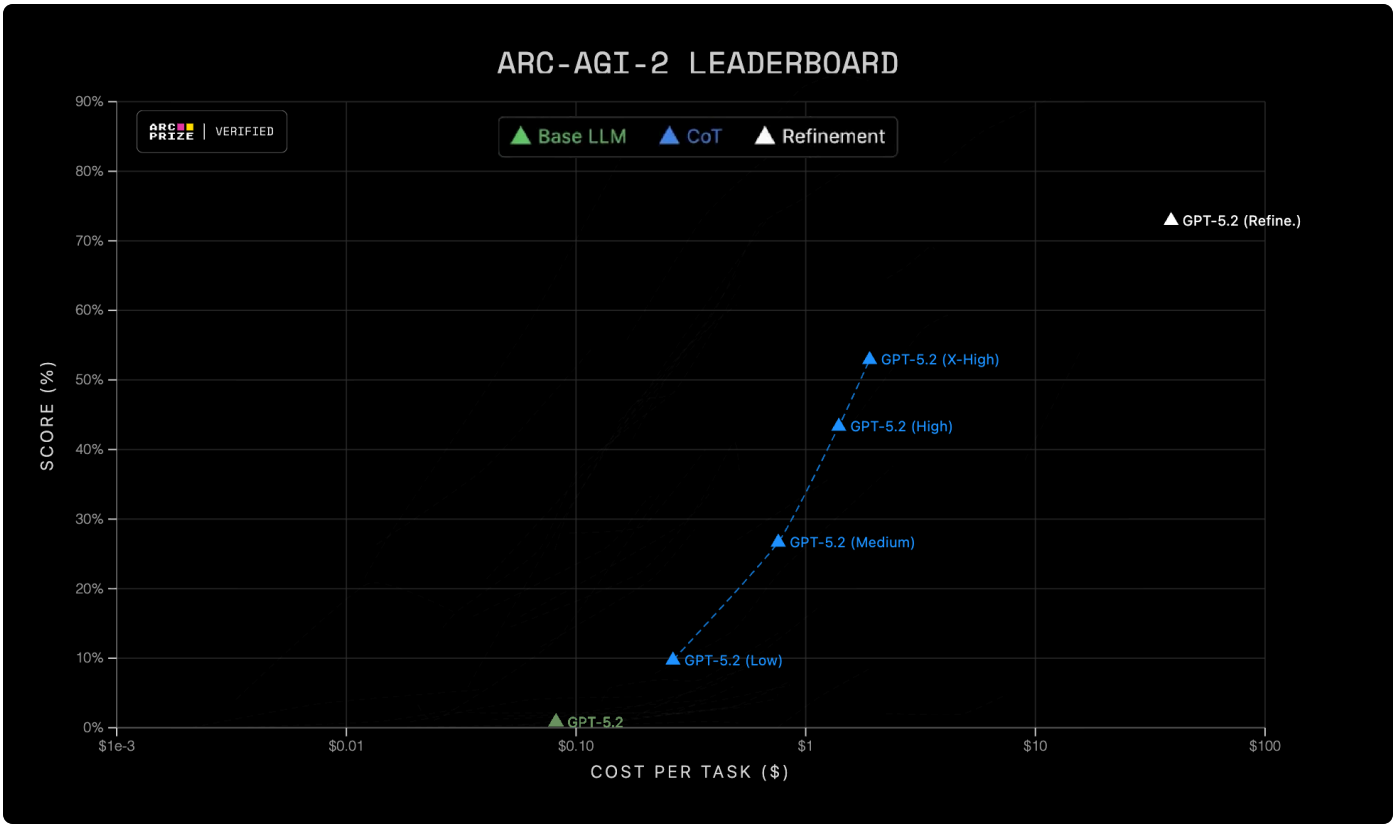
SotA ARC-AGI-2 Results with REPL Agents

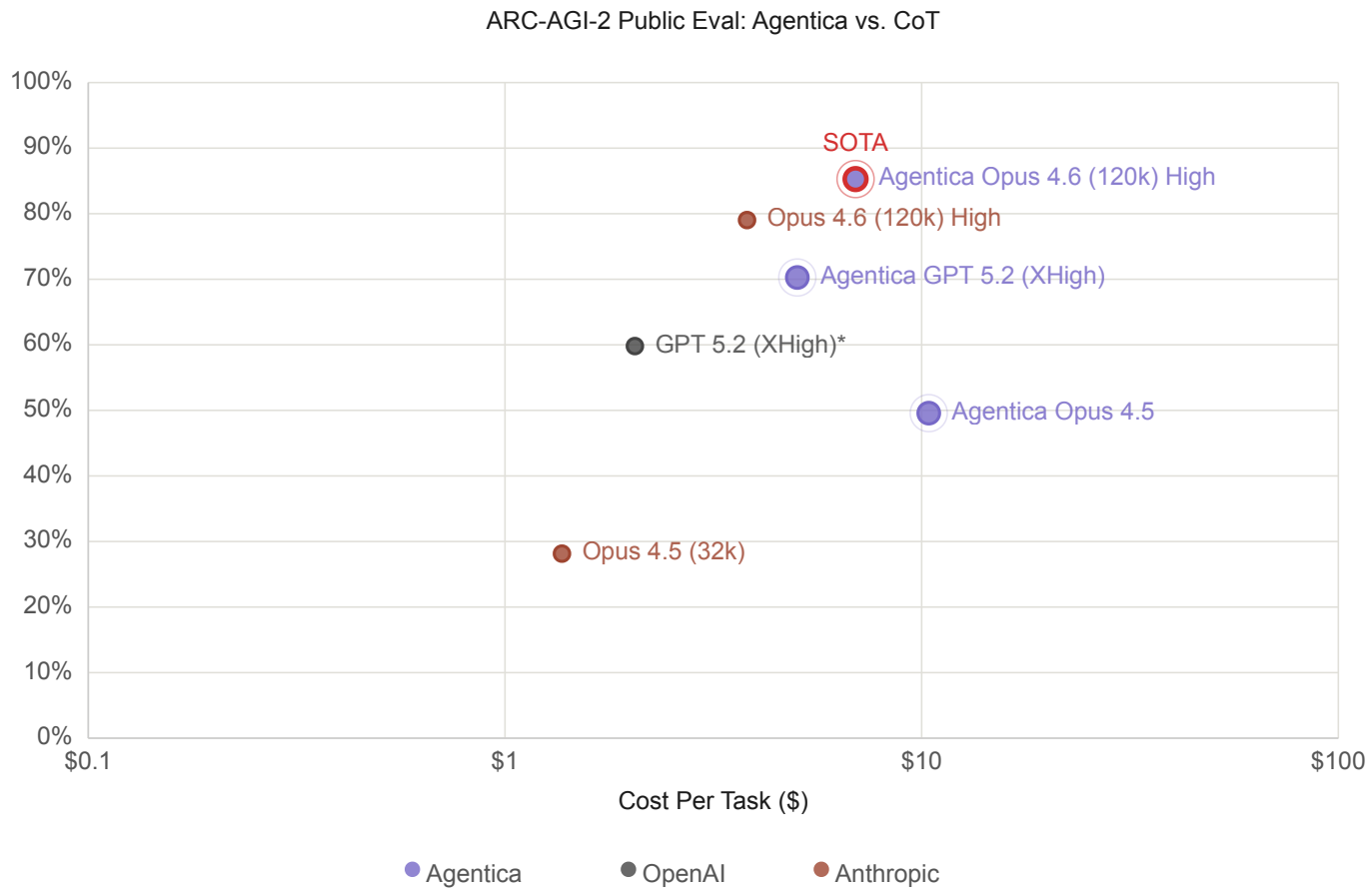
symbolica.ai/blog/arcgentica



The Agentica framework by Symbolica improves ARC-AGI-2 performance across three frontier models. The approach builds on code-mode agents [] and Recursive Language Models (RLMs) [].

Our implementation achieves a score of **85.28% with Opus 4.6 (120k) High** and increase the scores of GPT 5.2 (XHigh) and Opus 4.5 by 10 and 20 percentage points respectively. The agent is 350 lines of Python and uses the Agentica framework.

Check out the code on **GitHub** symbolica-ai/arcgentica



Submission

Public Eval Score (%)

Cost ($/task)

Agentica Opus 4.6 (120k) High

85.28

6.94

Opus 4.6 (120k) High

79.03

3.81

Agentica GPT 5.2 (XHigh)

70.27

5.03

GPT 5.2 (XHigh)*

59.81

2.05

Agentica Opus 4.5

49.58

10.40

Opus 4.5 (32k)

28.15

1.37

Figure 1. A comparison of the score and cost per task on the ARC-AGI-2 public eval set between Chain of Thought (CoT) models and an Agentica agent for Opus 4.6 (120k) High, GPT 5.2 (XHigh) and Opus 4.5. The cost per task for Agentica Opus 4.6 (120k) High and Agentica Opus 4.5 assume a cache hit rate of 85% on all input tokens.
*The public eval score for OpenAI GPT 5.2 (XHigh) is for a subset of 107 problems, with a score of 53.33% on the full 120 problems.

Agents and ARC-AGI

ARC-AGI

ARC-AGI was introduced in 2019 as a test of fluid intelligence [][] and has since become a primary reasoning benchmark in the AI community, with its second iteration ARC-AGI-2 released in 2025 []. Submissions are categorized into base LLM models, chain-of-thought (CoT) , and refinements that enhance base model outputs through custom harnesses []. Agentica's ARC-AGI agent falls into the latter category.

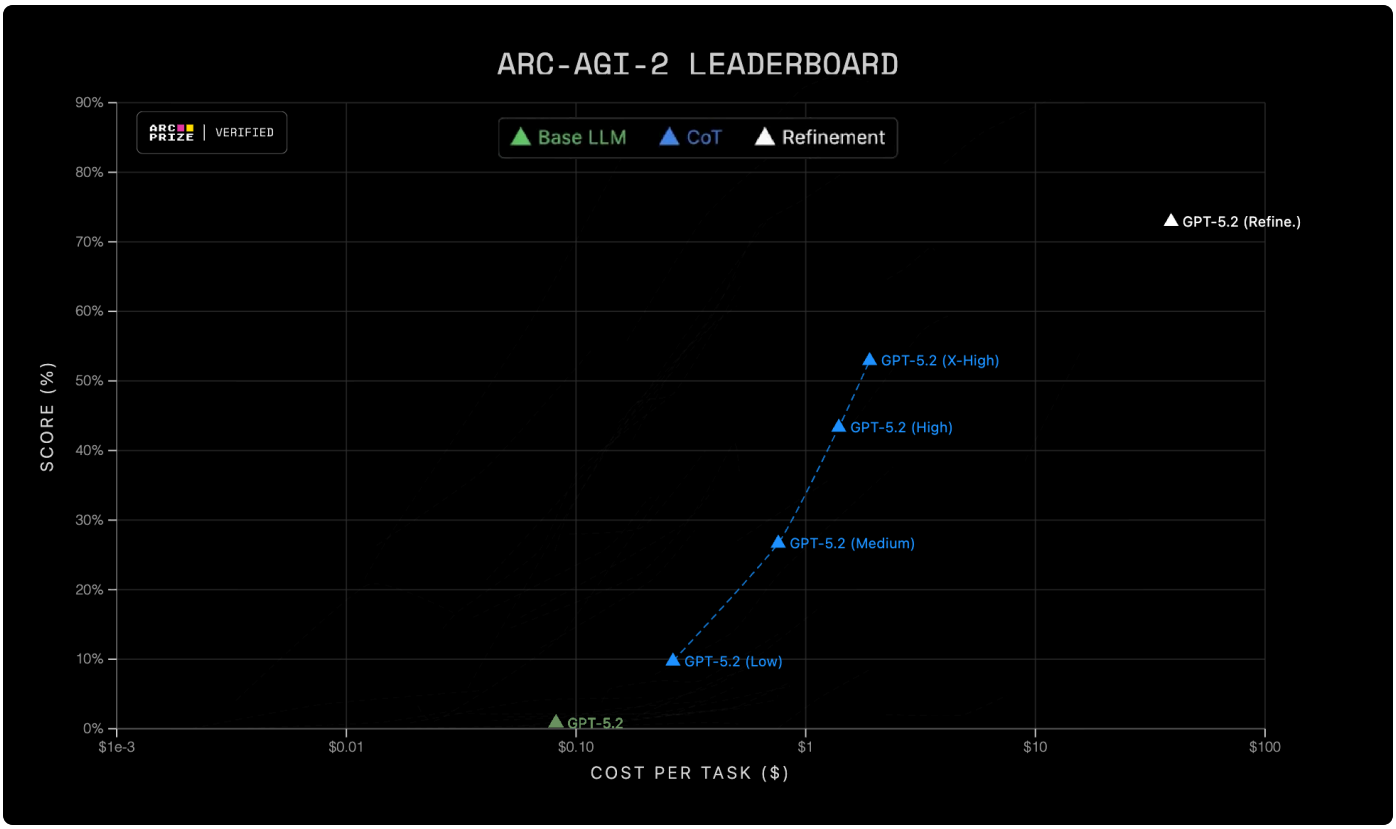


Figure 2. Comparison of the ARC-AGI-2 score between the base model, CoT, and a refinement approach [7] for GPT 5.2.

Initially, base models scored in the low single digits on ARC-AGI-2. Chain-of-thought prompting pushed scores substantially higher (). More recently, refinement approaches have driven further gains, turning ARC-AGI-2 into a testing ground for new harnesses and agentic loops [].

The agentic frontier

ARC-AGI tasks can be framed as a program synthesis problem: given an example set of training inputs and outputs, infer a program that correctly predicts the outputs for a set of test inputs.

Early LLM-based approaches exploited this structure by pairing chain-of-thought [] reasoning with code generation []. Encouraging models to articulate intermediate reasoning before committing to an implementation led to more coherent hypotheses and more structured candidate programs, pushing performance beyond low single-shot baselines.

Now, leading submissions leverage agentic workflows: solving ARC tasks is an inherently iterative process that aligns well with the agentic structure. An agent can propose a rule in the form of a Python program, implement it as executable code, run it against training examples, and use any mismatches as concrete feedback for revision [][][]. This loop is fast to evaluate, easy to automate, and supports exploring multiple candidate solutions.

Limitations

Most refinement harnesses encode domain-specific assumptions in the strategy itself, whether that be natural language feedback in an outer loop or the inclusion of multi-modality. While the performance of agents often depends on the underlying model, mixtures of domain-specific agents introduce additional dependencies. Applying such strategies to reasoning tasks outside their intended domain is often non-trivial or ineffective.

In the majority of harnesses, the control flow is fixed from the start. Some approaches choose depth, allowing the underlying model access to a persistent environment that accumulates state []. This relies on the reasoning or interleaved thinking (reasoning while calling tools) of the underlying model. Others choose width, employing parallel sampling of the same or various agents, aggregating results at the end [][]. In both cases, agents lack the autonomy to dynamically choose one or the other.

Agentica and ARC-AGI

Agentica is a framework for building agents with a persistent Python REPL. It provides a stateful workspace where agents can run code, keep objects in memory, and call user-defined Python objects and tools via code.

This persistent REPL allows agents to:

- Use tools in a more complex manner than schema-based tool calls allow
- Keep and mutate objects in memory
- Interleave reasoning and execution in one persistent workspace

In practice, this improves performance on long-context tasks. As is standard, types are enforced in the REPL, avoiding compounding errors throughout prolonged reasoning.

User-defined objects are virtualized in the agent's REPL and, from the agent's perspective, they are native Python objects that it has access to. This also allows agents to return objects to the user or pass objects into the REPL of another agent. Consequently, the framework supports building both code mode agents and RLMs [].

Addressing limitations

As mentioned above, a stateful execution environment allows agents to practice interleaved thinking, reasoning while intermittently executing code. The RLM capabilities of the framework mean agents can also autonomously delegate to other agents and pass on any existing state. This enables both depth and width: an agent might explore several hypotheses in parallel, or check whether its proposed solution to an ARC-AGI task generalizes from the training inputs to test inputs. This is a key difference compared to other implementations of stateful code-based solvers []. In theory, this strategy is not domain-specific and is generalizable to any complex reasoning task.

ARC-AGI Agent

ARC AGI agent REPL scope (truncated)

```
@dataclass class FinalSolution: transform_code: str explanation: str def
soft_accuracy(pred: Output, truth: Example) -> float: ... def call_agent(task:
str, return_type: type[Any], **objects: Any,): ...
```

Figure 4. Truncated stubs of some of the objects virtualized in the ARC-AGI agent's Python REPL and the function used to spawn and call the agent. See `arc agent/` for details.
The objects listed in `scope` in are virtualized in the agent's REPL as native Python types and functions. The agent is constrained to output a `FinalSolution` object via the `return_type` parameter on the initial call. This contains the code for a Python function that solves the particular task and an accompanying explanation of how it does so. Task-dependent training examples and test inputs are also passed into the agent's `scope` on the initial call via the `objects` parameter.

Recursive delegation

Recursive delegation in the style of RLMs is a key differentiating capability. An agent can spawn sub-agents for specific subtasks and pass only the relevant state into their REPL, as needed. This distributes context across sub-agents, avoiding context rot.

This also allows agents to explore multiple hypotheses in parallel and delegate the analysis of individual examples to other agents. Rather than hard-coding a branching strategy for the whole domain, an agent can dynamically decide whether delegation is necessary based on the task.

As shown in , the framework allows for recursive delegation since `call_agent` can be added to the agent's `scope`.

Further reading:

- Full ARC-AGI agent implementation: arcgentica/arc_agent/agent.py
- Agentica documentation: Quickstart and How it works.

Results

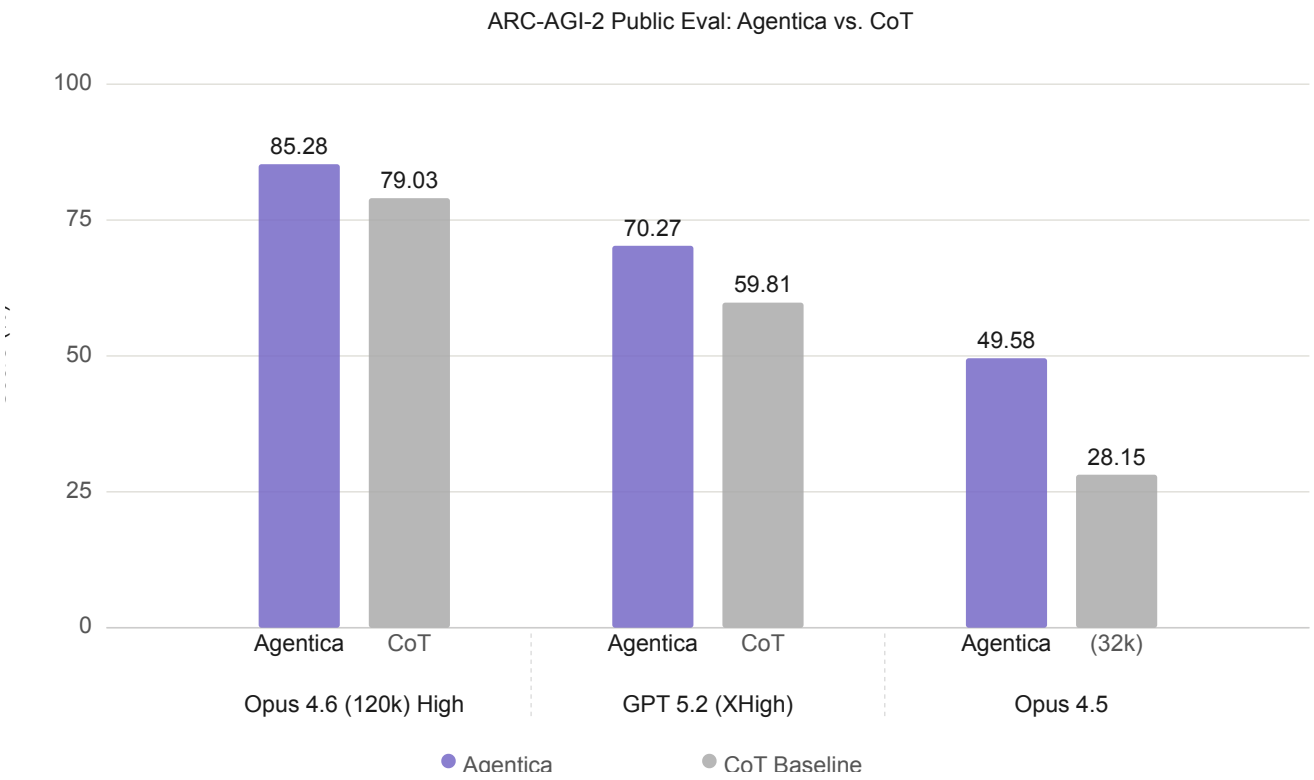


Figure 5. A comparison of the ARC-AGI-2 public eval score between Chain of Thought (CoT) baselines and an Agentica agent for Opus 4.6 (120k) High, GPT 5.2 (XHigh) and Opus 4.5. *The public eval score for OpenAI GPT 5.2 (XHigh) CoT is for a subset of 107 problems, with a score of 53.33% on the full 120 problems.
compares the performance of our implementation for Opus 4.6, GPT 5.2 (XHigh) and Opus 4.5 with their CoT analogues on the ARC-AGI-2 public evaluation set. All results for the ARC-AGI Agentica agent were run with a maximum of 9 recursively spawned sub-agents (excluding the initial agent) per attempt per task, and two attempts overall per task on all 120 tasks.

Opus 4.6 (120k) High

Our implementation achieved a score of 85.28% at $6.94 per task compared to 79.03% at $3.81 per task. The average number of agents used per task attempt was 2.6.

The result was run with Anthropic as the inference provider using their Messages API, with thinking effort set to `high`, `max_tokens` set to 128k and `cache_ttl` set to `1hr`. A few errors occurred due to invalid prompt exceptions. For each attempt, the number of retries due to framework or API-related errors was set to 3.

Agentica GPT 5.2 (XHigh)

Our implementation achieved a score of 70.27% at $5.03 per task compared to 59.81% at $2.05 per task. Note that the score of 59.81% for GPT 5.2 (XHigh) was calculated on a 107-problem subset, decreasing to 53.33% on all 120 problems. This score is comparable to the 68.33% of Chakravorty et al. [], who created a bespoke Python REPL harness, without the option for the recursive spawning of agents. It is also comparable to the 72.9% score of Land [] on the ARC-AGI-2 semi-private eval set, with a cost of $38.99 per task.

All experiments with GPT 5.2 (XHigh) were run with OpenAI as the inference provider using their Responses API, with reasoning set to `xhigh`. Caching was handled internally by OpenAI. Errors included invalid prompt exceptions from the OpenAI Responses API. For each attempt, the number of retries due to framework or API-related errors was set to 1.

Opus 4.5

Our implementation achieved a score of 49.58% at $10.40 per task compared to 28.15% at $1.37 per task. The average number of recursively spawned sub-agents per task attempt was 1.4, including the initial agent itself.

The result was run with OpenRouter as the inference provider using their Chat Completions API, with reasoning set to `high`, for which OpenRouter calculates the thinking token budget as `budget_tokens = max(min(max_tokens * 0.8, 128000), 1024)` [], where `max_tokens` was omitted in the request to OpenRouter. Costs exclude cache writes and assume a cache hit rate of 85% for input tokens, since caching was not enabled. For each attempt, the number of retries due to framework or API-related errors was set to 1.

Conclusion

ARC-AGI agents built with the Agentica framework improved the ARC-AGI-2 scores of the frontier reasoning models GPT 5.2 (XHigh), Opus 4.6 (120k) High, and Opus 4.5 by at least 6 percentage points, for a cost increase of at most $9.03 per task. The implementation achieved a state-of-the-art score of 85.28% for $6.94 per task with Opus 4.6 (120k) High. This project aims to be a first step in tackling reasoning problems using a domain-agnostic strategy, allowing agents access to a stateful REPL with which they can **dynamically and autonomously** explore each domain with both depth and width.

Check out the code on **GitHub** symbolica-ai/arcgentica

Acknowledgements

This article was written by Victoria Klein. Thanks to Charlie Lidbury, Terence Le Huu Phuong, Nick Kouris and Jonathan Frydman for their help on this project and consistent feedback on this post. Thanks also to Anthropic and OpenAI for the donation of credits.

Call for contributions

The Agentica framework by Symbolica is open-source and MIT licensed. It includes a Python SDK, a TypeScript SDK and a Server. Contributions to the project are welcome and encouraged.

Check out the repos **GitHub** symbolica-ai and join the Discord **Symbolica**.

References

[1] Symbolica. Beyond Code Mode: Agentica. Symbolica Blog.

[2] Alex L. Zhang *et al.* Recursive Language Models. arXiv.

[3] ARC Prize. ARC Prize. ARC Prize.

[4] François Chollet. On the Measure of Intelligence. arXiv.

[5] ARC Prize. ARC-AGI-2. arXiv.

[6] ARC Prize. ARC Prize Policy. ARC Prize.

[7] Johan Land. Johan Land Solver. Kaggle.

[8] ARC Prize. ARC Prize Leaderboard. ARC Prize.

[9] Jeremy Berman. How I Got a Record 53.6% on ARC-AGI. Substack.

[10] CT Pang. ARC-AGI-2 SOTA Efficient Evolutionary. Substack.

[11] Poetiq AI. Poetiq AI ARC-AGI Solver. GitHub.

[12] Chakravorty et al. Agentic Coding for ARC-AGI. Pivotools Blog.

[13] OpenRouter. Reasoning Tokens. OpenRouter Documentation.

[14] OpenAI. Responses API Benefits. OpenAI Documentation.

[15] Anthropic. Claude Models Migration Guide. Anthropic Documentation.

Appendix

A note on API endpoints and providers

As is the case with most model providers, it is expected that using the most recent API endpoint per provider yields the best results. OpenAI describes numerous benefits of using their Responses API []. Likewise, the adaptive thinking parameter, reccomended for Opus 4.6, is available via Anthropic's Messages API []. For this reason, experiments with GPT 5.2 used the OpenAI Responses API directly and those with Opus 4.6 used the Anthropic Messages API directly, supported by the Agentica framework as of v0.4.0. However, experiments with Claude Opus 4.5 used the Chat Completions API via OpenRouter since Anthropic Messages API support was not yet available.

A note on pass@2 and program synthesis

For ARC-AGI-1 and ARC-AGI-2, the score per task is calculated as the normalized pass@2 for all test output grids. Given two candidate output grids for *i* test inputs, each pair of candidate output grids receives a score of 1/*i* if one is correct and 0 otherwise. These scores are summed and normalized over the number of test inputs in that task. This is an interesting metric in the context of program synthesis, since the official score is calculated as the pass@2 test output grids, meaning it is not necessarily the case that if all *i* test outputs are correct, they are outputs of the same program.

Parallelization at the very start of the context window can be achieved by running *n* independent attempts for a given task at the same time, before ranking and selecting candidate solutions at the end, similar to []. If *n* agents are run on a task with *i* test inputs, then there are at most min(*i*,*n*) distinct programs contributing to the score on that particular task. An alternative metric, perhaps more aligned with the goal of program synthesis, would be to calculate the number of tasks where there was **at least one program synthesized by an agent that solved all training and test inputs** correctly. The exploration of the effect of scaling test time compute in this way is left for future work.