ARC PRIZE 2025 RESEARCH-DRIVEN SOLVER DEVELOPMENT PROMPT

OVERVIEW AND MISSION

You are tasked with developing an AI systems to efficiently learn new skills and solve open-ended problems, rather than depend exclusively on systems trained with extensive datasets for the ARC Prize 2025 (Abstraction and Reasoning Corpus) competition on Kaggle. This is a research-driven development process where you will:

1. **Comprehensively review** all provided research documents and literature
2. **Synthesize** state-of-the-art approaches from the research
3. **Design** a novel or hybrid solution architecture based on your analysis
4. **Implement** the solution through iterative refinement
5. **Deliver** a production-ready Kaggle notebook (.py file)

Critical Principle: Your approach must be **derived from research**, not predetermined. You will analyze the literature, identify promising directions, and make informed architectural decisions based on empirical evidence and theoretical foundations from the research community.

EXECUTION MODE: This is a single-pass, complete workflow. Execute all steps sequentially without pausing. For each step, produce the specified deliverables in full before proceeding to the next step.
KAGGLE ENVIRONMENT:

- Your .py file is the solution code (what you submit to Kaggle)

- Input data path: `/kaggle/input/arc-prize-2025/arc-agi_test_challenges.json`

- Your code must write: `submission.json` in current directory

- Do NOT reference training/evaluation/solution files in the final .py code

- The .py code must be self-contained and work with only the test_challenges.json file

STEP 1: COMPREHENSIVE RESEARCH ANALYSIS AND PROBLEM UNDERSTANDING

Prompt:

```
Conduct a thorough analysis of the ARC Prize 2025 challenge by synthesizing
all available resources in this directory.e.g everything in
/home/legend/Documents/AGI/Kaggle, and sub folders like
/home/legend/Documents/AGI/Kaggle/literature:


### A. Document Review and Synthesis
```

1. **Primary Problem Documents Analysis:**

   – Read ALL provided files about the ARC Prize 2025

   - Extract the exact problem specification:

     * Task structure (JSON format, train/test pairs)

     * Grid constraints (dimensions, color values)

     * Submission format requirements (pass@2, attempt_1/attempt_2)

     * Scoring methodology (exact matching)

   - Identify competition constraints:

     * Computational limits (12-hour runtime, no internet access)

     * Hardware specifications (GPU availability, memory)

     * Kaggle-specific requirements (file paths, deterministic execution)

2. **Existing Research Literature Review:**

   - Analyze the solutions so far and research documents provided (e.g., "/home/legend/Documents/AGI/Kaggle/literature/papers, /home/legend/Documents/AGI/Kaggle/arc_prize_2025_submission and sun folders, /home/legend/Documents/AGI/Kaggle/literature, any paper, code, approach, etc)

   - Extract key methodologies mentioned: e.g

     * Domain-Specific Language (DSL) approaches

     * Program synthesis techniques

     * LLM-augmented methods

     * Object-centric neuro-symbolic approaches

   - Identify what has worked historically:

     * Performance benchmarks from literature and online searches

     * Winning solutions from previous competitions

     * Theoretical foundations cited

3. **Online Research Supplementation:**

   Use web_search to gather:

   - Recent papers on ARC solving (2023-2025)

   - Kaggle discussion forums for ARC Prize 2025

   - Current leaderboard standings and publicly shared approaches

- GitHub repositories of successful ARC solvers
- Blog posts and technical write-ups from competition participants

B. Technical Problem Decomposition

Based on your research, answer:

1. **What makes ARC tasks difficult?**
   - Core cognitive requirements
   - Why traditional ML/LLMs fail
   - Specific reasoning challenges

2. **What are the latest best proven solution paradigms?**
   - Categorize approaches from literature (e.g symbolic, neural, hybrid etc)
   - Identify success rates and limitations of each
   - Note computational trade-offs

3. **What are the key design decisions?**
   - DSL expressiveness vs. search space size
   - Search strategy (e.g breadth-first, beam search, Monte Carlo etc)
   - Verification mechanisms
   - Heuristic design
   - Two-attempt strategy formulations

C. Comparative Analysis

Create a detailed comparison table:

- List 5-7 major approaches from literature
- For each: core principle, performance, computational cost, implementation complexity
- Identify gaps or opportunities for hybrid approaches

DELIVERABLE:

Research synthesis document (800-1000 words) including:

- Problem specification summary (200 words)

- Key findings from literature (300 words)

- Comparative analysis table (3-5 approaches)

- Preliminary architectural recommendation (200 words)

STEP 2: ARCHITECTURAL DESIGN AND APPROACH SELECTION

Prompt:

Based on your research synthesis, design a solver architecture:

A. Architecture Selection Rationale example: Use better one if you find a better one in your research

1. **Choose Your Core Approach:**

   Justify your selection using research evidence:

   - Will you use pure DSL-based program synthesis?

   - Hybrid neuro-symbolic?

   - LLM-augmented synthesis?

   - Novel combination?

   For your choice, provide:

   - Supporting evidence from literature (cite specific papers/results)

   - Why this approach suits ARC Prize 2025 constraints

   - Expected performance range based on benchmarks

   - Known limitations and mitigation strategies

2. **Component Architecture Design:**

   Define the major system components:

   - **Perception/Analysis Layer**: How will you analyze training pairs?

   - **Reasoning/Synthesis Layer**: How will you generate candidate solutions?

- **Verification Layer**: How will you validate candidates?
- **Execution Layer**: How will you apply solutions to test inputs?

For each component:
- Technical approach (specific algorithms/methods)
- Research citations supporting the approach
- Expected computational complexity
- Failure modes and handling strategies

3. **Two-Attempt Strategy Design:**
   Design how you'll generate diverse attempts:
   - What defines "diversity" in your approach?
   - How will attempt_1 and attempt_2 differ?
   - Research evidence for ensemble/multi-strategy benefits

B. Implementation Specification

Create detailed specifications WITHOUT writing code:

1. **DSL Design (if applicable):**
   - List of primitive operations (justified by task analysis)
   - Operation categories (geometric, color, structural, etc.)
   - Parameter spaces for each operation
   - Composability rules

2. **Search Strategy:**
   - Algorithm choice (beam search, A*, MCTS, etc.)
   - Search space pruning heuristics
   - Stopping criteria
   - Computational budgets per task

3. **Heuristic Design:**

   - Feature extraction from training pairs

   - How features guide search

   - Priority ordering mechanisms

   - Adaptation strategies

4. **Verification Mechanism:**

   - Exact matching on training pairs

   - Handling of ambiguous cases

   - Confidence scoring (if applicable)

C. Theoretical Analysis

1. **Completeness Analysis:**

   - What percentage of ARC tasks is your approach theoretically capable of solving?

   - What categories of tasks will it struggle with?

2. **Computational Feasibility:**

   - Estimated time per task

   - Scalability to 100+ tasks

   - Memory requirements

3. **Expected Performance:**

   - Predicted accuracy range based on literature benchmarks

   - Comparison to current leaderboard (from online research)

   - Conservative vs. optimistic estimates

DELIVERABLE:

Architecture design document (600-800 words):

- Approach selection rationale with citations (300 words)

- Component specifications (200 words)

- Theoretical performance analysis (150 words)

STEP 3: ITERATION 1 - INITIAL IMPLEMENTATION

Prompt:

Implement your first version of the solver based on your architectural design:

A. Implementation

Write a complete, production-ready Python file that:

1. Implements ALL components from your architecture

2. Loads data from Kaggle paths: `/kaggle/input/arc-prize-2025/arc-agi_test_challenges.json`

Guve kaggles notebook starter as provided by Kaggle {# This Python 3 environment comes with many helpful analytics libraries installed

```
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python

# For example, here's several helpful packages to load


import numpy as np # linear algebra

import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)


# Input data files are available in the read-only "../input/" directory

# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory


import os

for dirname, _, filenames in os.walk('/kaggle/input'):

    for filename in filenames:

        print(os.path.join(dirname, filename))


# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a version using "Save & Run All"
```

You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session}

3. Generates valid `submission.json` with correct format

4. Handles all edge cases gracefully

5. Runs deterministically (set seeds)

6. Includes clear code structure and comments

Implementation Guidelines:

- Follow your architecture specification exactly

- Implement complete functionality (no stubs or TODOs)

- Use only libraries available on Kaggle (numpy, scipy, no internet-dependent packages)

- Include error handling and fallback mechanisms

- Ensure computational efficiency where possible

B. Theoretical Correctness Analysis

Conduct focused self-review (300-400 words total):

- Logic verification: Walk through 2-3 example tasks mentally

- Compliance check: Verify all Kaggle requirements

- Critical assessment: 3 key strengths, 3 key weaknesses, performance estimate

DELIVERABLE:

- `arc_solver_v1.py`

- Self-assessment (300-400 words)

2. **Mental Task Walkthrough:**

   Select 3 example task types (from your understanding of ARC):

   - Simple pattern (e.g., color remapping)

   - Geometric transformation (e.g., rotation + scaling)

   - Complex compositional (e.g., multi-rule interaction)

For each, trace through your code mentally:

- Would it extract correct features?
- Would it generate appropriate candidates?
- Would verification work correctly?
- Would it produce valid output?

3. **Research Benchmark Comparison:**
   - Compare your approach to methods in literature
   - Identify where your implementation might underperform
   - Identify potential advantages

C. Compliance and Feasibility Review

1. **Competition Requirements:**
   - [ ] Correct input file path
   - [ ] Correct submission.json format
   - [ ] Two attempts per test input
   - [ ] All tasks covered (no missing task_ids)
   - [ ] Grids are valid (0-9 values, reasonable dimensions)
   - [ ] No internet access required
   - [ ] Deterministic execution

2. **Computational Complexity Analysis:**
   - Estimate runtime per task (worst case, average case)
   - Calculate total expected runtime for 100 tasks
   - Identify computational bottlenecks
   - Assess if it fits in 12-hour limit

3. **Code Quality:**
   - Readability and maintainability
   - Proper error handling

   - Modularity and extensibility

D. Critical Self-Assessment

Write an honest critique:

- **Strengths**: What does this implementation do well?

- **Weaknesses**: Where is it likely to fail?

- **Theoretical Performance**: Estimate accuracy (e.g., "5-15%")

- **Comparison to SOTA**: How far from competitive solutions?

- **Key Improvements Needed**: Prioritized list

DELIVERABLE:

- `arc_solver_v1.py` (complete, runnable file)

- Self-assessment document (800-1200 words)

- Identified improvements for next iteration

STEP 4: ITERATION 2 - REFINEMENT AND ENHANCEMENT

Prompt:

Refine your solver based on V1 analysis and additional research:

A. Gap Analysis and Research

1. **Review V1 Weaknesses:**

   - Identify the top 3-5 critical issues from V1 assessment

   - For each issue, conduct targeted research:

     * Use web_search for specific solutions (e.g., "ARC task color symmetry detection")

     * Look for papers addressing these specific challenges

     * Search Kaggle discussions for practical solutions

2. **Enhancement Opportunities:**
   - Research advanced techniques not in V1:
     * More sophisticated search strategies
     * Better heuristics
     * Additional DSL operations
     * Improved verification methods
   - Find specific algorithmic improvements from recent work

B. Implementation V2

Create an improved version:

1. **Address Critical Issues:**
   - Fix logical errors from V1
   - Improve components that were weak
   - Add missing functionality

2. **Add Enhancements:**
   - Implement research-backed improvements
   - Expand DSL (if applicable) with new operations
   - Improve search efficiency
   - Better feature extraction

3. **Maintain Strengths:**
   - Keep working components from V1
   - Preserve successful design decisions
   - Ensure compliance still met

C. Advanced Evaluation

1. **Cross-Validation Against Research:**

   - Compare V2 approach to 3-5 recent papers
   - Identify which techniques from literature you've incorporated
   - Note which proven techniques you're still missing

2. **Theoretical Task Coverage:**
   - Categorize ARC task types (from problem understanding)
   - For each category, assess: Can V2 solve it?
   - Estimate coverage: "V2 should handle X% of Y-type tasks"

3. **Computational Profile:**
   - Re-analyze runtime complexity
   - Compare to V1: Is it faster/slower?
   - Is it still within 12-hour budget?

4. **Online Benchmarking:**
   - Use web_search to find:
     * Current Kaggle leaderboard scores
     * Discussion of what scores are competitive
     * Public solution approaches and their performance
   - Compare V2's expected performance to these benchmarks

D. Critical Assessment V2

1. **Progress Evaluation:**
   - Quantify improvements over V1
   - Justify why changes should help
   - Estimate new performance range

2. **Remaining Gaps:**
   - What still doesn't work?
   - What tasks will V2 still fail on?

   - Why?

3. **Path to V3:**

   - Identify THE most impactful improvement for V3

   - Research backing for this improvement

   - Implementation strategy

DELIVERABLE:

- `arc_solver_v2.py`

- V1→V2 improvement summary (400-500 words)

- Performance estimate with justifications

STEP 5: ITERATION 3 - OPTIMIZATION AND ROBUSTNESS

Prompt:

Create the most robust and competitive version before finalization:

A. State-of-the-Art Alignment Research

1. **Deep Dive into Top Approaches:**

   - Use web_search to find:

     * Recent competition winners' approaches (2024-2025)

     * Highest-performing published methods

     * Novel techniques from latest papers

   - Extract specific implementation details

   - Identify "secret sauce" components

2. **Competitive Gap Analysis:**

   - Current leaderboard scores (from online research e.g https://www.kaggle.com/competitions/arc-prize-2025/leaderboard)

   - Your V2 estimated performance

   - Gap size and components causing the gap
   - Feasibility of closing gap with V3

B. Implementation V3

Final pre-production version:

1. **Incorporate SOTA Techniques:**
   - Add most promising technique from research
   - Implement sophisticated optimizations
   - Enhance robustness and edge case handling

2. **Optimization Pass:**
   - Profile computational bottlenecks (theoretically)
   - Implement caching where beneficial
   - Optimize critical paths
   - Balance quality vs. speed

3. **Robustness Hardening:**
   - Comprehensive error handling
   - Graceful degradation for hard tasks
   - Ensure NO crashes regardless of input
   - Validate all outputs before submission

C. Comprehensive Evaluation

1. **Multi-Dimensional Analysis:**

   Correctness:
   - Mental walkthrough of 5+ diverse task types
   - Verify logic for each component

   - Check for subtle bugs or edge cases

   Compliance:
   - Re-verify all Kaggle requirements
   - Check submission format rigorously
   - Confirm deterministic behavior

   Performance:
   - Estimate accuracy on:
     * Simple tasks (pattern matching)
     * Medium tasks (2-3 operation compositions)
     * Complex tasks (4+ operations, interactions)
   - Overall accuracy estimate

   Efficiency:
   - Task time distribution (fast/slow tasks)
   - Total expected runtime
   - Memory usage assessment

2. **Research-Based Validation:**
   - Compare V3 to each major approach from Step 1 literature review
   - For each: What techniques did you adopt? What did you skip? Why?
   - Identify your solver's unique contributions or combinations

3. **Online Competitive Analysis:**
   - Latest Kaggle leaderboard standings
   - Discussion forum insights
   - Comparison: Where does V3 likely rank?
   - What would push it higher?

D. Failure Mode Analysis

1. **Identify Task Categories V3 Will Fail:**

   - Specific examples from ARC task taxonomy

   - Why will it fail? (insufficient DSL, search timeout, etc.)

   - Could these be addressed? At what cost?

2. **Known Limitations:**

   - Computational constraints causing shortcuts

   - Theoretical limitations of approach

   - Trade-offs made and their implications

3. **Risk Assessment:**

   - Probability of submission errors

   - Probability of timeout

   - Expected score variance (best/worst case)

E. V3 Critical Assessment

Final honest evaluation:

- **Architecture Quality**: Is the design sound?
- **Implementation Quality**: Is the code production-ready?
- **Expected Performance**: Realistic accuracy estimate with reasoning
- **Competitive Standing**: Where in leaderboard range?
- **Confidence Level**: High/Medium/Low and why
- **Key Strengths**: Top 3 advantages
- **Key Weaknesses**: Top 3 limitations

DELIVERABLE:

- `arc_solver_v2.py`
- V1→V2 improvement summary (400-500 words)
- Performance estimate with justification

STEP 6: FINAL VERSION - PRODUCTION IMPLEMENTATION

Prompt:

```
Create the final, production-ready solver for Kaggle submission:


### A. Final Implementation Decision


Based on V3 assessment:

1. **If V3 is strong**: Polish and finalize it

2. **If critical issue found**: Make targeted fix, creating V3.1

3. **If major rework needed**: Explain what and why, then implement


### B. Production-Ready Code


Create `arc_solver_final.py`:


1. **Code Quality:**

   - Clean, well-commented code

   - Proper structure and organization

   - Professional naming conventions

   - Comprehensive docstrings


2. **Robustness:**

   - Handles ALL edge cases

   - Never crashes or hangs

   - Always produces valid submission

   - Graceful degradation for hard tasks


3. **Kaggle Optimization:**

   - Efficient imports (only necessary libraries)
```

   - Proper file paths for Kaggle environment
   - Deterministic with fixed random seeds
   - Appropriate logging/progress indicators

4. **Submission Guarantee:**
   - Always generates valid submission.json
   - Correct format for every task
   - Two attempts per test input
   - No missing or malformed entries

C. Final Validation

1. **Mental Execution Test:**
   - Trace through the complete pipeline
   - Verify data flow from input to submission
   - Check all branches and conditions
   - Confirm error handling paths work

2. **Requirements Checklist:**
   - [ ] Correct Kaggle input path
   - [ ] Valid submission.json output
   - [ ] Pass@2 format (two attempts)
   - [ ] No internet dependencies
   - [ ] Deterministic execution
   - [ ] Runs within time limit (estimated)
   - [ ] Handles all 100+ tasks
   - [ ] Values in [0-9] range
   - [ ] Grid dimensions valid (1x1 to 30x30)

3. **Final Complexity Analysis:**
   - Best case runtime

- Average case runtime
   - Worst case runtime
   - Confidence in 12-hour completion

D. Documentation Package

Create concise documentation:

1. **Approach Evolution** (600 words):
   - Research foundation and key decisions (250 words)
   - Version progression V1→V2→V3→Final (250 words)
   - Implementation highlights (100 words)

2. **Final Approach Explanation** (600 words):
   - Architecture and core components (250 words)
   - Algorithmic strategy and two-attempt design (200 words)
   - Expected performance and limitations (150 words)

DELIVERABLE:
- `arc_solver_final.py`
- `approach_documentation.md` (both sections combined, ~1200 words total)

STEP 7: FINAL RESEARCH REFLECTION AND SUBMISSION PACKAGE

Prompt:

STEP 7: FINAL PACKAGE

Organize all deliverables with brief summary:
- Final performance prediction (confidence intervals)
- List of all 4 Python files produced
- 200-word reflection on research-to-implementation journey

DELIVERABLE:

- Summary document (3000 words)