NORTHWESTERN UNIVERSITY

A Domain Specific Language Approach to Solving Grid Type Problems with Large Language Models

PROJECT

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Computer Science

By

Jack O'Keefe

EVANSTON, ILLINOIS

September 2025

ABSTRACT

Large Language Models (LLMs) have shown remarkable capabilities across a wide range of natural language and coding tasks, but they continue to struggle with compositional reasoning. This thesis investigates the extent to which LLMs can generalize from atomic reasoning primitives to more complex compositions in the context of ARC-style abstract reasoning tasks. Using a synthetic dataset of Domain-Specific Language (DSL) functions, we fine-tuned two instruction-tuned models, Qwen2.5-Coder-7B-Instruct and Phi-3-mini-4k-instruct, and evaluated their performance on both atomic and chained transformations.

Results show that while both models achieve strong performance on isolated primitives (91% and 52% test accuracy, respectively, representing a 60% and 21% improvement over their base counterparts), their accuracy collapses to below 5% on multi-step compositions. This sharp decline underscores a pronounced distribution shift: mastery of atomic transformations does not translate into robust compositional generalization. Qualitative analysis further suggests that occasional successes arise from heuristic shortcuts rather than systematic multi-step reasoning.

These findings provide empirical evidence for a compositionality gap in LLM reasoning. The work highlights the limitations of scale and narrow fine-tuning, and argues that meaningful progress will require chain-aware supervision, intermediate feedback signals, decoding constraints, and adaptive test-time computation. By clarifying where current models succeed and fail, this thesis contributes to a deeper understanding of the challenges in building models capable of genuine abstract reasoning.

Contents

List of Figures

Chapter 1

Background and Related Work

1.1 Introduction

Large Language Models have achieved remarkable success across natural language and code generation tasks, yet their ability to generalize and perform systematic reasoning remains limited. A central challenge is that LLMs are trained as next-token predictors, making them highly effective pattern matchers but prone to failure when problems require compositional or abstract reasoning [1].

The Abstraction and Reasoning Corpus (ARC) was introduced as a benchmark to evaluate such generalization. ARC tasks require transforming input grids into outputs according to latent rules that must be inferred from only a few examples. Humans typically solve these tasks by leveraging abstraction and flexible reasoning, but most machine learning approaches struggle.

This work explores whether LLMs can improve on ARC-style reasoning when augmented with structured tools. Specifically, I provide models with access to a domain-specific language designed for shape manipulation and grid transformations. By combining supervised fine-tuning with this DSL interface, I aim to bridge the gap between raw language model capabilities and the structured reasoning demands of ARC.

I evaluate two models—Qwen2.5-Coder-7B-Instruct and Microsoft Phi-3-mini-4k-instruct—both in their base forms and after fine-tuning with DSL integration. My results highlight the benefits and limitations of this approach, shedding light on the extent to which LLMs can be guided toward more systematic problem solving through explicit function access.

1.2 Background of Grid Puzzles

The grid puzzles were generated with a set of grid generators I created. They utilize the base DSL functions to generate the processes shown in the puzzles. There are nine different colors possible for shapes and backgrounds in the dataset, which are indicated by numbers 0-9. Below is a text example of a grid puzzle from the training set.

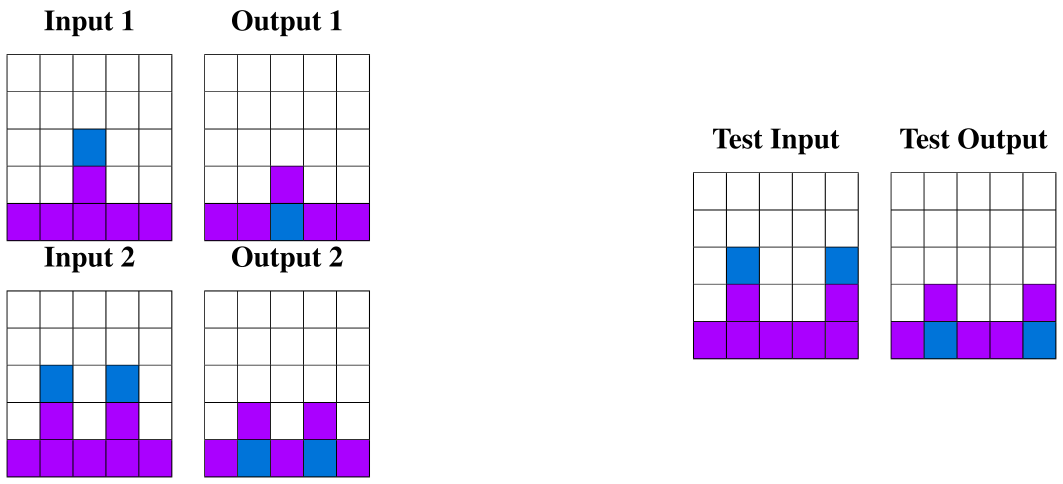


Figure 1.1: Gravity transformation example showing downward gravity applied to blue objects. Blue cells (value 1) fall downward until blocked by purple cells (value 5). The purple horizontal bar acts as a floor/barrier, while purple cells in columns also serve as obstacles. Blue objects move down one row, swapping positions with the purple cell directly above the floor. With my DSL, this would be solved by using the filter objects function and then the apply gravity function, but full examples will be shown later.

1.3 Related Work

The ARC Challenge has motivated a wide range of approaches, spanning program synthesis, neural methods, and reinforcement learning. Early work focused on explicit program induction using domain-specific languages, though these methods often struggled with scalability. DreamCoder [2], for example, demonstrated how programs could be composed hierarchically through a wake-sleep learning cycle, but required substantial computation and careful DSL engineering.

More recent efforts have combined symbolic and neural methods. The 2024 best paper, *Combining Induction and Transduction for Abstract Reasoning* [3], proposed a dual strategy: induction, where an LLM generates Python code to solve puzzles, and transduction, where a neural network directly predicts outputs from inputs. When ensembled, these complementary approaches achieved state-of-the-art performance.

Test-time computation has also emerged as an important theme. *Searching Latent Program Spaces* [4] showed that learning a continuous latent space of programs enables efficient gradient-based search at inference, while *The Surprising Effectiveness of Test-Time Training for Abstract Reasoning* [5] demonstrated that adapting model parameters during inference can yield substantial accuracy gains. These results suggest that allocating computation at test time can be as impactful as model architecture.

Reinforcement learning provides another promising direction. DeepSeek-R1 [6] applied RL directly to base models, achieving reasoning behaviors such as self-verification and reflection without supervised fine-tuning. Its success highlights the potential of RL-guided test-time adaptation as an alternative to program synthesis.

Finally, given the small size of the ARC dataset, data augmentation has been central. RE-ARC [7] introduced a DSL for generating new puzzles in the ARC style, providing additional training data and enabling more robust evaluations.

Together, these works underscore both the diversity of strategies applied to ARC and the difficulty of systematic generalization. This thesis contributes to this line of research by exploring whether fine-tuning LLMs with access to a shape-manipulation DSL can provide a complementary path toward more structured reasoning on ARC-like tasks.

Chapter 2

Experimental Approach

2.1 Dataset Creation and Augmentation

2.1.1 Motivation and Scope

The ARC training dataset contains 400 tasks with 1,302 total examples. My analysis reveals a strong correlation between grid size and puzzle complexity: only 6.8% of tasks (27 tasks) have all examples fitting within 5×5 grids, while 93.2% require larger grids for complete representation. However, 14.5% of tasks (58 tasks) contain at least some examples where both input and output are $\leq 5 \times 5$, forming my experimental subset.

2.1.2 What constitutes a valid ARC Example?

The purpose of the ARC Challenge is to provide a wide distribution of testing examples that don't fall in the distribution of the training set to prove true model generalization. Thus, it is difficult to create a classifier to show what is valid vs. invalid. What can be done, however, is working with the training data as a seed for data augmentation, which is what was done. The training dataset provides examples where the same transformation is done one or more times so that when the testing input is shown, the model should understand to do the same transformation. Thus, I solved every 5x5 training example by hand with my DSL so that there exists a confidence that the DSL convers the training dataset. It is not, guaranteed, however, that it generalizes to the testing dataset, which is the purpose of the ARC Challenge.

2.1.3 DSL Function Dataset (Training Set)

The primary training data consist only of single function examples. Each example provides:

1. Two (input, output) training pairs generated by one DSL function (same transformation semantics across both pairs).
2. One held out test input requiring the model to emit executable Python that sets `output_grid`.

I generated approximately 40,000 such examples (90% train, 5% validation, 5% test) by uniformly sampling across the DSL function families and their internal parameter spaces (coordinates, colors, repetition counts, symmetry axes, etc.). No multi step (chained) compositions appear in this training corpus.

2.1.4 DSL Function Families

The DSL exposes concrete spatial primitives:

- Mirroring: horizontal, vertical
- Rotation: clockwise, counterclockwise (applied in 90° steps)
- Symmetry completion: horizontal, vertical, diagonal, rotational
- Flood fill: seeded region recolor
- Shape translation: move an existing connected component
- Counting and marking: count shapes, mark by size, mark by frequency, count colors
- Shape creation: stamp a new shape from a list of points

- Point connection: draw a straight or diagonal line between two points
- Pattern extraction: most frequent color, largest shape, non-background mask
- Pattern repetition: horizontal, vertical, diagonal, tile
- Gravity: objects fall in specified direction (up, down, left, right) until blocked
- Neighbor modification: fill cells between pairs, mark adjacent neighbors (4-connected, 8-connected)
- Object filtering: keep/remove by color, frequency (minority/majority), or position (row/column)
- Color mapping with transform: simultaneous color remapping and spatial transformation (shift, rotate, flip) with optional region constraints
- Background pattern generation: geometric patterns around center points (diamond, cross, ring, star, checkerboard, diagonal stripes, concentric squares)

These functions act as a constrained DSL so the model learns to produce short, interpretable code instead of arbitrary free form logic.

2.1.5 Representative Single Function Generation Pseudocode

We illustrate the dataset generation process using rotation as a representative example. The key principle is **parameter consistency**: each generated puzzle (consisting of two training pairs and one test pair) uses the *same* transformation rule, allowing the model to infer the pattern from the training examples and apply it to the test case.

Generation Process. For each puzzle, we:

1. Sample transformation parameters once (e.g., rotation angle $k \in \{90°, 180°, 270°\}$ and direction $dir \in \{\text{cw}, \text{ccw}\}$).

2. Generate three distinct random shapes.

3. Apply the *same* transformation (parameterized by k and dir) to all three shapes, producing three input-output pairs.

4. Package two pairs as training examples and one as the test case.

5. Store the executable Python solution code that, when executed on the test input, produces the correct output.

Algorithm 1 Generate Rotate Shape Examples (Representative)

```
1:  function GENERATEROTATESHAPEEXAMPLES($N, size$)
2:      $examples \leftarrow []$
3:      for $i = 1$ to $N$ do
4:          $k \leftarrow$ RANDOMINT$(1, 3)$                                  ▷ Number of 90° steps
5:          $dir \leftarrow$ RANDOMCHOICE$(\{\text{cw}, \text{ccw}\})$
6:          $(train1\_in, meta) \leftarrow$ RANDOMSHAPE$(size)$
7:          $train1\_out \leftarrow$ APPLYROTATE$(train1\_in, k, dir)$
8:          $(train2\_in, meta) \leftarrow$ RANDOMSHAPE$(size)$
9:          $train2\_out \leftarrow$ APPLYROTATE$(train2\_in, k, dir)$
10:         $(test\_in, meta) \leftarrow$ RANDOMSHAPE$(size)$
11:         $test\_out \leftarrow$ APPLYROTATE$(test\_in, k, dir)$
12:         $solution \leftarrow$ Python line that applies rotation with $k$, $dir$
13:         Append {train1, train2, test, solution} to $examples$
14:     end for
15:     return $examples$
16: end function
```

Concrete Example. Consider a puzzle with $k = 1$ (two 90° steps) and $dir = \text{cw}$ (clockwise). Figure 2.1 shows how three different shapes undergo identical clockwise rotation. The model observes Training Pairs 1 and 2, then must emit code to transform the test input.

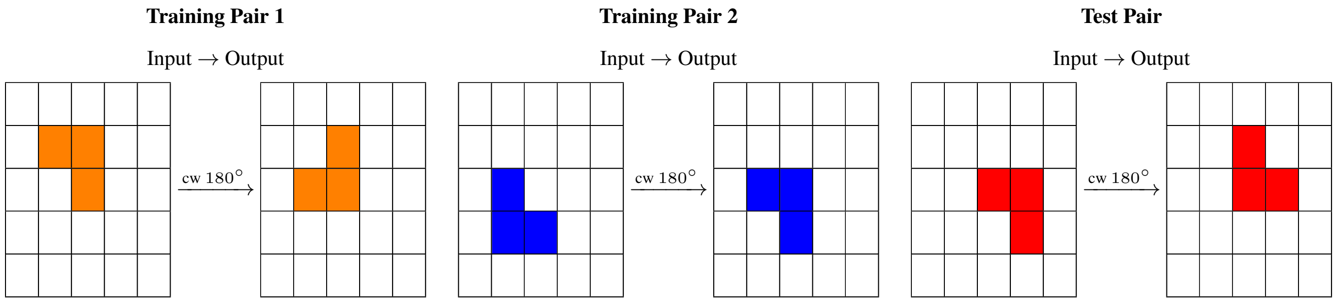


Figure 2.1: Three shapes (orange, blue, red) undergoing identical 180° clockwise rotation.

The model must emit Python code that rotates the test input 180° clockwise:

```
output_grid = RotateShapeGenerator.rotate_shape_on_grid(
    test_input, 2, 2, 2, 'clockwise')
```

This few-shot structure mirrors the original ARC benchmark format. All 15 DSL primitive families follow this template, varying only in sampled parameters (e.g., mirror axis, flood fill seed position, repetition count).

This ensures that the model must learn to generalize the transformation rule from the training pairs, rather than memorizing specific grid configurations. For a full example, see Appendix .2 for the code that outlines the generator.

2.1.6 Determinism and Validation

For every generated example (single function and chained):

- Parameter sampling is recorded implicitly via emitted code so execution is reproducible.
- An interpreter injects only the sanctioned DSL functions and re executes the solution to verify `output_grid` matches the stored test output. Please see the interpreter in the Appendix .1 . It ensures that the code is run precisely so that the DSL is isolated from the LLM and then the output grid is tested against the test grid.
- Any example failing validation is discarded and regenerated.

2.1.7 Compositional Evaluation Dataset (Unseen During Training)

To assess compositional generalization I build a separate Chained Pipeline evaluation set (300 examples) where each example applies a sampled chain of 2, 3, or 4 transformations (balanced counts across lengths). Each record contains:

- Two single chain training pairs (same ordered transform sequence)
- One test input with the same chain
- `transform_chain` list and `chain_length`
- Multi-line executable `solution` reproducing the final grid

I did not train on these chained compositions. They are purely for measuring how well a model trained only on atomic DSL operations can generalize to sequences.

Concrete Example. Consider a puzzle with a 2-step chain: (1) connect two points with a diagonal line, then (2) repeat the pattern diagonally. All three examples undergo the same transformation sequence, but start from different initial configurations.

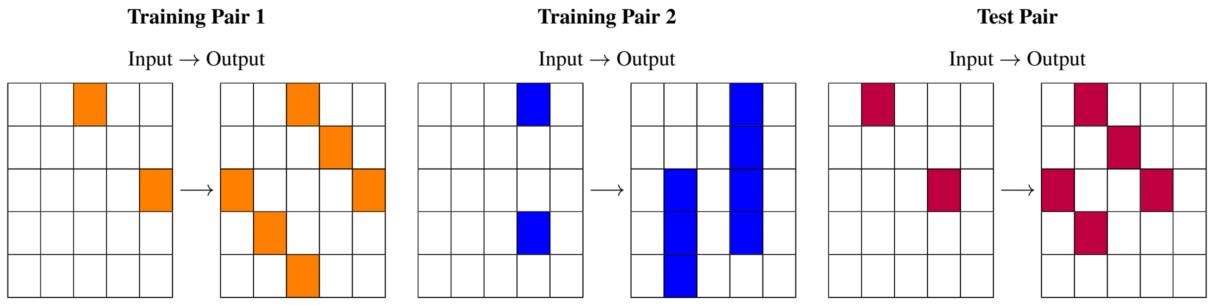


Figure 2.2: Three examples undergoing identical 2-step chain: (1) connect two detected marker points with a line using the same color, then (2) down 2 and to the left 2.

The model observes the two training transformations and must emit multi-line Python code that applies both steps to the test input:

```
step_1 = ConnectGenerator.connect(test_input, (0, 1), (2, 3), 8)
output_grid = ConnectGenerator.connect(step_1, (2, 0), (3, 1), 8)
```

This few-shot chained structure extends the atomic format (Figure 2.1). Models trained exclusively on single-function examples typically collapse to single-line guesses, omitting the intermediate `step_1` variable and failing to compose operations correctly. The connect-then-repeat chain requires both spatial reasoning (line drawing) and pattern recognition (diagonal replication).

2.1.8 Rationale

By first training exclusively on atomic DSL functions, I ensure the model internalizes the primitive operational vocabulary. The separate chained dataset then isolates true compositional generalization rather than memorization of multi step templates.

2.2 Fine-Tuned Qwen and Phi Models

2.2.1 Instruction Formatting

Prompt template (atomic training examples only):

1. Natural language directive (always the same concise instruction).
2. Two serialized (JSON-like) training input/output grid pairs.
3. Test input terminated by the literal token sequence "`Test Output:`" (no target grid shown).

Target generation: minimal Python lines that construct `output_grid`. No chain-of-thought, comments optional, extraneous prose discouraged.

2.2.2 Training Scope

Only atomic (single DSL function) transformations used for fine-tuning. No multi-step / chained compositions shown during training. Compositional generalization evaluated separately on the held-out chained dataset (chain lengths 2–4).

2.2.3 Parameter-Efficient Adaptation

LoRA applied with 4-bit base weight loading (QLoRA style) and mixed precision (bf16/fp16) where supported.

- Base models: Qwen2.5-Coder-7B-Instruct and Phi-3 Mini (4k).
- LoRA rank experiments: Qwen (r = 8, 16), Phi (r = 16, 32); a lower rank was used for Qwen as it is a larger model and would not fit on the GPU with the higher rank that Phi used.
- Target modules: primary projection and value/attention linear layers (kept consistent across models).
- Optimizer: low learning rate to preserve pretrained code priors; no aggressive warmup.
- Gradient checkpointing enabled to fit higher rank runs under memory constraints.
- Uniform sampling over atomic DSL function types each epoch to avoid frequency bias.
- Context window sized to worst-case prompt + anticipated multi-line solution (ensuring no truncation of longer valid code completions).

2.2.4 Evaluation and Stratification

Generated code executed inside a constrained interpreter namespace containing only DSL primitives plus safe utilities. Exact-match accuracy = structural equality of predicted `output_grid` and reference grid. Reported:

- Overall atomic test accuracy.
- DSL Function Test slice (fine-grained primitive coverage).
- Chained dataset accuracy stratified by `chain_length` $(2, 3, 4)$ using gold solutions (upper bound) and model generations.

2.2.5 Quality Controls

1. Deterministic dataset synthesis (seeded sampling + explicit transform specs) enabling replay.
2. 100% interpreter pass rate for released (gold) solutions prior to model evaluation.
3. Versioned Hugging Face release (`v0.1.0`) with documented features (`chain_length`, `transform_chain`, solution lines).
4. Fully synthetic supervision; no human ARC solutions or external code corpora leaked into fine-tuning data.

2.2.6 Outcome

Pipeline yields an auditable compositional code generation benchmark: controllable complexity via chain length, granular primitive attribution, and strict execution-based scoring to isolate reasoning versus pattern memorization.

2.2.7 Example

```
<|im_start|>system
You are Qwen, created by Alibaba Cloud. You are a helpful assistant.<|im_en
<|im_start|>user
Write Python code that transforms test_input into the expected test output.
Training Example 1:
Input: [[1, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
Output: [[1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, 0]

Training Example 2:
Input: [[6, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
Output: [[6, 0, 0, 0, 0], [0, 6, 0, 0, 0], [0, 0, 6, 0, 0], [0, 0, 0, 6, 0]
```

```
Test Input: [[7, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0
Test Output: <|im_end|>
<|im_start|>assistant
output_grid = RepeatPatternGenerator.repeat_diagonal(test_input, 5)<|im_end
```

The Phi and Qwen models were formatted according to the tokenizer format provided by the models. The only context given was the text before the `<|im_start|>assistant` token, and the loss was only calculated on the text generated after this token.

2.2.8 Fine-tuning Configuration

Both models were fine-tuned using supervised fine-tuning with the following hyperparameter configurations:

2.2.8.1 Training Hyperparameters

- **Batch Size**: Per-device batch size of 128 for Phi-3-mini and 90 for Qwen2.5-Coder-7B (adjusted for memory constraints)
- **Learning Rate**: 3×10^{-4} with cosine decay schedule
- **Optimizer**: AdamW with weight decay of 0.01
- **Warmup**: 5% of total training steps
- **Training Duration**: 1 epoch on the atomic DSL function dataset
- **Sequence Length**: Maximum of 512 tokens
- **Gradient Checkpointing**: Disabled for both models to improve training speed

2.2.8.2 Data Processing

- **Packing**: Disabled to maintain clear example boundaries
- **Data Loading**: 4 parallel workers for efficient data pipeline
- **Tokenization**: Models were formatted according to their native chat templates, with context provided before the `<|im_start|>assistant` token and loss calculated only on the assistant's response

2.2.8.3 Evaluation and Logging

- **Evaluation Strategy**: Step-based evaluation every 10 training steps for the Phi model and every 100 for the Qwen model, due to different batch sizes
- **Logging**: Weights & Biases (W&B) integration for loss tracking
- **Model Saving**: Epoch-based checkpointing with automatic push to Hugging Face Hub

The configuration prioritized stable training with conservative learning rates and standard regularization (weight decay, warmup) to preserve the models' pre-trained code generation capabilities while adapting to the DSL-specific task.

2.2.9 Training Dynamics and Loss Curves

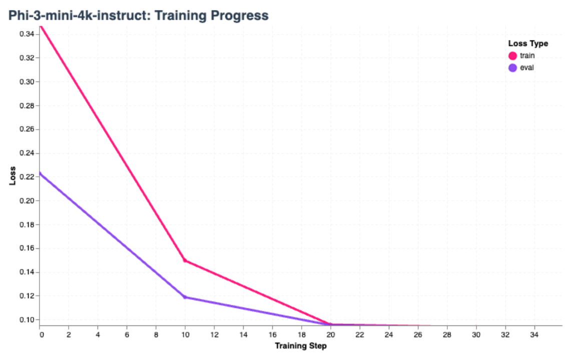
2.2.9.1 Loss Function

Both models were trained using cross-entropy loss, calculated only on the model's generated code tokens (after the `<|im_start|>assistant` delimiter). This selective loss computation ensures the model learns to generate solutions rather than memorizing the input format. The loss function can be expressed as:

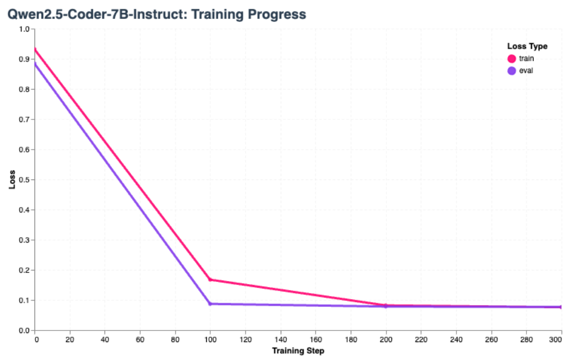
$$\mathcal{L} = -\sum_{t \in \text{assistant}} \log p_\theta(x_t | x_{<t})$$

where the summation is restricted to tokens in the assistant's response portion of the sequence.

2.2.9.2 Training Loss Analysis



(a) Phi-3-mini-4k-instruct



(b) Qwen2.5-Coder-7B-Instruct

Figure 2.3: Training and evaluation loss curves for both models over the course of fine-tuning.

Figure 2.3a and Figure 2.3b show the training and evaluation loss curves for Phi-3-mini-4k-instruct and Qwen2.5-Coder-7B-Instruct respectively.

Phi-3-mini-4k-instruct The model exhibits rapid initial learning with loss decreasing from approximately 0.35 to 0.15 within the first 10 steps. The training loss (pink) starts higher than evaluation loss (purple) due to the running average including early high-loss steps. Both curves converge to approximately 0.10 by step 30, indicating stable learning without overfitting. The smooth exponential decay pattern suggests appropriate learning rate selection and stable gradient flow.

Qwen2.5-Coder-7B-Instruct Starting from a higher initial loss (0.95), Qwen shows a steeper learning curve, dropping to 0.15 within 100 steps. The model achieves a lower final loss (0.07) compared to Phi-3, potentially due to its larger capacity (7B vs 3B parameters). The extended training duration (300 steps vs 30 for Phi-3) reflects the dataset size and batch size differences. Notably, both training and evaluation losses remain closely aligned throughout training, demonstrating good generalization.

Comparative Observations

- Both models show healthy convergence with no signs of overfitting (evaluation loss does not diverge)
- The initial gap between training and evaluation loss in early steps is expected behavior due to loss averaging mechanics as the train loss is averaged across the 10 or 100 steps, whereas the evaluation loss is solely computed at that step
- Qwen's lower final loss suggests better fit to the DSL task, though this must be validated through downstream accuracy metrics
- The cosine learning rate schedule appears effective for both models, with no instability or oscillations observed

Chapter 3

Results

3.0.1 Atomic DSL Function Test Performance with Base Models

The unmodified base models were evaluated on all 1,859 atomic DSL transformation tasks using a simple instruction prompt: "Write Python code that transforms `test_input` into the expected test output. Only output executable Python lines setting `output_grid` as the variable holding the final output." Because the prompt (for this diagnostic run) exposed both the input and the target grid, the models frequently bypassed transformation reasoning and directly emitted a literal assignment (e.g., `output_grid = [[0,0,0,0,0],[...]]`). Thus, the reported accuracies largely reflect verbatim reproduction of the target rather than application of the intended DSL operation. The base models were able to correctly output the correct grid for simple transformations like `connect` and at times were able to answer correctly for more complex transformations like `count_shapes`. It is notable that both models performed within their margins of error despite Qwen being nearly double the size of the Phi model.

As additional reference points, two trivial baselines were evaluated. A random guessing baseline by sampling each grid cell uniformly from 0–9 achieves an expected exact-match accuracy of less than 10^{-24} on 5×5 grids, effectively 0%. An identity baseline that simply returns the input grid as output also yields 0%, since nearly all DSL transformations modify at least one cell. These baselines establish a lower bound and confirm that even the un-fine-tuned models' $\approx 30\%$

accuracy reflects nontrivial pattern imitation rather than chance.

Model	Accuracy (95% CI)	Parameters
Random baseline	0.0%	–
Identity baseline	0.0%	–
Phi-3-mini-4k-instruct	29.4% [27.4, 31.5]	3.8B
Qwen2.5-Coder-7B-Instruct	31.2% [29.1, 33.3]	7B

Table 3.1: Base model exact-match accuracy on atomic DSL function tasks (1,859 examples). Models often solved items by literal reproduction of the target grid rather than applying transformations.

3.0.2 Atomic DSL Function Test Performance with Fine-tuned Models

Table 3.2 presents the test set accuracy for both fine-tuned models on the held-out atomic DSL function dataset containing 1,859 examples.

Model	Test Accuracy	Parameters
Phi-3-mini-4k-instruct	52.0% [49.7, 54.3]	3.8B
Qwen2.5-Coder-7B-Instruct	91.0% [89.6, 92.2]	7B

Table 3.2: Test accuracy on atomic DSL function transformations (Wilson 95% confidence intervals in brackets).

The substantial performance gap between the two models (39 percentage points) reflects several architectural and training differences. Qwen2.5-Coder-7B achieved 91% accuracy, demonstrating strong acquisition of the DSL primitives after just one epoch of training. In contrast, Phi-3-mini reached 52% accuracy, suggesting partial but incomplete mastery of the transformation patterns.

The superior performance of Qwen2.5-Coder can be attributed primarily to its larger model capacity. Despite using a lower LoRA rank (r=8 for Qwen vs r=16 for Phi-3), the Qwen model's 7B parameter base provides approximately 1.8× more parameters than Phi-3-mini's 3.8B. This additional capacity translates to more expressive power for learning the mapping between grid patterns and their corresponding DSL operations. The total trainable parameters with LoRA still favored Qwen due to the larger base dimensions of its attention layers, even with the reduced rank factorization.

Additionally, Qwen2.5-Coder's specialization as a code-generation model may provide better inductive biases for learning structured transformations compared to Phi-3-mini's more general-purpose training. The 91% accuracy on atomic functions establishes a strong foundation for evaluating compositional generalization on the chained transformation dataset.

3.0.3 Chained DSL Function Test Performance

Task. Zero-shot compositional generalization: models fine-tuned only on *atomic* (single-function) DSL transformations are evaluated on held-out problems requiring a sequence of 2–4 ordered transformations (mirror, rotate, symmetry completion, flood fill, move, connect, repeat, create shape, pattern extraction, count/transform, etc.). No chained examples were seen during training. This was done to explore if fine-tuning on single transformation examples could generalize to multi-transformation examples, displaying emergent behavior. The same Phi-3-mini-4k-instruct + LoRA and Qwen2.5-Coder-7B-Instruct + LoRA models were used in this test.

Metric. Exact-match of the final output grid after executing the emitted Python (single decoding pass, greedy/standard settings; no self-consistency, beam search, or repair). Intermediate states are unsupervised.

Model	Overall	Length 2	Length 3	Length 4
Phi	4.33% [2.55, 7.27]	6.32% [2.93, 13.10]	3.96% [1.55, 9.74]	2.88% [0.99, 8.14]
Qwen	2.67% [1.36, 5.17]	3.16% [1.08, 8.88]	1.98% [0.54, 6.93]	1.92% [0.53, 6.74]

Table 3.3: Exact-match accuracy on chained DSL transformation tasks (Wilson 95% confidence intervals in brackets).

Example. Below is an example that shows how the Phi model correctly answered one of the multi transformation answers. Every single correct multi-transformation puzzle that was answered correctly by the Phi or Qwen models was answered correctly because a single transformation could capture multiple transformations together. Below is an interesting example, where even though there are four transformations, the composition of those transformations is equivalent to the flood fill transformation, which the Phi model correctly answered with.

```
# Model-generated by Phi-3-mini-4k-instruct + LoRA
output_grid = FloodFillGenerator.flood_fill(test_input, (4, 3), 2)

# Code that generated testing output (4 composed DSL operations)
step_1 = RepeatPatternGenerator.repeat_diagonal(test_input, 3)
step_2 = RepeatPatternGenerator.tile_pattern(step_1)
step_3 = RotateShapeGenerator.rotate_clockwise(step_2)
output_grid = SymmetryCompleteGenerator.complete_rotational_symmetry(step_3)
```

Listing 3.1: Illustrative chained example showing single-step generation vs. original multi-step composition.

Overall Observations.

1. Large drop from atomic accuracy (91% / 52%) to chained (2.5–4.17%) highlights a pronounced distribution shift.

2. Accuracy decays with chain length, consistent with multiplicative error propagation.

3. Smaller Phi slightly exceeds Qwen despite lower capacity, suggesting reduced overconfident heuristic collapse, or is just a fragment of stochasticity, and that nothing greater was actually learned by the model, but this warrants further investigation.

Takeaway. Training solely on atomic primitives yields strong primitive mastery but weak emergent composition. Narrow improvements (longer decoding, more parameters) do not substitute for structural supervision and intermediate incentives. Addressing the compositional gap will likely require (i) explicit chain-aware data, (ii) intermediate execution feedback, and (iii) decoding constraints that enforce multi-step program structure over heuristic one-shot guesses.

3.1 Conclusion

This work explored the compositional reasoning gap in large language models when applied to ARC-style abstract reasoning tasks. By constraining training to a synthetic dataset of atomic DSL primitives, I demonstrated that models such as Qwen2.5 and Phi-3-mini can achieve strong mastery of isolated transformations (91% and 52% test accuracy, respectively, which is a 60% and 21% improvement over the base models' performance). However, when evaluated on chained compositions of these primitives, accuracy collapsed to less than 5%. This sharp drop confirms that fluency with atomic operations does not translate into robust compositional generalization.

The experimental results highlight several important dynamics. First, performance degrades as chain length increases, reflecting multiplicative error propagation and the absence of structural guidance. Second, the smaller Phi model slightly exceeded Qwen on chained tasks despite lower capacity, which may indicate less reliance on brittle heuristics or simply stochastic variance. Finally, qualitative examples suggest that models occasionally succeed by collapsing multi-step compositions into a single memorized transformation, rather than reasoning over intermediate steps. Together, these observations illustrate that structural supervision, rather than scale alone, is the key missing ingredient for abstract reasoning.

3.2 Limitations and Future Work

While this thesis demonstrates that fine-tuning large language models with access to a domain-specific language (DSL) can improve performance on ARC-style reasoning tasks, several limitations remain. Addressing these challenges presents opportunities for future research.

3.2.1 Dataset Limitations

A fundamental limitation is the small size of the ARC dataset. With only 400 training tasks, each containing one or two examples, supervised fine-tuning is constrained by data scarcity. Although RE-ARC and other augmentation methods provide additional synthetic puzzles, these may not fully capture the diversity and difficulty of original ARC tasks. Future work could explore more systematic approaches to augmentation, including procedural generation of puzzles with controlled abstractions or leveraging human-designed extensions of ARC.

3.2.2 Model Limitations

The models tested in this work—Qwen2.5-Coder-7B-Instruct and Microsoft Phi-3-mini-4k-instruct—are relatively small by modern standards. While this makes experimentation more computationally feasible, it may also limit generalization capacity. Larger models, or models pretrained on tasks more closely aligned with structured reasoning (e.g., code-heavy corpora or symbolic domains), might yield stronger results. Exploring whether DSL-augmented fine-tuning scales effectively to larger models remains an open question.

3.2.3 Evaluation Limitations

The evaluation in this thesis focused primarily on exact-match accuracy across a fixed test set. While this metric is standard in ARC evaluations, it does not capture partial progress toward a solution or qualitative differences in reasoning strategies. For example, a model that produces near-correct outputs but consistently fails on color transformations may have learned useful abstractions that are not reflected in accuracy alone. Future work could incorporate richer evaluation protocols, including error categorization, human-in-the-loop assessments, or process-based metrics such as intermediate reasoning steps.

3.2.4 Potential of Test-Time Adaptation

One major theme in recent ARC research is the importance of test-time computation. Techniques such as test-time training, Monte Carlo tree search, and reinforcement learning have all been shown to enhance reasoning performance by allowing models to adapt dynamically. This thesis did not implement such methods, but they represent a natural next step. In particular, reinforcement learning approaches such as GRPO could be combined with DSL access to encourage exploration of novel transformations at inference time. Such hybrid approaches may help overcome the limitations of static fine-tuning.

3.2.5 Broader Implications

Beyond the ARC benchmark, the findings of this thesis speak to a central challenge in artificial intelligence, which how to enable models that excel at surface-level pattern recognition to also engage in systematic, compositional reasoning. The gap observed here, between mastery of atomic primitives and failure on their compositions, mirrors broader limitations seen in domains such as code synthesis, theorem proving, and symbolic planning. By showing that explicit DSL integration can anchor LLMs in more interpretable and structured operations, this work suggests a pathway toward hybrid neuro-symbolic systems that combine the flexibility of modern language models with the rigor of formal reasoning. While limited in scope, these results contribute to the ongoing discussion of how to move from predictive fluency toward genuine reasoning, a shift that is likely to be essential for deploying AI systems in domains requiring reliability, abstraction, and trustworthiness.

3.2.6 Summary

In summary, the limitations of this project highlight promising directions: expanding datasets, scaling models, enriching evaluation, and integrating test-time adaptation. Addressing these open questions would not only strengthen performance on ARC but also contribute to the broader challenge of enabling machine learning systems to reason more flexibly and systematically.

Bibliography

[1] François Chollet. On the measure of intelligence. *CoRR*, abs/1911.01547, 2019.

[2] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning, 2020.

[3] Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M. Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, Wei-Long Zheng, Zenna Tavares, Yewen Pu, and Kevin Ellis. Combining induction and transduction for abstract reasoning, 2024.

[4] Matthew V Macfarlane and Clément Bonnet. Searching latent program spaces, 2025.

[5] Ekin Akyürek, Mehul Damani, Adam Zweiger, Linlu Qiu, Han Guo, Jyothish Pari, Yoon Kim, and Jacob Andreas. The surprising effectiveness of test-time training for few-shot learning, 2025.

[6] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie 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, J. L. 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, Minghui 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, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin 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, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin 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, 2025.

[7] Michael Hodel. Addressing the abstraction and reasoning corpus via procedural example generation, 2024.

.1 Rotation Shape Generator Implementation

The following code implements the rotation shape generator used to create training examples in our dataset. This generator creates few-shot examples where shapes are rotated around a fixed center point.

```python
import random
import copy
from .base import ExampleGenerator

class RotateShapeGenerator(ExampleGenerator):
    """
    A class to generate few-shot examples for the 'rotate shape' task.
    """

    def __init__(self, size=5):
        super().__init__(size)

    @staticmethod
    def rotate(points, center, times, direction):
        """
        Rotate a set of points around a center point.

        Args:
            points (list): List of (row, col) tuples representing the shape
            center (tuple): (row, col) center of rotation
            times (int): Number of 90-degree rotations (1, 2, or 3)
            direction (str): 'clockwise' or 'counterclockwise'
        """
        rotated_points = []
        center_row, center_col = center

        angle = (times * 90) % 360
        if direction == 'counterclockwise':
            angle = 360 - angle

        for row, col in points:
            translated_row = row - center_row
            translated_col = col - center_col

            if angle == 90:
                new_row = translated_col
                new_col = -translated_row
            elif angle == 180:
                new_row = -translated_row
                new_col = -translated_col
            elif angle == 270:
                new_row = -translated_col
                new_col = translated_row
            else:
                new_row = translated_row
                new_col = translated_col

            final_row = new_row + center_row
            final_col = new_col + center_col
            rotated_points.append((final_row, final_col))

        return rotated_points

    @staticmethod
    def rotate_shape_on_grid(grid, center_row, center_col, times, direction):
        """Rotate a shape on a grid using the rotate function."""
        shape_coords, value, _ = RotateShapeGenerator.detect_shape(grid)
        if not shape_coords:
            return grid

        new_grid = [[0 for _ in range(len(grid[0]))] for _ in range(len(grid))
            ]
        center = (center_row, center_col)
        rotated_points = RotateShapeGenerator.rotate(shape_coords, center,
            times, direction)

        for row, col in rotated_points:
            if 0 <= row < len(new_grid) and 0 <= col < len(new_grid[0]):
                new_grid[row][col] = value

        return new_grid

    def create_fewshot_examples(self, num_examples=100):
        """Generate few-shot examples with consistent rotation parameters."""
        examples = []

        for i in range(num_examples):
            grid_center_row = self.size // 2
            grid_center_col = self.size // 2

            train1 = self.generate_rotate_example()
            times = train1['times']
            direction = train1['direction']

            train1['output'] = self.rotate_shape_on_grid(
                train1['input'], grid_center_row, grid_center_col, times,
                    direction
            )

            train2 = self.generate_rotate_example()
            train2['output'] = self.rotate_shape_on_grid(
                train2['input'], grid_center_row, grid_center_col, times,
                    direction
            )

            test = self.generate_rotate_example()
            test['output'] = self.rotate_shape_on_grid(
                test['input'], grid_center_row, grid_center_col, times,
                    direction
            )

            solution = [
                f"output_grid = RotateShapeGenerator.rotate_shape_on_grid(
                    test_input, {grid_center_row}, {grid_center_col}, {times},
                     '{direction}')"
            ]

            examples.append({
                "train_input1": train1["input"],
                "train_output1": train1["output"],
                "train_input2": train2["input"],
                "train_output2": train2["output"],
                "test_input": test["input"],
                "test_output": test["output"],
                "solution": solution
            })

        return examples
```

.2 Solution Interpreter Implementation

The interpreter validates generated solutions by executing them in a controlled namespace containing all DSL functions and generator classes. This ensures that generated code is both syntactically correct and produces the expected output.

```python
import json
import random
import copy
import inspect
from generators import *

class ExampleInterpreter:
    """
    Interprets and tests JSON examples against their corresponding generator
       classes.
    """

    def __init__(self, json_dir="JSON_training"):
        self.json_dir = json_dir
        self.generator_classes = self.discover_generator_classes()

    def discover_generator_classes(self):
        """Discover all available generator classes."""
        generator_classes = {}
        import generators
        for name in dir(generators):
            obj = getattr(generators, name)
            if inspect.isclass(obj) and name.endswith('Generator'):
                generator_classes[name] = obj
        return generator_classes

    def create_execution_namespace(self):
        """Create namespace with all generators and DSL functions."""
        namespace = {
            'copy': copy,
            '__builtins__': __builtins__,
        }

        namespace.update(self.generator_classes)

        # Add function aliases for different naming conventions
        for class_name, class_obj in self.generator_classes.items():
            try:
                for attr in dir(class_obj):
                    if attr.startswith('_'):
                        continue
                    attr_obj = getattr(class_obj, attr)
                    if callable(attr_obj):
                        if attr not in namespace:
                            namespace[attr] = attr_obj

                        # Add capitalized variant
                        parts = attr.split('_')
                        if len(parts) > 1:
                            cap_variant = '_'.join([parts[0].capitalize()] +
                                parts[1:])
                            if cap_variant not in namespace:
                                namespace[cap_variant] = attr_obj
            except Exception:
                continue

        return namespace

    def test_example(self, example):
        """Test a single example by executing its solution."""
        test_input = example['test_input']
        expected_output = example['test_output']
        solution_lines = example['solution']

        namespace = self.create_execution_namespace()
        namespace['test_input'] = copy.deepcopy(test_input)

        try:
            full_solution = '\n'.join(solution_lines)
            exec(full_solution, namespace)
            output_grid = namespace.get('output_grid', namespace.get('grid',
                None))

            if output_grid is None:
                return False, None

            success = output_grid == expected_output
            return success, output_grid
        except Exception as e:
            return False, None

    def test_json_file(self, json_filename, num_tests=5):
        """Test random examples from a JSON file."""
        examples = self.load_json_examples(json_filename)
        test_examples = random.sample(examples, min(num_tests, len(examples)))

        passed = 0
        for example in test_examples:
            success, _ = self.test_example(example)
            if success:
                passed += 1

        return passed, len(test_examples)

    def test_all_json_files(self, num_tests_per_file=5):
        """Test all JSON files in the directory."""
        json_files = [f for f in os.listdir(self.json_dir) if f.endswith('.
            json')]

        total_passed = 0
        total_tests = 0

        for json_file in json_files:
            passed, total = self.test_json_file(json_file, num_tests_per_file)
            total_passed += passed
            total_tests += total

        return total_passed, total_tests
```

The interpreter performs three key functions: (1) dynamically discovers all available generator classes and their methods, (2) creates an execution namespace that supports multiple naming conventions for robustness, and (3) validates that executed solutions produce outputs matching the gold-standard transformations.