 2025) co-evolves a coder and unit tester via interaction-based rewards without requiring ground-truth solutions. More broadly, Self-Questioning Language Models (Chen et al., 2025) propose an auto-curriculum where a proposer generates tasks (and, for coding, tests) that are neither trivial nor impossible, pushing a solver to improve through self-generated challenges. Together, these approaches align with cognitively motivated views of reasoning as the acquisition and reuse of reusable procedures, supported by external checks and iterative correction.

3 METHOD

3.1 PROBLEM FORMULATION AND BASELINES

Code generation can be framed as learning a policy π_θ that, given a natural language problem description q, generates executable code c that satisfies a set of test cases. We compare COACH against two baseline approaches that represent standard paradigms in LLM code generation.

3.1.1 CHAIN-OF-THOUGHT PROMPTING

Chain-of-Thought (CoT) prompting is a simple yet remarkably effective way to induce reasoning in LLMs. By explicitly prompting the LLM to think step-by-step, along with guiding it to generate a formatted output, we query the base LLM to output code for a given problem. The detailed prompt can be found in the appendix: A.1.

3.1.2 GRPO-BASED RLVR

Reinforcement Learning with Verifiable Rewards (RLVR) has emerged as a viable paradigm to train LLMs when a verifier is available to validate the generated outputs. We use Group Relative Policy Optimization (GRPO), a variant of PPO adapted for LLM training.

Given a distribution over problems $P(Q)$ and a language-model policy π_θ, GRPO samples G rollouts per problem to generate solutions. We define the probability ratio and clipped ratio as:

$$r_t(\theta) = \frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})}, \quad r_t^{\text{clip}}(\theta) = \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon) \tag{1}$$

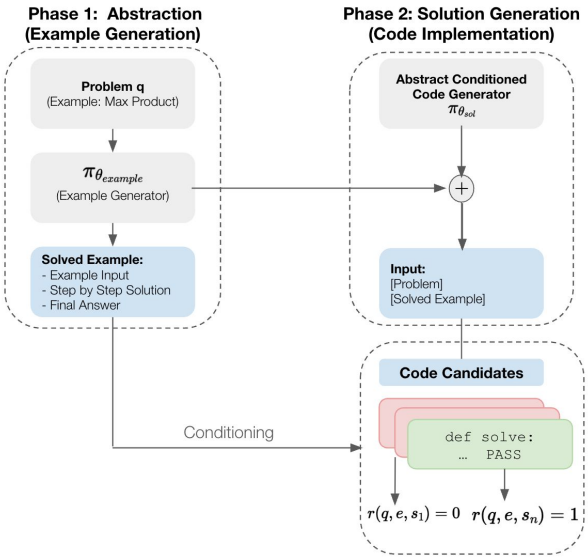


Figure 2: Two-stage worked-example–conditioned code generation in COACH. **Phase 1 (Abstraction):** an example generator policy $\pi_{\theta_{\text{example}}}$ produces a solved example (example input, step-by-step solution, and final answer) for a given problem q. **Phase 2 (Solution Generation):** a code generator $\pi_{\theta_{\text{sol}}}$ conditions on [Problem, Solved Example] to sample multiple code candidates, which are evaluated with an execution-based reward $r(q, e, s)$ (pass/fail) for RL training.

The GRPO objective optimizes a clipped surrogate loss with a KL penalty:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim P(Q),\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}} \left[\frac{1}{G}\sum_{i=1}^{G} \frac{1}{|o_i|}\sum_{t=1}^{|o_i|} \left\{ \min\left(r_t(\theta)\cdot\hat{A}_{i,t},\ r_t^{\text{clip}}(\theta)\cdot\hat{A}_{i,t} \right) - \beta D_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}] \right\} \right] \tag{2}$$

where $\hat{A}_{i,t}$ is the group-relative advantage computed as:

$$\hat{A}_{i,t} = \frac{R_{i,t} - \text{mean}_G(R_{i,t})}{\text{std}_G(R_{i,t})} \tag{3}$$

Here, $R_{i,t}$ is the reward for solution o_i at token t, and the advantage is normalized relative to the group of G rollouts. For our experiments, $R_{i,t}$ is a binary 0/1 reward based on whether the generated code passes **all** test cases for the given problem. The KL penalty term $D_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}]$ prevents the policy from deviating too far from the reference policy π_{ref}.

3.2 COACH: OUR APPROACH

3.2.1 OVERVIEW AND COGNITIVE MOTIVATION

Inspired by the worked-example effect from cognitive science, COACH (**CO**gnitive **A**bstraction **C**onditioning for code **H**elp) decomposes the code generation process into two cognitively-motivated stages: (1) generating a concrete solved example that demonstrates the solution pattern (cognitive abstraction), and (2) conditioning the code generation on this example (guided execution). This mirrors how humans approach coding problems—first understanding through examples, then implementing solutions.

We employ two models: an *example generator* $\pi_\theta^{\text{example}}$ that produces step-by-step solved examples, and a *solution generator* π_θ^{sol} that conditions on these examples to produce code. Both models are trained jointly using GRPO with the same execution-based reward signal, creating an incentive structure where the example generator learns to produce examples that genuinely aid the solution generator.

3.2.2 WARM-START VIA SUPERVISED FINE-TUNING

Since we use a relatively small LLM (Qwen-1.7B), we observe sparse reward signals during initial COACH training due to brittleness in instruction following. To address this, we employ a warm-start phase where we perform supervised fine-tuning (SFT) on the example generator using oracle examples generated by a more capable model (GPT-4o-mini).

Given a dataset $\mathcal{D}$ of problem-example pairs (q, e), the SFT objective is:

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = \max_{\theta} \mathbb{E}_{(q,e)\sim\mathcal{D}} \left[\sum_{t=1}^{|e|} \log \pi_\theta\big(e_t \mid q, e_{<t}\big) \right] \tag{4}$$

where π_θ represents our policy (language model) with parameters θ, e_t is the example token at position t, and $e_{<t}$ denotes all preceding tokens. This warm-started model serves as the initialization for joint GRPO training.

3.2.3 JOINT TRAINING ALGORITHM

For each problem $q \sim D$, we generate N_e examples using $\{e_i\}_{i=1}^{N_e} \sim \pi_\theta^{\text{example}}(q)$, and for each example, generate N_s solutions using $\{s_i\}_{i=1}^{N_s} \sim \pi_\theta^{\text{sol}}(q, e_i)$. The reward for each solution is calculated as:

$$r(\mathbf{q}, \mathbf{e}, \mathbf{s}) := \text{Acc}_q(s, s^*) \tag{5}$$

where s^* represents the ground truth test cases for problem q and Acc_q is 1 if and only if s passes all test cases. Crucially, the same reward is used for both the example and solution generator, incentivizing the example generator to produce examples that maximize downstream solution success.

The complete training procedure is formalized in Algorithm 1.

Algorithm 1: COACH Training

Input: Two models: an example generator $\pi_\theta^{\text{example}}$ and a solution generator π_θ^{sol}.
Output: Updated parameters θ.

while *not converged* **do**
  foreach *problem* $q \in D$ **do**
    Sample N_e examples $\{e_i\} \sim \pi_\theta^{\text{example}}(\cdot \mid q)$
    for *each example* e_i **do**
      Sample N_s solutions $\{s_j\} \sim \pi_\theta^{\text{sol}}(\cdot \mid q, e_i)$
      Compute each solution's rewards using equation (4)
      Compute advantages $\hat{A}_{i,t}^{\text{example}}$ for $\pi_\theta^{\text{example}}$ using equation (3)
      Compute advantages $\hat{A}_{j,t}^{sol}$ for π_θ^{sol} using equation (3)
    end
    Update $\theta^{example}$ using GRPO (equation 2);
    Update θ^{sol} using GRPO (equation 2);
  end
end

4 EXPERIMENTAL SETUP

4.1 DATASETS AND BENCHMARKS

We conduct our experiments on the *Mostly Basic Python Problems* (`MBPP`) dataset Austin et al. (2021), a widely-used benchmark for code generation consisting of approximately 1,000 crowd-sourced Python programming problems. Each problem includes a natural language task description, a reference solution, and three automated test cases for verification. The problems are designed to be solvable by entry-level programmers and cover fundamental programming concepts, standard library usage, and basic algorithms. We use the standard train/validation/test splits provided with the dataset. For COACH, we augment the training set by generating solved examples using GPT-4o-mini, which provides step-by-step solutions that serve as demonstrations for the example generator model.

4.2 Models and Implementation

We use `Qwen-1.7B` as our base language model for all experiments. While relatively small, this model size allows us to conduct rapid iterations and isolate the effects of our worked-example conditioning approach. We compare three model variants: (1) the base Qwen-1.7B model with chain-of-thought prompting, (2) Qwen-1.7B fine-tuned with single-player GRPO-based RLVR, and (3) COACH with separate example generator and solution generator models, both initialized from Qwen-1.7B. For COACH, the example generator is warm-started via supervised fine-tuning on GPT-4o-mini-generated examples before joint GRPO training. All models use the same base architecture to ensure fair comparisons.

We used **AWS** and the **Bridges-2** cluster at Pittsburgh Supercomputing Center (PSC) for all our experiments. For training the single-player GRPO baseline, we used the open-source **verl: Volcano Engine Reinforcement Learning for LLMs** library. Given that there is no open-source equivalent for COACH training available, we forked and made substantial changes to VeRL to implement the COACH training loop.

4.3 Evaluation Metrics

Following standard practice in code generation Austin et al. (2021), we evaluate using the **pass@k** metric, which measures the percentage of problems for which at least one correct solution is generated among k samples. Specifically, we report **pass@1** (accuracy with a single generation) and **pass@16** (accuracy with 16 generations). A solution is considered correct if and only if it passes all provided test cases for the problem. To ensure fair comparison across methods with different sampling budgets, we adjust k to maintain equivalent test-time compute: for COACH which generates both an example and a solution, we use $k = 8$ to match the total number of forward passes of the single-player baseline with $k = 16$. All results are reported on the held-out test set.

5 Results

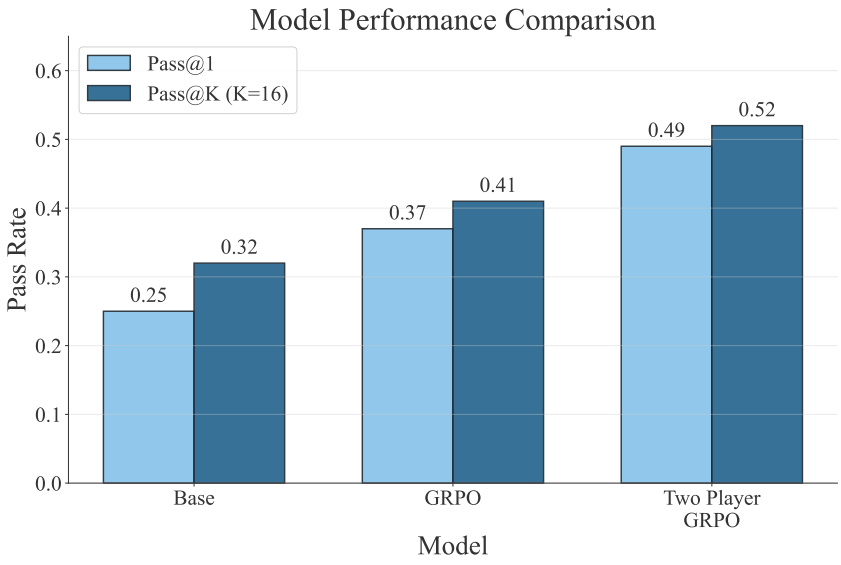


Figure 3: ss:change the captionComprehensive results comparing GRPO to COACH with worked-example conditioning. (a) Absolute performance (pass@1 and pass@5) across the base model, GRPO baseline, and COACH. (b) Relative improvement over the base model for each training setup, highlighting larger gains from COACH cognitive abstraction conditioning. (c) Performance progression across training variants, showing consistent improvements from base $\rightarrow$ GRPO $\rightarrow$ COACH. (d) Validation reward curves over training steps, where COACH achieves higher reward and better convergence than GRPO.

5.1 Quantitative Analysis

Our results are summarized in Table 1. We see that the simple GRPO baseline improves over the baseline model by over **12%** in absolute accuracy. COACH improves over the GRPO baseline by a further **12%** in absolute accuracy. To have a fair comparison, we use a value of $K = 8$ for COACH to have the same test-time compute budget across our experiments.

Model	**Pass@1**	**Pass@K** ($K = 16$)
Qwen-1.7B	0.25	0.32
Qwen-1.7B + GRPO baseline	0.37	0.41
Qwen-1.7B + COACH	**0.49**	**0.52**

Table 1: Performance Comparison of Qwen Models. The best performing method is highlighted. The numbers indicate accuracies on the test set.

5.2 Qualitative Analysis

An example walkthrough of the generated outputs is given in the Appendix: A.2.

The coding problem is to find the maximum product pair in an array of integers. We see that the base model is unable to converge to writing code, and uses away all its token budget in reasoning. This is because it is not trained to format code explicitly. On doing RLVR with GRPO, we see that the model does output only code, hence it learns to format the output appropriately. However it misses the corner case of having both negative numbers as the maximum product in its generated code. The example generator in COACH, handles the both negative case in a solved example, and also gives a clean step by step algorithm for the solution generator to implement. Thus the solution generator in COACH is able to cleanly implement the algorithm and does better.

6 Discussion

6.1 Examples as Useful Reasoning Abstractions

In this work, we set out to answer the question "Are solved examples a useful abstraction for coding problems?". We answer this question in affirmative through our experiments. By training COACH with an example generator and a solution generator, we were able to improve over the vanilla RLVR baseline and show the superiority of using solved examples in guiding solution generation.

6.2 Training stability and practical considerations

One practical consideration for our approach is the longer training time and compute needed to train two models. Moreover, one would need a larger GPU-VRAM at test time to load and run inference on two models, raising a potential question of whether the gains are worth the added cost and complexity. We believe a more systematic study of our approach's performance on larger and more challenging datasets can help answer this question.

6.3 Failure cases

We observe that while examples help improve performance over the baseline, they are still unable to fully solve all the examples in the test set. We group the remaining failure modes into the following buckets:

- **Solution Generator Unable to Convert Example to Correct Code**: A detailed example is given in the appendix: A.3. With reference to this example, we see that while the example generator generates a concise solved example, the solution generator is unable to convert it into the correct code. This hints at needing harder problems to train on or a curriculum strategy to progressively train on harder problems to improve on this performance.

- **Limited Information in Coding Prompt**: A detailed example is given in the appendix: A.4. With reference to this example, we see that the prompt only asks to zip two tuples and expects handling of the case when the two lists are not of the same length. This is not clear from the question at all. The generated code is wrong, as it takes the maximum of two list lengths which can lead to out of bounds exception, however, the prompt is also incomplete on how to format the input, the input specifications and other minor problem details. We believe moving on to a more verbose dataset like Apps for training and evaluation can help mitigate.

6.4 Limitations and Future Work

Our work has several limitations. The MBPP dataset allowed for quick experimentation and proof-of-concept development given the scope of a class project, however is not greatly indicative of the complexity of real world competitive coding problems. We aim to address this in future work by moving on to harder datasets like the Apps (Hendrycks et al., 2021) and Codeforces (Penedo et al., 2025) datasets.

Our experiments were small scale which was due to the limited computational resources available at our disposal. Future work could also involve scaling this approach to larger models and datasets and run more complex experiments with COACH.

Author Contributions

All authors contributed equally to this work.

Acknowledgments

We thank the Pittsburgh Supercomputing Center (PSC) for providing compute resources through the Bridges-2 cluster. We also acknowledge the open-source VeRL library which served as the foundation for our implementation.

References

Robert K. Atkinson, Sharon J. Derry, Alexander Renkl, and Donald Wortham. Learning from examples: Instructional principles from the worked examples research. *Review of Educational Research*, 70(2):181–214, 2000. doi: 10.3102/00346543070002181.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Lili Chen, Mihir Prabhudesai, Katerina Fragkiadaki, Hao Liu, and Deepak Pathak. Self-questioning language models, 2025. URL https://arxiv.org/abs/2508.03682.

Michelene T. H. Chi, Miriam Bassok, Matthew W. Lewis, Peter Reimann, and Robert Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13(2):145–182, 1989a. doi: 10.1207/s15516709cog1302_1.

Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science*, 13(2):145–182, 1989b.

Gaël Gendron, Qiming Bao, Michael Witbrock, and Gillian Dobbie. Large language models are not strong abstract reasoners. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, IJCAI '24, 2024. ISBN 978-1-956792-04-1. doi: 10.24963/ijcai.2024/693. URL https://doi.org/10.24963/ijcai.2024/693.

Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2): 155–170, 1983. doi: 10.1207/s15516709cog0702_3.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Zi Lin, Sheng Shen, Jingbo Shang, Jason Weston, and Yixin Nie. Learning to solve and verify: A self-play framework for code and test generation. *arXiv preprint arXiv:2502.14948*, 2025.

Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piqueres Lajarín, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. Codeforces. `https://huggingface.co/datasets/open-r1/codeforces`, 2025.

Yuxiao Qu, Anikait Singh, Yoonho Lee, Amrith Setlur, Ruslan Salakhutdinov, Chelsea Finn, and Aviral Kumar. Rlad: Training llms to discover abstractions for solving reasoning problems, 2025. URL `https://arxiv.org/abs/2510.02263`.

Alexander Renkl. Learning from worked-out examples: A study on individual differences. *Cognitive science*, 21(1):1–29, 1997.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2): 257–285, 1988a.

John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2): 257–285, 1988b. doi: 10.1207/s15516709cog1202_4.

John Sweller and Graham A Cooper. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and instruction*, 2(1):59–89, 1985a.

John Sweller and Graham A. Cooper. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2(1):59–89, 1985b. doi: 10.1207/s1532690xci0201_3.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning. *arXiv preprint arXiv:2506.03136*, 2025.

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

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Yueke Zhang, Yifan Zhang, Kevin Leach, and Yu Huang. Codegrad: Integrating multi-step verification with gradient-based llm refinement. *arXiv preprint arXiv:2508.10059*, 2025.

A Appendix

A.1 Prompt Templates Used

Python Coding Prompt

You are a helpful Python coding assistant. Given a task, output ONLY valid Python code that defines the required function(s). Do not include explanations or markdown fences. Do not include any docstrings or anything other than the python function(s). Enclose the entire solution within “‘ `python` and “‘.

Task:
`{problem}`

Constraints:

- Write clean, minimal Python and no other language.
- Define the function(s) exactly as implied by the tests.
- Do NOT print; just return values.
- Output ONLY code (no backticks, no explanations).
- Enclose the entire solution within “‘ `python` and “‘.

Prompt to Generate Solved Example for a Problem

Given the following programming task: `{problem}`
Generate a clear and complete solved example for this problem.

- Do not write any code.
- Choose simple toy values to illustrate the process.
- Show the step-by-step reasoning used to solve the example.
- Clearly present the initial input, intermediate steps, and final output.
- Format the solution neatly using bullet points or equations where appropriate.

Structure your response with the following sections:

- Problem Recap
- Example Input
- Step-by-Step Solution
- Final Answer

Python Coding Prompt with Generated Solved Example

You are a helpful Python coding assistant.
Given a task, output ONLY valid Python code that defines the required function(s).
Do not include explanations or markdown fences.
Do not include any docstrings or anything other than the python function(s).
Enclose the entire solution within “‘ `python` and “‘.

Task:
Write a python function to find smallest power of 2 greater than or equal to n.

Constraints:

- Write clean, minimal Python and no other language.
- Define the function(s) exactly as implied by the tests.
- Do NOT print; just return values.
- Output ONLY code (no backticks, no explanations).
- Enclose the entire solution within “‘ `python` and “‘.

To aid you in solving the problem, here is(are) step by step solved examples to illustrate how to solve the problem

1. Problem Recap
The task is to determine the smallest power of 2 that is greater than or equal to a given number n. A power of 2 is any number that can be expressed as 2^k, where k is a non-negative integer (i.e., $0, 1, 2, 3, \ldots$).
2. Example Input
Let's choose a simple example to illustrate the process. Suppose we have the following input:
- $n = 9$
3. Step-by-Step Solution
To find the smallest power of 2 greater than or equal to 9, follow these steps:
- ****Step 1:**** Identify the powers of 2 starting from 2^0 and continue until we find a power that is greater than or equal to 9.
- $2^0 = 1$ (Not greater than or equal to 9)
- $2^1 = 2$ (Not greater than or equal to 9)
- $2^2 = 4$ (Not greater than or equal to 9)
- $2^3 = 8$ (Not greater than or equal to 9)
- $2^4 = 16$ (Greater than or equal to 9)

- ****Step 2:**** Compare the powers of 2 with 9, as calculated:
- $2^3 = 8$ is less than 9.
- $2^4 = 16$ is the first power of 2 that is greater than or equal to 9.
- ****Step 3:**** Conclude that 2^4 is the smallest power of 2 that satisfies the condition.
4. Final Answer
- The smallest power of 2 greater than or equal to 9 is 16.

A.2 Qualitative Analysis of Generated Outputs

Base Model Qwen3-1.7B (Exhausts response length)

<*think*> Okay, I need to write a Python function that finds a pair of integers with the highest product from a given array. Let's think about how to approach this.
First, the problem says to find a pair with the highest product. So, the array can have any number of elements, but we need to find two numbers whose product is the maximum possible.
Hmm, what's the best way to do this? Well, the straightforward approach would be to check all possible pairs and compute their product, then keep track of the maximum. But that's $O(n^2)$ time complexity, which might not be efficient for very large arrays.
Wait, but maybe there's a smarter way. For example, if the array has negative numbers, the maximum product could be the product of two negative numbers. So, the approach would need to consider both positive and negative numbers.
So, the plan is to find the two largest numbers and the two smallest numbers, then compare their products. <*/think*>

[Output Truncated due to length...]

GRPO Model WITHOUT Example Guidance (Wrong output)

```
def max_product_pair(arr):
    max_product = -float('inf')
    second_max = -float('inf')

    for num in arr:
        if num > max_product:
            second_max = max_product
            max_product = num
        elif num > second_max:
            second_max = num

    return (max_product, second_max)
```

GRPO Model WITH Example Guidance (Correct output)

```
def find_max_product_pair(arr):
    max_product = -float('inf')
    max_pair = (0, 0)

    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            product = arr[i] * arr[j]
            if product > max_product:
                max_product = product
                max_pair = (arr[i], arr[j])

    return max_pair
```

A.3 Failure Mode Analysis: Unable to Parse Example to Code

Problem: Write a python function to reverse an array upto a given position.

COACH Failure Mode

```python
def reverse_array_up_to_position(arr, position):
    # Reverse the array up to the given position
    arr[:position+1] = arr[:position+1][::-1]
    return arr
```

Generated Solved Example by Example Generator

1. Problem Recap
The task is to reverse an array up to a given position. This means that for a given index k, all elements from the start of the array up to index $k-1$ should be reversed, while the elements from index k onwards remain in their original order.

2. Example Input

- Array: `[1, 2, 3, 4, 5]`
- Position: `3`

3. Step-by-Step Solution
To solve this, we identify the sub-segment of the array defined by the position parameter and reverse only those elements.

- **Initial Array**: `[1, 2, 3, 4, 5]`
- **Position**: `3` (This means we reverse the first 3 elements of the array: indices 0, 1, and 2).
- **Step 1: Identify the elements to reverse**
  Elements from index 0 to index 2: `[1, 2, 3]`
- **Step 2: Reverse these elements**
  Reversing `[1, 2, 3]` gives `[3, 2, 1]`
- **Step 3: Replace the elements in the original array**
  Update the original array with the reversed elements:
  - Original array: `[1, 2, 3, 4, 5]`
  - After reversing the first 3 elements: `[3, 2, 1, 4, 5]`

4. Final Answer

- **Final Array**: `[3, 2, 1, 4, 5]`

A.4 Failure Mode Analysis: Limited Information in Coding Prompt

Problem Statement: Write a function to zip the two given tuples.

Ground Truth Test Cases

```
assert zip_tuples((7, 8, 4, 5, 9, 10), (1, 5, 6)) == [(7, 1),
    (8, 5), (4, 6), (5, 1), (9, 5), (10, 6)]
assert zip_tuples((8, 9, 5, 6, 10, 11), (2, 6, 7)) == [(8, 2),
    (9, 6), (5, 7), (6, 2), (10, 6), (11, 7)]
assert zip_tuples((9, 10, 6, 7, 11, 12), (3, 7, 8)) == [(9, 3),
    (10, 7), (6, 8), (7, 3), (11, 7), (12, 8)]
```

COACH Generated Solution

```
def zip_tuples(tuple1, tuple2):
    return tuple(tuple1[i] + tuple2[i] for i in range(max(len(
        tuple1), len(tuple2))))
```

A.5 Oracle Training Results

Along with our experiments above, we also run a proof-of-concept on using only the GPT o4-mini generated examples along with the prompt and doing a GRPO based RLVR using that. The results of this experiment are given below:

Model	**Pass@1**	**Pass@K** ($K = 16$)
Qwen-1.7B	0.25	0.32
Qwen-1.7B + GRPO baseline	0.37	0.41
Qwen-1.7B + GRPO with Example Oracle	0.47	0.52
Qwen-1.7B + COACH	0.49	0.52

Table 2: Oracle Example Conditioning Results along with other experiments.

Based on the results in table 2, we see that COACH surpasses the oracle model on pass@1 and achieves the same performance on pass@k. We attribute this to the fact that COACH is warm started using the examples generated from the oracle. As a results, it is able to retain the performance of the oracle and generalize to the test scenario reasonably well. We also believe that the train and test distributions are highly similar, so this generalization does not take a hit for the example generator. The pass@1 improvement is not statistically significant given 2 out of 100 examples in our test scenario, and we observe that they go away with higher 'k'. This can thus be explained by the fact that the model simply re-orders the correct solution on these examples and the result does not indicate a qualitative improvement in model performance.

A.6 HYPERPARAMETERS

Our hyperparameters are summarized in table 3

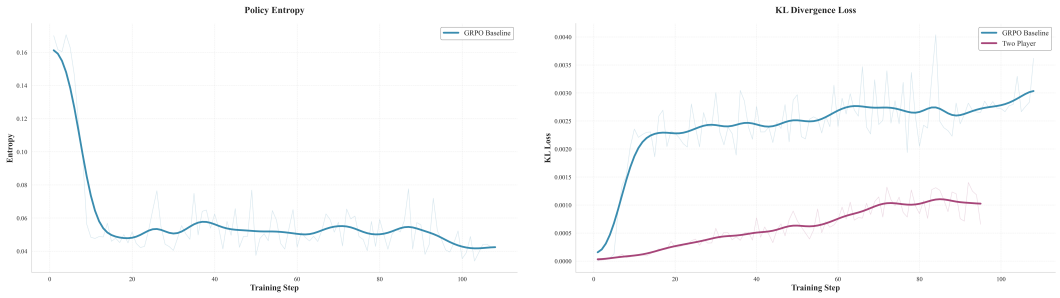
Param	Value
LR (η)	1×10^{-6}
Batch Size	32
Micro Batch	4
Optimizer	Adam
Epochs	4
Prompt Len	2048
Resp Len	512
Rollouts	8

Table 3: GRPO Baseline Hyperparameters

Param	Value
Ex. Rollouts	4
Sol. Rollouts	8

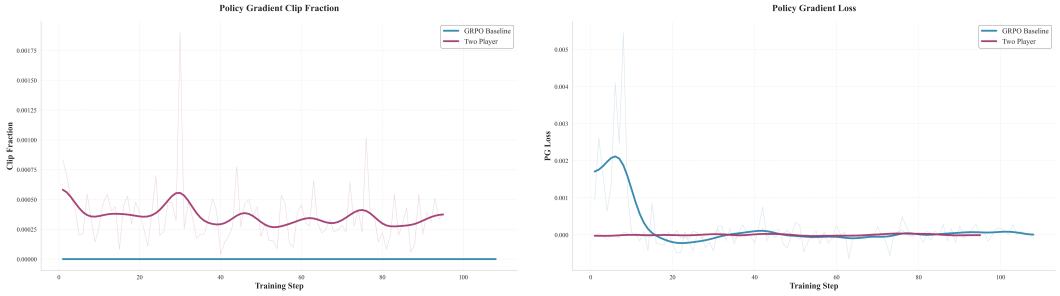
Table 4: COACH Additional Parameters

A.7 TRAINING PLOTS



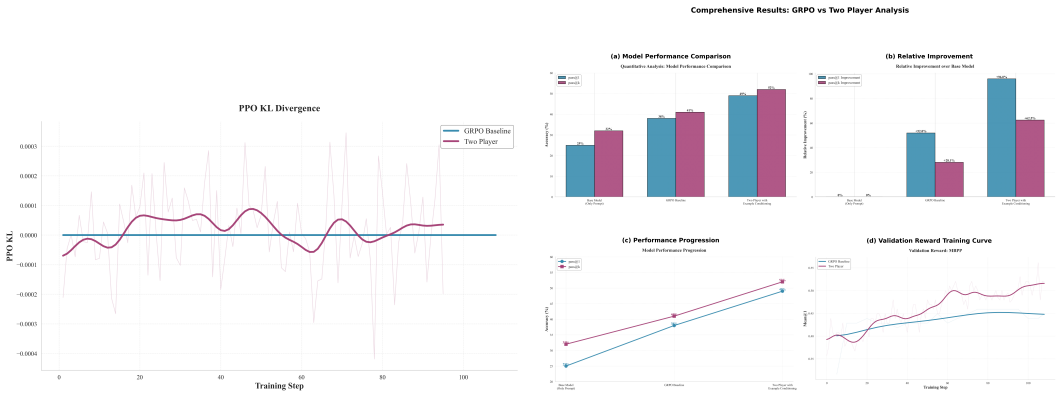
(a) Actor entropy **(b)** Actor KL loss

Figure 4: Training diagnostics (1/8): actor optimization statistics.



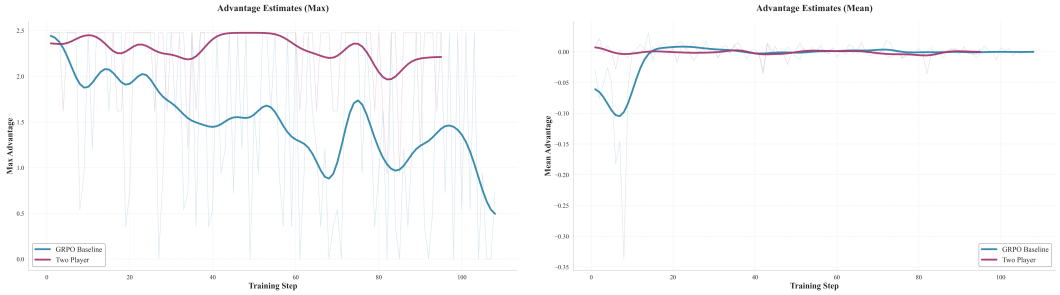
(a) PPO clip fraction **(b)** Policy gradient loss

Figure 5: Training diagnostics (2/8): actor optimization statistics (continued).



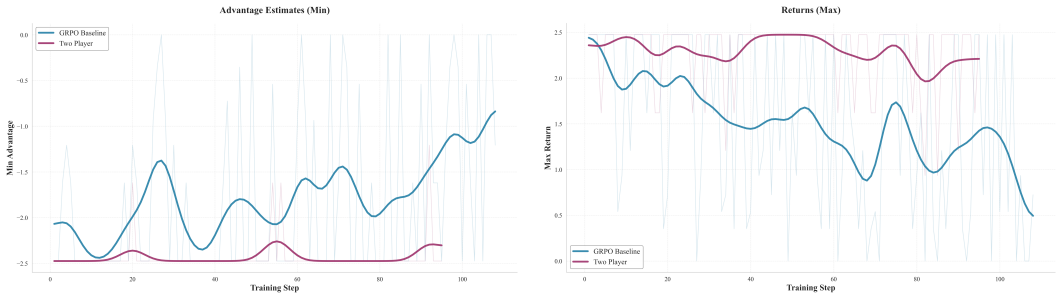
(a) PPO KL **(b)** Aggregate results

Figure 6: Training diagnostics (3/8): KL behavior and aggregate performance.



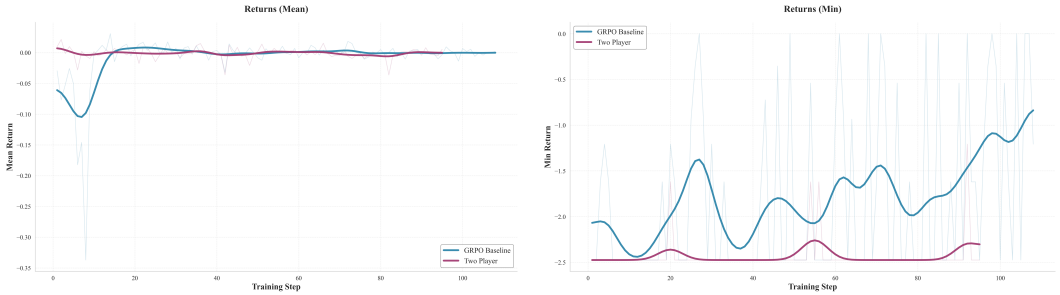
(a) Advantages (max) **(b)** Advantages (mean)

Figure 7: Training diagnostics (4/8): advantage statistics.



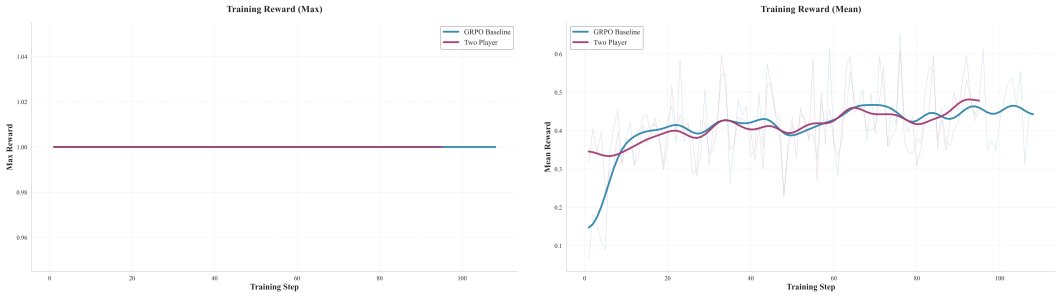
(a) Advantages (min) **(b)** Returns (max)

Figure 8: Training diagnostics (5/8): advantages and return behavior.



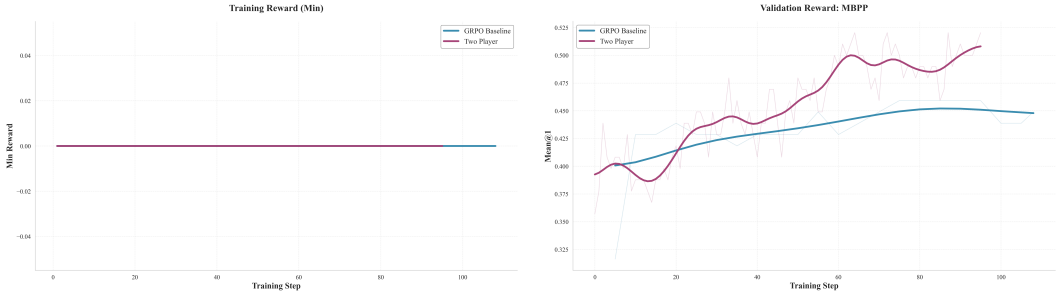
(a) Returns (mean) **(b)** Returns (min)

Figure 9: Training diagnostics (6/8): return statistics (continued).



(a) Training reward (max) **(b)** Training reward (mean)

Figure 10: Training diagnostics (7/8): training reward statistics.



(a) Training reward (min) **(b)** Validation reward (MBPP)

Figure 11: Training diagnostics (8/8): training vs. validation reward.