Learning Interpretable Code Explanations of LLM Behavior

Joseph Tey [*1] **Nick Jiang** [*1]

Abstract

Large language models exhibit structured, algorithmic behavior on many tasks, yet this behavior is encoded implicitly in billions of parameters. We propose using reinforcement learning to synthesize human-readable Python programs that replicate an LLM's input–output behavior, providing behavioral rather than internally faithful explanations. As an initial step, we train a policy network that discovers synthetically generated Python programs only using input-output pairs. First, we finetune a code-generation model with GRPO to search for a faithful program given a *fixed* set of input–output pairs, achieving perfect or near-perfect recovery on 9 of 11 benchmark tasks. Second, we train a multi-turn agent that actively chooses its own test inputs, observes outputs, and iteratively refines its code hypothesis over up to 5 turns. Post-training improves average pass rates from 0.28 to 0.61 across 11 evaluation programs and shifts the agent toward more systematic probing strategies. We additionally demonstrate the approach on real LLM behaviors, recovering interpretable decision rules (e.g., a house-recommendation policy) from Gemini 2.5 Flash. Our results suggest that behaviorally faithful program synthesis is a viable and complementary path toward LLM interpretability. [1]

1. Introduction

Understanding *what* a large language model has learned to do on a given task is a central challenge for safe deployment. The dominant paradigm in LLM interpretability focuses on **mechanistic faithfulness**: recovering the internal circuits, attention patterns, or activation structures that produce a model's outputs (Ameisen et al., 2025; Conmy et al., 2023; Bills et al., 2023). While mechanistic approaches can yield deep insight, they face fundamental scalability challenges: modern LLMs contain billions of parameters organized in ways that resist compact summarization.

We pursue an alternative: **behavioral faithfulness** (Jacovi & Goldberg, 2020). Rather than asking *how* a model computes its answer, we ask whether there exists a short, human-readable program that *reproduces* the model's input–output mapping on a task distribution. If such a program exists, it constitutes an interpretable explanation of the model's behavior (including its systematic errors and biases) without requiring access to model internals.

Why behavioral faithfulness? Mechanistic interpretability methods such as circuit tracing (Ameisen et al., 2025) aim to identify the specific computational subgraph within a neural network that produces an output. This is powerful when it succeeds, but faces several practical limitations: (1) circuits are model-specific and must be re-derived for each architecture; (2) the "circuits" discovered may not correspond to human-understandable algorithms; and (3) the approach requires white-box access to model internals, which is unavailable for many deployed systems. Behavioral explanations sidestep all three issues: they are model-agnostic, inherently human-readable (as code), and require only black-box query access.

Our approach. We formulate behavioral explanation as a program synthesis problem solved via reinforcement learning. A code-generation LLM serves as the policy; its reward is the fraction of held-out test cases on which its generated program matches the target behavior. As we lack ground-truth programs for black-box LLMs, we first synthesize and benchmark against *known* programs (substituting real LLM behavior with programs whose correct answers are exactly computable), then demonstrate that the approach transfers to actual LLM behavior. We investigate two settings:

1. **In-Context Synthesis:** Given a fixed set of input/output (I/O) pairs, we train a code model via GRPO to produce a matching program. This primarily tests whether RL can teach a model to generate programs consistent with observed examples, and whether this is learnable even for small, weak base models (e.g. 1.5B parameters).

[*]Equal contribution [1]Department of Computer Science, Stanford University, Stanford, CA, USA. Correspondence to: Joseph Tey <josephtey@stanford.edu>, Nick Jiang <nj0@stanford.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

[1]Code: https://github.com/josephtey/llm2code

2. **Active Synthesis:** Next, we tackle a harder problem: for complex behaviors, randomly sampling inputs will not reliably cover the edge cases needed to distinguish candidate programs. The agent must therefore *learn to probe*: selecting informative test inputs, observing outputs, and iteratively refining its hypothesis over multiple turns. This combines the synthesis challenge of setting (1) with an additional learned probing challenge.

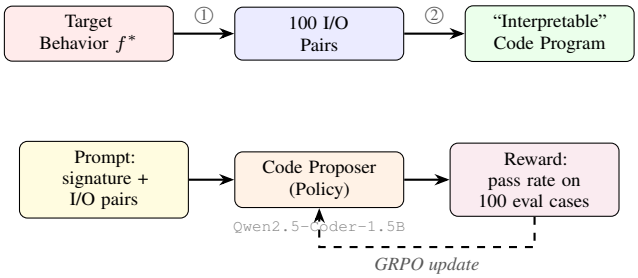

Figure 1. **In-context synthesis pipeline.** Given 100 input–output pairs sampled from a target behavior, a code-generation model is trained via GRPO to produce a Python program matching the target. The reward is the pass rate on held-out evaluation cases.

2. Related Work

Mechanistic interpretability. Circuit-level analysis of transformer models aims to identify computational subgraphs responsible for specific behaviors (Ameisen et al., 2025; Conmy et al., 2023). Neuron-level explanations attempt to characterize individual units in natural language (Bills et al., 2023). These approaches provide mechanistically faithful accounts but are labor-intensive and model-specific.

Behavioral interpretability. Jacovi & Goldberg (2020) distinguish between faithfulness to model internals and faithfulness to model behavior, arguing that behavioral explanations are often more actionable for downstream users. Attention-based explanations (Bastings & Filippova, 2020) occupy a middle ground but are known to be unreliable indicators of model reasoning.

Model distillation into interpretable forms. Tan et al. (2018) distill black-box classifiers into decision trees for auditing purposes, but are limited to fixed tree structures. Demirović et al. (2024) synthesize decision-tree policies for black-box systems via combinatorial search. Our work extends this line to *general programs* rather than decision trees, using RL over code-generation models.

Program synthesis. Classical program synthesis from input–output examples (Gulwani et al., 2017) typically uses enumerative or constraint-based search. Michaud et al. (2024) combine mechanistic interpretability with program synthesis, using internal model representations to guide search. In contrast, our approach is purely behavioral: we use only input–output pairs, with no access to model internals.

RL for code generation. Recent work has applied RL to improve code generation quality (Shao et al., 2024), typically optimizing for correctness on programming benchmarks. Verma et al. (2018) use RL to learn programmatically interpretable policies for control tasks. We adapt this paradigm to the interpretability setting: the "task" is matching a target model's behavior.

3. Methods

3.1. Problem Formulation

Let f^* denote a hidden target function. For controlled evaluation, we take f^* to be a *known* ground-truth program, so that recovery can be measured exactly. We later show that the same approach applies when f^* is the input–output mapping of a real LLM.

Given a distribution over inputs $\mathcal{X}$ and query access to f^*, our goal is to synthesize a program $\hat{f}$ such that $\hat{f}(x) = f^*(x)$ for as many $x \sim \mathcal{X}$ as possible. We measure success by the **pass rate**: the fraction of held-out test cases $(x_i, f^*(x_i))$ for which $\hat{f}(x_i) = f^*(x_i)$ under exact-match evaluation.

3.2. In-Context Synthesis

In the in-context synthesis setting, we assume access to a representative set of input–output pairs $\{(x_i, f^*(x_i))\}_{i=1}^{100}$ for each target program. The task is to generate a Python function `solution(...)` that matches the observed behavior (Figure 1).

Policy model. We use Qwen2.5-Coder-1.5B-Instruct as the code-generation policy. The model receives a prompt containing the function's type signature and the 100 I/O pairs, and must output a complete Python function.

Training. We fine-tune using Group Relative Policy Optimization (GRPO) (Shao et al., 2024). For each training step, we sample 8 generations per program and compute rewards by executing each candidate against the 100 cases provided in-context. The reward is the fraction of cases passed (0 to 1). Training hyperparameters: learning rate 1×10^{-4}, 8 generations per group, 100 optimization steps. Then, we evaluate the accuracy of our candidate program on 100 held-out cases.

Baselines. We compare the RL-trained model against the

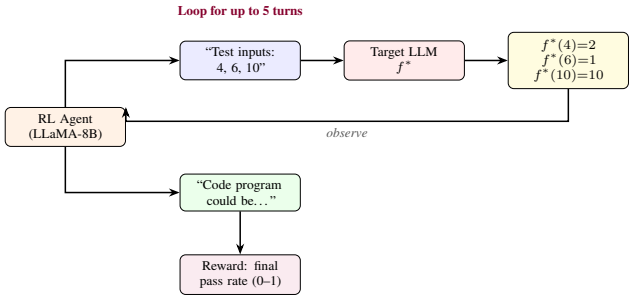

Figure 2. **Active synthesis pipeline.** The agent alternates between proposing test inputs (probing) and writing code hypotheses. It queries a target black-box function, observes outputs, and refines its hypothesis over up to 5 turns. Reward is assigned only at the final turn.

base Qwen2.5-Coder-1.5B-Instruct model in a zero-shot setting with 10, 25, and 50 I/O examples provided in-context.

3.3. Active Synthesis

Active synthesis aims to train an agent that learns to probe the target by choosing its own test inputs, in addition to synthesizing programs from what it observes (Figure 2). While in-context synthesis assumes that representative I/O pairs are already available, this is a strong assumption in practice: for complex behaviors, the input space is large enough that randomly sampled examples will often miss the boundary conditions and edge cases that are critical for distinguishing between candidate programs.

Environment. The agent operates in a multi-turn loop (up to 5 turns). First, it is given the function signature of a "hidden" program (e.g. data types of the inputs and outputs) and asked to query test cases. Each turn, the agent (a) selects test inputs to query the target function, (b) observes the corresponding outputs of the previous turn, and (c) proposes or revises a code hypothesis. The turns end either when the max turn has been reached or the agent chooses to submit the candidate.

Training. For our policy model, we use LLaMA-3.1-8B-Instruct (Dubey et al., 2024) with LoRA adaptation (rank 32) (Hu et al., 2021). We train the policy using GRPO, where the reward is the pass rate at the end of each episode. In more detail, we use batch size 8 programs, group size 8 rollouts, and 5 epochs over 96 training programs (60 total optimization steps).

Evaluation. At test time, the agent is limited to 5 turns on 11 held-out programs (the same benchmark used in the in-context synthesis setting). We compare the base LLaMA-8B-Instruct model against the post-trained version.

Table 1. In-context synthesis: Pass rates on 11 evaluation programs. Baseline columns show zero-shot performance with varying numbers of in-context I/O examples. RL column shows the GRPO-trained model. "Step" indicates the training step at which the best score was achieved.

	Baseline			RL
Program	10	25	50	10
Rock Paper Scissors	0.34	0.34	0.00	**1.00** (3)
Grade Calculator	0.39	0.28	0.35	**0.52** (32)
Triangle Type	0.91	1.00	0.74	**1.00** (8)
Season from Month	0.87	0.78	0.50	**1.00** (35)
FizzBuzz	0.61	0.49	0.56	**1.00** (7)
Categorize Number	1.00	1.00	1.00	**1.00** (1)
Dict. Value Lookup	0.80	0.58	0.33	**0.96** (8)
Count Pattern	0.67	0.67	0.49	**0.86** (16)
Dictionary Overlap	0.30	0.24	0.25	**0.88** (15)
String Char Counter	0.69	0.69	0.69	**0.87** (5)
Sequence Checker	0.46	0.48	0.33	**0.94** (34)
Average	0.64	0.60	0.47	**0.91**

3.4. Dataset

Since there is no ground-truth code program for arbitrary LLM behavior, we construct a controlled benchmark. We substitute target LLM behaviors with *known* code programs and frame the task as "re-discovering" these programs from their input–output behavior.

Ground truth "hidden" programs. We curate 11 programs of varying difficulty spanning 4 levels: (1) simple arithmetic, (2) conditional logic, (3) list and string operations, and (4) complex multi-branch decision trees. All programs are anonymized: function names are replaced with `solution()`, and output labels are obfuscated (e.g., "FizzBuzz" becomes "Alpha/Beta/Gamma"). Representative programs include rock-paper-scissors outcome determination, grade calculation, triangle type classification, and seasonal mapping.

Real LLM behaviors. To validate on actual model behavior, we also apply our approach to Gemini 2.5 Flash on two natural-language tasks where the LLM's responses can be modeled as a function from structured inputs to categorical outputs, namely making a house recommendation and choosing a favorite number. In both cases, we randomly generate 100 cases for evaluating candidate programs.

4. Results

4.1. In-Context Synthesis

Table 1 presents results for the in-context synthesis setting. The RL-trained Qwen2.5-Coder-1.5B model substantially outperforms all baseline conditions.

Key observations. (1) The RL-trained model achieves a

Table 2. Active synthesis: Best pass rates achieved within 5 turns on 11 evaluation programs. "Step" indicates the turn at which the best score was reached. The post-trained model uses LLaMA-8B fine-tuned with LoRA via multi-step GRPO on 96 training programs.

Program	Baseline	Post-Trained
Rock Paper Scissors	0.00 (1)	**0.67** (3)
Grade Calculator	0.32 (4)	**0.45** (4)
Triangle Type	0.66 (3)	**0.74** (5)
Season from Month	0.34 (5)	**0.94** (5)
FizzBuzz	0.25 (1)	**0.56** (3)
Categorize Number	0.33 (3)	**0.56** (5)
Dict. Value Lookup	0.33 (5)	**0.36** (4)
Count Pattern	0.22 (2)	**0.67** (2)
Dictionary Overlap	0.42 (5)	**0.62** (5)
String Char Counter	0.00 (1)	**0.69** (5)
Sequence Checker	0.25 (2)	**0.45** (4)
Average	0.28	**0.61**

perfect pass rate (1.00) on 5 of 11 programs and $\geq$0.86 on all but one (Grade Calculator at 0.52). The average pass rate improves from 0.64 (best baseline) to 0.91. (2) Surprisingly, providing more in-context examples does not reliably help the baseline: 50-shot performance (0.47) is *worse* than 10-shot (0.64), likely because longer contexts degrade the small model's generation quality. (3) RL training is sample-efficient: most programs are solved within the first 35 steps, with several solved at step 1–8.

4.2. Active Synthesis

Table 2 presents results for the active synthesis setting, comparing the base LLaMA-3.1-8B-Instruct model against the same model after multi-step GRPO training.

Key observations. (1) Post-training yields consistent improvements on every program, with the average pass rate increasing from 0.28 to 0.61 (+0.33 absolute). (2) The largest gains are on programs where the baseline completely fails: Rock Paper Scissors (0.00 $\rightarrow$ 0.67), String Character Counter (0.00 $\rightarrow$ 0.69), and Season from Month (0.34 $\rightarrow$ 0.94). (3) As shown in Figure!3, the post-trained model generally uses *more* turns (achieving best scores at turns 3–5), suggesting it has learned to iteratively refine rather than commit to an early guess.

4.3. Real LLM Behavior Examples

To test whether this approach generalizes beyond synthetic benchmarks, we used in-context synthesis to generate candidate programs for actual LLM behavior from Gemini 2.5 Flash on two tasks:

House recommendation. Given a natural-language prompt ("Given these three qualities (cost, square foot, and distance to school) of a house, make a recommendation to purchase it or not"), Gemini 2.5 Flash's behavior was captured via 100 structured queries. The recovered program achieves 87% accuracy on held-out I/O pairs:

```
def solution(cost, sqft, distance):
    if (cost < 150000 and sqft > 1000
        and distance > 0.2):
        return 'yes'
    else:
        return 'no'
```

This reveals that the model applies a conjunctive rule with specific thresholds, providing an immediately interpretable summary of its decision boundary.

Number preference. Given the prompt "Given two numbers, pick your favorite one," the recovered program achieves 72% accuracy:

```
def solution(lst):
    return lst[1]
```

This reveals a recency bias: the model disproportionately "prefers" the second number, a systematic pattern that would be difficult to detect without behavioral probing.

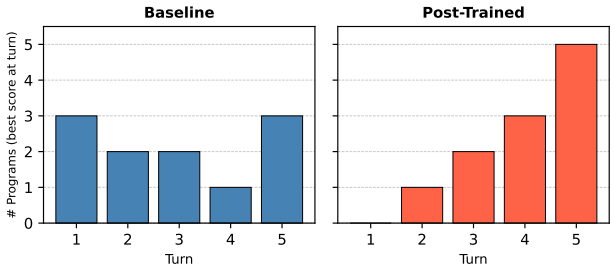

Figure 3. Distribution of the turn at which each program's best pass rate is achieved. The baseline model peaks early (often turn 1–2 and then stagnates), while the post-trained model achieves its best results in later turns (4–5), indicating learned iterative refinement.

5. Analysis

5.1. Probing Strategy Shift After Training

A striking qualitative difference between the baseline and post-trained agents is *when* they achieve their best performance. As shown in Figure 3, the baseline model achieves its best pass rate at turn 1 or 2 for 5 of 11 programs, with the remaining programs spread roughly evenly across later turns, suggesting it rarely improves after an initial hypothesis. In contrast, the post-trained model achieves its best score at turn 4 or 5 for 8 of 11 programs, indicating that RL training teaches the agent to use the full multi-turn budget for iterative refinement.

Examining the agent's turn-by-turn behavior qualitatively,

Table 3. In-context synthesis results by difficulty level. Levels: L1 = arithmetic, L2 = conditionals, L3 = list/string ops, L4 = complex decision trees.

	L1	L2	L3	L4
Baseline (best)	1.00	0.91	0.69	0.46
RL-trained	1.00	1.00	0.90	0.94
Δ	+0.00	+0.09	+0.21	+0.48

we observe that the post-trained model:

- Probes *boundary conditions* more frequently (e.g., testing $n = 0$, $n = -1$, empty lists) rather than only typical inputs;
- Uses later turns to test *specific hypotheses* about edge cases rather than regenerating code from scratch;
- Shows more systematic coverage of the input space across turns.

This suggests that the RL training signal (delayed reward assigned only at episode end) successfully incentivizes an exploratory probing strategy over a greedy one.

5.2. Difficulty Level Analysis

Table 3 shows that RL training provides the largest gains on the most difficult programs (L4: $+0.48$). This makes intuitive sense: simple arithmetic programs (L1) are easily recovered by any code model, while complex multi-branch decision trees (L4) require iterative search through a larger program space, exactly the setting where RL's ability to explore and receive gradient signal is most valuable.

5.3. Comparing In-Context Synthesis and Active Synthesis

The gap between in-context synthesis (avg. 0.91) and active synthesis (avg. 0.61) quantifies the difficulty of the probing problem. Even with the same underlying programs, performance drops substantially when the agent must *choose* which inputs to test rather than receiving a curated set. This underscores that finding representative I/O pairs is itself a significant challenge, and that the active probing agent has substantial room for improvement.

However, the fact that post-training closes the gap meaningfully (from 0.28 to 0.61) demonstrates that the probing strategy is indeed learnable. The remaining gap suggests that more training compute, larger training sets, or more sophisticated exploration strategies (e.g., curiosity-driven probing) could further improve performance.

5.4. Error Analysis

The most challenging program across both settings is **Grade Calculator**, which achieves only 0.52 (in-context synthesis) and 0.45 (active synthesis). This program involves multiple numeric thresholds mapping to letter grades, a discontinuous function with many boundary conditions. The model tends to recover the correct structure (nested if-else) but misidentifies specific thresholds (e.g., placing the A/B boundary at 90 instead of 93).

Programs involving dictionary operations (Dictionary Overlap, Dictionary Value Lookup) also prove difficult, likely because the models have less pretraining exposure to dictionary manipulation patterns compared to numeric logic.

6. Discussion

Behavioral faithfulness as a practical tool. Our results demonstrate that RL-based program synthesis can recover faithful behavioral explanations for a range of algorithmic tasks. The recovered programs are not just quantitatively accurate; they are qualitatively interpretable. A 4-line program capturing a model's house-recommendation policy is arguably more useful to a practitioner than a circuit diagram of attention heads, at least for the purpose of auditing and understanding model behavior.

Complementarity with mechanistic approaches. We do not view behavioral and mechanistic interpretability as competing paradigms. A behaviorally faithful program tells us *what* a model does; mechanistic analysis can explain *how* it does it. The two can be used in concert: a recovered program can serve as a hypothesis for mechanistic investigation ("does the model implement this specific algorithm?"), and mechanistic insights can constrain the program search space.

Limitations of the synthetic benchmark. Our controlled evaluation uses programs as stand-ins for real LLM behavior. Real LLM behavior may not always be expressible as a short program; models may exhibit stochastic, context-dependent, or genuinely complex behavior that resists compact programmatic description. The real LLM examples (Section 4) provide initial evidence of applicability, but more extensive evaluation on diverse model behaviors is needed.

The probing problem. The active synthesis setting highlights that choosing informative inputs is as important as the program synthesis itself. Our current approach trains the probing and synthesis skills jointly via end-to-end RL. An alternative would be to decompose the problem: train a specialized probing policy (optimized for information gain) separately from the synthesis policy. We leave this exploration to future work.

Scaling considerations. Both experiments use relatively

small models (1.5B and 8B parameters). Larger code models would likely achieve higher synthesis quality, and longer training would allow exploration of more complex program spaces. The Tinker distributed training infrastructure we use supports scaling to larger models and longer rollouts.

7. Conclusion

We present an RL-based approach to learning interpretable code explanations of LLM behavior. The in-context synthesis setting demonstrates that GRPO can efficiently search for faithful programs given representative I/O pairs, achieving 0.91 average pass rate on a diverse benchmark. The active synthesis setting shows that an RL-trained multi-turn agent can learn to actively probe a black-box target and iteratively discover programs, improving from 0.28 to 0.61 average pass rate after training. We additionally demonstrate the approach on real Gemini 2.5 Flash behaviors, recovering interpretable decision rules.

Our work opens several directions: (1) incorporating model internals (activations, attention patterns) as additional signal for program synthesis; (2) scaling to more complex and naturalistic LLM behaviors; and (3) developing specialized probing strategies that maximize information gain. More broadly, we believe that behaviorally faithful program synthesis offers a practical and complementary path to understanding what large language models have learned to do.

Acknowledgements

We thank the Tinker team at Thinking Machines Lab for providing the distributed RL training infrastructure, and the CS224R course staff at Stanford for guidance and feedback.

Author Contributions

JT wrote the core infrastructure, led GRPO-training for the ground truth programs, and wrote the baseline harness for in-context synthesis. NJ came up with the project direction, wrote the baseline harness for active synthesis, and led experiments with real LLM behaviors.

References

Ameisen, E., Lindsey, J., and Anthropic. Circuit tracing: Revealing computational graphs in language models. *Transformer Circuits Thread*, 6:16318–16352, 2025.

Bastings, J. and Filippova, K. The elephant in the interpretability room: Why use attention as explanation? *arXiv preprint arXiv:2010.05607*, 2020.

Bills, S., Cammarata, N., Mossing, D., Tillman, H., Gao, L., Goh, G., Sutskever, I., Leike, J., Wu, J., and Saunders, W. Language models can explain neurons in language models. *OpenAI Blog*, 2023.

Conmy, A., Mavor-Parker, A. N., Lynch, A., Heimersheim, S., and Garriga-Alonso, A. Towards automated circuit discovery for mechanistic interpretability. *Advances in Neural Information Processing Systems*, 36, 2023.

Demirović, E., Hou, P., Stuckey, P. J., and Walsh, T. In search of trees: Decision-tree policy synthesis for black-box systems via search. *arXiv preprint arXiv:2409.03260*, 2024.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Gulwani, S., Polozov, O., and Singh, R. Program synthesis. *Foundations and Trends in Programming Languages*, 4 (1-2):1–119, 2017.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Jacovi, A. and Goldberg, Y. Towards faithfully interpretable NLP systems: A survey. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4198–4205, 2020.

Michaud, E. J., Liao, I., Lad, V., Liu, Z., Mudide, A., Loughridge, C., Guo, Z. C., Kheirkhah, T. R., Vukelić, M., and Tegmark, M. Opening the AI black box: Program synthesis via mechanistic interpretability. *arXiv preprint arXiv:2402.05110*, 2024.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Tan, S. H., Caruana, R., Hooker, G., and Lou Koch, Y. Distill-and-compare: Auditing black-box models using transparent model distillation. pp. 303–310, 2018.

Verma, A., Murali, V., Singh, R., Kohli, P., and Chaudhuri, S. Programmatically interpretable reinforcement learning. *Proceedings of the 35th International Conference on Machine Learning*, pp. 5045–5054, 2018.

A. Evaluation Program Catalog

Below we list the 20 held-out evaluation programs used in our benchmark, organized by difficulty level. All programs are anonymized: the model sees only the type signature and I/O examples, never the function name, description, or source code.

A.1. Level 1: Simple Arithmetic

```
# eval_L1_001
def solution(a, b):
    return a + b

# eval_L1_002
def solution(a, b):
    return a - b

# eval_L1_003
def solution(a, b):
    return a / b

# eval_L1_004
def solution(a, b):
    return a * b

# eval_L1_005
def solution(a, b):
    return a + 2 * b
```

A.2. Level 2: Conditionals

```
# eval_L2_001
def solution(a, b):
    return a > b

# eval_L2_002
def solution(a, b):
    return a < b

# eval_L2_003
def solution(a, b):
    if a > b:
        return "X1"
    else:
        return "X2"

# eval_L2_004
def solution(a, b):
    return max(a, b)

# eval_L2_005
def solution(a, b):
    return min(a, b)
```

A.3. Level 3: List and String Operations

```
# eval_L3_001
def solution(lst):
    return sum(lst)

# eval_L3_002
def solution(lst):
    return sum(1 for x in lst if x

# eval_L3_003
def solution(lst):
    return max(lst)

# eval_L3_004
def solution(a, b):
    return a

# eval_L3_005
def solution(lst):
    return len(lst)
```

A.4. Level 4: Complex Decision Trees

```
# eval_L4_001  Range classifier
def solution(n):
    if n == 0:   return "Q"
    elif n > 0:
        if n < 10: return "R1"
        elif n < 100: return "R2"
        else:      return "R3"
    else:
        if n > -10: return "S1"
        elif n > -100: return "S2"
        else:      return "S3"

# eval_L4_002  Grade-like classifier
def solution(x, y):
    if y < 75:   return "T1"
    elif x >= 90: return "T5"
    elif x >= 80: return "T4"
    elif x >= 70: return "T3"
    elif x >= 60: return "T2"
    else:         return "T1"

# eval_L4_003  FizzBuzz variant
def solution(n):
    if n
    elif n
    elif n
    else:         return n

# eval_L4_004  Rock-paper-scissors
def solution(x, y):
    if x == y:   return "Z"
    elif (x=="alpha" and y=="gamma") or \
         (x=="gamma" and y=="beta") or \
         (x=="beta" and y=="alpha"):
        return "X"
    else:         return "Y"

# eval_L4_005  Triangle classifier
def solution(x, y, z):
    if x<=0 or y<=0 or z<=0:
        return "type_d"
    elif x+y<=z or x+z<=y or y+z<=x:
        return "type_d"
    elif x == y == z:
        return "type_a"
    elif x==y or y==z or x==z:
        return "type_b"
    else:
        return "type_c"
```

B. Training Program Statistics

The 96 training programs used in the active synthesis setting are synthetically generated with the following distribution:

Level	Count	Example types
L1 (Arithmetic)	∼24	$a^2 + b^2$, $\|a - b\|$, modular arithmetic
L2 (Conditionals)	∼24	Comparisons, clamping, sign detection
L3 (List/String)	∼24	Aggregations, filtering, counting
L4 (Decision trees)	∼24	Multi-branch classifiers, nested logic

Each program has 10 informative I/O pairs (selected to cover boundary conditions) and 100 randomly sampled evaluation cases. All function names are anonymized to `solution()` and output labels are obfuscated.

C. Hyperparameter Details

Table 4. Full hyperparameter settings for both experiments.

Parameter	**In-Context Synthesis**	**Active Synthesis**
Base model	Qwen2.5-Coder-1.5B	LLaMA-3.1-8B
Adaptation	Full fine-tune	LoRA (rank 32)
Learning rate	1×10^{-4}	4×10^{-5}
RL algorithm	GRPO	GRPO
Group size	8	8
Batch size (programs)	—	8
Optimization steps	100	60
Max tokens per turn	512	512
Max turns per episode	1	10 (train), 5 (eval)
Reward	Pass rate (0–1)	Final-turn pass rate
Training programs	11 (same as eval)	96 (separate)
Eval programs	11	11 (same set)