

Zindi ARC Challenge: A Week-Long Sprint Towards Conditioned Reasoning

Cédric Manouan

manouancedric@gmail.com

1 Background

The ARC (Abstraction and Reasoning Corpus),¹ introduced in 2019 by François Chollet, was designed to probe a system’s capacity for fluid intelligence. It involves solving visual pattern puzzles from very limited examples — mimicking the kind of reasoning humans perform when generalizing from few demonstrations. Unlike typical machine learning benchmarks, it doesn’t reward brute-force data-hungry solutions, but instead focuses on generalizable reasoning strategies.

The Zindi ARC Challenge Africa² serves as a simplified version of the original ARC benchmark, built to encourage creative experimentation with the original competition format.

Competition objective: Develop a machine learning model that can successfully generalize from a small number of examples to solve abstract reasoning tasks. This capability is evaluated through the ability for the model to:

- Infer underlying patterns and rules from a limited set of input-output grid pairs.
- Apply these inferred rules to a new test input grid to generate the correct output grid.
- Demonstrate “fluid intelligence,” the ability to solve novel problems without relying on a vast amount of prior knowledge or training data.

It is important to acknowledge that the tasks in this Zindi competition are adapted from the original ARC-AGI dataset developed by the ARC team, which is publicly available. While the competition organizers curated a modified version to define custom train/test splits, the core task formulations remain closely aligned with the original. As such, leveraging models that were pre-trained on or exposed to the ARC dataset—especially if those models had access to the original public training tasks—introduces a high risk of (unintended) memorization. This compromises the intended generalization objective of the benchmark, where models are expected to reason abstractly over unseen problems rather than recall previously encountered solutions.

2 Project Goal

This sprint explored the question: *Can domain intuition and a few LLM prompts solve complex visual reasoning tasks competitively without extensive training or compute?*

I hypothesized that with careful problem analysis and handcrafted strategies, one could score up to 40-50% on this competition.

In line with the original intent of the ARC benchmark, which emphasizes solving novel tasks without prior exposure or memorization, I further aimed to assess whether few-shot prompting alone could yield competitive results—without any fine-tuning. This approach preserves the spirit of the challenge, where the solving agent (human or machine) must infer abstract rules from limited examples, rather than rely on

¹<https://arcprize.org>

²<https://zindi.africa/competitions/the-arc-challenge-africa>

extensive training. The goal was to simulate a human-like problem-solving process: general reasoning guided by structure and analogy, rather than pattern recall.

3 Experiment

The code for this solution available in the project repository at <https://github.com/dric2018/zindi-arc-agi>. The utility functions (`utils.py`) and model-specific implementations (`model.py`) are saved in the `src` folder. This folder also comprises a base prompt to be used a `system prompt` for LLM setup.

3.1 Day 1–2: EDA and Problem Analysis

3.1.1 Setup and Tooling

- Hardware (compute capabilities): Apple M4 MBP (24GB RAM), NVIDIA RTX 6000 (48GB VRAM)
- Visualization tools extended from Kaggle starter code by Oleg X³
- Custom utilities for: grid parsing, data sampling, output shape inference, pixel accuracy calculation, and submission generation.

Overall Config

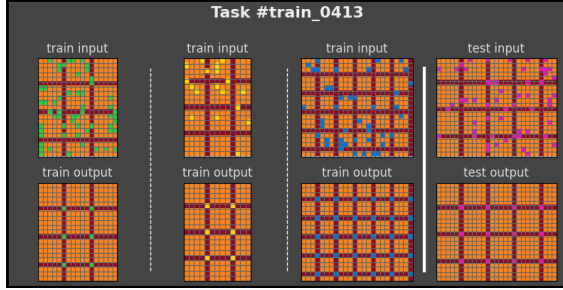
```
from matplotlib import colors
import os.path as osp
import torch

class Config:
    # I/O
    root = "../"
    data_path = osp.join(root, "data")
    submission_path = osp.join(root, "submissions")
    model_zoo = osp.join(root, "models")
    experiment = "llm-fs"
    # 0:black, 1:blue, 2:red, 3:green, 4:yellow, # 5:gray, 6:magenta, 7:orange, 8:sky, 9:brown
    CMAP = colors.ListedColormap(
        ['#000000', '#0074D9', '#FF4136', '#2ECC40', '#FFDC00',
         '#AAAAAA', '#F012BE', '#FF851B', '#7FDBFF', '#870C25'])

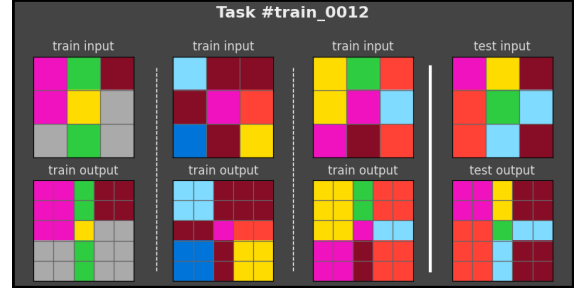
    NORM = colors.Normalize(vmin=0, vmax=9)
    DEFAULT_BG_VALUE = 7 # (orange not black...from EDA)
    # model vars
    base_llm = "Qwen/Qwen2.5-14B-Instruct"
    model_name = "arc-solver-"+base_llm.split("/)[-1]
    device = "cuda" # or "mps" when running on Mac
    target_platform = "cuda" # or "mlx" when running on Mac
    dtype = torch.bfloat16
    max_tokens = 4096
    temperature = 0.1
    top_p = 0.9
    repetition_penalty = 1.1
    do_sample = True
    trust_remote_code = True
    quantize_model = True
    MAX_N = 30
```

³<https://www.kaggle.com/code/allegich/arc-agi-2025-starter-notebook-eda/notebook>

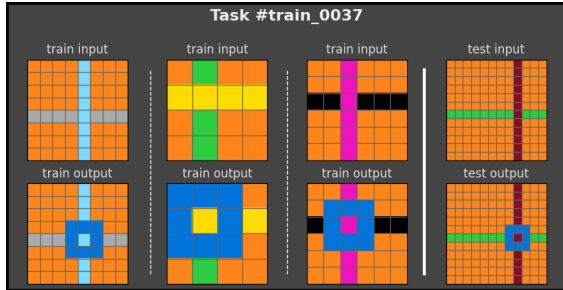
Figure 1 shows representative examples of ARC puzzles. These highlight the diverse range of tasks: from geometric symmetry and color manipulation (Fig. 1a, 1c) to pattern replication and counting-based inference (Fig. 1b, 1d). Such diversity makes rule induction highly non-trivial and often ambiguous, motivating both analytical heuristics and perhaps LLM-based reasoning strategies.



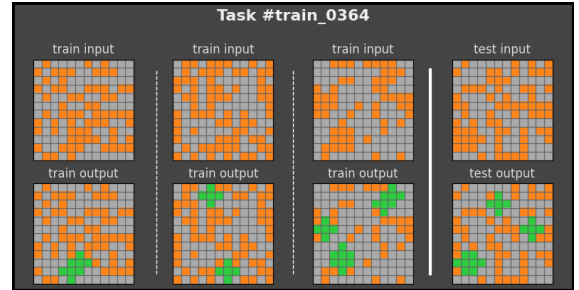
(a) Example 1



(b) Example 2



(c) Example 3



(d) Example 4

Figure 1: Examples of ARC visual puzzles from the training set. Each task consists of one or more input-output grid pairs used to infer a transformation rule, followed by a test input grid to apply that rule. Tasks vary in color schemes, layout complexity, and spatial logic.

3.1.2 Key Insights from EDA

ARC-AGI is inherently a visual reasoning challenge, where understanding spatial structures, color patterns, and transformations is essential. As such, having effective visualization tools is crucial for meaningful exploration and analysis. While a public desktop application is available, developed by the authors of the original competition⁴, it is not optimized for programmatic interaction or integration within experimentation pipelines. To streamline the workflow, I opted to build a custom visualization suite directly into the codebase, enabling rapid inspection, debugging, and comparative evaluation of model predictions in a notebook-friendly environment.

- Each task contains input-output grid pairs (up to 8 per sample), plus a test input
- Colors encoded as integers from 0 to 9 (see overall config in section 3.1.1); each grid of main size 1×1 and max size 30×30
- Background color 7 (orange) dominates the dataset; using 0 (black) for padding hurts accuracy
- Crucial design bias: test output **row count is provided** (and column count could be inferred from the sample submission), allowing constrained generation

Public (baseline) Score: 11.72% (only predicting a full orange grid with the inferred output shape)

⁴<https://github.com/fchollet/ARC-AGI/tree/master/apps>

3.2 Day 2: Heuristics and Baseline Solver

Building upon the results from the EDA stage, I implemented a fallback solver that:

- Infers output shape based on common patterns in input-output pairs
- Falls back to copying the input when no transformation is obvious
- Uses color frequency (mode) to pad or complete non-rectangular grids

Public Score: 31.33%

Quantized Inference Setup

```
import ast

from src.__init__ import logger
from src.config import Config

from mlx_lm import load, generate

import numpy as np

import re

import time
import torch

from transformers import AutoTokenizer, AutoModelForCausalLM, GenerationConfig, BitsAndBytesConfig
from typing import Tuple

from src.utils import infer_out_shape, get_grid_shape

class ARCModel:
    """
    A simple ARC solver.
    """
    def __init__(
        self,
        dtype:torch.dtype=Config.dtype,
        model_name:str=Config.model_name,
        system_prompt_path="base_prompt.txt",
        base_llm_name:str=Config.base_llm,
        verbose:bool=True,
        target_platform:str="mlx"
    ):
        self.model_name = model_name
        self.verbose = verbose
        self.target_platform = target_platform

        with open(system_prompt_path, "r") as f:
            self.base_prompt = f.read().strip()

    def solve(
        self,
        task,
        expected_rows:int=None,
```

```

        verbose:bool=False
    ):
        train = task["train"]
        test = task["test"][0]

        if "output" in test.keys():
            # adjust expected row num
            expected_rows = len(test["output"])

        inferred_out_shape = infer_out_shape(
            train_pairs=train,
            test_input=test["input"],
            expected_rows=expected_rows,
            verbose=verbose
        )

        out_grid = self.copy_input_solver(
            test["input"],
            inferred_out_shape,
            verbose=verbose
        )

        return out_grid

    def copy_input_solver(
        self,
        test_input,
        expected_shape=None,
        verbose:bool=False
    ):
        if verbose:
            logger.info("Fallback - Copy input solver...")

        output = [row[:] for row in test_input]

        if expected_shape:
            output = self.adjust_rows_and_columns(
                grid=output,
                expected_shape=expected_shape,
                verbose=verbose
            )

        if verbose:
            logger.info("Done!")

        return output

    def extract_out_grid(self, response):
        # Find all list-of-lists patterns (non-greedy, multiline)
        matches = re.findall(r"\[s*\[.*?\]\s*\]", response, re.DOTALL)
        output_grid = []

        if matches:
            try:
                output_grid = ast.literal_eval(matches[-1])

```

```

        except Exception as e:
            logger.error(e)
            raise e

    return output_grid

def pixel_accuracy(self, pred, target):
    try:
        pred_arr = np.array(pred)
        target_arr = np.array(target)

        if pred_arr.shape != target_arr.shape:
            return 0.0

        total = pred_arr.size
        match = np.sum(pred_arr == target_arr)
        return float(match / total) if total > 0 else 0.0
    except:
        return 0.0

def evaluate(self, task, predictions):
    assert "test" in task and all("output" in t for t in task["test"]), "Missing test outputs"

    correct = 0
    total = len(task["test"])
    pixel_scores = []

    for pred, test_case in zip(predictions, task["test"]):
        target = test_case["output"]

        if pred == target:
            correct += 1
            pixel_scores.append(1.0)
        else:
            match = self.pixel_accuracy(pred, target)
            pixel_scores.append(match)

    return {
        "exact_match": correct / total,
        "mean_pixel_accuracy": sum(pixel_scores) / total
    }

def adjust_rows_and_columns(
    self,
    grid,
    expected_shape,
    fill_value=Config.DEFAULT_BG_VALUE,
    verbose=bool=False
):
    if verbose:
        logger.info("Adjusting grid...")
        logger.info(f"{expected_shape=}")

    exp_rows, exp_cols = expected_shape
    adjusted = []

```

```

for i in range(exp_rows):
    if i < len(grid):
        row = grid[i][:exp_cols] # crop if needed
        if len(row) < exp_cols:
            row += [fill_value] * (exp_cols - len(row))
        else:
            row = [fill_value] * exp_cols # pad grid
        adjusted.append(row)

if verbose:
    final_shape = get_grid_shape(adjusted)
    logger.info(f"{final_shape=}")

return adjusted

```

3.3 Day 3–5: LLM Integration

I integrated large language models (LLMs) using structured few-shot prompts that describe the grid as a matrix and ask the model to return the transformed grid. Models used include:

- **Mistral-Nemo-Instruct (4bit/8bit)** — prompt compliant, fast, but limited on large inputs
- **Phi-4-mini-reasoning** — deep reasoning, but overly verbose
- **Qwen2.5-14B-Instruct** — highly performant especially on CUDA-enabled GPU, fairly accurate, and consistent

After a few trials, I kept **Qwen2.5-14B-Instruct** as my base LLM in this competition.

From a resource standpoint, the deployment of large language models (LLMs) was carefully optimized to fit within constrained hardware environments. For instance, the Qwen2.5-14B-Instruct model, when quantized to 8-bit, occupied approximately 15GB of memory on a MacBook Pro (M4) and around 21GB on an NVIDIA RTX 6000 GPU. This quantization step proved essential for enabling local inference, particularly on consumer-grade GPUs like the RTX 4090, which has a maximum of 24GB VRAM.

Despite this, the model struggled to process larger input grids (greater than 15×15) on the M4, often leading to crashes or memory exhaustion.

On the aforementioned GPU setup, a full pass over the test set typically required around two hours. Notably, all LLM experiments—including quantization, generation, and evaluation—were conducted within a budget of less than \$30, making the exploration both computationally and financially feasible for a short-term hackathon setting.

Quantized Inference Setup

```

from transformers import BitsAndBytesConfig

quantization_config = BitsAndBytesConfig(
    load_in_8bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype="float16",
    bnb_4bit_use_double_quant=True
)

llm = AutoModelForCausalLM.from_pretrained(
    base_llm_name,
    torch_dtype=Config.dtype,
    low_cpu_mem_usage=True,

```

```

device_map="auto" if torch.cuda.is_available() else None,
trust_remote_code = Config.trust_remote_code,
quantization_config=quantization_config
)

```

Best Score with LLM (pre-sealing of leaderboard): 28.84%

PS: A bug in grid parsing overwritten LLM predictions with background pixels. After fixing it, I noticed that the LLM-based implementation effectively achieved a score of **xx.xx%**, post competition.

3.4 Final Solution Architecture

- A shared `ARCModel` class handles heuristics, LLM calls, grid resizing, and evaluation
- Custom prompt defines rules for LLM task-solving using a few-shot approach
- Optional: Candidate outputs are validated using metrics (exact match, pixel accuracy)
- Final predictions are padded/cropped based on known row counts and submitted as CSV

4 Discussion & Possible Future Work

The approach detailed in this write-up demonstrates several key insights. First, incorporating task-specific priors—such as the known number of rows in the output grid and set a default background pixel value—can lead to substantial improvements over naive baselines. Second, simple and interpretable heuristics, when well-crafted, can outperform unstructured deep learning models that lack domain-specific guidance. Finally, leveraging LLMs through carefully designed few-shot prompts proves to be a competitive strategy, often yielding results on par with more traditional algorithmic and non ML-based approaches.

Despite limitations like compute budget, prompt length, and model constraints, the performance achieved with little tuning seems to confirm the power of careful problem design and domain awareness. Further experiments are required to fully validate this result.

If resources (time and compute availability) allow, I aim to:

- Explore multimodal architectures combining vision and LLM reasoning
- Train symbolic rule extractors from grid pairs to aid LLM planning
- Extend to the full ARC dataset and tackle the official ARC Prize benchmark
- Build a toolformer-style solver where the LLM can invoke utility functions for shape inference or color analysis mid-prompt

5 Acknowledgments

This project was developed in close collaboration with GPT-4o as a coding assistant. Its contributions included brainstorming experimental directions, assisting with iterative code completion and refactoring, and supporting the development of grid evaluation routines, prompt engineering, and strategy testing. The interactive workflow allowed for rapid prototyping and refinement, significantly accelerating the pace of experimentation, given the short-term setting.

I would like to express my gratitude to the organizers for setting up this challenge, which not only provided a stimulating and well-structured benchmark, but also rekindled my engagement with the Zindi platform after a long break. The competition offered a refreshing opportunity to explore abstract reasoning tasks in a focused, time-constrained setting.