

GIFARC: Synthetic Dataset for Leveraging Human-Intuitive Analogies to Elevate AI Reasoning

Woochang Sim
GIST

woochang@gm.gist.ac.kr

Hyunseok Ryu
GIST

omnyx2@gmail.com

Kyungmin Choi
GIST

youvegotmail@gm.gist.ac.kr

Sungwon Han
GIST

lion4152@gmail.com

Sundong Kim
GIST

sundong@gist.ac.kr

Abstract

The Abstraction and Reasoning Corpus (ARC) poses a stringent test of general AI capabilities, requiring solvers to infer abstract patterns from only a handful of examples. Despite substantial progress in deep learning, state-of-the-art models still achieve accuracy rates of merely 40–55% on 2024 ARC Competition, indicative of a significant gap between their performance and human-level reasoning. In this work, we seek to bridge that gap by introducing an analogy-inspired ARC dataset, GIFARC¹. Leveraging large language models (LLMs) and vision-language models (VLMs), we synthesize new ARC-style tasks from a variety of GIF images that include analogies. Each new task is paired with ground-truth analogy, providing an explicit mapping between visual transformations and everyday concepts. By embedding robust human-intuitive analogies into ARC-style tasks, GIFARC guides AI agents to evaluate the task analogically before engaging in brute-force pattern search, thus efficiently reducing problem complexity and build a more concise and human-understandable solution. We empirically validate that guiding LLM with analogic approach with GIFARC affects task-solving approaches of LLMs to align with analogic approach of human.

1 Introduction

Recent advances in AI have produced impressive results in specialized domains, ranging from image classification to natural language processing. Despite these achievements, many of these systems remain limited in their capacity for genuinely flexible, general-purpose reasoning – a core characteristic often associated with artificial general intelligence (AGI). In this context, the Abstraction and Reasoning Corpus (ARC) was introduced as a benchmark aiming to measure the ability of artificial general intelligence systems to abstract visual patterns and reason about them in a manner akin to human cognition [1]. ARC tasks are intentionally designed to challenge AI the ability of high-level abstract reasoning by requiring solutions derived from minimal few-shot examples. This requirement closely mirrors the cognitive flexibility inherent to human task-solving, thus making ARC an essential dataset for probing the fundamental limitations and capabilities of current AI systems in relation to achieving human-level AGI.

Despite the excitement surrounding deep learning, current best-performing models in the 2024 ARC Prize competition left a significant gap compared to human-level proficiency [2]. While traditional deep neural networks typically excel with ample data, they struggle in example-deficient tasks that

¹Full dataset is available on Hugging Face at https://huggingface.co/datasets/DumDev/gif_arc. All 10,000 tasks generated with GIFARC are visualized in <https://gifarc.vercel.app/>.

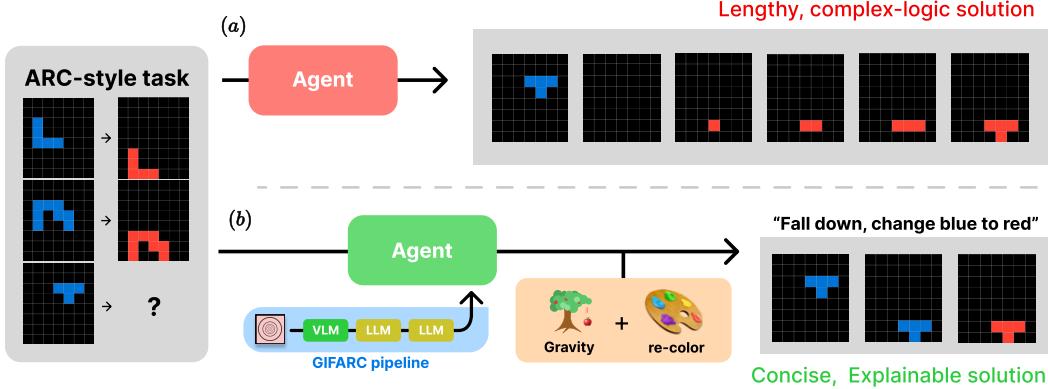


Figure 1: Illustration of how differently an agent solves ARC-style task when it is guided with or without analogic approach. In (a), an agent attempts to solve the ARC-style task in brute-force manner and is in risk to result logically complex solutions. In contrast, When the agent has been guided to think analogically with GIFARC dataset, as depicted in (b), it finds out a more concise and human-intuitive solution.

require understanding. This shortcoming underscores the limitations of purely data-driven approaches, especially when the underlying task involves rich forms of abstract and compositional reasoning [3, 4]. As a result, ARC provides strong evidence that deep learning alone does not yet capture the breadth of human reasoning, pointing to critical shortcomings in its capacity for generalization, pattern inference, and systematic task-solving.

Why does this gap exist? One crucial factor lies in how humans leverage analogies to map new visual or conceptual clues onto existing knowledge. When people see a grid transformation as in Figure 1, they can instantly map this visual pattern onto familiar concepts such as gravity and color change. These priors, accumulated from a lifetime of diverse visual and conceptual experiences, provide immediate insight into the task constraints and sharply reduce the complexity of the tasks. Rather than searching blindly through every possible transformation, humans can reject lengthy or counterintuitive candidates and focus on the small subset of patterns consistent with their analogies.

In contrast, deep learning models rarely have an explicit mechanism for analogy-making [5]. Even advanced neural networks typically rely on statistical correlations learned from large training sets, lacking the means to “tag” or “reframe” new tasks in terms of previously understood scenarios [6]. Consequently, these models miss a powerful heuristic: an analogy-driven search strategy that narrows down the solution space by linking visual abstractions to real-world events.

Need of analogy-contained dataset. To help bridge this gap between human cognition and AI, the agent should obtain human-level amount of analogies in prior to any task-specific training. Since the models we aim to train are predominantly data-driven, the need for training dataset that consists of useful analogies naturally arise. The dataset we seek contains following criteria;

- (i) *ARC-style examples* : the dataset should contain analogies in forms of ARC-style examples. Since the agent is evaluated via ARC benchmark, the agent has the world perception in form of 2D grids. In addition, the analogies one needs to know should be expressible onto an ARC-style 2D grid. Thus, the analogy we intend to train should also be expressed and given in ARC-style 2D grid in order for the agents to understand and utilize during evaluation.
- (ii) *Executable codes* : In order for the agent to capture how to manipulate 2D grid in accordance to the analogy, analogy should also be expressed in executable codes. Therefore, the analogy of transformation should also be given as lines of executable codes. With those 2D grids and codes, the agent should be able to correlate ARC-style transformation with its underlying analogy and find it during evaluation.

In this paper, we propose an analogy-inspired synthetic ARC dataset, GIFARC. Drawing on large language model (LLM) and vision-language model (VLM), we developed a framework to synthesize ARC-style tasks from online GIF images by extracting various visual patterns and the analogies

they evoke. These GIFs encompass a range of scenarios (e.g., snowfall, blooming flowers, moving objects), each providing a different type of visual transformation. We encode these transformations in newly generated ARC-style tasks and, crucially, supply ground-truth analogy labels that clarify the underlying conceptual mapping (e.g., “water flow blocked by an obstacle”).

Role of GIF as the basis of ARC-style task generation. Using GIFs as the starting ingredient of ARC-style tasks, our GifArc synthesis pipeline takes great advantages in picking human-understandable and visualizable analogies, as GIFs are series of images that are intentionally drawn to visualize certain ideas. In addition, GIF also has advantages in holding a sufficiently small number of analogies. In terms of analogy extraction simplicity, it is critical for the analogies to be implied in the image series without any entanglement to one another. Longer image series such as videos are more likely to hold excessive number of analogies and consequently have a higher risk of entanglement in between. Thus, GIF is an adequate form of analogy-implying image series.

These new ARC-style tasks generated from GIFs will possess a variety of intuitive analogies. We expect that the agent trained with this dataset can gain the grasp of analogical approaches and utilize them in the solving step. By incorporating diverse analogies into the dataset, we hope to facilitate models that can “think analogically” before searching for solutions, thereby reducing the complexity of task-solving. In addition, by converting the visual data to verbose expression before generating the ARC-style tasks, we expect that the agent will specifically gain the ability to verbally point out the analogy intended in the ARC-style task.

Our GifArc dataset synthesis pipeline operates in three sequential stages. First, a VLM converts raw GIF into a structured JSON record listing the objects in the scene, invariant backgrounds, dynamic trajectories, and concise analogy. LLM then compresses that verbose abstraction into a task sketch comprising concepts and descriptions for ARC-style tasks. Conditioned on the sketch, LLM finally generate executable ARC-style tasks via retrieval-based in-context learning.

Running this pipeline on motion-rich GIFs produces 10k unique high-quality ARC-style tasks spanning 20 analogy categories. We empirically demonstrated that the analogy information in our GifArc dataset guides LLMs to reason through ARC tasks in a manner more aligned with human-like reasoning.

2 Related Work

Abstraction and Reasoning Corpus. The Abstraction and Reasoning Corpus (ARC), introduced by Chollet (2019) [1], is an open-ended benchmark designed to evaluate broad generalization from minimal demonstrations. Its goal is to test how efficiently an agent can acquire and apply abstract patterns with just a few examples. ARC-AGI-1 (often referred to as “ARC-1”) quickly became a focal point in the research community. The initial Kaggle competition in 2020 yielded a top solve rate of only 21%, significantly below human-level performance. With recent advances in large language models, OpenAI o3-preview model scored 75% on ARC-AGI-1 using low level of computation and reached 87% accuracy with higher level of computation, marking the first effective solution to the challenge [7]. This milestone sparked interest and led to the development of ARC-AGI-2 [8], which was designed to pose even greater challenges while preserving accessibility to humans.

Synthetic ARC Generation. To address the data scarcity inherent in ARC, which disturbs both symbolic program synthesis and the training of neural models, researchers have turned to generating programmatic data using human-curated, AI-generated, or hybrid pipelines [9–12]. One such effort is ReARC, which manually scripts generative procedures for each task to produce an effectively infinite set of in-distribution examples [13]. LARC (Language-Complete ARC) crowd-sources natural language explanations of tasks, enabling models to learn from natural programs instead of raw grids [14]. Other benchmarks such as ConceptARC, Mini-ARC, and 1D-ARC use manually crafted rules to produce diverse yet interpretable task distributions [15–17]. Recent approaches automate this process at scale. For instance, BARC generates 400,000 training examples by prompting GPT-4 to mutate a curated set of 160 Python-based solvers [18]. Each seed includes a transformation function, an input generator, and natural language description. Similarly, MC-LARC transforms ARC tasks into multiple-choice formats, enabling scalable evaluation of language models’ intermediate reasoning stages [19].

Different from existing in-distribution resampling or code-mutation pipelines, our work introduces GIFARC, a new dataset that mines human-intuitive GIFs, distills their analogical structure with vision-language models, and auto-compiles each analogy into an executable ARC-domain specific language (ARC-DSL) code, explicitly teaching models how to think by analogy rather than simply giving them more examples to memorize.

3 Method

3.1 Overview

Problem formulation. Given a set of GIF images $\mathcal{G} = \{g_j\}_{j=1}^N$, our goal is to construct an analogy-grounded ARC dataset, GIFARC, containing ARC-style tasks and represented as triples $\mathcal{T} = (\mathcal{E}, \alpha, \phi)$ where (i) $\mathcal{E} = \{(x_i, y_i)\}_{i=1}^K$ is a set of K input–output grid pairs of ARC-style task. Here, $x_i \in \{0, \dots, 9\}^{h_{x,i} \times w_{x,i}}$ is the input grid, $y_i \in \{0, \dots, 9\}^{h_{y,i} \times w_{y,i}}$ is the output grid, with $h_{G,i}, w_{G,i}$ respectively being the height and width of the G grid (either grid x or y) in i -th pair of \mathcal{E} ; (ii) α is a short natural-language analogy description (e.g., “blocked water flow”). (iii) ϕ is a Python program implementing $\mathcal{F}: \mathcal{X} \rightarrow \mathcal{Y}$, the latent deterministic transformation such that $y_i = \mathcal{F}(x_i)$; GIFARC ($\mathcal{D} = \{\mathcal{T}_j\}_{j=1}^N$) therefore pairs each ARC task with its provenance and analogy.

Pipeline summary. Our data synthesis pipeline is a three-stage process that transforms raw GIFs into fully executable, analogy-grounded ARC-style tasks (see Figure 2 for illustrative examples).



- Step 1. A VLM (GPT o1) analyzes each GIF g and outputs a structured JSON record that captures its scenario, objects, static/dynamic patterns, interactions, and core reasoning principles $\mathcal{A}(g)$.
- Step 2. A LLM (GPT o3-mini) converts the JSON abstraction into a concise ARC-style task sketch: a set of concepts with a natural-language description that implies the extracted patterns.
- Step 3. Conditioned on the sketch and a handful of retrieved ARC task examples, the LLM (GPT o3-mini) synthesizes Python code implementing `generate_input` (stochastic examples) and `main` (deterministic transformation). The output is a complete ARC-style task comprising (i) an input–output demonstration set \mathcal{E} , (ii) the Python solution program ϕ , and (iii) its analogy label α .

Details of Step 1 appear in Section 3.2, Step 2 in Section 3.3, and Step 3 in Section 3.4. All LLM calls use Azure OpenAI endpoints.

3.2 Step 1. Extracting Visual Abstractions from GIFs

Our first stage converts a raw GIF into a structured, readable summary that captures the analogy-laden visual logic hidden in the clip. Given an animated GIF g depicting a short visual phenomenon (e.g., snow piling up, light reflecting, pipes clogging), we wish to automatically extract the key elements $\mathcal{A}(g)$ that a human would use when reasoning by analogy: what the scene contains, which parts move or remain fixed, how objects interact, and which general principle explains the observed dynamics. The output, called a visual abstraction, serves as the input for the later stage.

GIF crawl. To maximize coverage of distinct visual regularities, we first assemble a visual transformation corpus of GIFs. Using the GIPHY API, we query a curated tag list `{nature, geometry, pattern, relational_pattern, sequence}` where these types of clips often exhibit dynamic change. After deduplication, we obtained total 10k GIFs.

Prompt for extracting visual abstractions. We then submit every GIF to the vision–language model GPT o1 under a prompt intentionally designed around two objectives: (i) *granularity* - to extract as many diverse visual analogies as possible from the GIF, (ii) *structure* – the reply must be valid JSON, and (ii) *coherence* – to ensure that each extracted visual analogy maintains alignment with the original GIF. Full prompt used in this step is reported in Appendix A.1. The system message

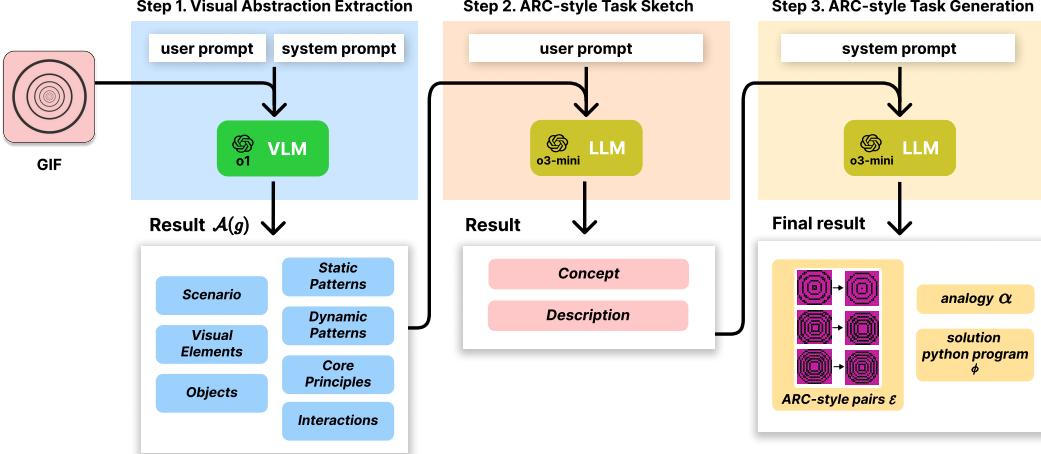


Figure 2: Illustration of GIFARC data synthesis pipeline that transforms a single GIF into a corresponding ARC-style task and supplementary data. In step 1, a vision language model (GPT o1) digests a GIF file and outputs a detailed text expression $\mathcal{A}(g)$ of the visual transformation implied in the GIF in a JSONL format. In step 2, a large language model (GPT o3-mini) reads step 1 result JSONL and outputs a text sketch an ARC-style task. In step 3, a large language model (GPT o3-mini) reads the step 2 sketch result and generates an ARC-style task \mathcal{E} , an implied analogy α , and the Python solution program ϕ . The example prompts used in each step are reported in Appendix A.

states the structural constraint (“return JSON with keys { scenario ... interactions} and use bullet lists only”). The user message enumerates the seven fields and attaches concrete bullet-level instructions.

These seven fields in $\mathcal{A}(g)$ serve complementary purposes: (i) **scenario** describes the overall narrative of GIF images; (ii) **visual_elements** stores the raw, static pixel facts; (iii) **objects** binds memorable names to entities so later prompts can reference them; (iv) **static_patterns** and (v) **dynamic_patterns** distinguish invariants and variants from transformations, respectively; (vi) **core_principles** captures the very abstraction (e.g., gravity, reflection, occlusion, ...) that we ultimately hope an ARC solver will recall as an analogy; and (vii) **interactions** records explicit or implicit relations (e.g., “snowflake collides with ground”, “fluid flow is blocked”) among objects.

This step is an extraction step to pull out intuitive analogies from GIF image. With this step, the GIFs can now be understood by text based LLMs in the later stages. In addition, converting pixel data into a more lightweight text descriptions lets the pipeline to efficiently digest the analogy faster through the remaining synthesis stages without having to handle heavy image data along. Also, in case that GIF holds multiple visual analogies, explicitly pointing out which visual analogy to focus on lets the agents in later stages interpret the target analogy without possible confusion. Furthermore, this stage is critical in abiding the GIPHY API terms of right. By irreversibly conceptualizing visual data into text description, we have generated a distinct visual analogy data pool that does not contain any explicit GIFs.

3.3 Step 2. From Visual Abstraction to Task Sketch

The visual abstraction $\mathcal{A}(g)$ obtained in previous step is rich but verbose: it contains a lot of observable details of the GIF, many of which may be irrelevant for task design (e.g., background textures, camera shakes). Before we can synthesize executable code, we therefore distill $\mathcal{A}(g)$ into a compact *task sketch* composed of two parts: (i) **Concepts:** 3–5 keyword phrases that name the core visual ideas (e.g., blocked_flow, rotational_symmetry, incremental_accumulation, ...); (ii) **Description:** a single paragraph that narrates how those concepts manifest on an ARC-style grid.

Prompt for task sketch. We obtain each sketch from a single completion of the text-only model GPT o1 using a three-block prompt: **Block 1** provides in-context demonstrations; we draw 75 concepts/description pairs at random from the 160 human-written example tasks released with the BARC corpus [18] and paste them. Because these examples cover symmetry, counting, physical

analogy, topological change, and more, they implicitly teach the model what “looks like” a solvable ARC-style task while leaving room for new content. **Block 2** provides the seven fields of $\mathcal{A}(g)$ from the previous step in bullet form. **Block 3** has a message that freezes the output format, exactly two comment headers for concepts and description.

We further instruct the model to reflect the full trajectory from initial to final state, maintaining semantic fidelity between the GIF dynamic behavior and the input-output structure of ARC-style grid. Full prompt used in this step is reported in Appendix A.2.

This selection step is crucial in two aspects. First, we expect LLM to filter out abstractions that cannot be turned into deterministic grid transformations (e.g., ‘clouds forming random shapes’). Second, it forces the retained content into the exact format expected by our code-generation prompt via in-context learning, thereby lowering entropy and reducing failure cases downstream.

3.4 Step 3. Generating Executable ARC-style Tasks

The final stage transforms each task sketch produced in Step 2, into a fully executable ARC-style task. The output comprises two parts: (i) `main`, A deterministic Python function implementing the latent transformation \mathcal{F} (rendered as Python code ϕ); (ii) `generate_input`, A stochastic Python function that draws fresh grid examples from the same distribution to generate input grids.

From this output, input grids generated with `generate_input` function are fed to the `main` function to produce corresponding output grids, thereby constructing a set of input-output grid pairs \mathcal{E} . The analogy α is copied from the task sketch created in Step 2, forming a ARC-style task $\mathcal{T} = (\mathcal{E}, \alpha, \phi)$.

Prompt for generating executable ARC-style tasks. Our code-generation prompt inherits the overall structure introduced in BARC [18], but differs in that the new task is grounded in the GIF-derived concepts and description produced in Step 2. To tighten this grounding while preserving the coding style that has been proven effective in earlier literature, we again adopt an in-context learning strategy, along with retrieval augmentation. Specifically, we embed both our description and the 160 human-authored example ARC-style tasks with `text-embedding-ada-002` model. The cosine similarity between sketch and examples is computed, and the top-4 semantically closest examples are pasted into the prompt before the new sketch. Each example already contains a working `generate_input` / `main` pair, so these few-shot examples show the model how high-level concepts and narrative prose translate into concise Python code. The full prompt example for generating executable ARC tasks is reported in Appendix A.3.

4 Dataset Analysis

To ground subsequent experiments, we first take a close look at the corpus itself. We here provide our analysis on a sampled subset of the dataset. A comprehensive analysis and full statistics of the entire dataset are reported in Appendix.

Descriptive Statistics. Table 1 summarizes coarse-grained corpus properties. Gifarc currently contains 10,000 tasks, each accompanied by 103,357 train–test input/output pairs and an executable solution. The average input grid is 420.404 cells with a standard deviation of 262.096, while target grids are at $1,125.484 \pm 2,894.980$. On average, a task uses 4.402 distinct colors (1.917 std), indicating that most tasks require multi-object reasoning or contain object with some complexity, rather than binary foreground/background segmentation.

Table 1: Summary of Gifarc’s descriptive statistics.

	Count	Mean	Std.
The number of tasks	10,000	—	—
Input-grid size ($h \times w$)	—	420.404	262.096
Target-grid size ($h \times w$)	—	1,125.484	2,894.980
The number of colors per task	—	4.402	1.917

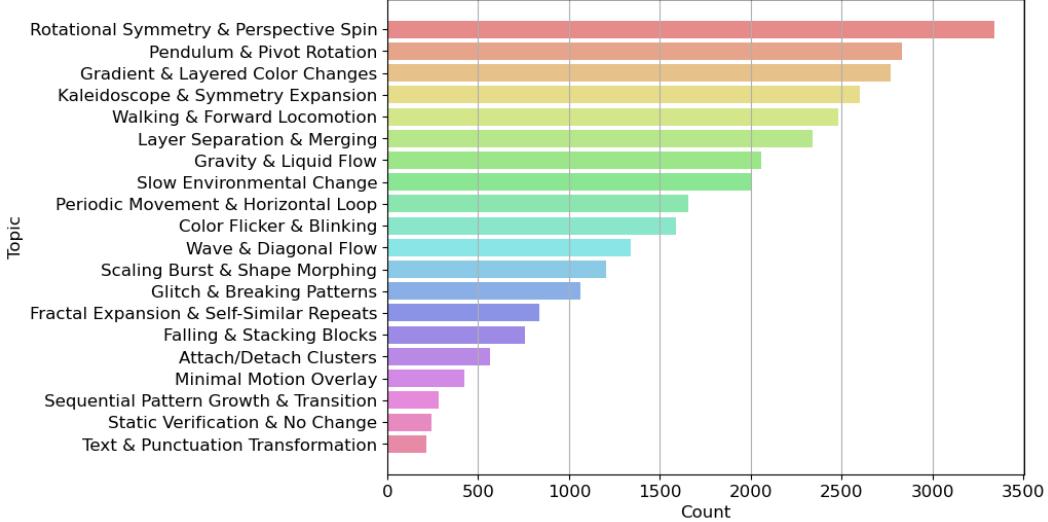


Figure 3: Histogram of task types occurrence in GIFARC.

Table 2: Code Complexity of GIFARC Tasks

	Mean	Std.
Lines of Code	12.12	38.43
Cyclomatic Complexity	6.59	3.66
Nesting Depth	4.37	1.52
Unique Ops Count	7.01	0.18

Table 3: Success rate of generation.

	Rate (%)
Visual abstraction stage	94.12
Task sketch stage	80.54
Executable ARC task stage	81.82

Distribution of Task Types. To understand which semantic concepts and human-intuitive analogies dominate the dataset, we visualized the distribution of task types in our dataset. We prompted GPT-4o to map each keyword in the task into one of 20 coarse types. Figure 3 shows the histogram of task types based on the keyword. Note that in the initial stage of crawling raw GIFs, we sampled 500 GIFs each from 20 different GIF categories (e.g., nature, geometry, animation, ...). The distribution of the resultant task type has shifted from the initial GIF crawling category distribution. The results indicate that the problems generated from GIFs span a diverse range of task types, with the ‘Rotational Symmetry & Perspective Spin’ type being the most dominant. Also, many problems contained a mixture of multiple task types.

Task Complexity. We assessed task complexity by measuring the complexity of the code used to generate GIFARC’s grids. To this end, we report metrics such as the average number of lines of code (Lines of Code), the number of branching points in the code (Cyclomatic Complexity), the maximum depth of nested blocks in the code (Nesting Depth), and the number of unique operators used in the code (Unique Ops Count). These metrics make it easy to gauge how complex the processes required by GIFARC tasks are.

Table 2 summarizes the code complexity metrics measured for GIFARC tasks. The average number of lines of code is 12.12 with a standard deviation of 38.43, while the cyclomatic complexity averages 6.59 (3.66 std). The maximum nesting depth in the code has a mean of 4.37 (1.52 std), and the number of unique operators averages 7.01 (0.18 std). These metrics quantitatively demonstrate that GIFARC contains tasks with diverse levels of complexity.

Generation Fidelity. We measure the fidelity of our generation pipeline by evaluating the pass ratio at each stage. A task is considered a failure if it fails to compile or does not produce outputs in the expected format. Table 3 reports the success rates, showing that the overall pass ratio is consistently high across stages.

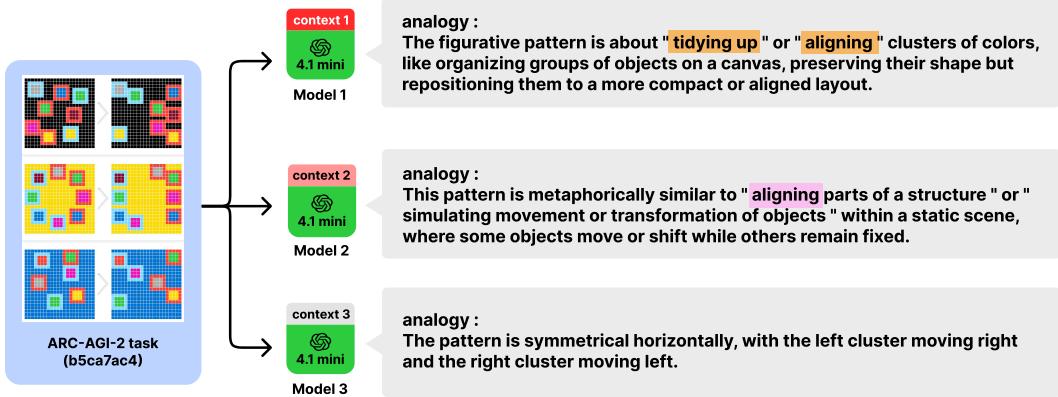


Figure 4: Experiment 1 case study: the LLMs learned in-context with richer analogy context result in more analogic interpretation of the ARC-AGI-2 task. Context 1, 2, 3 respectively stands for full description sample, description sample without analogy, description sample without analogy and without solution.

5 Applications

In this section, we conducted two experiments to examine: (1) how in-context learning with appropriately selected guiding examples influences the task-solving strategy of a LLM, and (2) how it affects the use of analogical information in GIFARC.

5.1 Verification Changes in Reasoning Steps

In order to check the importance of each element in the dataset, GPT 4.1-mini was instructed to return the thought process while solving ARC-AGI-2 public evaluation dataset in context of sampled data of GIFARC. As described above, since ARC-AGI-2 dataset requires a more diverse approach than the predecessor, analogic approach in the task-solving process would be more clearly shown and used for the comparison of the reasoning step.

GPT 4.1-mini was first learned in-context with 3 types of sampled GIFARC:

- full description:** 15 well-figuratively described data $(\mathcal{E}, \alpha, \phi)$ chosen from GIFARC by repeated multi-turn refinement prompts to the LLM.
- description without analogy:** 15 data $(\mathcal{E}, \alpha_{\text{flat}}, \phi_{\text{flat}})$ from full description but gone through a ‘flattening’ process where researchers replaced analogic terms in α and ϕ into their corresponding lower-level synonyms.
- description without analogy and without solution:** 15 data $(\mathcal{E}, \alpha_{\text{flat}})$ from description without analogy but without solution ϕ_{flat} .

Note that smaller numbered description contains richer analogies. These in-context learned models were then prompted to solve the tasks of ARC-AGI-2. In order to effectively check whether an analogic approach was used, models were instructed to specify two points in the result; first identify the underlying analogy in the given task, and implement it as a detailed solution.

As a result of the solution comparison between models, we could confirm instances that analogies in GIFARC significantly helped identify the analogy of ARC-AGI-2 task. GPT 4.1-mini with full description expressed the analogy applied to the ARC-AGI-2 problem using an intuitive analogy. Figure 4 depicts one of the well-performed result cases. While GPT 4.1-mini with description without analogy and without solution mainly described the task analogy with grid-level terms (symmetrical, horizontal, left, right), GPT 4.1-mini with full description was able to describe with words from several domains (such as ‘tidying up’ and ‘organizing’). In particular, it was observed that GPT 4.1-mini perceived full description as not only as an ‘analogy phrase pool’, but also recognized it as an ‘example of analogic approach’ and could use external domain terminologies those are not present in context. This result shows that the analogic terms that were wiped during the flattening process are critical for the LLMs to get the grasp of analogic approach needed in solving ARC-style tasks.

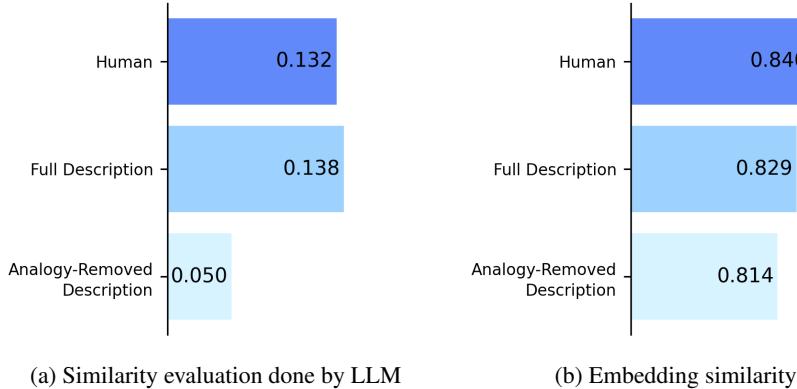


Figure 5: Similarity analysis between task-implied analogy found by Gifarc-trained LLMs and its corresponding hand-crafted analogy. Figure 5a illustrates that the analogy generated by GPT 4.1 mini with full description showed better similarity than that of GPT 4.1 mini with analogy-removed description by 0.087 pp, and embedded cosine similarity by 0.014 pp, as depicted in Figure 5b.

5.2 Alignment between Context-Generated and Ground-Truth Analogies

The second experiment was conducted to verify if LLMs with Gifarc-sampled context can successfully identify underlying analogy in ARC-style tasks. Here, we used full description and analogy-removed description, which we further removed the analogical expressions that remained in description (such as “gravity”, “spiral spin”, and “flicker”) and replaced them with grid-level descriptions (such as “move down”, “rotate”, and “color change”). Next, GPT 4.1-mini with full description and GPT 4.1-mini with analogy-removed description were instructed to find the analogy implied in the 12 ARC-style tasks. Along with two models, three humans (who know the ARC well) also collected analogies implied in the 12 tasks. The outputs of two models and three humans were then compared with the ground-truth analogies of the task generated with Gifarc pipeline.

We first instructed GPT o3-mini to evaluate semantic similarities by providing a one-shot guideline about measuring similarity. This evaluator LLM evaluated the semantic similarities between analogies generated by solvers (two in-context learned LLMs and three humans) and the ground truth analogy. Figure 5a depicts the similarity result done by LLM. It resulted that the output of GPT 4.1-mini with full description showed 0.137 similarity, while the output of GPT 4.1 mini with analogy-removed description only measured 0.050. Secondly, we embedded each outputs and compared the cosine similarity between. It resulted that the output of GPT 4.1-mini with full description showed 0.829 similarity, while the output of GPT 4.1 mini with analogy-removed description only scored 0.814. This shows that context sample from Gifarc played a significant role in changing the task-solving approach of LLM to align with human-level analogic approach. Since Gifarc is a superset of the tested samples, it contains a much various types of analogies. Thus, this experiment shows the potential of Gifarc as the ‘analogic approach’ guiding dataset.

6 Conclusion

We introduced Gifarc, a large-scale synthetic ARC-style dataset grounded in human-intuitive analogies extracted from GIF images. Gifarc framework narrows the human-AI reasoning gap by embedding explicit analogy labels and executable solutions that help language-vision models to map abstract grid transformations with everyday experiences. This approach has potential to improve AI-assisted education, scientific discovery, and decision-making support systems. No significant negative societal impacts are anticipated from this framework at present. Empirical results on the original ARC benchmark confirm that both fine-tuning on Gifarc and leveraging analogy cues in the reasoning step boost solver accuracy. However, our data generation pipeline is inherently dependent on a single GIF for analogy extraction, which places a limitation on the diversity and scope of analogies. In future work, we will scale Gifarc beyond motion-rich GIFs to videos and 3D simulations, extend evaluation to additional reasoning suites, and explore the fusion of multiple visual analogies to generate much more complex ARC tasks.

References

- [1] François Chollet. On the Measure of Intelligence. *arXiv:1911.01547*, 2019.
- [2] Lab42. ARC Prize, 2024.
- [3] Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. Faith and fate: Limits of transformers on compositionality. In *NeurIPS*, 2023.
- [4] Seungpil Lee, Woochang Sim, Donghyeon Shin, Sanha Hwang, Wongyu Seo, Jiwon Park, Seokki Lee, Sejin Kim, and Sundong Kim. Reasoning Abilities of Large Language Models: In-Depth Analysis on the Abstraction and Reasoning Corpus. *arXiv:2403.11793*, 2024.
- [5] Melanie Mitchell. Abstraction and analogy-making in artificial intelligence. *Annals of the New York Academy of Sciences*, 1505(1):79–101, 2021.
- [6] Mikel Bober-Irizar and Soumya Banerjee. Neural networks for abstraction and reasoning. *Scientific Reports*, 14(1):27823, 2024.
- [7] Rolf Pfister and Hansueli Jud. Understanding and Benchmarking Artificial Intelligence: OpenAI’s o3 Is Not AGI. *arXiv preprint arXiv:2501.07458*, 2025.
- [8] Francois Chollet, Mike Knoop, and Greg Kamradt. Abstraction and Reasoning Corpus for Artificial General Intelligence v2. <https://github.com/arcprize/ARC-AGI-2>, 2025.
- [9] Rim Assouel, Pau Rodriguez, Perouz Taslakian, David Vazquez, and Yoshua Bengio. Object-Centric Compositional Imagination for Visual Abstract Reasoning. In *ICLR Workshop on the Elements of Reasoning: Objects, Structure, and Causality*, 2022.
- [10] Solim LeGris, Wai Keen Vong, Brenden M Lake, and Todd M Gureckis. H-ARC: A Robust Estimate of Human Performance on the Abstraction and Reasoning Corpus Benchmark. *arXiv preprint arXiv:2409.01374*, 2024.
- [11] Gustaw Opielka, Hannes Rosenbusch, Veerle Vijverberg, and Claire E Stevenson. Do Large Language Models Solve ARC Visual Analogies Like People Do? In *CogSci*, 2024.
- [12] Yonggang Qi, Kai Zhang, Aneeshan Sain, and Yi-Zhe Song. PQA: Perceptual Question Answering. In *CVPR*, 2021.
- [13] Michael Hodel. Addressing the Abstraction and Reasoning Corpus via Procedural Example Generation. *arXiv:2404.07353*, 2024.
- [14] Samuel Acquaviva, Yewen Pu, Marta Kryven, Theodoros Sechopoulos, Catherine Wong, Gabrielle Ecanow, Maxwell Nye, Michael Tessler, and Joshua B. Tenenbaum. Communicating Natural Programs to Humans and Machines. In *NeurIPS*, 2022.
- [15] Subin Kim, Prin Phunyaphibarn, Donghyun Ahn, and Sundong Kim. Playgrounds for Abstraction and Reasoning. In *NeurIPS Workshop on Neuro Causal and Symbolic AI*, 2022.
- [16] Arseny Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. The ConceptARC Benchmark: Evaluating Understanding and Generalization in the ARC Domain. *Transactions on Machine Learning Research*, 2023.
- [17] Yudong Xu, Wenhao Li, Pashootan Vaezipoor, Scott Sanner, and Elias B Khalil. LLMs and the Abstraction and Reasoning Corpus: Successes, Failures, and the Importance of Object-Based Representations. *Transactions on Machine Learning Research*, 2024.
- [18] Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, et al. Combining induction and transduction for abstract reasoning. *arXiv preprint arXiv:2411.02272*, 2024.
- [19] Donghyeon Shin, Seungpil Lee, Klea Lena Kovacec, and Sundong Kim. From Generation to Selection Findings of Converting Analogical Problem-Solving into Multiple-Choice Questions. In *EMNLP*, 2024.
- [20] Max Peepkorn, Tom Kouwenhoven, Dan Brown, and Anna Jordanous. Is Temperature the Creativity Parameter of Large Language Models?, 2024.

Appendix

A Examples of Prompts

A.1 Full Prompt Example for Extracting Visual Abstractions

The extraction of visual abstractions constitutes the first step in the data generation process. This process involves extracting comprehensive information from GIFs, including `scenario`, `visual_elements`, `objects`, `static_patterns`, `dynamic_patterns`, `core_principles`, and `interactions`. The specific content requirements for these seven types of information and the response format for the LLM are detailed in both the user prompt and system prompt below. The following content presents the complete specification of the aforementioned prompts.

User Prompt: Extracting Visual Abstractions from GIFs

Please analyze the given GIF by extracting core reasoning elements. For each section below, provide your response as a list of clearly separated items (not long-form paragraphs).

[scenario]

- Describe the complete narrative of the GIF in 1-3 concise sentence(s).
- Include initial state, key changes, and final state.
- Include only observable facts and avoid speculation.

[visual_elements]

- Identify the key visual elements, objects, and actors in the GIF.
- List the major visual components in the scene (e.g., objects, colors, spatial arrangement, shapes).

[objects]

- List all visual objects appearing in the GIF using this format:

```
{  
  "name": "object name",  
  "type": "explicit" or "implicit"  
}
```
- explicit = physical objects with clear boundaries
- implicit = patterns or structures formed by multiple elements
- Identify 2-8 key objects.

[static_patterns]

- Identify the elements or relationships that remain consistent throughout the GIF.
- Describe repeating spatial arrangements, consistent backgrounds, or fixed design features.
- List all patterns or objects that remain constant throughout the GIF.

[dynamic_patterns]

- Describe how the elements interact and change over time.
- Consider whether the changes occur gradually (step-by-step), abruptly, or through multi-stage transformations (e.g., directional shifts, scaling, rotation).
- List the distinct changes or interactions that occur over time.

[core_principles]

- Identify all general reasoning principles or mechanisms that explain **how and why** the dynamic elements change over time.
- Each principle should be **generalizable beyond the current GIF** and help abstract the reasoning structure behind the transformation.
- List one or more such principles that account for the observed dynamics.
- After listing them, **summarize the single most fundamental principle in one concise sentence.**

- Examples include: physical forces (e.g., gravity causes downward movement), goal-oriented behaviors (e.g., movement toward a target), causal chains (e.g., one event triggers another), symmetry-based transformations (e.g., reflection or alignment), and repetitive or cyclic patterns.

[interactions]

- List object interactions using this format:

```
{
  "objects_involved": ["object1", "object2"],
  "interaction_type": "clear/ambiguous/constraint",
  "interaction_parameters": ["parameter1", "parameter2"]
}
```

- **interaction_type** definitions:
 - **clear**: distinct physical interactions (collisions, contact, etc.)
 - **ambiguous**: indirect or unclear interactions
 - **constraint**: interactions that establish limitations or boundaries
- Record 2-6 key interactions, including only those directly observed in the GIF.

System Prompt: Extracting Visual Abstractions from GIFs

You are an analysis assistant. Your role is to extract structured and reproducible information from a visual GIF using the five categories below.

You **MUST** focus on observable phenomena, transformation structures, and etc.

Do **NOT** invent unobservable events or make abstract generalizations. Describe only what can actually be seen in the GIF.

For each category below, provide a plain list of items (one per line or bullet point). Avoid paragraph-style narration. Focus only on concrete, observable phenomena and transformation patterns.

Return your response strictly in the following JSON format:

```
{
  "scenario": "<detailed description of the overall narrative>",
  "visual_elements": [
    "<list of observable objects, colors, spatial arrangements, or notable visual traits>"
  ],
  "objects": [
    {
      "name": "<object name>",
      "type": "explicit/implicit"
    }
  ],
  "static_patterns": [
    "<list of all objects or properties that remain unchanged throughout the GIF>"
  ],
  "dynamic_patterns": [
    "<list of all distinct transformations or movements that occur over time>"
  ],
  "core_principles": [
    "<list of general reasoning principles behind the transformation (e.g., gravity causes vertical motion)>"
  ],
  "interactions": [
    {
      "objects_involved": ["<object1>", "<object2>"],
      "interaction_type": "clear/ambiguous/constraint",
      "interaction_parameters": ["<parameter1>", "<parameter2>"]
    }
  ]
}
```

A.2 Full Prompt Example for Task Sketch

The task sketch represents the second step in the data generation process. The objective of this step is to design ARC-style tasks by dividing them into concepts and description components, utilizing the seven types of information extracted in Step 1. The user prompt provides the detailed specifications for this process.

User Prompt: Task Sketch

You've generated these on previous requests:
{examples}

Based on your previous GIF analyses, I'd like you to create a new concept that incorporates the key elements from this analysis:

```
scenario: {scenario}
objects: {objects}
static_patterns: {static_patterns}
dynamic_patterns: {dynamic_patterns}
interactions: {interactions}
core_principles: {core_principles}
```

Brainstorm one more.

If the above informations involve gradual changes over time or a process of reaching a clear goal state, the input and output grids should not merely depict a short-term or single-frame transition. Instead, they should be designed to capture the entire transformation process from the initial state to the final state.

When temporal progression or cumulative change is central to the analogy, construct the task code so that this flow and its outcome are clearly reflected through the difference between input and output.

In this case, please create concepts and descriptions that effectively incorporate the above visual elements, static patterns, dynamic patterns, and core principles, following the format below, while referencing previous examples:

```
# concepts:
# <concepts in your new generation>
# description:
# <description of your new generation>
```

A.3 Full Prompt Example for Generating Executable ARC Tasks Version 1

We generated ARC-style tasks using the analogies extracted from Step 1 and Step 2, employing two versions of prompts. Version 1 utilizes original BARC seeds that were used to generate input and output grid pairs. The following prompt contains the detailed explanation for Step 3 of Version 1.

User Prompt: Puzzle Implementation from Description (Version 1)

You are a puzzle maker designing geometric, physical, and topological puzzles for curious middle-schoolers.

Each puzzle consists of uncovering a deterministic rule, pattern, procedure, algorithm, or transformation law that maps inputs to outputs. Both the inputs and outputs are 2D grids of colored pixels. There are 10 colors, but the order of the colors is never relevant to the puzzle.

The middle schoolers are trying to discover this deterministic transformation, which can be implemented as a Python function called `main`. Designing a puzzle involves also creating example inputs, which can be implemented as a Python function called `generate_input`. Unlike `main`, the `generate_input` function should be stochastic, so that every time you run it, you get another good example of what the transformation can be applied to.

Here is a overview of the puzzle you are designing:

{description}

Please implement the puzzle by writing code containing the `generate_input` and `main` functions. Use the following standard library (`common.py`):

{common_lib}

Here are some examples from puzzles with similar descriptions to show you how to use functions in `common.py`:

{examples}

Your task is to implement the puzzle, following these steps:

1. Inspect the example puzzle implementations, making note of the functions used and the physical/geometric/topological/logical details
2. Inspect the new puzzle's description
3. Brainstorm a possible implementation for the new puzzle
4. Generate a code block formatted like the earlier examples with a comment starting `# concepts:` listing the concepts and `# description:` describing the inputs and transformation from the given description.

When implementing code, please avoid using float type variables, numbers with decimal points, and the math library. These elements make it difficult to intuitively identify patterns between input and output grids. Instead, use only integer operations and basic arithmetic operators to clearly reveal the essence of the pattern. Please strictly follow this constraint when implementing your code.

Also, When implementing code, please use the minimum number of lines possible. As code gets longer, its complexity increases, and if it becomes too detailed and complicated, people will find it difficult to intuitively understand the puzzle's rules just by looking at the input and output grids. Situations where one needs to analyze the code to understand the rules should be avoided. Please write concise and efficient code that clearly reveals the core pattern.

Be sure to make the transformation `main` deterministic. Follow the description closely.

System Prompt: Generating Executable ARC Task (Version 1)

You need to help me write code containing the `generate_input` and `main` functions according to the given puzzle design. You must use the standard library (`common.py`). Create an appropriate puzzle following the given puzzle design concepts and description.

When writing code, please use variable names that are meaningfully related to the core concepts of the problem. For example, if the problem involves snow falling phenomena, use variable names like `snowflake`, `precipitation`, `accumulation`, `gravity`, `obstacle`, etc. Specifically, when implementing the `generate_input` function and `main` function, make sure each variable name is directly associated with the concepts in the problem. For instance, use `gravity_strength` for a variable representing the intensity of gravity, and `obstacle_positions` for storing the locations of obstacles — choose names that clearly reveal the role and meaning of each variable.

Additionally, in the `generate_input` function, please restrict the grid size to be between `1x1` and `30x30`. Do not create grids larger than `30x30`. Implement the `generate_input` function that creates inputs appropriate for the problem and the `main` function that utilizes them while following these constraints.

When implementing code, please avoid using float type variables, numbers with decimal points, and the math library. These elements make it difficult to intuitively identify patterns between input and output grids. Instead, use only integer operations and basic arithmetic operators to clearly reveal the essence of the pattern. Please strictly follow this constraint when implementing your code.

Also, when implementing code, please use the minimum number of lines possible. As code gets longer, its complexity increases, and if it becomes too detailed and complicated, people will find it difficult to intuitively understand the puzzle rules just by looking at the input and output grids. Situations where one needs to analyze the code to understand the rules should be avoided. Please write concise and efficient code that clearly reveals the core pattern.

When doing this, please output your solution following the JSON format specified below.

```
{  
    "library": "<Write only the libraries used in the code. Ex. from  
    common import* \n import numpy as np \n ....>",  
    "main_code": "<Write the main code part.>",  
    "generate_input_code": "<Write the generate input code part.>",  
    "total_code": "<Write total code including libraries, main,  
    generate_input and given concepts and description>"  
}
```

B ARC Tasks Generate Pipeline Version 2

We renewed Version 2 of task generation pipeline for Step 3. This version bump was carried out because the BARC seeds are unsuitable for GIFARC task generation. In GIFARC dataset generated by the initial pipeline, we spotted several tasks which failed to express objects appearing in GIF. This was a critical problem, as it resulted in tasks that are irrelevant to the original GIF. Thus, we added an additional stage that explicitly extracts objects that are presented in GIF which is Step 3-1. We extracted a list of codes that generates a grid representation of each object appearing in the GIF with Step 3-1. The object-generation code extracted in this additional step was human-intuitive and well-known.

B.1 Architecture of ARC Tasks Generate Pipeline (Version 2)

After completing Step 1, the detected objects are then categorized into explicit or implicit types. Objects that have clear physical quantities or concrete shapes were categorized as explicit objects (i.e., human, dog, text), whereas implicit objects are metaphorical entities formed by compositing multiple explicit objects (i.e., circle: peoples are sitting around in circles, big fish: school of small fish form a shape of big fish to protect themselves from shark.). We instructed LLM (GPT 4.1) to write each object generation code not only with rich comments and rich analogies with great details, but also with meanings aligned to its pixel representations. Next, we 1) paired each block of code with corresponding object name, 2) filtered out duplicates, and 3) embedded the names of the objects into vectors using all-MiniLM-L6-v2. Only explicit objects were stored in the vector database, as implicit objects are composites of explicit objects by definition and therefore cannot serve as fundamental units. With those preparations completed, these codes are additionally supplied to Step 3-1 and Step 3-2, along with the results of their former step results. During Step 3-2, we queried the embedded objects name to the vector database searching for the explicit object name and code with the highest cosine similarity score. We use the code as the seed to build the prompt, which then is used in task generation. An example seed is reported below in Listing below.

Listing 1: Example seed of object generation code.

```

1 def generate_amplifier_bitmap(width=9, height=8, body_color=1,
2     knob_color=2, speaker_color=0, border_color=0):
3
4     # width: Number of horizontal pixels (minimum 3, maximum 30)
5     # height: Number of vertical pixels (minimum 3, maximum 30)
6     # body_color: Amplifier body color (values from 0 to 9)
7     # knob_color: Knob color (values from 0 to 9)
8     # speaker_color: Speaker area color (values from 0 to 9)
9     # border_color: Border color (values from 0 to 9)
10
11    width = max(3, min(30, width))
12    height = max(3, min(30, height))
13    bitmap = [[speaker_color for _ in range(width)] for _ in
14              range(height)]
15    # Draw border
16    for x in range(width):
17        bitmap[0][x] = border_color
18        bitmap[height-1][x] = border_color
19    for y in range(height):
20        bitmap[y][0] = border_color
21        bitmap[y][width-1] = border_color
22    # Draw body (main amp area)
23    for y in range(1, height-2):
24        for x in range(1, width-1):
25            bitmap[y][x] = body_color
26    # Draw knobs (top row, spaced)
27    knob_count = max(2, width//4)
28    for i in range(knob_count):
29        x = 1 + i * ((width-2)//(knob_count-1)) if knob_count>1 else
30             width//2
31        bitmap[1][x] = knob_color
32    # Draw speaker (bottom row, center)

```

```

30     speaker_width = max(1, width//5)
31     speaker_start = (width - speaker_width)//2
32     for x in range(speaker_start, speaker_start+speaker_width):
33         bitmap[height-2][x] = speaker_color
34     return bitmap

```

B.2 Example of Prompt (Version 2)

In Version 2, we refined the seeds to convert the object data identified in Step 2 into a grid-based representation. The gridded objects express analogical relationships through explicit object interactions and transformations, serving as the core components of GIFARC task generation. The improved seeds thus enable the integration of a broader range of analogical information into ARC-style grids.

Below are full prompt examples for Version 2.

User Prompt: Executable Object generation code (Version2)

```

# My instruction:
Look at the given # Word and create a bitmap.
The bitmap must use numbers 0-9, each number standing for a pixel
color.
Strictly follow the # Format shown below.

# Word:
{template}

# Format:

{
    "bitmap": [[[]],
    "pixel_meaning": {
        "0": "Background", ...
    },
    "parameter_desc": {
        "width": "Bitmap width - determines the overall horizontal
size.",
        "height": "Bitmap height - includes any frame, legs, etc.",
        ...
    },
    "function_code": "...",
    "sample_execute_code": "..."
}

# Additional requirements:
- Provide parameterized Python generator code so the bitmap can be
created with various parameters (width, height, etc.) for the
given # word.
- The resulting JSON must include both pixel_meaning
(number-to-meaning map)
and parameter_desc (parameter explanations).
- function_code and sample_execute_code must be valid, runnable
Python.
- function_code must contain only the Python function.
- In sample_execute_code, each result variable must be named
like test_bitmap_(word)_(idx) - e.g. test_bitmap_bed_1,
test_bitmap_bed_2, test_bitmap_bed_3.

```

- Also show example outputs for bitmap sizes from 3×3 up to 30×30 .
- Neither width nor height may ever exceed 30.
- Ensure the code contains no infinite loops.
- Pixel color mapping is fixed as follows:
 0: "Black", 1: "Blue", 2: "Red", 3: "Green", 4: "Yellow",
 5: "Grey", 6: "Pink", 7: "Orange", 8: "Teal", 9: "Maroon".

User Prompt: Generating Executable ARC Task (Version 2)

My Instruction

The object generation code in the following #materials creates a bitmap of the object, mentioned in the function name and returns it. By combining these objects, provide code that implements the interactions and transformations described in #Interaction and Change, with extensive comments throughout. The return value must be JSON, and both input and output must contain detailed comments.

Requirements

1. At least 2 pairs of example problems (input/output)
2. Use of predictable transformation laws or analogies
3. Clear definition of each object's role, interaction, and rule
4. Emphasis on fun/creativity and adjustable difficulty
5. Complex and creative Puzzle
6. *input_bitmap_generation_code* must be executable Python code, when it generate a bitmap, bitmap should be contained in the parameter which has formatted name, (*short description, in 10 char*)_*input_bitmap_(idx)* is the format you should keep, However the name of bitmap should start with alphabet so that it keeps proper python variable name
7. *solution_code* must be executable Python

Creating an ARC-Style Puzzle Problem

- At least 2 pairs of example input-output
- There is a pattern/rule
- The human/AI infers the rule and solves the problem

Detailed Objectives

You *must* use the provided functions to create objects and generate bitmaps with them.

{seed}

Using this, create at least 2–3 pairs of input–output bitmap examples and a puzzle problem with a consistent rule. The rule must be something humans are meant to infer!

```
# Example Result Format
{
    "input_bitmap_generation_code": "... #( this must use object
generate function )",
    "used_concept": "...",
    "solution_code": "def main(input): ..."
}
```

(If possible) Answer generation code, Do not give me a bitmap, just give me the code to generate it. Input bitmap generation code must use object generate function to generate input bitmap.

```
# Problem Story  
{description}  
  
# Required Information  
{story}
```

(e.g.: triangle + pillar, pattern + triangle, etc.) Output rule: (e.g.: change color of overlapping area, only certain pattern positions change, etc.) Bitmap size: (between 10x10 and 30x30)

C Implementation Details

C.1 Model hyperparameters

In pipeline Version 1, we set the temperature to 0 in order to build deterministic datasets. In pipeline Version 2, we set the temperature hyperparameter to approximately 0.2 to produce more diverse outputs. [20]. Note that reasoning models like GPT o-series do not support temperature configuration, so this parameter was not set for those models. For more detailed hyperparameters, refer to Tables 4 and 5.

Table 4: OpenAI API Hyperparameters Verison 1

Process Name	Role	Model Name	Max Tokens	Top p	Temperature
Method Step 1	Data Generator	GPT o1	2,048	1.0	-
Method Step 2	Data Generator	GPT o3-mini	2,048	1.0	-
Method Step 3	Data Generator	GPT o3-mini	2,048	1.0	-
Application	LLM Evaluator	GPT o3-mini	40,000	1.0	-
Application	LLM Solver	GPT-4.1-mini	32,768	1.0	0.0

Table 5: OpenAI API Hyperparameters Version 2

Process Name	Role	Model Name	Max Tokens	Top p	Temperature
Method Step 1	Data Generator	GPT o1	2,048	1.0	-
Method Step 2	Data Generator	GPT o3-mini	2,048	1.0	-
Method Step 3-1	Data Generator	GPT 4.1	2,048	1.0	0.2
Method Step 3-2	Data Generator	GPT 4.1	2,048	1.0	0.2

C.2 Data Generation

GIFARC builds on the data generation process established by the prior work BARC [18]. The original BARC framework generated tasks using manually crafted functions called seeds. A key difference from BARC is that GIFARC employs GPT o1 to extract seven key pieces of information from GIFs in Step 2, which are then used to design tasks in Step 3. In Version 1, generation is based on the `Description`, whereas in Version 2, it utilizes `Pattern_information` and `Object_information`, leveraging the `object_bitmap` seeds constructed in GIFARC. Based on the designed tasks, we use GPT o3-mini to generate code in Step 3 of Version 1 and use GPT 4.1 to generate code in Step 3-1 and Step 3-2 of Version 2 for both the `generate_input` function that creates inputs and the `main` function that produces outputs. In Version 2, the input bitmap is generated using the given `object_bitmap` functions based on the semantic meaning of pixel values, and the output bitmap is produced through the solution code. We create input and output grids and apply a filtering process using this generated code.

The filtering criteria consist of ten conditions adapted from BARC, as shown in Table 6. Input-output grid pairs that meet any of these conditions are filtered out. In GIFARC, creating tasks that incorporate analogies often results in complex code. Therefore, we set a maximum time limit of 300 seconds per task to provide sufficient processing time. Cases requiring more time are considered to be caught in infinite loops and are filtered out as timeout failures. This filtering process aims to enhance the quality of GIFARC-generated tasks.

Version 2 develops code to convert object information extracted in Step 2 into grid representations, thereby enriching tasks with deeper analogical information. These gridified objects serve as seed functions for generating GIFARC tasks where analogies manifest as explicit object interactions or transformations. Specifically, we generate code that can represent objects such as ‘dog’, ‘tree’, ‘human’, and ‘pacman’ extracted in Step 2 as grids through GPT 4.1. By using these codes as seeds, we enable the analogical changes and patterns in GIFARC tasks to emerge more clearly. During task generation, these object-level grid fragments are combined as seed functions to construct input grids containing various objects. The corresponding output grids are generated by applying inter-object interactions and transformations to these input grids. Unlike the BARC seeds used in Version 1,

Table 6: Filtering Conditions

Filtering Condition	Meaning
Non-Deterministic	The output grid differ when the same transformation is applied multiple times to the same input.
Non Color Invariant	1. Cases where the transformation function itself fails during the color permutation process. 2. Either the permuted grid or the input grid is not well-formed. 3. The results differ even when only the colors are changed.
Identity	The input and output are completely identical.
Non-Well Formed Output	The transformation result is not well-formed (i.e., not a 2D list with equal row lengths and integer values between 0-9).
Black Output	The output consists entirely of 0s (black pixels).
Timeout	Overall time limit exceeded.
Non-Well Formed Input	The input is not well-formed (i.e., not a 2D list with equal row lengths and integer values between 0-9).
Duplicate Input	The generated input is duplicated with existing ones.

Version 2 seeds are semantically structured, generating tasks with clearer and more interpretable analogical patterns. Finally, we apply the same 300-second timeout and filtering pipeline as in Version 1 to ensure that only efficiently and reliably executable tasks are retained.

C.3 Application

We conducted experiments on Section 5.1 which is “Verification Changes in Reasoning Steps” and Section 5.2 which is “Alignment between Context-Generated and Ground-Truth Analogies”. For these experiments, we developed various prompt context conditions: full description, description without analogy, and description without analogy and without solution.

The full description was generated using o3-mini to enhance the analogies in both the α and ϕ of the GIFARC dataset. To prevent the inclusion of irrelevant analogies, we performed manual refinement of the generated content. For the description without analogy, we utilized o3-mini to remove analogy-related information from the full description. Subsequently, we converted any remaining analogical elements in both α and ϕ to grid-level representations. The description without analogy and without solution used the same analogical framework as the description without analogy. Across all kinds of descriptions, the grid information remained consistent.

In our experiments examining Section 5.1 and Section 5.2, we employed both the full description and the analogy-removed description. The full description was identical to that used in the Section 5.1 application. The analogy-removed description was a refined version of the description without analogy used in the Section 5.1 experiment, where we completely eliminated any remaining analogical information by either removing it entirely or converting it to grid-level representations, thus ensuring the complete absence of analogical content.

D Guidelines for human evaluators assessing the given tasks

In the section 5 “Application”, we conducted human evaluation on 12 tasks. Three experts familiar with ARC were asked to describe the analogies for 12 GIFARC tasks. To obtain responses of sufficient quality for our experiment, we provided minimal guidelines. The following is a brief outline of the guidelines provided to the human evaluators.

Guidelines for Human Evaluators

Please write the pattern analogy in English sentences for the provided 13 tasks (input-output pairs), referring to the content below:

1. What the provided input-output pairs are visualizing
2. What kind of analogy the pattern changing from input to output can be related to or expressed as
3. What the colors or object shapes shown in the input/output symbolize or represent
4. Please write the expected rules if you don't know the task's analogy.
Ex-1) - It appears to be spreading from the center.
Ex-2) - The movement seems to change randomly and the colors are changing.

[Working Example]
{Example task}

Analogy: In this task the grid represents a scene where small buds (colored gray: 5) transform into blooming flowers. The transformation simulates the process of blooming: each bud expands its presence by turning its neighboring cells (up, down, left, right) into petal cells (colored gray: 5) while the original bud cell becomes the highlight of the bloom (colored yellow: 4).

[GIFARC Task 1]
{Input-Output Pairs}

:

[GIFARC Task 12]
{Input-Output Pairs}

Based on this guideline, participants described the analogies for the given GIFARC tasks. The information collected contained only content about the analogies, and no personal or sensitive information was gathered.

E Example Visualizations of GIFARC

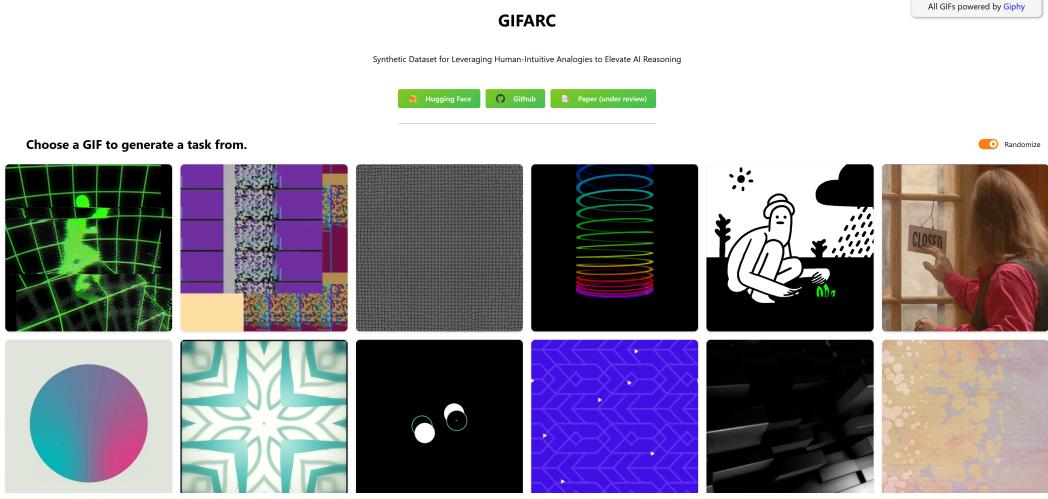


Figure 6: Landing Page of GIFARC Visualization website. It is publicly released at <https://gifarc.vercel.app/>

All 10,000 tasks generated with GIFARC are visualized in <https://gifarc.vercel.app/>. Following are example visualizations of GIFARC-generated ARC-style tasks of each task type.

E.1 Example Task of Type 1 - Rotational Symmetry & Perspective Spin

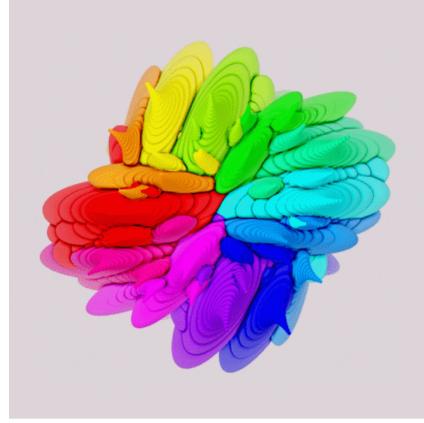


Figure 7: Snapshot of GIF used for generating task 962.

- Concepts : fractal expansions, symmetrical pulsations, radial transformations, iterative growth
- Description : In the input, you will see a sequence of grids (frames) on a black background. Each grid depicts: - A large circular region in the center, with multiple concentric rings and radial lines diverging outward. - A stacked-curve fractal anchored at the bottom-left corner. - A set of spire fractals anchored at the bottom-right corner. As you move from one frame to the next in the input, these elements undergo iterative transformations: - The radial lines repeatedly shift in thickness and visual intensity, while preserving rotational symmetry. - The concentric rings in the circular region pulsate by alternately expanding and contracting. - The stacked-curve fractal on the bottom-left changes its curve density, fractally adding or removing segments. - The right-side spire fractals expand and contract in repeated vertical segments. Your task is to replicate and apply these transformations for the entire sequence, and produce the final frame of the animation as the output:

1) Ensure the anchoring of the fractals at the bottom edge remains the same. 2) Preserve the radial symmetry of the central circular region and its common center point. 3) For each pulsation step in the input, magnify or contract the rings, lines, and fractals accordingly. 4) Continue until the final pulsation step is reached. That final state is your output grid. The essential principle is that symmetry-based fractal transformations and repeated cyclical expansions/contractions produce the overall pulsating effect. By reflecting each iterative change step by step, you reconstruct the final pulsating pattern visible at the end of the sequence.

- Full web view is available at <https://gifarc.vercel.app/task/962>.

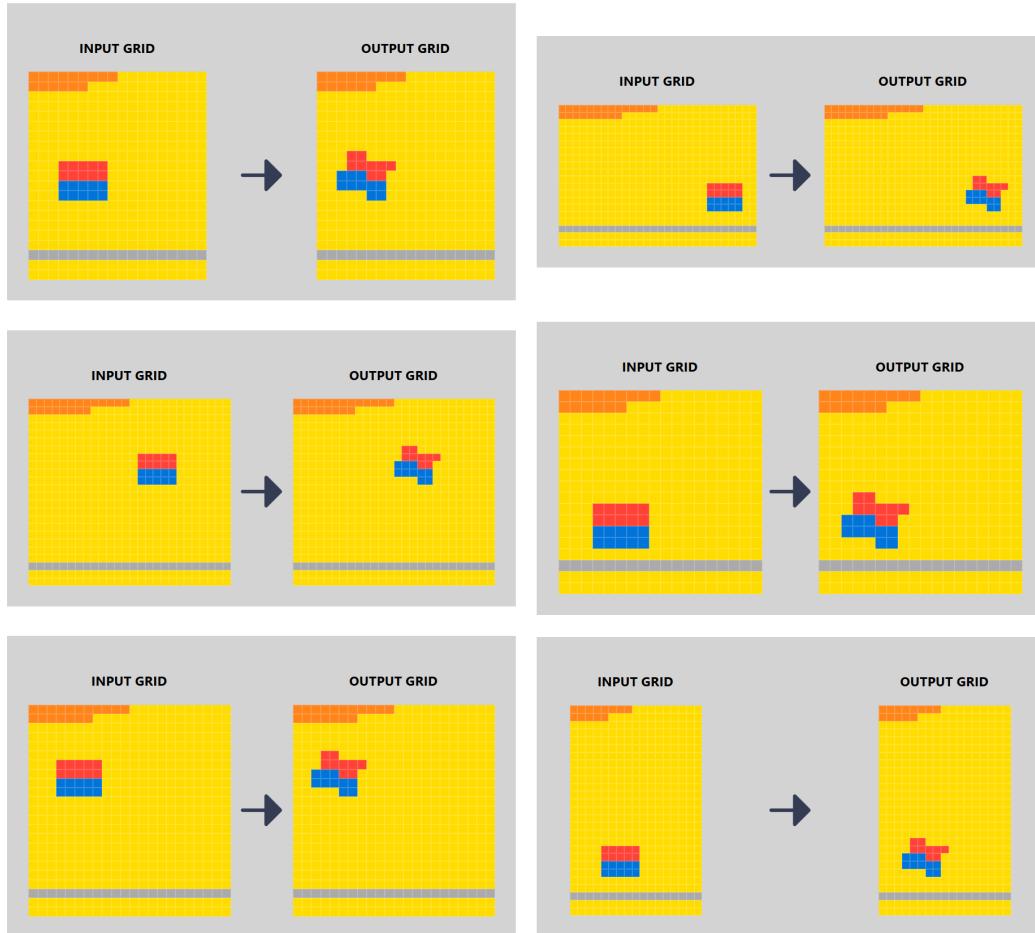


Figure 8: GIFARC-generated task 962.

Listing 2: Python code solution for task 962.

```

1  from common import *
2  import numpy as np
3  import random
4
5  # concepts:
6  # rotational motion, center-of-mass shifting, multi-step
7  # transformation, static background
8
9  # description:
10 # The input grid shows a paved surface (Color.YELLOW) with static
11 # features: a fence (Color.GREY) drawn near the bottom and
12 # mountains (Color.ORANGE) at the top.
13 # A dynamic object-a wheelchair with its occupant-is represented by a
14 # 4x5 sprite: the top two rows are the occupant (Color.RED) and the
15 # bottom two rows the wheelchair (Color.BLUE).
16 # The wheelchair rotates around an axle (computed as the average
17 # position of the blue pixels) so that the occupant's mass shifts
18 # further off support.
19 # The transformation in main erases the original dynamic pixels and
20 # re-blits them using an integer-approximated clockwise rotation
21 # (about 22 degrees) around the axle, while leaving the static
22 # background unchanged.
23
24 def main(input_grid):
25     out=input_grid.copy()
26     dyn=[]
27     for i in range(len(input_grid)):
28         for j in range(len(input_grid[0])):
29             if input_grid[i,j] in [Color.BLUE,Color.RED]:
30                 dyn.append((i,j,input_grid[i,j]))
31                 out[i,j]=Color.YELLOW
32     if not dyn: return out
33     ax_i=sum(p[0] for p in dyn)//len(dyn)
34     ax_j=sum(p[1] for p in dyn)//len(dyn)
35     cos_val, sin_val = 92,38
36     for i,j,col in dyn:
37         di=i-ax_i; dj=j-ax_j
38         ni=ax_i+((cos_val*di+sin_val*dj)//100)
39         nj=ax_j+((-sin_val*di+cos_val*dj)//100)
40         if 0<=ni<len(out) and 0<=nj<len(out[0]): out[ni,nj]=col
41     return out
42
43 def generate_input():
44     H,W=random.randint(15,30),random.randint(15,30)
45     grid=np.full((H,W),Color.YELLOW)
46     for j in range(W): grid[H-3,j]=Color.GREY
47     for i in range(2):
48         for j in range(W//(2+i)):
49             grid[i,j]=Color.ORANGE
50     sprite=np.full((4,5),Color.BLACK)
51     sprite[:2,:]=Color.RED; sprite[2:,:]=Color.BLUE
52     x,y=random_free_location_for_sprite(grid,sprite,background=
53                                         Color.YELLOW)
54     blit_sprite(grid,sprite,x,y)
55     return grid

```

E.2 Example Task of Type 2 - Kaleidoscope & Symmetry Expansion

- Concepts : kaleidoscopic transformation, mirrored folding, cyclical animation, vertical symmetry
- Description : In this puzzle, the input is a multi-frame sequence depicting a pale-yellow corridor-like structure with a central rectangular opening and mirrored arches aligned along a vertical axis of symmetry. Over several frames, the corridor's walls fold inward and outward in a smooth, kaleidoscopic motion, always returning to the initial configuration. The output is the full cycle of transformations repeated in a loop, preserving the corridor's pale-yellow hue and central frame while continuously reflecting the mirrored arch patterns around the vertical axis. This cyclical approach showcases the corridor morphing into itself, completing each transformation stage and then re-starting at the original symmetrical configuration.

• Full web view is available at <https://gifarc.vercel.app/task/7442>.

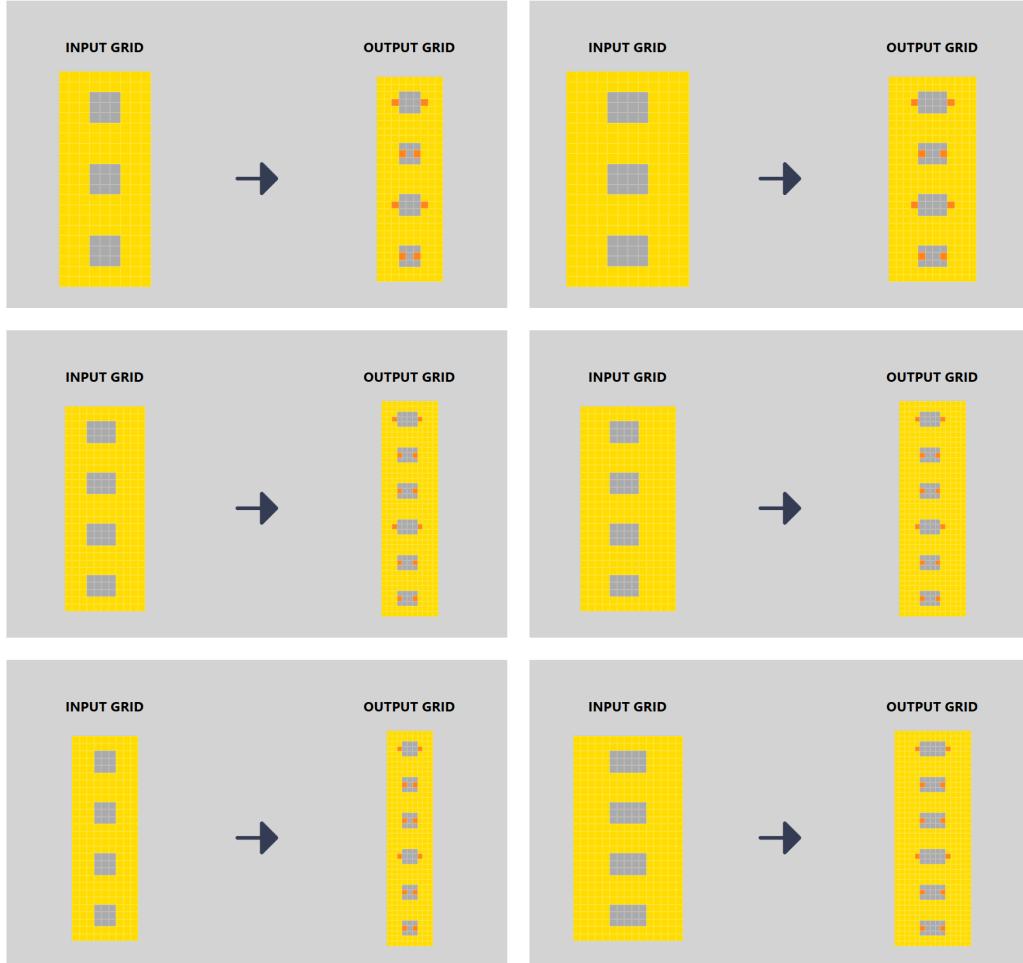


Figure 9: GIFARC-generated task 7442.

E.3 Example Task of Type 3 - Pendulum & Pivot Rotation

- Concepts : cyclical rotation, pivot anchoring, animation frames
- Description : Inspired by previous examples that repeated or translated objects across multiple frames, in this puzzle the input is a single snapshot showing a blue curved shape on a black background. For the output, construct a multi-frame grid depicting the same shape completing a full 360-degree rotation around a central pivot. Each frame rotates the shape by a fixed angle (e.g., 15° increments), and the frames are arranged in a 3x3 or 4x4 grid until the shape returns to its original orientation. The black background and the blue color remain consistent across frames, and the shape never goes outside the canvas boundary. This cyclical rotation from start to finish, then repeating, is the puzzle's core concept.
- Full web view is available at <https://gifarc.vercel.app/task/1037>.

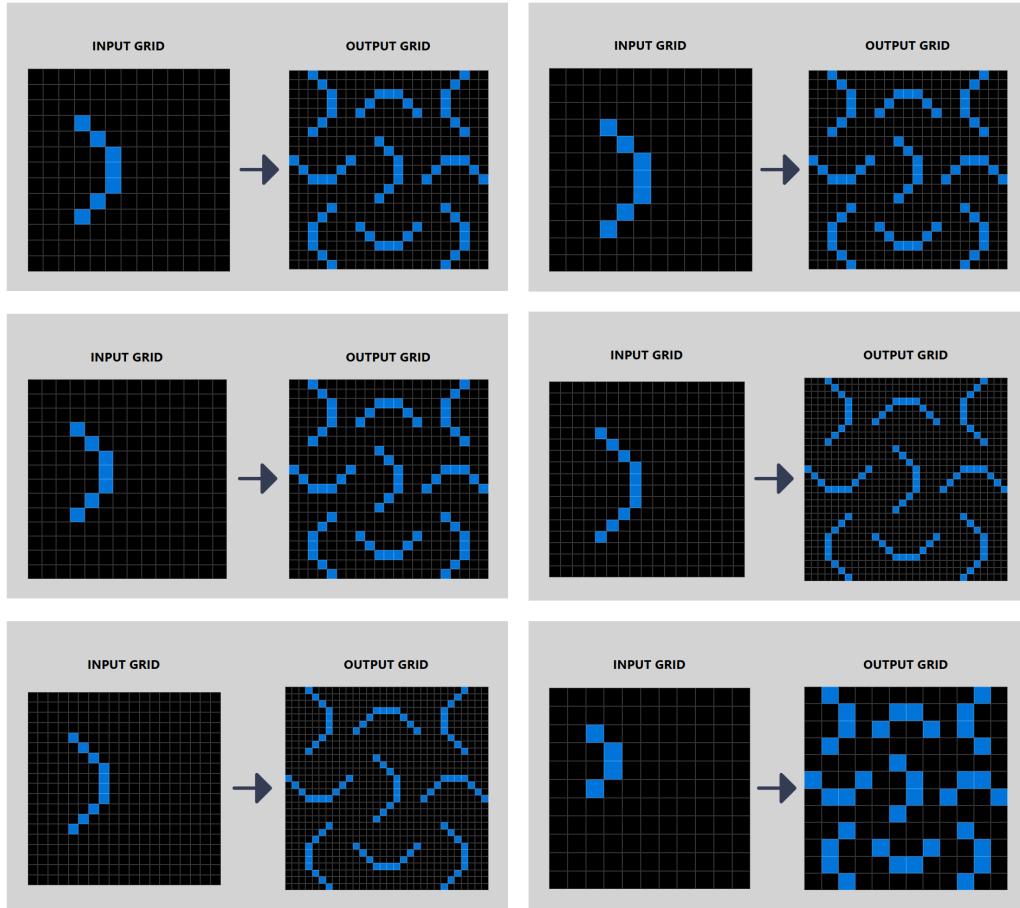


Figure 10: GIFARC-generated task 1037.

E.4 Example Task of Type 4 - Walking & Forward Locomotion

- Concepts : cyclical motion, repetition, background constraint
- Description : In the input you will see an animation (or multiple frames) of pink human-shaped figures in a light-blue background. White wavy lines span the background and shift subtly as the pink figures travel diagonally from the top-left to the bottom-right. When a pink figure leaves the bottom-right edge, it reappears at the top-left edge at the same vertical position, making a continuous loop. The translucent blue overlay patterns remain visually consistent on top of the background while the white lines gently oscillate as the figures pass. To produce the output, replicate this cyclical motion fully at least once, preserving continuity. The final frame must line up so that if we start again from there, the pink figures continue repeating the same path through the light-blue background in a seamless loop.
- Full web view is available at <https://gifarc.vercel.app/task/2767>.

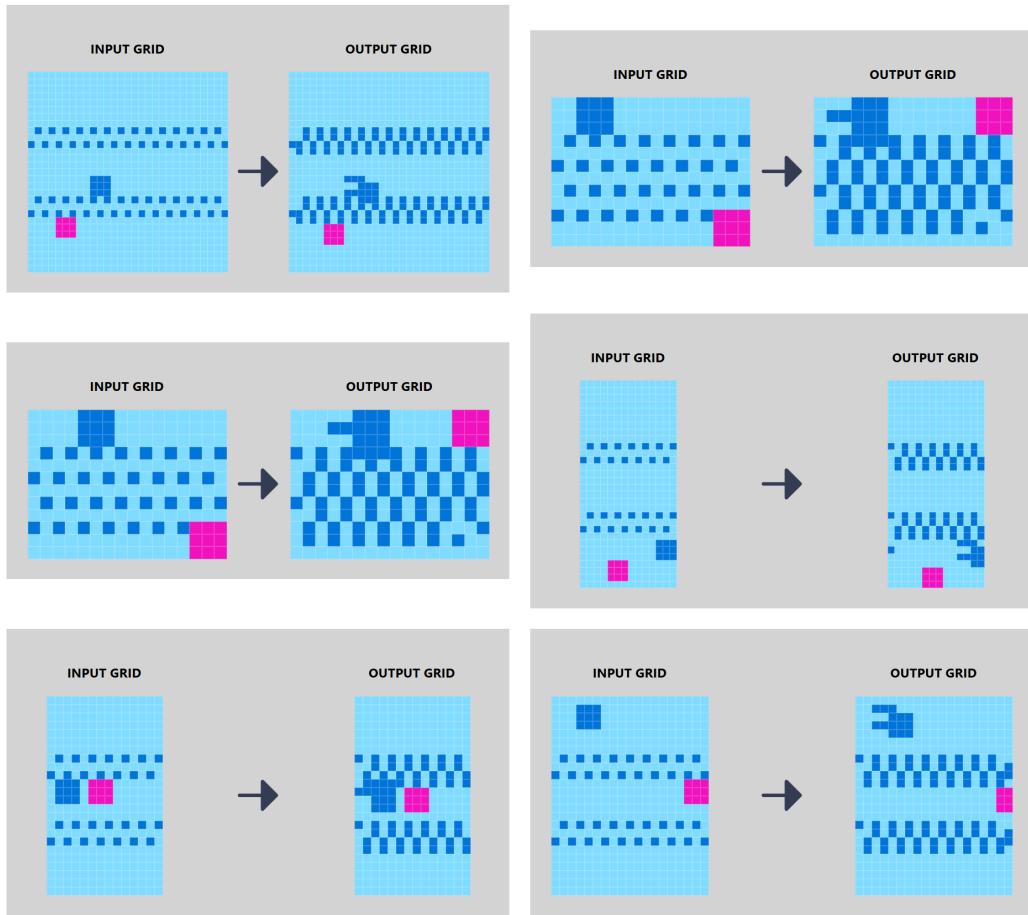


Figure 11: GIFARC-generated task 2767.

E.5 Example Task of Type 5 - Falling & Stacking Blocks

- Concepts : handheld device, screen area, gravity, falling blocks, stacking, looping animation
 - Description : In the input, you will see a pink background with a grey handheld gaming device in the center. The device has a bright green rectangle acting as its screen, with a d-pad on the left and action buttons on the right. Initially, you will see Tetris-like blocks of various shapes hovering above the upper edge of this green screen area. To create the output, you must simulate the blocks falling downward into the screen, stacking on top of one another or on previously fallen blocks. Keep each block fully within the green screen boundary; if a falling block collides with another block or the bottom of the screen, it must stop. Once the screen is completely filled to the top, the stack disappears, effectively resetting the screen to empty and repeating the cycle. In other words, show the progression from “empty screen” → “stacked blocks” → “reset,” producing a looping effect of the falling Tetris-like blocks within the green screen.
 - Full web view is available at <https://gifarc.vercel.app/task/556>.

The figure illustrates three examples of convolutional operations with a stride of 2:

- Example 1 (Top Left):** An input grid of size 4x4 (grey) with a pink border. A green 2x2 kernel is applied. The output grid (pink border) has a green center and a red corner, indicating a non-zero output from the top-left input unit.
- Example 2 (Top Right):** An input grid of size 4x4 (grey) with a pink border. A green 2x2 kernel is applied. The output grid (pink border) has a green center and a red corner, indicating a non-zero output from the top-left input unit.
- Example 3 (Bottom Left):** An input grid of size 4x4 (grey) with a pink border. A green 2x2 kernel is applied. The output grid (pink border) has a green center and a red corner, indicating a non-zero output from the top-left input unit.
- Example 4 (Bottom Right):** An input grid of size 4x4 (grey) with a pink border. A green 2x2 kernel is applied. The output grid (pink border) is entirely green, indicating no output from the top-left input unit due to zero padding.

Figure 12: GIFARC-generated task 556.

E.6 Example Task of Type 6 - Periodic Movement & Horizontal Loop

- Concepts : cyclical motion, frame-based animation, horizontal repetition, background preservation
- Description : In the input you will see several frames depicting a single brown triceratops, side-profile, walking across a beige background. The frames show a gradual shift in the triceratops' position from left to right while its legs cycle through a walking gait. To create the output, stack or tile these frames into a continuous strip (or series) that repeats seamlessly. The background stays beige and does not change between frames, and the triceratops remains the same brown color and orientation. The final output shows a horizontally looping animation strip where the last frame transitions smoothly back into the first, preserving the cyclical walking motion and the uniform background.

• Full web view is available at <https://gifarc.vercel.app/task/3690>.

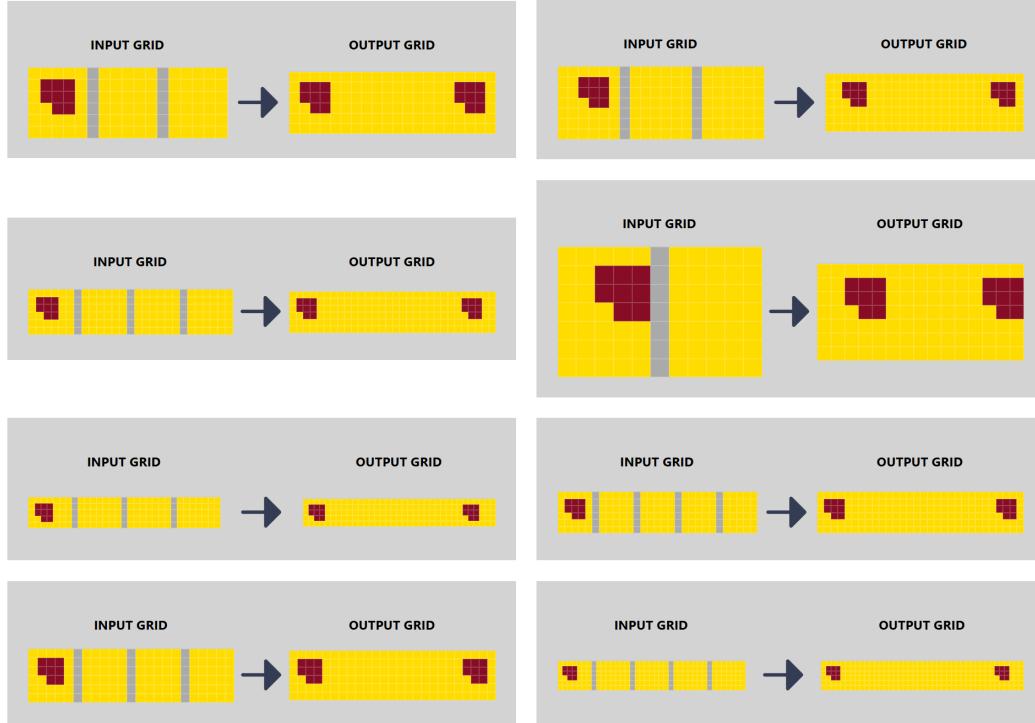


Figure 13: GIFARC-generated task 3690.

E.7 Example Task of Type 7 - Color Flicker & Blinking

- Concepts : cyclical color flicker, hinge constraints, multi-frame puzzle, looping animation
- Description : The input consists of multiple frames showing a hand-drawn scene: a large circular shape with concentric scribbles/colors in its center, topped by a smaller hinged circle, with various scribbles at the right (ladder-like), left (darker scribble), and near the hinge (green). Over time, the concentric scribbles in the center circle flicker through different color states while all objects remain in the same relative positions.
- Full web view is available at <https://gifarct.vercel.app/task/385>.

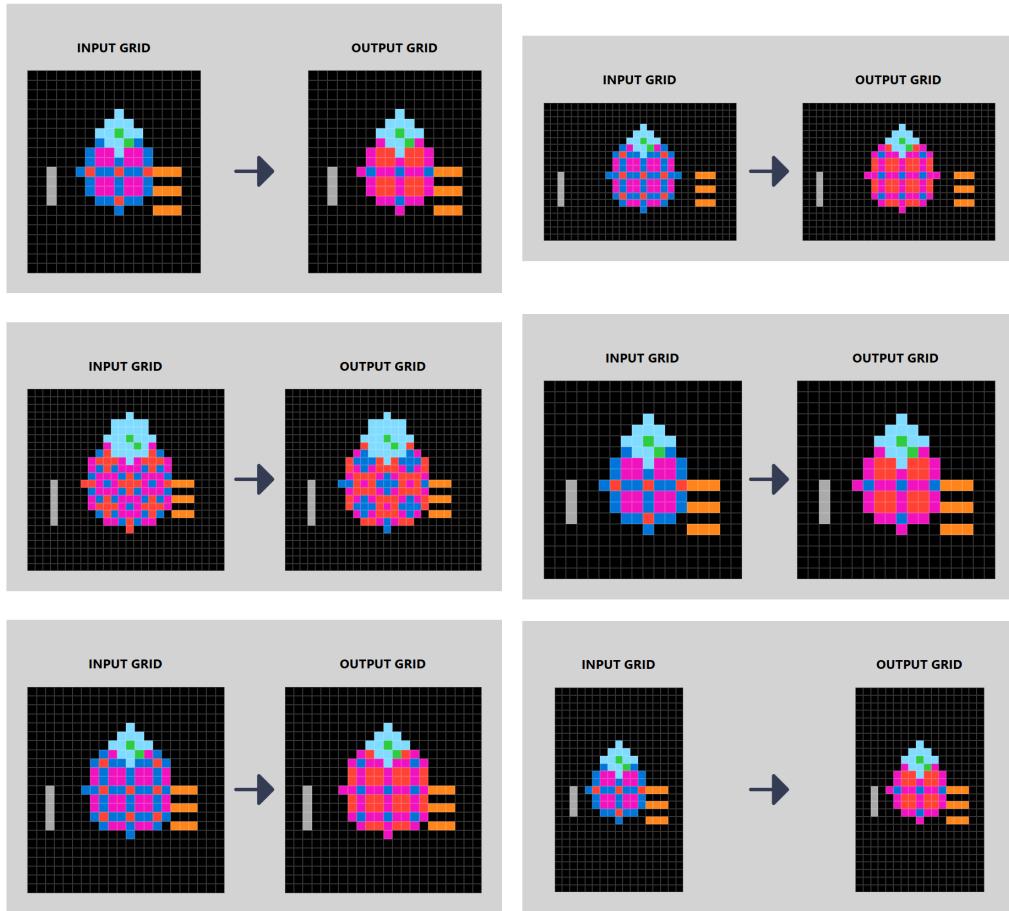


Figure 14: GIFARC-generated task 385.

E.8 Example Task of Type 8 - Gradient & Layered Color Changes

- Concepts : radial color cycling, symmetry, continuous transformations, layering
- Description : In the input, you will see a single circular shape on a purely black background. Inside the circle is a radial, symmetrical pattern composed of multiple colors. Each concentric ring of the circle shifts colors in a periodic cycle, from an inner ring (earliest phase) to an outer ring (later phase), repeatedly. The circle retains its size and position, and the background remains black.
- Full web view is available at <https://gifarc.vercel.app/task/117>.

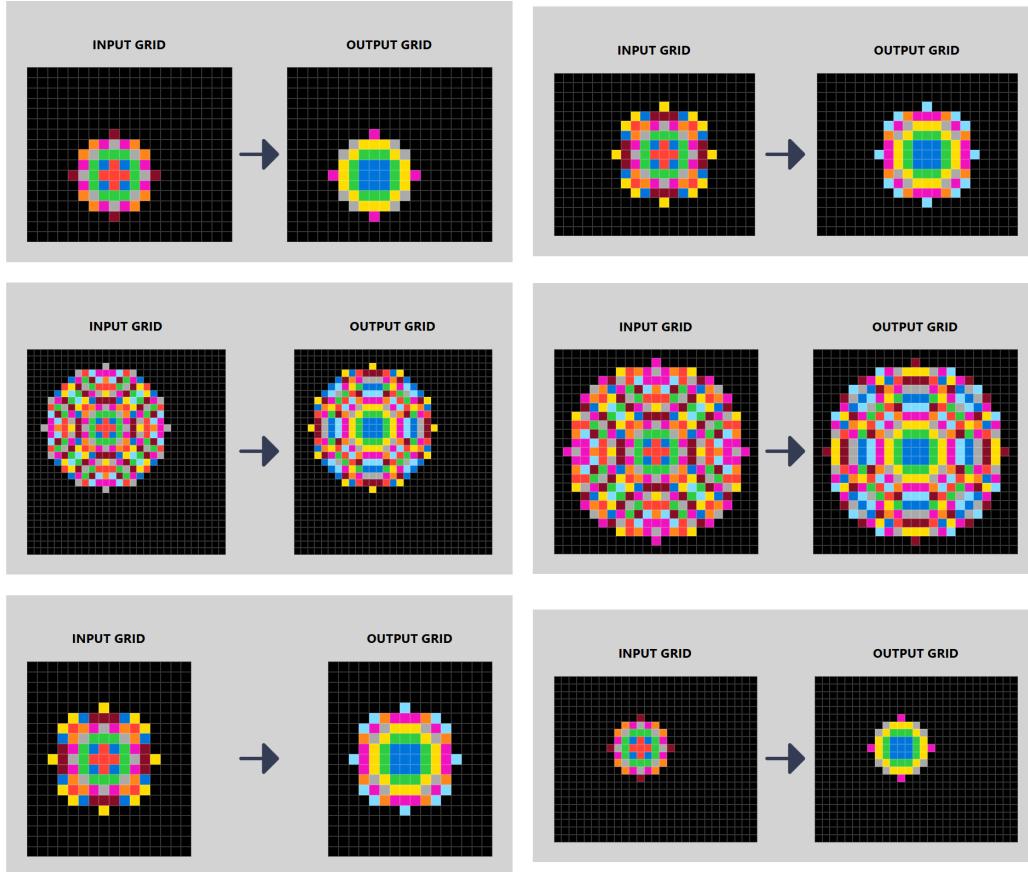


Figure 15: GIFARC-generated task 117.

E.9 Example Task of Type 9 - Glitch & Breaking Patterns

- Concepts : glitch patterns, color flickering, repetitive animation, partial stability
- Description : In the input you will see multiple horizontal bars stacked vertically over a black background. Each bar is composed of green, purple, and yellow pixels arranged in rectangular stripes. Some small clusters of yellow pixels flicker like digital noise, intensifying toward the center and right side of each bar. The bars remain in the same positions throughout the sequence.
- Full web view is available at <https://gifarc.vercel.app/task/410>.

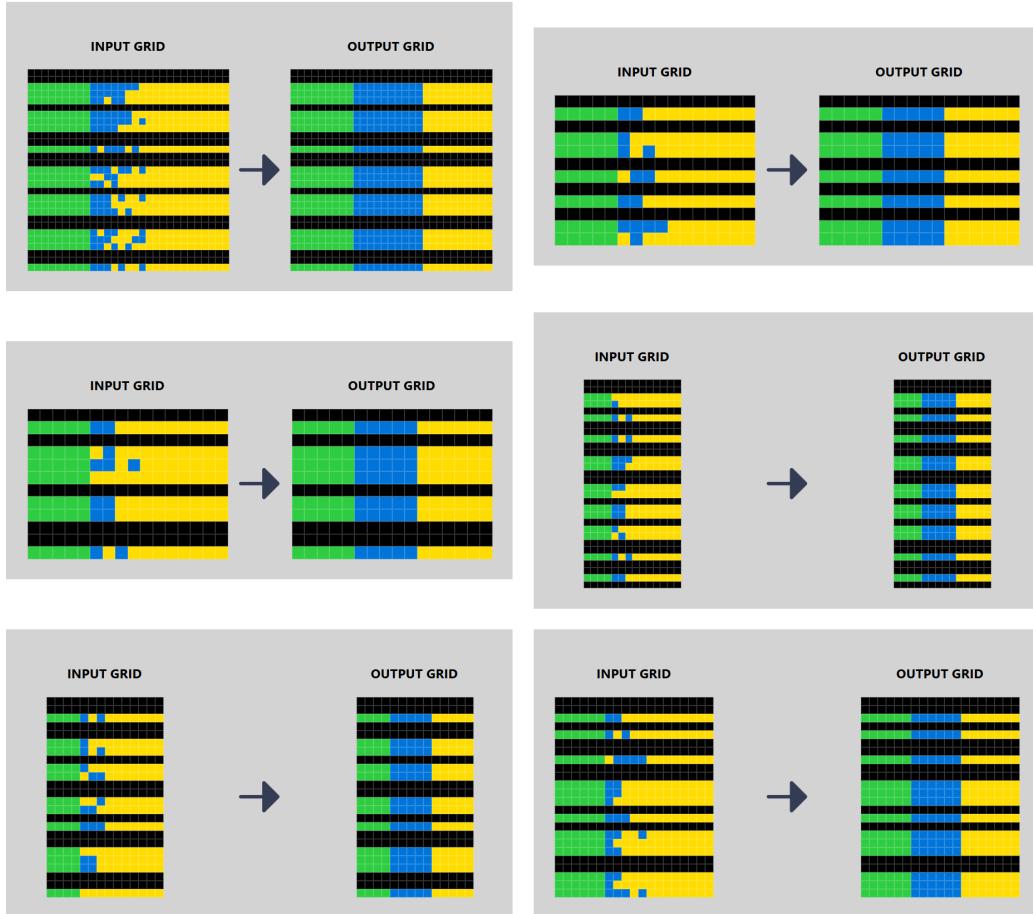


Figure 16: GIFARC-generated task 410.

E.10 Example Task of Type 10 - Wave & Diagonal Flow

- Concepts : cyclical pulsing, polygonal wave propagation, time-lapse transformation
- Description : In the input, you will receive a sequence of frames representing a dark polygonal surface with green glowing points at each polygon vertex. Each frame gradually shifts the brightness of the polygons in wave-like motions, causing shimmering highlights that travel along the connected polygon faces. The initial frame and the final frame share the same luminous intensity, forming a seamless loop of pulsing and returning. Your task: 1. Identify the green points at the vertices, which remain spatially fixed throughout the sequence. 2. Track the wave-like shimmering across the polygon faces. 3. Ensure that the final frame's brightness pattern matches the initial frame's, capturing the cyclical nature of the pulse. 4. Output the entire transformation as a collection of frames (or a metadata structure) that visually loops back to the start. The key principles are:
 - The polygon mesh arrangement does not change; only the luminous intensity on the polygon faces and green vertices fluctuates.
 - Shimmering waves spread across the surface repeatedly, then recede, creating a looping timeline.
 - The result is a time-lapse style output in which the final state seamlessly resets to the first state, preserving the thought of continuous pulsing.
- Full web view is available at <https://gifarc.vercel.app/task/202>.

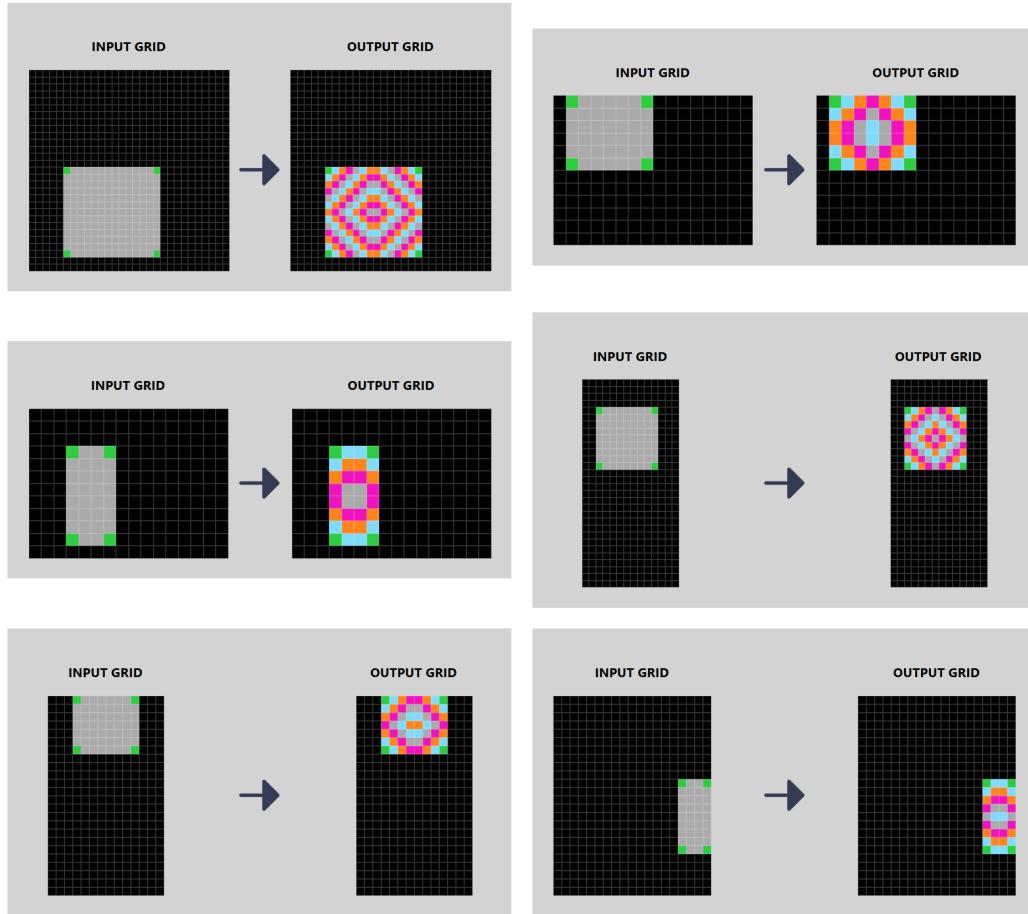


Figure 17: GIFARC-generated task 202.

E.11 Example Task of Type 11 - Gravity & Liquid Flow

- Concepts : time-lapse, incremental changes, layering, chart visualization
- Description : In the input you will see multiple "frames" of a chart with vertical bars and a pink line, each frame on a separate grid. The black background, rectangular grid, and bounding box remain constant across these frames. In each subsequent frame, the vertical bars shift in height and flicker in intensity, while the pink line oscillates up and down with a net upward trend.
- Full web view is available at <https://gifarc.vercel.app/task/1413>.

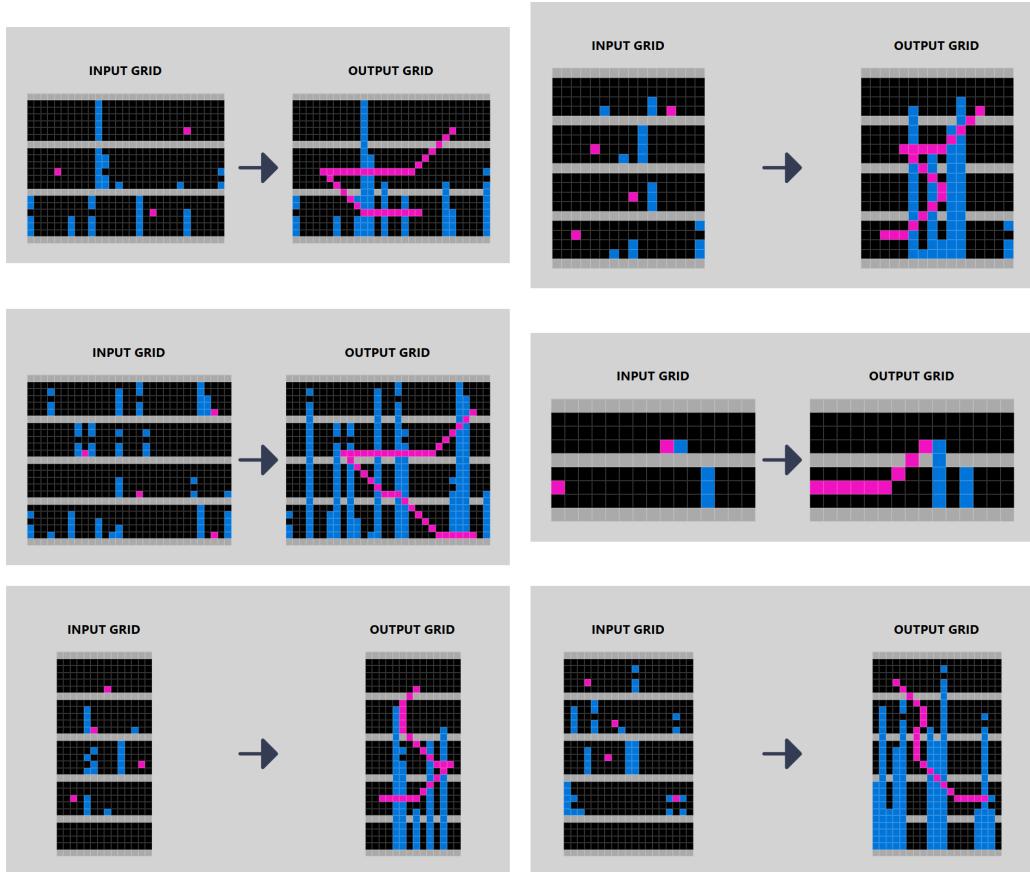


Figure 18: GIFARC-generated task 1413.

E.12 Example Task of Type 12 - Slow Environmental Change

- Concepts : time-lapse, incremental changes, layering, chart visualization
- Description : In the input you will see multiple "frames" of a chart with vertical bars and a pink line, each frame on a separate grid. The black background, rectangular grid, and bounding box remain constant across these frames. In each subsequent frame, the vertical bars shift in height and flicker in intensity, while the pink line oscillates up and down with a net upward trend.
- Full web view is available at <https://gifarc.vercel.app/task/2500>.

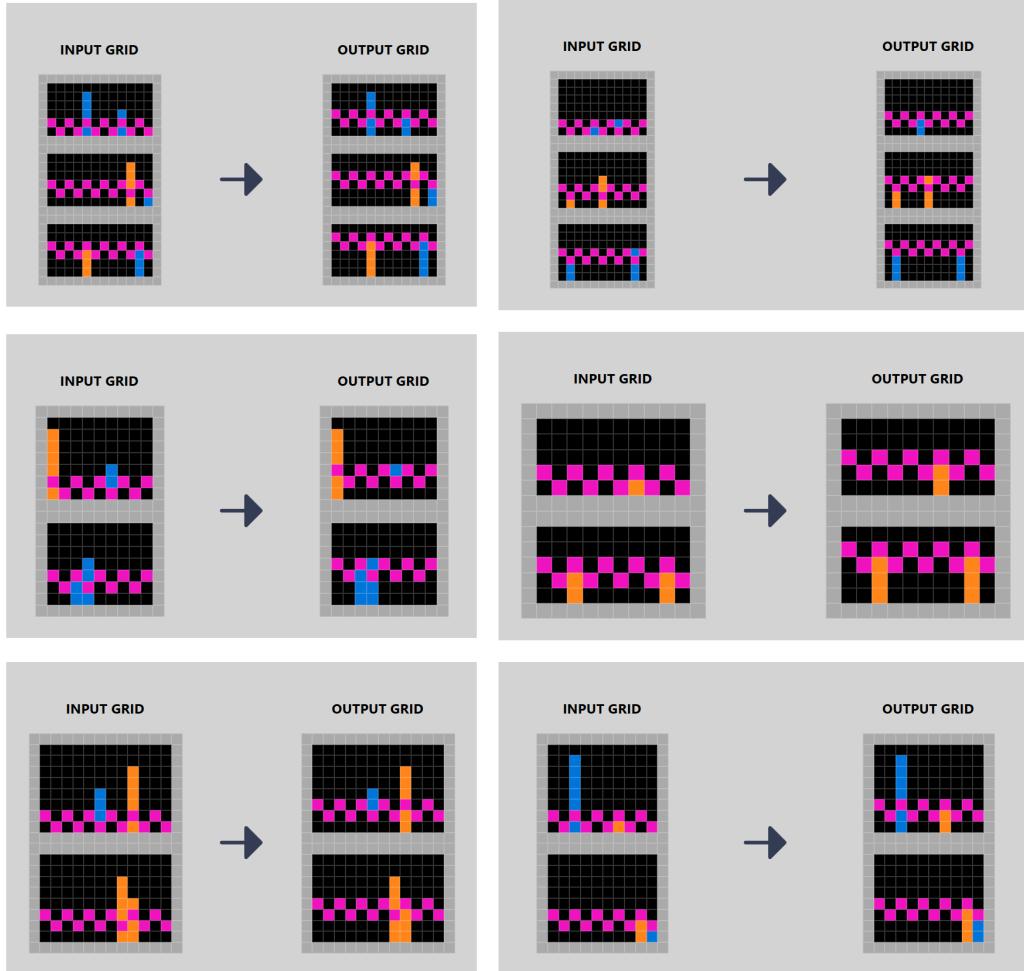


Figure 19: GIFARC-generated task 2500.

E.13 Example Task of Type 13 - Scaling Burst & Shape Morphing

- Concepts : repetitive scaling, cyclical emergence, timed transformations
- Description : In the input you will see a pink background with one or more black rings near the center. Each ring should continuously expand outward from the center until it goes off the grid, at which point it disappears. Meanwhile, whenever an existing ring crosses halfway toward the border, a new smaller ring is spawned in the center. This process loops indefinitely—old rings vanish at the boundary, and new rings keep forming in the center. The output should capture this entire cycle, from the initial state to the eventual large rings that disappear, and newly formed rings that repeat the pattern. In a static puzzle context, illustrate at least one full cycle of rings moving outward and disappearing, while new rings appear at the center.
- Full web view is available at <https://gifarc.vercel.app/task/651>.

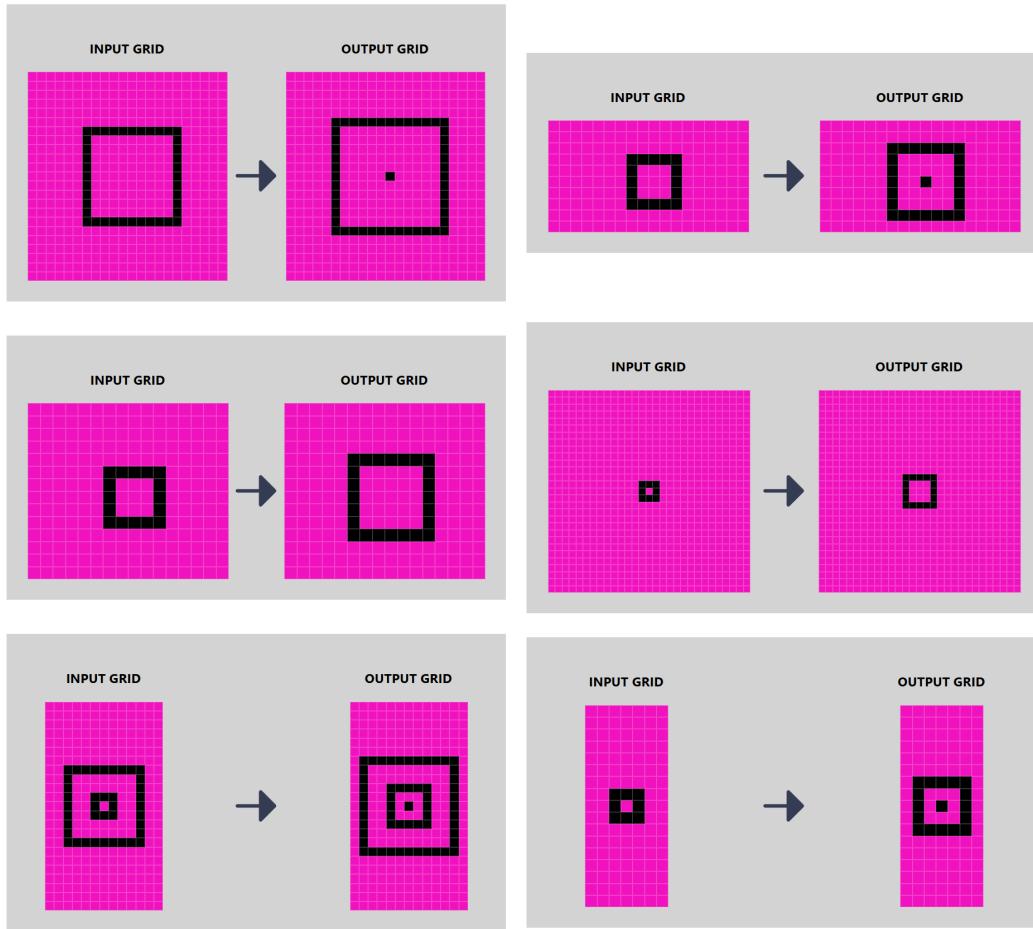


Figure 20: GIFARC-generated task 651.

E.14 Example Task of Type 14 - Attach/Detach Clusters

- Concepts : multi-frame transformation, attachment/detachment, motion, object grouping
- Description : The input represents a time-sequence (e.g., multiple "frames") showing a suited host seated at a desk with several static background objects (gift bag, city backdrop, building spire). The host reaches for a black mug, and we see that the tie is inadvertently attached to the mug (one frame shows the tie moving upward with the mug). In subsequent frames, the host detaches the tie from the mug and returns both to their normal positions. The core principle to illustrate is that if two objects are "attached" they move together until they become detached. Your output is the final frame where the tie is no longer stuck to the mug. The puzzle solution must show how the tie becomes free again, returning to its intended resting position, while the mug is back on the desk.
- Full web view is available at <https://gifarc.vercel.app/task/7128>.

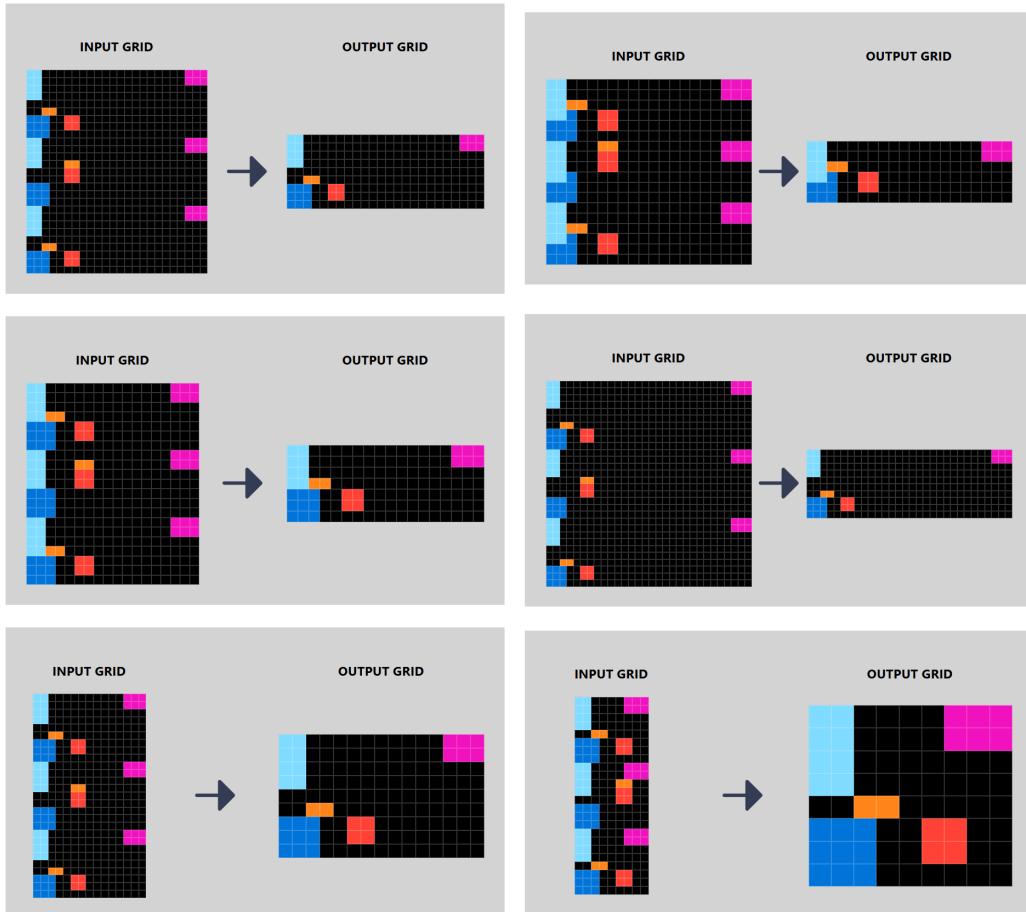


Figure 21: GIFARC-generated task 7128.

E.15 Example Task of Type 15 - Layer Separation & Merging

- Concepts : color gradient, layered squares, temporal progression, partial transparency
 - Description : In the input you will see a multi-frame sequence of an overlapping grid of squares starting in dark blues at the top, transitioning gradually to lighter, more yellow tones at the bottom. The squares remain in a consistent arrangement, but their colors shift slightly from frame to frame. Overlaps create partial transparency effects that slightly modify the underlying tones. To produce the output, you must show the final frame in which the color gradient transitions all the way to lighter shades near the bottom. Capture the entire transformation by preserving the arrangement of squares, ensuring that no abrupt color changes occur between consecutive frames, and maintaining a seamless progression of hues through the final arrangement.
 - Full web view is available at <https://gifarc.vercel.app/task/214>.

- Full web view is available at <https://gifarc.vercel.app/task/214>.

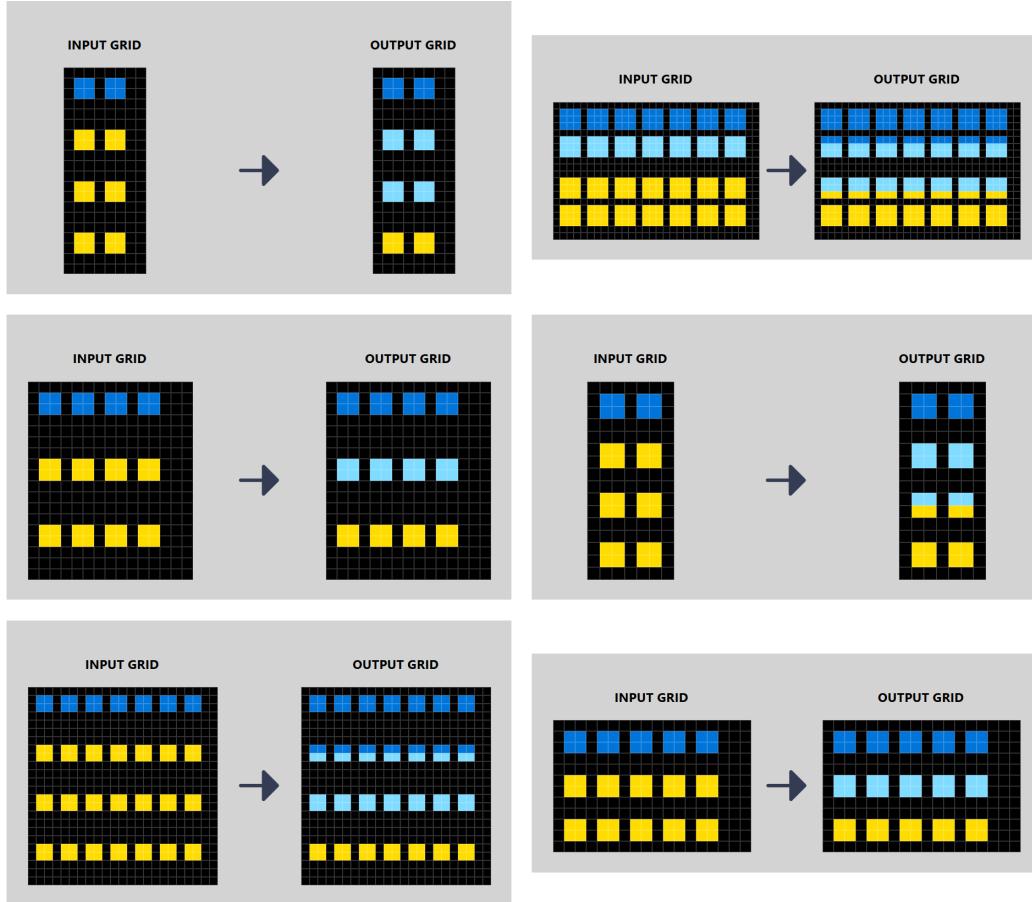


Figure 22: GIFARC-generated task 214.

E.16 Example Task of Type 16 - Text & Punctuation Transformation

- Concepts : frame-based animation, looping, minimal movement, static overlay
- Description : In this puzzle, the input consists of multiple frames showing a black background with a rectangular banner. On the left side of the banner is the static text "I'm waiting...", and on the right side is a small clock icon that subtly changes appearance from frame to frame. To produce the output: 1. Keep the banner and text exactly the same in each frame, preserving their positions. 2. Continuously cycle the clock frames in a loop so that it appears to be animating. 3. The clock must remain contained within the banner boundary throughout all frames. 4. The final output (or sequence of frames) should repeat the frames of the clock's movement in an endless loop, preserving the overall layout within each frame.

• Full web view is available at <https://gifarc.vercel.app/task/2222>.



Figure 23: GIFARC-generated task 2222.

E.17 Example Task of Type 17 - Minimal Motion Overlay

- Concepts : multi-frame static verification, scene consistency, no-change detection
- Description : In the input, you will see several frames that together depict a stylized scene: - A top sphere positioned at the upper center - A curved band placed directly below the sphere - Two large symmetrical side shapes in yellow - A vertical pink line in the center - A signature text in the bottom right corner These elements are repeated exactly across all frames with no movement or color transitions. To make the output, you must check if all frames are truly identical. If they are, return a single-frame grid replicating the scene exactly. If there is any discrepancy (in shape, color, or position) among the frames, then the output should be a black canvas of the same size, indicating the scenes are not perfectly static.

• Full web view is available at <https://gifarc.vercel.app/task/7262>.

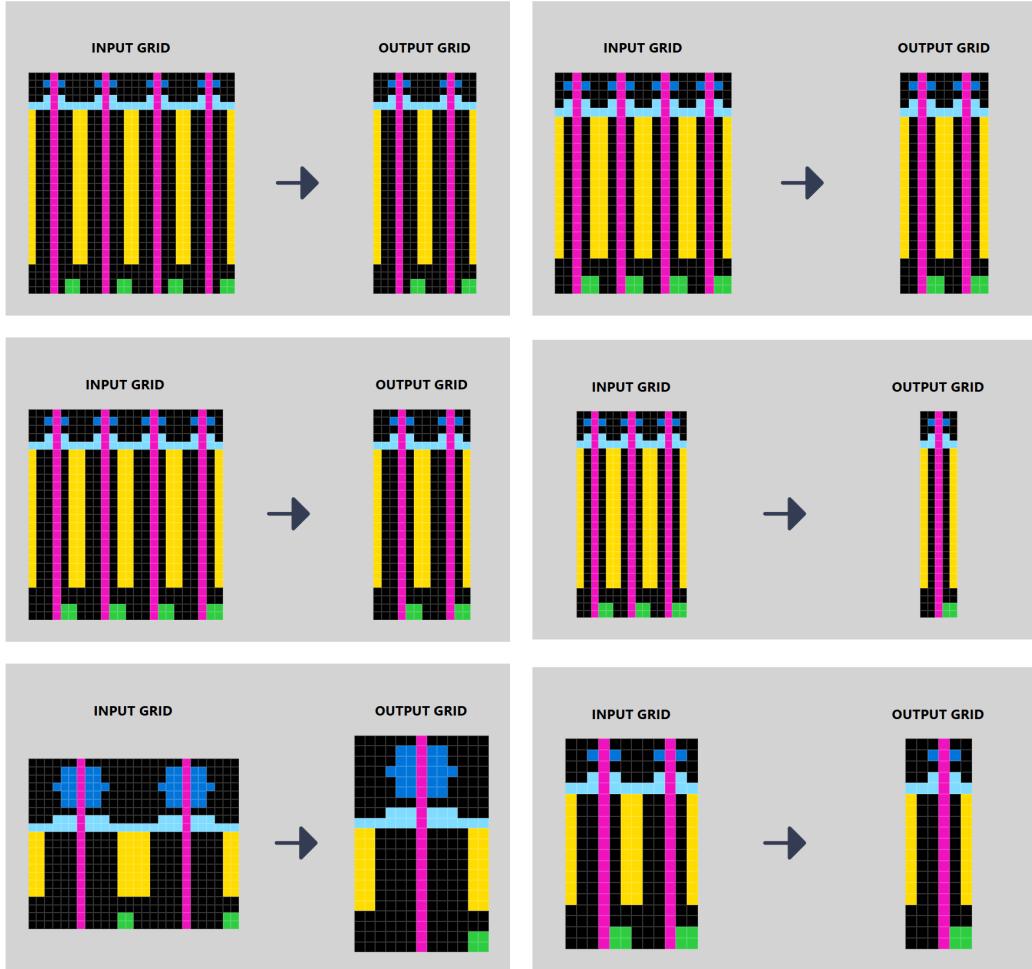


Figure 24: GIFARC-generated task 7262.

E.18 Example Task of Type 18 - Static Verification & No Change

- Concepts : multi-frame static verification, scene consistency, no-change detection
- Description : In the input, you will see several frames that together depict a stylized scene: - A top sphere positioned at the upper center - A curved band placed directly below the sphere - Two large symmetrical side shapes in yellow - A vertical pink line in the center - A signature text in the bottom right corner These elements are repeated exactly across all frames with no movement or color transitions. To make the output, you must check if all frames are truly identical. If they are, return a single-frame grid replicating the scene exactly. If there is any discrepancy (in shape, color, or position) among the frames, then the output should be a black canvas of the same size, indicating the scenes are not perfectly static.
- Full web view is available at <https://gifarc.vercel.app/task/413>.

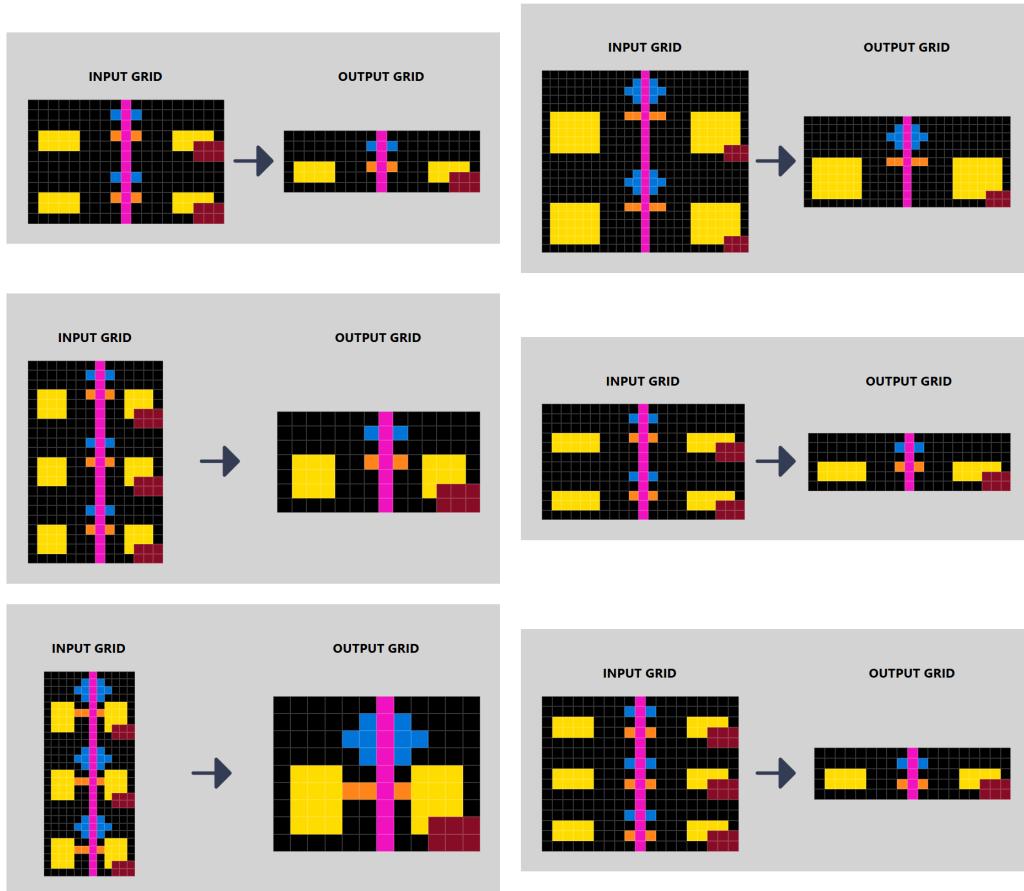


Figure 25: GIFARC-generated task 413.

E.19 Example Task of Type 19 - Fractal Expansion & Self-Similar Repeats

- Concepts : fractal expansions, symmetrical pulsations, radial transformations, iterative growth
- Description : In the input, you will see a sequence of grids (frames) on a black background. Each grid depicts: - A large circular region in the center, with multiple concentric rings and radial lines diverging outward. - A stacked-curve fractal anchored at the bottom-left corner. - A set of spire fractals anchored at the bottom-right corner. As you move from one frame to the next in the input, these elements undergo iterative transformations: - The radial lines repeatedly shift in thickness and visual intensity, while preserving rotational symmetry. - The concentric rings in the circular region pulsate by alternately expanding and contracting. - The stacked-curve fractal on the bottom-left changes its curve density, fractally adding or removing segments. - The right-side spire fractals expand and contract in repeated vertical segments. Your task is to replicate and apply these transformations for the entire sequence, and produce the final frame of the animation as the output: 1) Ensure the anchoring of the fractals at the bottom edge remains the same. 2) Preserve the radial symmetry of the central circular region and its common center point. 3) For each pulsation step in the input, magnify or contract the rings, lines, and fractals accordingly. 4) Continue until the final pulsation step is reached. That final state is your output grid. The essential principle is that symmetry-based fractal transformations and repeated cyclical expansions/contractions produce the overall pulsating effect. By reflecting each iterative change step by step, you reconstruct the final pulsating pattern visible at the end of the sequence.
- Full web view is available at <https://gifarc.vercel.app/task/155>.

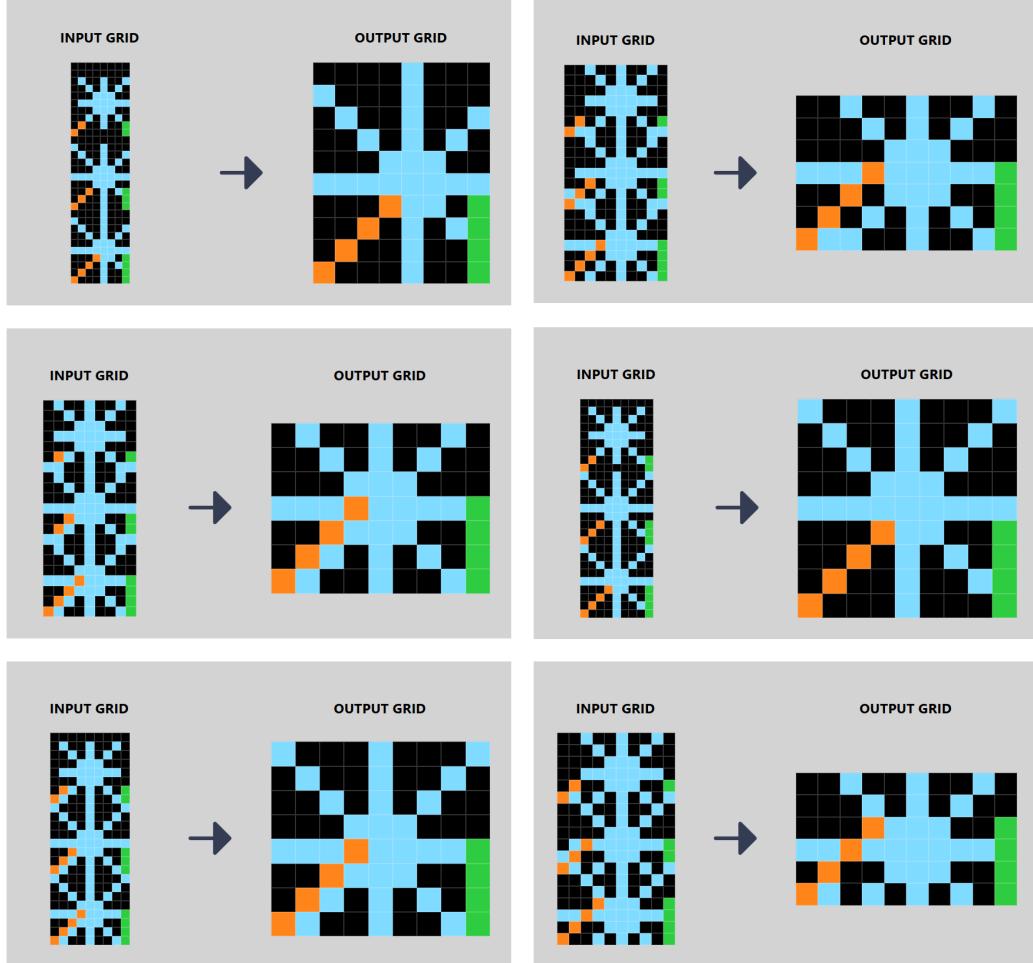


Figure 26: GIFARC-generated task 155.

E.20 Example Task of Type 20 - Sequential Pattern Growth & Transition

- Concepts : fractal expansions, symmetrical pulsations, radial transformations, iterative growth
- Description : In the input, you will see a sequence of grids (frames) on a black background. Each grid depicts: - A large circular region in the center, with multiple concentric rings and radial lines diverging outward. - A stacked-curve fractal anchored at the bottom-left corner. - A set of spire fractals anchored at the bottom-right corner. As you move from one frame to the next in the input, these elements undergo iterative transformations: - The radial lines repeatedly shift in thickness and visual intensity, while preserving rotational symmetry. - The concentric rings in the circular region pulsate by alternately expanding and contracting. - The stacked-curve fractal on the bottom-left changes its curve density, fractally adding or removing segments. - The right-side spire fractals expand and contract in repeated vertical segments. Your task is to replicate and apply these transformations for the entire sequence, and produce the final frame of the animation as the output: 1) Ensure the anchoring of the fractals at the bottom edge remains the same. 2) Preserve the radial symmetry of the central circular region and its common center point. 3) For each pulsation step in the input, magnify or contract the rings, lines, and fractals accordingly. 4) Continue until the final pulsation step is reached. That final state is your output grid. The essential principle is that symmetry-based fractal transformations and repeated cyclical expansions/contractions produce the overall pulsating effect. By reflecting each iterative change step by step, you reconstruct the final pulsating pattern visible at the end of the sequence.

- Full web view is available at <https://gifarc.vercel.app/task/2061>.

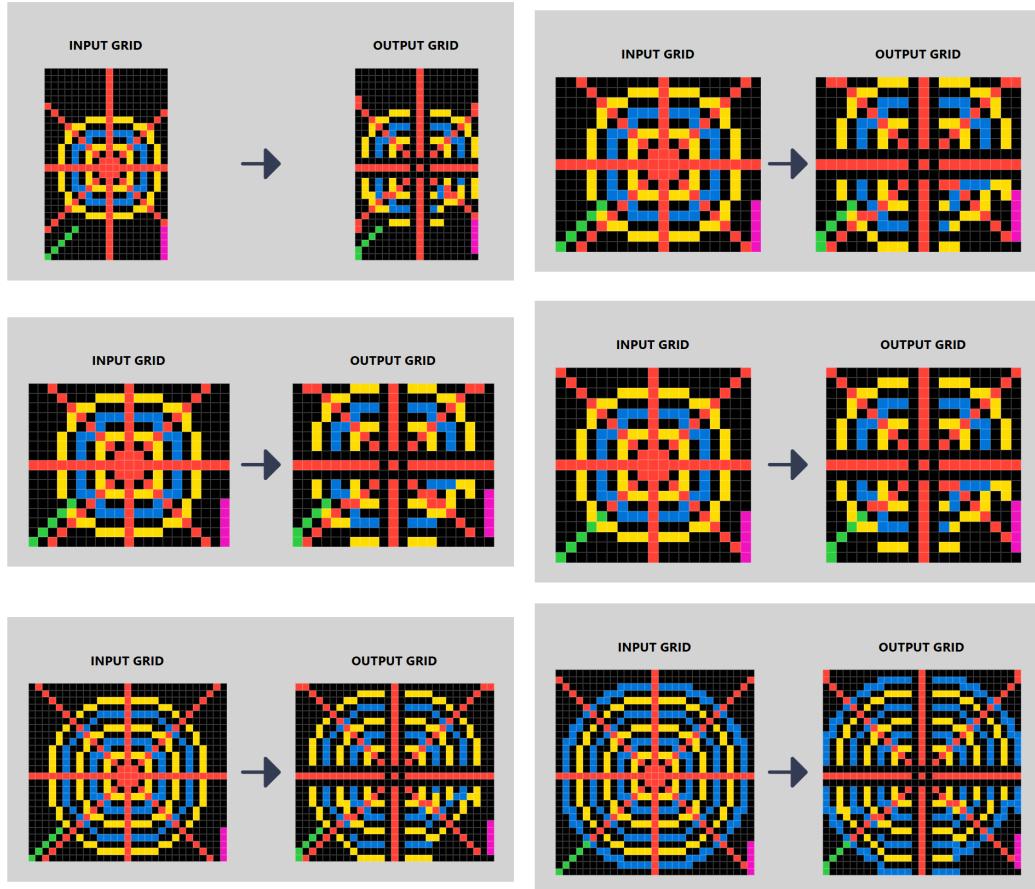


Figure 27: GIFARC-generated task 2061.