ARC-∞: AUTOMATING THE DESIGN OF RECURSIVELY GROWING INTELLIGENCE TESTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Conceptual bootstrapping, using known concepts to acquire new ones, is a hallmark of human intelligence, yet existing intelligence tests cannot measure it: their tasks are independent or fixed, and once solved, leave nothing to measure. Measuring bootstrapping requires a test that grows recursively, with each environment building on the concepts of earlier ones, which is impractical to design by hand. We introduce the **Abstraction and Reasoning Continuum (ARC-∞)**, a framework that automates the design of recursively growing intelligence tests. From a seed world, a Designer agent repeatedly produces the next world, yielding an unbounded worldline whose depth and breadth are controlled by two hyperparameters. We show that these controls steer growth as intended and that concepts accumulate across worlds. Evaluating state-of-the-art agent harnesses along a generated worldline, we find that they bootstrap from earlier worlds but fail to sustain it, and that a simple library learning step that consolidates prior knowledge extends their bootstrapping.

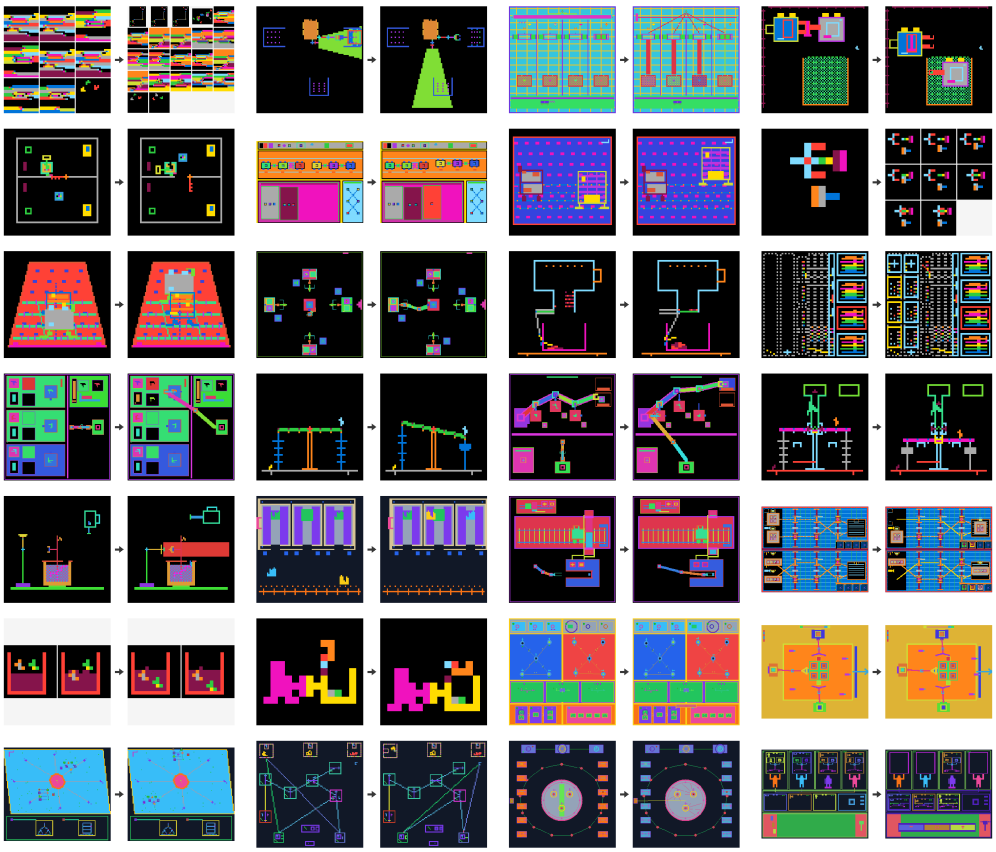


Figure 1: Input–output pairs ($x \to y$) from worlds generated by ARC-∞, sampled from different worldlines.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1 INTRODUCTION

Humans rarely learn a new concept from scratch. A child who understands counting uses it to learn addition, and addition in turn becomes a building block for multiplication. This process, in which a learner uses concepts it already knows to acquire new ones and thereby continually expands the range of concepts it can represent, is known as *conceptual bootstrapping* (Piantadosi et al., 2012; Sfard, 1991; Dehaene et al., 2022; Zhao et al., 2024). Bootstrapping is inherently recursive, as concepts built from earlier ones become, in turn, the building blocks of later ones. It is widely regarded as a hallmark of human intelligence: what makes a learner powerful is not only how well it solves the task in front of it, but how much what it learned today helps it learn tomorrow. As AI agents are increasingly expected to operate over long horizons and to keep learning after deployment, measuring whether they can bootstrap concepts becomes a central question for intelligence evaluation.

Existing intelligence tests are poorly suited to answering this question. Benchmarks for abstraction and skill acquisition, such as ARC (Chollet, 2019; Chollet et al., 2025), and procedurally generated reasoning suites (Moffitt, 2025; Taoudi-Benchekroun et al., 2026) present each task as an independent puzzle, so solving one task says little about whether its concepts can be reused in the next. ARC-AGI-3 (ARC Prize Foundation, 2026) carries learning across the levels of a game, but its games are fixed and do not build on one another. Continual-learning benchmarks (Wołczyk et al., 2021; Zheng et al., 2025; Asawa et al., 2026) do present tasks in sequence and measure transfer and forgetting, but their tasks are fixed in advance and are not constructed so that later concepts are *built from* earlier ones.

Evaluating conceptual bootstrapping therefore requires a test that grows recursively, as a sequence of environments in which each environment builds its concepts from those of earlier ones. Such a test is impractical to design by hand. Grounding every new concept in earlier ones is laborious, so a human designer can produce only a few stages with a fixed set of tasks, and once a system solves them, the test has nothing left to measure, as with ARC. The test must therefore be generated automatically, so that it can grow without bound, not only deeper, with each environment building on earlier ones, but also broader, with each environment introducing new objects, properties, and interactions.

Automatically designing such a test is challenging. Recent work shows that foundation models can generate executable environments, games, and benchmarks (Faldor et al., 2025; Zhang et al., 2026; Verma et al., 2025; Ying et al., 2026), and open-ended environment design has long produced ever-harder challenges (Wang et al., 2019; Dennis et al., 2020; Parker-Holder et al., 2022). However, these systems generate tasks either to train an agent or to add independent tests. They do not ensure that each new task develops the concepts of earlier ones. Naively asking a model to extend an environment also tends to fail in two ways. The model either stays close to what is already implemented, producing little new, or rewrites earlier components instead of building on them, so that nothing accumulates.

In this paper, we introduce the **Abstraction and Reasoning Continuum** (ARC-∞, pronounced *arc infinity*), a framework that automates the design of recursively growing intelligence tests. Each environment, called a *world*, consists of an operation library that defines its rules, training observations from which a learner can infer them, and ARC-style test problems. Starting from a seed world, a *Designer* agent repeatedly produces the next world from earlier ones, yielding a *worldline*. The Designer separates planning from implementation. A *Director* reads only the observations of earlier worlds and writes a *growth intent* for how the next world should develop their concepts. A *Creator* then extends the previous codebase with operations composed from existing ones and generates data that makes the new compositions visible. Two hyperparameters, α_d and α_b, control how deeply each world builds on earlier results and how broadly it introduces new concepts.

Our contributions are as follows:

1. We formalize the evaluation of conceptual bootstrapping as learning along a *worldline*, a recursively growing sequence of worlds, each building its concepts from those of earlier worlds.
2. We propose ARC-∞, a framework that automates the design of such intelligence tests without human involvement. Its Designer, split into a Director and a Creator, grows worldlines with explicit controls over the depth and breadth of growth.
3. We show that α_d and α_b steer the depth and breadth of growth, and identify the design choices that let compositions accumulate across worlds.
4. We find that agent harnesses bootstrap from earlier worlds but fail to sustain it, and that a simple library learning step (Ellis et al., 2021; Grand et al., 2024) extends their bootstrapping.

2 ABSTRACTION AND REASONING CONTINUUM FRAMEWORK

In this section, we formalize the two parts of ARC-∞, the worlds that make up a recursively growing intelligence test and the Designer that creates them. In ARC-∞, each environment in which a learner is trained and tested is called a *world*. ARC-∞ generates a sequence of worlds $\mathcal{L} = (W_0, W_1, \ldots, W_T)$, called a *worldline*, in which each world builds its concepts from those of earlier worlds. We call the index t of a world W_t its *age*. The worldline has no fixed length, and T can keep increasing for as long as there is something left to measure. Worlds are created by the *Designer*, which produces each new world W_t from the earlier worlds $W_{<t} = (W_0, \ldots, W_{t-1})$.

A learner is trained on each world in turn and evaluated on its test problems, but designing a good learner is outside the scope of this work. The Designer never observes the learner or its performance, so the same worldline can be used to evaluate any learner.

2.1 WORLD STRUCTURE

A world $W = (G, D^{\text{train}}, D^{\text{test}})$ consists of an operation library G, a training set D^{train}, and a test set D^{test}. The library G is the set of operations that govern the world, D^{train} consists of observations from which a learner can infer them, and D^{test} consists of problems that evaluate how well it has inferred them. The learner sees only inputs and outputs, never G or the programs that produced them.

Operation library. The operation library $G = \{f_1, \ldots, f_K\}$ is a set of operations that implement the concepts of the world. For example, $f_{\texttt{move_right}}$ takes an input grid and moves its objects one cell to the right. An operation may call other operations, including those introduced in earlier worlds, so the library of W_t builds recursively on those of $W_{<t}$. Operations are composed into *task programs* $z = f^{(L)} \circ \cdots \circ f^{(1)}$ with each $f^{(\ell)} \in G$. For example, composing $f_{\texttt{move_right}}$ and $f_{\texttt{change_color_to_red}}$ gives a task program that moves the objects right and colors them red.

Training set. The training set $D^{\text{train}} = \{(x_i, z_i(x_i))\}_{i=1}^{N}$ is a collection of input–output pairs, or *observations*, each generated by its own task program z_i. It lets the learner infer the operations in G before it faces the test problems.

Test set. The test set D^{test} is a collection of problems $P = (S, x^\star, y^\star)$ in the ARC format. The support $S = \{(x_j, z(x_j))\}_{j=1}^{m}$ contains m pairs generated by a task program z, and the query output is $y^\star = z(x^\star)$ for a new input $x^\star$. The learner receives S and $x^\star$, and a problem is counted as correct when its prediction $\widehat{y}^\star$ exactly matches $y^\star$. Different problems may use different task programs.

2.2 GROWING A WORLD WITH THE DESIGNER

To evaluate conceptual bootstrapping, the intelligence test itself must unfold as a sequence of worlds, such that concepts learned in earlier worlds help the learner learn concepts in subsequent ones. The Designer thus develops worlds so that earlier concepts become building blocks for new ones.

Starting from a seed world W_0, the Designer generates each successor $W_t = (G_t, D_t^{\text{train}}, D_t^{\text{test}})$ from the preceding worlds: $W_t = \texttt{designer.create}(W_{<t})$.

The Designer can introduce new operations and combine existing operations into reusable ones. The training set should support learning the new concepts, and the test set should evaluate how well the learner has learned them. To make these sets, the Designer selects task programs and inputs that demonstrate and test the concepts. Section 3 describes how the Designer carries out this process.

Evaluating a learner. A learner meets the worlds in order, starting from an empty state M_{-1}. At age t, it updates its state from the training set while keeping what it learned before, $M_t = \text{Learn}(M_{t-1}, D_t^{\text{train}})$, and then answers each test problem $(S, x^\star, y^\star) \in D_t^{\text{test}}$ as $\widehat{y}^\star = \text{Solve}(M_t, S, x^\star)$. The learner bootstraps when M_{t-1} helps it learn in W_t. Bootstrapping can be measured against the same learner restarted from an empty state at every age. The difference between the two shows how much the earlier worlds help.

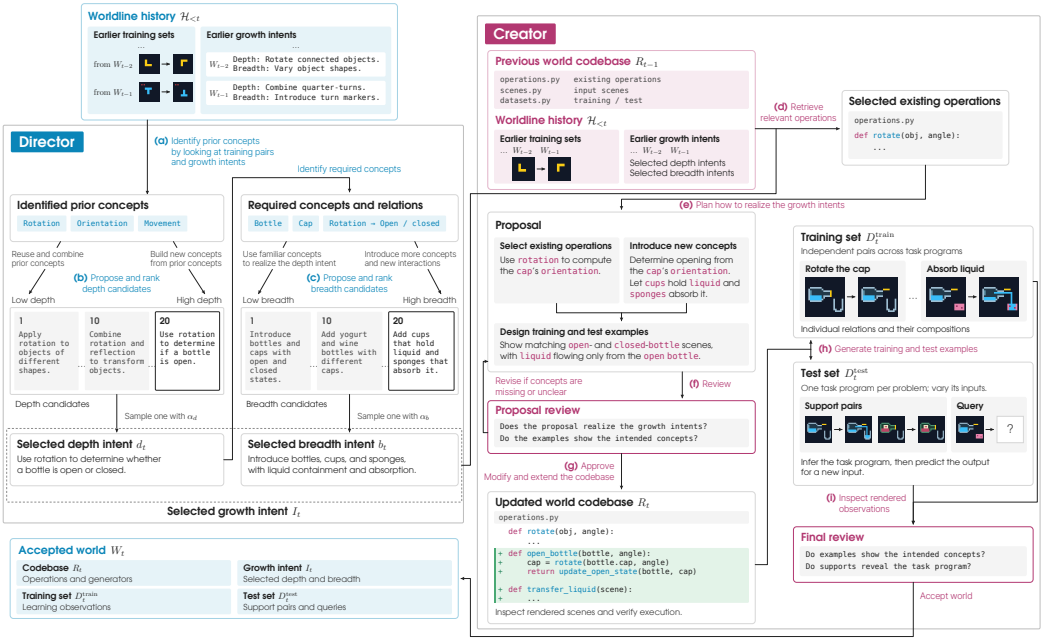


Figure 2: The ARC-∞ Designer. **(a)** The Director identifies concepts by comparing earlier training inputs and outputs and tracing them through earlier growth intents. **(b)** It proposes developments of these concepts, from varying their use to building new concepts on their results, ranks them by depth, and samples one with α_d. **(c)** For the chosen development, it proposes candidates that add different ranges of concepts, ranks them by breadth, and samples one with α_b. **(d)** The Creator retrieves and selects the existing operations that implement the concepts the growth intent develops. **(e)** Its proposal builds new concepts from the selected operations and plans the training and test examples that show them. **(f)** A review checks whether the proposal realizes the growth intent. **(g)** After approval, the Creator extends the codebase into R_t. **(h)** It generates the training and test sets, with one task program per test problem. **(i)** A final review checks the rendered sets and accepts the world.

3 The Designer: Director and Creator

We implement the Designer with two agents, the Director and the Creator (Figure 2). The Director decides how the next world develops the concepts of earlier worlds. The Director's output is a *growth intent*, a natural-language plan for this development. The Creator implements the intent by extending the operation library of the previous world and generating the next world's training and test sets. Appendix A gives the full procedure and the prompts of the Designer.

3.1 The Director

The Director receives the history of training sets and growth intents of earlier worlds, denoted by $\mathcal{H}_{<t}$, and produces a growth intent I_t. The intent $I_t = (d_t, b_t)$ consists of a *depth intent* d_t and a *breadth intent* b_t, generated through two-stage planning. First, the Director proposes a natural-language plan describing how to develop earlier concepts and how complex this development should be, forming d_t. It then forms b_t by specifying the range of concepts the next world introduces to realize d_t.

Input. We make the Director's input $\mathcal{H}_{<t}$ exclude the codebase of the previous world, because a plan made from the code tends to rewrite earlier operations instead of extending them. The history covers all earlier worlds, so the plan can build on concepts introduced at any earlier age.

Depth intent. The depth intent decides how the next world develops earlier concepts. To form d_t, the Director first identifies concepts that could be learned from the training sets of earlier worlds. It compares inputs and outputs to find what changes and under which visible conditions those changes occur. It reads these observations alongside each world's growth intent to trace how the concepts recur, combine, and take on new roles across worlds. The Director records each identified concept

together with its supporting training pairs. The Director works from these observations, so it judges what a learner could infer from them.

From these concepts, the Director proposes candidates at different *depths*. For a low-depth candidate, it selects an established concept and varies the objects or conditions it applies to, or combines it with familiar concepts. For example, rotation may apply to objects of new shapes, or combine with reflection. Such a candidate consolidates what the learner already knows. For a higher-depth candidate, the Director determines what can be computed with an earlier concept and proposes a new concept whose target, rule, or condition depends on that result. For example, a cap's orientation after rotation may decide whether a bottle is open. Rotation then becomes a part of opening, and the learner must learn the new concept of opening. The Director generates a sorted list of $C = 20$ intent candidates ranked by their depths. We sample d_t with a hyperparameter $\alpha_d \in [0, 1]$, where higher α_d values favor higher-depth candidates. A single plan tends to stay close to what already exists, so ranking many candidates lets α_d decide how far the next world goes.

Breadth intent. The breadth intent specifies the range of concepts with which the next world realizes d_t. To form b_t, the Director first identifies the concepts that d_t requires. Opening a bottle by rotation, for example, requires bottles and caps whose orientation can be seen. The Director then proposes $C = 20$ candidates that realize the same d_t with different ranges of new concepts. A low-breadth candidate adds only these required concepts and otherwise uses familiar ones. A high-breadth candidate adds further objects, properties, and interactions that take part in the development. For example, cups may hold the released liquid and sponges may absorb it, introducing containment and absorption alongside opening by rotation. The Director ranks the candidates from narrowest to broadest, and we sample b_t with a hyperparameter $\alpha_b \in [0, 1]$, with higher values favoring broader candidates. Appendix A.6.1 gives the candidate prompts and Appendix A.3 the sampling rule.

3.2 The Creator

The Creator receives the growth intent I_t from the Director, the history $\mathcal{H}_{<t}$, and the codebase R_{t-1} of the previous world, and produces the next world W_t. The codebase contains the operation library G_{t-1} and the code that generates the training and test sets. The Creator builds W_t in three stages. First, it turns the abstract growth intent into a concrete plan, called a *proposal*. The proposal selects the operations in G_{t-1} that implement the concepts the growth intent develops, specifies how to extend the operation library to realize the growth intent, and plans which examples the training and test sets should contain to show the new concepts. Next, the Creator implements the proposal by extending G_{t-1} into G_t. Finally, it writes the code that generates the training set D_t^{train} and the test set D_t^{test} from task programs composed of the operations in G_t. The extended operation library and this code form the next codebase R_t.

Proposal. The growth intent is an abstract plan written in natural language, so it describes the concepts it develops but not the operations that implement them. For example, the depth intent "Use rotation to determine whether a bottle is open or closed" names rotation but not the specific operation that computes it. The Creator therefore first selects the operations in G_{t-1} that implement these concepts, such as `rotate`, and runs them on sample inputs to confirm what they compute. It also finds the training pairs in $\mathcal{H}_{<t}$ that show the selected operations, because the learner's knowledge could be limited to what its training sets showed.

The Creator then builds the new concepts from the selected operations. Following the depth intent, it combines the selected operations with each other, or uses their results as the inputs or conditions of new operations. Following the breadth intent, it adds further new concepts and the operations that implement them. For example, a new operation decides whether a bottle is open from the cap's orientation that `rotate` computes, and further operations, added for breadth, let cups hold the released liquid and sponges absorb it.

Finally, the Creator plans which examples the training and test sets should contain to show the new concepts. A composition implemented in code can still be hidden in the examples, because the learner sees only their inputs and outputs. If every example of a composition also fits a simpler rule without the selected operation, the learner can solve the examples without using that operation. The Creator therefore designs *contrasts* for each composition. A contrast is a pair of examples that differ in one condition and whose outputs differ because of that condition. For example, an open bottle and a closed bottle tilted the same way show that the open state decides whether liquid leaves the bottle.

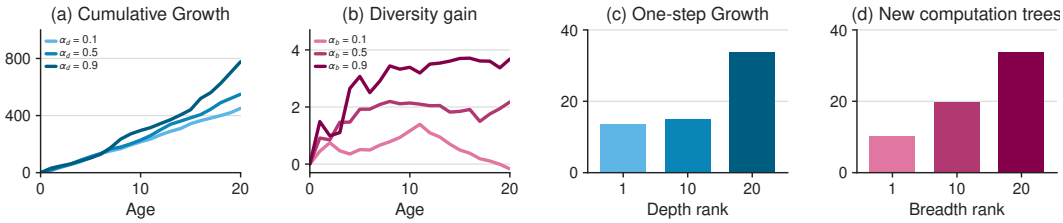


Figure 3: Controlling development across worlds and within one step. (a) Cumulative Growth for three depth settings and (b) Diversity gain for three breadth settings. (c) One-step Growth and (d) new computation trees for successors of the same parents built at depth and breadth ranks 1, 10, and 20.

Before implementation, a review with access to the codebase checks whether the proposal realizes the growth intent. It runs the selected operations to confirm that they implement the concepts the growth intent develops, and it checks that the planned contrasts show the new concepts. If the proposal misses either, the Creator revises it.

Implementation. After approval, the Creator implements the planned operations and adds them to the operation library, forming G_t. Because the learner sees only inputs and outputs, the Creator also updates how inputs and outputs are generated and rendered, so that everything the new operations act on or change can be seen in them. It checks the implementation by running the new operations on sample inputs and inspecting the rendered results.

Training and test sets. The Creator then writes the code that generates the training and test sets planned in the proposal. It samples inputs on which each composition and the simpler rule give different outputs, so that the planned contrasts appear in the sets. Each test problem applies one task program to its support pairs and query, and the supports vary the conditions that change the output.

Review and acceptance. A final review then inspects the rendered training and test sets. It checks that the examples planned in the proposal appear in the training and test sets and that the supports of each test problem are enough to infer its task program. If either check fails, the Creator revises the problematic code. After acceptance, the world's training set and growth intent join the history. The updated history and codebase R_t are the inputs for planning and building the next world.

4 Experiments

We analyze the worlds generated by ARC-∞ and evaluate agent harnesses trained on the generated worlds. Appendix C explains generated worldlines in detail, including how they recursively grow concepts and where generation fails. Section 4.1 tests the effect of the control hyperparameters and our design choices. Section 4.2 trains agent harnesses along a worldline and analyzes what they learn.

4.1 Growing Worlds

We first test whether the depth and breadth controls steer how worlds grow. We then change the Designer's design choices one at a time, and compare gradual growth with one-shot generation.

4.1.1 Controlling World Growth

Metrics. We describe how a worldline grows with two metrics, *Growth* and *Diversity*. Growth measures how far each world builds on earlier results. We parse the code that generates each world into a graph of expressions and their dependencies, and treat nodes that match the code of earlier worlds as earlier results. The longest path of new nodes that starts from an earlier result is the *one-step Growth* of a world, and Growth sums it across the worldline. Diversity is the mean pairwise distance between code embeddings of the operations in the library, and we report its change from the seed world as *Diversity gain*. Within one step, we measure how broadly a world grows by the number of *new computation trees*, connected groups of new nodes that do not depend on earlier results.

Setup. From one seed world, we generate a worldline of age 20 for each combination of $\alpha_d, \alpha_b \in \{0.1, 0.5, 0.9\}$. To see how depth and breadth act within one step, we also build successors of the

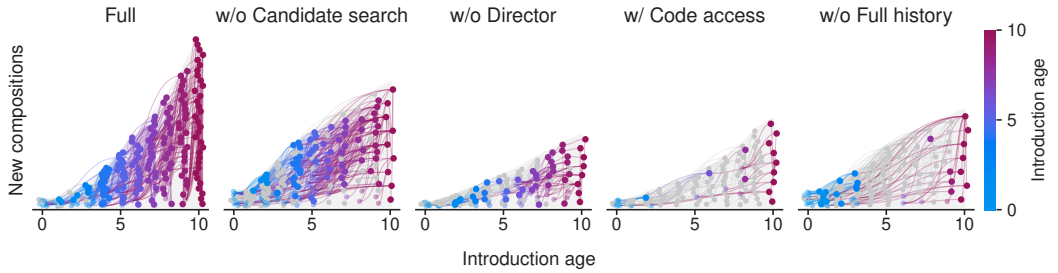


Figure 4: New compositions along each worldline. Points show new compositions and the operations they call, placed at their introduction ages. Lines connect callers to callees. Color marks the introduction age, and gray points were replaced before the final world.

	Final world		Operations per pair		New compositions		
Method	Diversity gain	Growth	Reused	New comp.	Introduced	Retained	Retention (%)
`Full`	+3.27	355	30.0	29.9	173	159	91.9
`w/o Candidate search`	+0.09	172	95.5	7.8	158	74	46.8
`w/o Director`	−0.08	191	48.0	6.6	94	44	46.8
`w/ Code access`	+3.84	213	27.0	9.0	85	13	15.3
`w/o Full history`	+3.88	178	31.8	7.0	102	25	24.5
`One-shot`	+0.20	59	6.0	2.0	–	–	–

Table 1: Growth, reuse, and accumulation for each condition.

same parent worlds from growth intents at depth or breadth ranks 1, 10, and 20. Appendix B gives the definitions and the results for all nine settings.

Depth. Higher α_d gives steeper Growth when breadth adds new operations (Figure 3a). Within one step, higher-ranked depth candidates build longer paths of new computation on earlier results (Figure 3c). Higher α_d selects such candidates more often, so their paths add up across the worldline.

Breadth. Higher α_b gives more diverse libraries (Figure 3b). Within one step, higher-ranked breadth candidates add more new computation trees as the worlds introduce more objects, properties, and interactions (Figure 3d). Breadth also raises Growth, as additional operations provide new uses for earlier results and extend the paths.

Property 1. Higher depth makes new operations build further on earlier results, and higher breadth broadens the library, both within one step and across a worldline.

4.1.2 Design Choices

Metrics. We measure how the worlds reuse and accumulate operations. For each training pair of the final world, we count the executed operations that appeared in earlier worlds, called *reused operations*, and the new operations that call them, called *new compositions*. To see whether new compositions accumulate, we count how many are *introduced* along the worldline and how many are *retained* in the library of the final world.

Setup. We generate a worldline of age 10 with the full Designer (`Full`) at $\alpha_d = 0.5$ and a high $\alpha_b = 0.9$, so that differences in how the library diversifies are visible. We compare it with four variants that each change one design choice of Section 3. `w/o Candidate search` has the Director write a single growth intent instead of choosing among ranked candidates. `w/ Code access` lets the Director read the previous world's codebase R_{t-1}. `w/o Director` has one agent both plan and implement the next world. `w/o Full history` gives the Director and the Creator only the preceding world $\mathcal{H}_{t-1}$, instead of $\mathcal{H}_{<t}$.

`Full` is the only condition that both keeps the library diversifying and accumulates new compositions. Figure 4 follows the compositions over time. In `Full`, nearly all new compositions remain available for recursive composition and growth in later worlds. The ablations fail in two ways.

Narrowing without alternatives. Without candidate search or a separate Director, Diversity stays at the seed level. Without candidate search, the Creator relies heavily on earlier operations but uses few new compositions in the generated problems. Without a separate Director, the single agent reads the code it is about to modify while choosing the direction, and it likewise extends the parent's operations in place. Both variants retain less of what they develop, leaving fewer compositions for later growth.

Property 2. Candidate search and separating the Designer into two agents move worldline development beyond the existing concepts and keep the library diversifying.

Variety without accumulation. Giving the Director code access or restricting history to the parent yields Diversity above `Full`'s, but few of the introduced compositions survive to the final world. In Figure 4, most of their points are gray. With code access, the Director plans from the implementation, and successors rewrite the previous age's compositions rather than extend them. With parent-only history, a composition must be carried by every intermediate world to remain available, and whole ages of operations disappear.

Property 3. Showing the Director the codebase or giving both agents only the preceding world makes later worlds replace earlier compositions instead of extending them, so compositions do not accumulate.

Gradual generation. `One-shot` builds directly from the seed with the same planning and review, the generation limits of `Full`'s final world, and `Full`'s total time budget. Its world has far fewer operations and compositions, and its Diversity barely rises above the seed (Table 1). `One-shot` adds one layer to the seed, while `Full` recursively composes operations from intermediate worlds.

Property 4. Under the same budget limit, gradual growth produces far more operations and compositions than one-shot generation, so it better supports complex, bootstrapped concepts.

4.2 Learning across Worlds

We train agent harnesses along a generated worldline and test them at each world. We analyze whether they bootstrap, using what they learned in previous worlds to learn the current world, and where the limits of their bootstrapping appear. We then test a way to overcome these limits.

4.2.1 Experimental Setup and Baselines

Setup. We generate a worldline of age 10 with $\alpha_d = 0.8$ and $\alpha_b = 0.5$. At each age, a harness learns from 10 training pairs of the world and then solves 20 test problems, each with three support pairs and one query. Appendix E analyzes success and failure cases in detail, Appendix C analyzes the ages at which performance drops most sharply, and Appendix F reports an additional evaluation on another worldline.

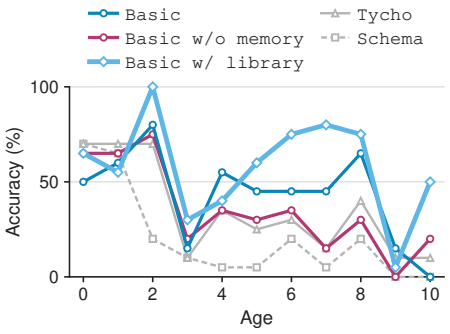


Figure 5: Accuracy along the worldline.

Baselines. `Basic` follows a general coding-agent loop, writing and executing code and keeping useful files across worlds, as in arc-code (Berman, 2026). `Basic w/o memory` is `Basic` with its memory reset at every age, so it learns from the current world alone. We also evaluate harnesses with near-perfect performance on the public games of ARC-AGI-3. `Schema` writes code that reproduces the observations and revises it when its outputs do not match (Zeng et al., 2026). `Tycho` pairs an actor that interprets observations with a builder that constructs models and solves queries (Lehmann et al., 2026). Appendix D describes the detailed procedure of each harness.

4.2.2 Results and Analysis

Bootstrapping from previous worlds. `Basic` solves more problems than `Basic w/o memory` at most ages and uses fewer tokens overall (Figure 5, Appendix E.1). `Basic` keeps learned concepts as code in its workspace, and bootstraps in new worlds. `Basic w/o memory` does not have any knowledge accumulation, so it cannot bootstrap from earlier concepts. This comparison shows

that `Basic` can bootstrap concepts, and that solving our worldline requires bootstrapping from the knowledge of earlier worlds.

Finding 1. Agent harnesses that accumulate knowledge across worlds bootstrap, learning new worlds more accurately at a lower cost.

Accumulation without optimization. The advantage of `Basic` becomes smaller in later worlds, and all four harnesses perform poorly in the final worlds (Figure 5). Each harness learns by fitting the current world, and none of them reorganizes what it has learned. This leads to two kinds of failure (Appendix E). First, a learner overfits to the current world. It keeps whatever can solve the current world, such as the current world's raw observations or a separate world model for this world. This knowledge works for the current world, but in later worlds, it fails to generalize. `Schema` overfits from the early worlds, and `Basic` also does so as the worlds grow. Second, a learner keeps the code that performs each transformation but not the rule that decides which transformation to apply. Later worlds build their new concepts on these decisions, so the learner has to infer them again from the few current examples. `Tycho` keeps files but fails in the same way as `Basic w/o memory`.

Finding 2. Agent harnesses need to optimize accumulated knowledge for bootstrapping. Otherwise they overfit to each world, and what they learn does not generalize to later worlds.

Optimizing accumulated knowledge. To test whether optimizing accumulated knowledge extends bootstrapping, we add a simple library learning step to `Basic` (Ellis et al., 2021; Grand et al., 2024), which rewrites retained code into reusable operations and composes them into a program for each test problem. The resulting `Basic w/ library` (Appendix D) solves more problems than `Basic` with fewer tokens (Figure 5, Appendix E.1).

Finding 3. A simple library learning step that rewrites accumulated knowledge into reusable operations extends bootstrapping, learning later worlds more accurately at a lower cost.

5 Related Work

Accounts of conceptual bootstrapping and abstraction describe how learned concepts or procedures become reusable units for later reasoning (Sfard, 1991; Dehaene et al., 2022; Zhao et al., 2024). Program-learning methods realize this idea by discovering functions that can be composed to solve new tasks or explain observations (Ellis et al., 2021; Grand et al., 2024; Baek et al., 2026).

Intelligence evaluation asks how well a system can learn and apply skills beyond its training tasks (Hernández-Orallo & Dowe, 2010; Hernández-Orallo, 2017; Chollet, 2019; Hernández-Espinosa et al., 2026; Ying et al., 2025). Open-ended systems invent their own problems (Schmidhuber, 2013; Wang et al., 2020), and LLM agents build benchmarks once people specify the ability that each benchmark should test (Butt et al., 2025; Dsouza et al., 2025). Appendix G extends this discussion.

6 Conclusion

We introduced ARC-∞, an automated framework that recursively grows intelligence tests for conceptual bootstrapping. Agent harnesses bootstrap from what they learned in previous worlds, but this advantage becomes smaller in later worlds because they do not optimize accumulated knowledge, and a simple library learning step extends their bootstrapping. The central contribution is the Designer and the philosophy of growth it embodies. With it, researchers can test their own models and agent harnesses on new worldlines until the limits of their systems become visible.

Limitations and outlook. Our worlds follow the support–query format of ARC. The Designer could also grow interactive worlds like those of ARC-AGI-3, where concepts unfold over time and through actions. Since the Designer is built from LLM agents, it does not always produce tests that are complete and logically sound, and it occasionally makes mistakes, as Appendix C.3 shows. Adding heuristic and neurosymbolic verification procedures to the Designer is a promising direction for making the generated tests more reliable. We also evaluated a few existing harnesses without extensive engineering. We believe that harnesses built for continual learning, such as agents that perform more principled library learning, can bootstrap much further, and ARC-∞ could measure how far they progress as the worldline grows.

Reproducibility Statement

Section 3 and Appendix A describe the Designer, including its full procedure and prompts. Appendix B gives the metrics and settings of the growth experiments, and Appendix D describes each harness and the evaluation protocol. We will organize and release the code of ARC-∞ and the generated worldlines used in this paper.

AI Use Statement

Large language models were used to draft and polish the writing of this paper. They were also used for research ideation, implementation, experiments, and the analysis of results. The worlds of ARC-∞, including their training and test sets, are synthetic data generated by LLM agents, as described in Section 3. The authors reviewed all AI-assisted work and take responsibility for the final content.

Ethics Statement

This work does not involve human subjects or personal data. The worlds of ARC-∞ are synthetic grid environments generated by LLM agents and contain no real people or sensitive content. ARC-∞ is designed to measure how AI systems learn new concepts, and we do not foresee direct harmful uses. Generating worlds and evaluating harnesses require substantial LLM computation, and Appendix B.9 reports the resources used for generation.

References

ARC Prize Foundation. ARC-AGI-3: A new challenge for frontier agentic intelligence. *arXiv preprint arXiv:2603.24621*, 2026.

Parth Asawa, Christopher M. Glaze, Gabriel Orlanski, Ramya Ramakrishnan, Benji Xu, Asim Biswal, Vincent Sunn Chen, Frederic Sala, Matei Zaharia, and Joseph E. Gonzalez. Continual learning bench: Evaluating frontier AI systems in real-world stateful environments. *arXiv preprint arXiv:2606.05661*, 2026.

Doojin Baek, Gyubin Lee, Junyeob Baek, Hosung Lee, and Sungjin Ahn. Learning to theorize the world from observation. In *International Conference on Machine Learning*, 2026.

Jeremy Berman. arc-code. GitHub repository, 2026. URL `https://github.com/jerber/arc-code`.

Natasha Butt, Varun Chandrasekaran, Neel Joshi, Besmira Nushi, and Vidhisha Balachandran. BENCHAGENTS: Automated benchmark creation with agent interaction. In *ICLR 2025 Workshop on Navigating and Addressing Data Problems for Foundation Models*, 2025.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

François Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. ARC-AGI-2: A new challenge for frontier AI reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025.

Heejae Chon, Seonghyeon Lee, Jinyoung Yeo, and Dongha Lee. Is functional correctness enough to evaluate code language models? exploring diversity of generated codes. *arXiv preprint arXiv:2408.14504*, 2024.

Stanislas Dehaene, Fosca Al Roumi, Yair Lakretz, Samuel Planton, and Mathias Sablé-Meyer. Symbols and mental programs: A hypothesis about human singularity. *Trends in Cognitive Sciences*, 2022.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. In *Advances in Neural Information Processing Systems*, 2020.

Amanda Dsouza, Harit Vishwakarma, Zhengyang Qi, Justin Bauer, Derek Pham, Thomas Walshe, Armin Parchami, Frederic Sala, and Paroma Varma. Automating benchmark design. *arXiv preprint arXiv:2510.25039*, 2025.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2021.

Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. OMNI-EPIC: Open-endedness via models of human notions of interestingness with environments programmed in code. In *International Conference on Learning Representations*, 2025.

Alexis Fox, Junlin Wang, Paul Rosu, and Bhuwan Dhingra. PRO-LONG: Programmatic memory enables long-horizon reasoning. *arXiv preprint arXiv:2607.20064*, 2026.

Gabriel Grand, Lio Wong, Maddy Bowers, Theo X Olausson, Muxin Liu, Joshua B Tenenbaum, and Jacob Andreas. LILO: Learning interpretable libraries by compressing and documenting code. In *International Conference on Learning Representations*, 2024.

Alberto Hernández-Espinosa, Luan Ozelim, Felipe S. Abrahão, and Hector Zenil. SuperARC: A test for artificial superintelligence based on compressed modelling, recursive prediction and problem complexity. *Nature Communications*, 2026.

José Hernández-Orallo. Evaluation in artificial intelligence: From task-oriented to ability-oriented measurement. *Artificial Intelligence Review*, 2017.

José Hernández-Orallo and David L. Dowe. Measuring universal intelligence: Towards an anytime intelligence test. *Artificial Intelligence*, 2010.

Jens Lehmann, Andrei Aioanei, and Sahar Vahdati. Tycho: Active abstraction with programmatic world models for ARC-AGI-3. *arXiv preprint arXiv:2607.28287*, 2026.

Michael D. Moffitt. ARC-GEN: A mimetic procedural benchmark generator for the abstraction and reasoning corpus. *arXiv preprint arXiv:2511.00162*, 2025.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. In *International Conference on Machine Learning*, 2022.

Steven T. Piantadosi, Joshua B. Tenenbaum, and Noah D. Goodman. Bootstrapping in a language of thought: A formal model of numerical concept learning. *Cognition*, 2012.

Jürgen Schmidhuber. PowerPlay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in Psychology*, 2013.

Anna Sfard. On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics*, 1991.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. CoRNStack: High-quality contrastive data for better code retrieval and reranking. In *International Conference on Learning Representations*, 2025.

Yassine Taoudi-Benchekroun, Klim Troyan, Pascal Sager, Stefan Gerber, Lukas Tuggener, and Benjamin Grewe. COGITAO: A visual reasoning framework to study compositionality & generalization. *arXiv preprint arXiv:2509.05249*, 2026.

Vivek Verma, David Huang, William Chen, Dan Klein, and Nicholas Tomlin. Measuring general intelligence with generated games. *arXiv preprint arXiv:2505.07215*, 2025.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Paired open-ended trailblazer (POET): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019.

Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeffrey Clune, and Kenneth Stanley. Enhanced POET: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. In *International Conference on Machine Learning*, 2020.

Maciej Wołczyk, Michał Zając, Razvan Pascanu, Łukasz Kuciński, and Piotr Miłoś. Continual world: A robotic benchmark for continual reinforcement learning. In *Advances in Neural Information Processing Systems*, 2021.

Lance Ying, Katherine M Collins, Prafull Sharma, Cedric Colas, Kaiya Ivy Zhao, Adrian Weller, Zenna Tavares, Phillip Isola, Samuel J Gershman, Jacob D Andreas, et al. Assessing adaptive world models in machines with novel games. *arXiv preprint arXiv:2507.12821*, 2025.

Lance Ying, Ryan Truong, Prafull Sharma, Kaiya Ivy Zhao, Nathan Cloos, Kelsey R Allen, Thomas L Griffiths, Katherine M Collins, José Hernández-Orallo, Phillip Isola, et al. AI GameStore: Scalable, open-ended evaluation of machine general intelligence with human games. *arXiv preprint arXiv:2602.17594*, 2026.

Guanning Zeng, Jiani Wang, Wenjie Ma, Shaofeng Yin, Chenyang Wang, Shichen Liu, Angjoo Kanazawa, Wode Ni, Xiuyu Li, Andrea Zanette, and Haiwen Feng. [schema]: Frontier models with the right harness achieve ∼99% on ARC-AGI-3 public. Impossible Research, 2026. URL `https://schema-harness.github.io/`.

Zheyuan Zhang, Zehao Wen, Alvin Zhang, Andrew Wang, Jianwen Xie, Daniel Khashabi, and Tianmin Shu. AgentOdyssey: Open-ended long-horizon text game generation for test-time continual learning agents. *arXiv preprint arXiv:2606.24893*, 2026.

Bonan Zhao, Christopher G. Lucas, and Neil R. Bramley. A model of conceptual bootstrapping in human cognition. *Nature Human Behaviour*, 2024.

Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. LifelongAgentBench: Evaluating LLM agents as lifelong learners. *arXiv preprint arXiv:2505.11942*, 2025.

A THE DESIGNER

A.1 ONE STEP OF THE WORLDLINE

Algorithm 1 One step of the worldline

Require: Previous codebase R_{t-1}; history $\mathcal{H}_{<t}$; controls α_d, α_b; candidate count C
Require: Limits: breadth candidates R_s, proposal revisions R_p, generated-world revisions R_r, validation repairs R_v
Ensure: Accepted world W_t and history $\mathcal{H}_{<t+1}$, or FAILURE

Director: decide what the next world should teach

$B_t \leftarrow \mathrm{Director.RecordConcepts}(\mathcal{H}_{<t})$ ▷ concepts shown so far, each with its supporting observations
$(d^{(1)}, \ldots, d^{(C)}) \leftarrow \mathrm{Director.RankDepth}(B_t)$ ▷ from revisiting a concept to building on its result
$i \sim q_{\alpha_d}$; $d_t \leftarrow d^{(i)}$ ▷ depth intent
$(b^{(1)}, \ldots, b^{(C)}) \leftarrow \mathrm{Director.RankBreadth}(B_t, d_t)$ ▷ from narrowest to broadest realization of d_t

Creator: plan and review the implementation

$\mathit{excluded} \leftarrow \varnothing$
for $s = 1, \ldots, R_s$ **do**
  $j \sim q_{\alpha_b}$ over ranks not in $\mathit{excluded}$; $b_t \leftarrow b^{(j)}$; $I_t \leftarrow (d_t, b_t)$ ▷ growth intent
  $Q \leftarrow \mathrm{Creator.Propose}(R_{t-1}, \mathcal{H}_{<t}, I_t)$ ▷ select earlier operations, design contrasts
  if $Q =$ INFEASIBLE **then**
    $\mathit{excluded} \leftarrow \mathit{excluded} \cup \{j\}$; **continue**
  end if
  for $r = 0, \ldots, R_p$ **do**
    $(\mathit{decision}, \mathit{feedback}) \leftarrow \mathrm{Creator.ReviewProposal}(R_{t-1}, I_t, Q)$
    if $\mathit{decision} =$ ACCEPT **then**
      goto line 24
    end if
    if $r = R_p$ **then**
      return FAILURE
    end if
    $Q \leftarrow \mathrm{Creator.ReviseProposal}(Q, \mathit{feedback})$
  end for
end for
return FAILURE

Creator: implement, generate, and review

$R_t \leftarrow \mathrm{Creator.Implement}(R_{t-1}, Q)$ ▷ operations, scene generation, and generators
for $r = 0, \ldots, R_r$ **do**
  $D_t^{\mathrm{train}}, D_t^{\mathrm{test}} \leftarrow \mathrm{Generate}(R_t)$; $W_t \leftarrow (G_t, D_t^{\mathrm{train}}, D_t^{\mathrm{test}})$
  $(\mathit{decision}, \mathit{feedback}) \leftarrow \mathrm{Creator.ReviewWorld}(W_t, I_t, Q)$ ▷ rendered cases and support evidence
  if $\mathit{decision} =$ ACCEPT **then**
    break
  end if
  if $r = R_r$ **then**
    return FAILURE
  end if
  $R_t \leftarrow \mathrm{Creator.Revise}(R_t, \mathit{feedback})$
end for

Automatic validation and acceptance

for $r = 0, \ldots, R_v$ **do**
  if $\mathrm{Validate}(W_t)$ passes **then**
    break
  end if
  if $r = R_v$ **then**
    return FAILURE
  end if
  $R_t \leftarrow \mathrm{Creator.Repair}(R_t, \mathrm{diagnostics})$; regenerate W_t
end for
$\mathcal{H}_{<t+1} \leftarrow \mathcal{H}_{<t} \cup \{(D_t^{\mathrm{train}}, I_t, B_t)\}$
return $W_t, \mathcal{H}_{<t+1}$

Algorithm 1 lists one step of the worldline as described in Section 3. Lines 1–4 are the Director of Section 3.1. Lines 5–23 are the proposal and its review, including the retry with another breadth candidate when the Creator finds the intent infeasible. Lines 24–35 are implementation, generation of the training and test sets, and the review of the generated observations. Lines 36–44 are the automatic checks. We use $R_s = 3$ breadth candidates, $R_p = 1$ proposal revision, $R_r = 1$ revision of the generated world, and $R_v = 2$ repairs after validation. When a limit is reached, the step returns FAILURE and the world is not added to the worldline. The sampling distributions q_{α_d} and q_{α_b} are given in Appendix A.3, what each role sees in Appendix A.2, the training and test generators in Appendix A.4, the automatic checks in Appendix A.5, and the prompts of each call in Appendix A.6.

A.2 What Each Role Sees

The Director and the Creator work in separate sessions. The Director writes both candidate lists in one session, and the Creator writes its proposal, implementation, and revisions in another. For the reviews, the Creator opens a new session. This session first reviews the proposal, and later reviews the generated world against the same growth intent and the proposal it approved.

Role	Available information	Output passed forward
Director	History $\mathcal{H}_{<t}$: the training sets of earlier worlds, the growth intents selected for them, and its own record of concepts	The record of concepts B_t, the ranked depth and breadth candidates, and the growth intent I_t
Creator	The growth intent I_t, the previous codebase R_{t-1}, the training sets, and the records of how earlier worlds were built and run	The proposal and the new codebase R_t
Review	The growth intent, the proposal, the codebase, and the generated training and test sets	Acceptance, or required revisions with supporting evidence
Learner	Training pairs, retained code and notes, and the test problem's support pairs and query input	Updated code and notes, and a query prediction

The Director's record B_t lists the concepts shown in earlier training sets, each with the observations that support it. The Director keeps this record and revises it at every age. Each depth candidate states which concepts in this record it builds on. The Creator receives these concepts with the growth intent.

A.3 Sampling a Candidate

The depth and breadth controls, α_d and α_b, each sample one intent from an ordered candidate list. With C candidates and a control value $u \in [0, 1]$, the central rank is

$$\mu(u) = 1 + (C - 1)u.$$

The probability of sampling rank k around this center is

$$q_u(k) = \frac{\exp\left(-\frac{(k-\mu(u))^2}{2\sigma^2}\right)}{\sum_{j=1}^{C} \exp\left(-\frac{(j-\mu(u))^2}{2\sigma^2}\right)}, \qquad k = 1, \ldots, C.$$

Depth sampling uses $u = \alpha_d$ and breadth sampling uses $u = \alpha_b$. We use $C = 20$ and $\sigma = 2$. So for example, if $u = 0.1$ or 0.9, the central ranks are 2.9 and 18.1.

If the Creator reports that the intent cannot be implemented (for example, it might require concepts that do not exist in the previous worlds), that breadth candidate is excluded and the probabilities are renormalized over the remaining candidates, with the depth intent fixed.

A.4 Training and Test Generators

The Creator writes the code of the generators that produce the training and test sets. This code specifies the distribution of the task programs and does not bound the number of observations, so

each set can be made as large as needed. The distribution is written as a set of sampling conditions, and each condition describes one kind of case that the world must show. For example, for a rule that depends on contact, touching objects and objects separated by a gap form two conditions. Each condition has a purpose, a minimum number of instances, and a sampling weight. The generator first produces the minimum number for every condition, so that every required case appears. It then fills the remaining instances in proportion to the weights. Some conditions are meant to be compared, such as the touching and the separated cases above. These conditions are linked, so the generator produces them as matched examples.

Training and test sets are generated separately. For a test problem, the generator fixes one task program and produces the support pairs and the query under it, recomputing in each scene which objects the program selects and what it produces. Each observation pair consists of rectangular integer arrays x, y of the same shape, with a fixed canvas size within a world. A learner can receive these arrays directly as grids or as rendered images. Following the convention of ARC in the evaluation of LLM agents, the agent harnesses in this paper receive them as grids.

A.5 Automatic Validation

After final review, a heuristic validator re-executes the candidate world and compares it with its parent. The reviews judge what the world teaches, and the validator catches mechanical errors that a review by reading can miss. It checks the code interfaces and size limits, deterministic replay, separation of training and test data, the format of observations and answers, and coverage of the specified sampling conditions. It also checks that the generated observations differ from those of the parent world, so that each accepted world shows something new. Failed checks return diagnostics to the Creator for repair, followed by another validation run.

A.6 Prompts

```
You are one role inside an artificial benchmark designer that constructs a
continuing genealogy of worlds. Each accepted decision becomes developmental
substrate for the worlds that follow. Read the accepted public evidence before
deciding what should develop next, and reason from what a learner can recover
from observable behavior.

The world develops alongside that knowledge. Earlier regularities recur,
independent lines can appear, and a structure learned at one age can become a
working constituent of a concept at a later age. Development may extend a
causal process, change which entities exist, alter what can be perceived,
introduce a new representation, or make several established lines interact.
Recurrence, consolidation, branching, composition, and conceptual revision
are all available developmental moves.

Explanations therefore have ancestry. Continuity makes earlier knowledge
useful, while conceptual invention keeps the space of explanation open. A new
concept may correspond to a familiar object or activity, or to a structured
regularity that has no established everyday name. Its scientific content lies
in the public distinctions it explains and the predictions those distinctions
support.
```

A.6.1 Depth and Breadth Candidates

This prompt produces and ranks the depth candidates of Section 3.1. The prompt calls the Director's record B_t the *basis*, and each candidate states which concepts in B_t it builds on.

```
Inspect the accepted arrays and contact sheets as needed. Update the basis
through {parent_world_id}; do not merely append a new summary. Keep only claims
needed to reason about future development. Each evidence reference must name a
world, record, or archived public artifact that supports the statement.

Then produce exactly {candidate_count} possible depth relations. Rank 1 is the
```

```
slowest legitimate continuation of the established basis. Rank
{candidate_count} is the earliest higher-order use that the accepted genealogy
has prepared. Intermediate ranks move gradually between them. A higher rank
reflects deeper uptake of prior knowledge, not a larger scene, new theme, or
unrelated mechanism. Candidate basis IDs may cite claims and changes, not open
uncertainties. The chronological list preserves relations that have fallen out
of the rolling basis; inspect their public evidence before bringing one back
into the current basis. Keep setting and concrete realization open for the
breadth choice and Creator.
```

This prompt produces and ranks the breadth candidates with the depth intent fixed.

```
The breadth control value is hidden. Produce exactly {candidate_count} successor
relations for {parent_world_id} -> {world_id}. Every candidate must realize the
same locked developmental relation over the same basis. Vary how far that
relation expands or reorganizes the present explanatory world.

Rank 1 is the smallest coherent extension. Rank {candidate_count} is the
broadest coherent extension supported by the genealogy. Build the set from
public behavior before any source vocabulary is available. Include genuinely
different explanatory directions when the genealogy supports them. A successor
may develop dynamics, perception, ontology, representation, agency, social
interaction, biological organization, or an emergent regularity; these are
possibilities rather than bins to fill. Do not fill the ordering with renamed
apparatus, selected-region motion, hidden decoders, or longer programs that
create no new explanatory demand.

Each candidate states one falsifiable public relation and the evidence needed
to distinguish it. It leaves concrete entities, morphology, layout, constants,
algorithms, and repository structure to the Creator.
```

A.6.2 Creator Proposal and Implementation

Before proposing, the Creator is instructed to distinguish the structure of the code from the relations that the observations show.

```
The inherited repository supplies working explanatory material, not a required
visual theme. Reusing source does not require retaining inherited nouns,
apparatus, or scenery. Choose a realization in which the locked relation is
ordinary and central to the scene. A familiar source symbol may therefore
support a living, natural, social, game-like, or formally organized successor
when its public behavior truly realizes the selected relation.

Keep computational, representational, and explanatory hierarchy distinct.
Nested source calls do not alone create a higher-level concept. A detailed
image governed by the old whole-region motion is still the old relation. The
intended relation should remain evident if suggestive source names are
anonymized.
```

This prompt produces the proposal of Section 3.2. In the proposal, `realization` describes the world and its problem family, including how earlier operations contribute to the new problems. The `decisive_contrasts` specify input conditions under which omitting or misapplying a dependency changes the output.

```
Design the world and its family of problems together. As relations
accumulate, develop problems that use them in increasingly varied and
interdependent combinations. A result established through one relation can
determine the participant, interpretation, condition, or transformation
required by another. The input should provide the evidence from which these
intermediate results can be constructed.
```

```
In realization, briefly describe the family of problems this world will
support. Identify the inherited computations used in composed cases, what
their intermediate results determine, and which valid variations change the
required combination or its outcome. Compare with the parent's released
experience so that useful established demands remain available as the focal
relation develops. Use decisive_contrasts to give public conditions under
which omitting or misapplying a claimed dependency changes the prediction.
```

This prompt guides the implementation of the approved proposal.

```
Implement the proposed realization, inherited explanatory work, visible
participant binding, decisive contrasts, and concise exact private provenance.
Implement the proposed problem family as well as its direct witnesses. Support
valid variation in the participating structures, their bindings, and the
dependencies between computations. Where a relation composes, allow the
relevant intermediate result to vary and determine the subsequent operation.
Use reusable world computations and scene construction so the release can
sample these cases across different configurations. Exercise the proposed
decisive contrasts in the actual x -> y implementation before final release
authoring.
```

A.6.3 Training and Test Generation

The Creator is asked to include composed applications throughout the generated training and test sets. To check a proposed dependency, it is instructed to run both the intended task program and a simpler program that omits or fixes an intermediate result, then sample inputs where they differ.

```
Design learning and problem populations from the accumulated world, its accepted
realization, and the parent's released experience. Learning should expose the
constituent relations and their connected use through direct witnesses,
informative contrasts, and composed applications. Evaluation should exercise
the developed problem family, including cases where one inferred result
changes what must be computed next. Give these cases a recurring role in the
sampled population, alongside simpler cases suited to this age.

Vary valid combinations, bindings, and intermediate outcomes beyond changes
of appearance. Sample configurations that exercise the actual choices in the
rule rather than a small set of templates that always have the same reduced
solution. A single world-wide rule can support such variation through the
structures supplied in x. When compositions differ across problems, sample
the shared rule once per problem and instantiate its supports and query under
that rule. Training and support evidence should establish the relations and
conventions needed to solve those problems.

Trace representative composed learning and query cases from x to y. Identify
the intermediate results that determine later selection, interpretation, or
transformation. Compare a claimed dependency with a plausible reduced
procedure that skips it, keeps it fixed, or reuses an earlier state. Execute
both accounts on valid varied scenes and locate where their predictions differ.
Use those conditions in the ordinary problem sampler and provide support
evidence for the governing relation.
```

A.6.4 Review

Proposal review checks the use of inherited operations and the contrasts specified in the proposal.

```
Inspect the cited source structures, runtime traces, public arrays, and source
ancestry. Accept when the proposal honestly grounds every locked basis item and
its decisive contrasts can establish the selected learner-facing relation.
Request revision only when one relationship-level correction is necessary.
```

Generated-world review checks whether the input and the shared task program determine each output, and whether the supports provide evidence for that program.

```
Inspect the output-producing source for every problem family. Trace the
selections, arguments, branches, and random draws that affect y, and check
that public x and the problem's shared rule determine them. Examine any
dependence on record position, support/query role, or per-example private
state. When a hidden choice appears to alter y independently, hold the shared
rule and x fixed and test that path by varying the choice. Request revision
with the source location and affected records if it changes the answer.
Matching condition metadata, function names, or deterministic replay alone
is insufficient. Separately inspect whether the supports provide evidence
for the rule used on the query. Creator retains responsibility for the
distribution and support/query scene design.

For revision, give one relationship-level required change and the record IDs
that reveal it.
```

B World Growth Experiments

All worldlines are generated with GPT-5.6 Sol. We measure how a worldline grows with two metrics. Growth follows the dependencies among the computations that generate a world's training and test sets, and Diversity compares the functions in the world's operation library. Both include every function that generates observations.

B.1 Seed World

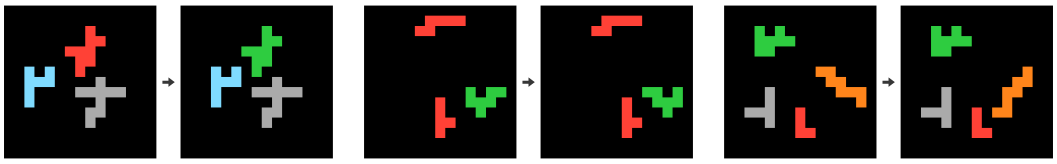


Figure 6: Three observations from the seed world, illustrating translation and recoloring (left), reflection and translation (middle), and rotation (right).

The control sweep starts from a world of colored objects on a 15×15 grid (Figure 6). Zero denotes the background, and values $1, \ldots, 9$ denote colors. An object is a connected, single-colored shape containing four to ten cells within a bounding box of at most 5×5 cells. Shapes are grown by adding adjacent cells. They are asymmetric under rotation and reflection, making the effects of these transformations visible. Initial scenes place objects apart, with a background margin around the grid. Objects may share a color, so their spatial extent also matters for distinguishing them.

The initial operation library contains the four transformations in Table 2. Each changes one selected object and keeps the others. Rotation and reflection operate within the object's local bounding box and retain its top-left position. A quarter-turn exchanges the height and width of the box. Recoloring preserves the object's geometry. Operations keep objects inside the scene without overlap.

Table 2: Operations in the seed world.

Operation	Effect on the selected object
Translation	Move one cell to the right.
Rotation	Rotate clockwise by 90°.
Reflection	Reflect about the vertical axis of its bounding box.
Recoloring	Replace foreground color c by $(c \bmod 9) + 1$.

Training scenes contain one to four objects. A transition applies either one transformation or a sequence of two distinct transformations, with the affected object selected at each step. The training distribution includes examples of each transformation and of their compositions. Each operation must change the rendered scene, and the final scene must differ from the input. The learner receives the pairs as an ungrouped collection and infers the transformations and the objects they affect.

Test problems share an operation sequence and an object-selection rule across their supports and query. The selection rule chooses either the first or the second object when bounding-box positions are ordered by row and then column. Every example contains two to four newly sampled objects, and the selected sequence acts on the same chosen object within that example. Thus, a problem may require rotating the second object and then recoloring it, with the supports demonstrating this program on different shapes and arrangements.

B.2 Growth

Growth measures how far each world builds on earlier results. Composed examples are chosen so that ignoring the earlier result gives a wrong output (Section 3.2). A learner therefore has to use the earlier concept to predict them.

Computation graph. We construct a directed graph whose nodes are computations reached during generation and whose edges are value and control dependencies. Starting from each function's returned values and state changes, we trace the computations and conditions behind them across the operation library. Structural matching normalizes local identifiers and preserves literal values.

Expressions and functions are matched against the earlier worlds $W_0, \ldots, W_{t-1}$ using normalized structure. Matching a function also checks the functions it calls during generation. A matched computation is treated as an earlier result. Before measuring computation length, we inline calls to new functions, so that computation length counts computations rather than how the code is divided into functions. Nodes with the same structure and the same dependencies are counted once.

Computation length above earlier results. Let V_t be the graph's nodes, $P(v)$ the predecessors of node v, and $h(v)$ an indicator that v is an earlier result. Whether a node depends on an earlier result is

$$b(v) = h(v) \vee \bigvee_{u \in P(v)} b(u). \tag{1}$$

Its computation length above an earlier result is

$$\ell(v) = \begin{cases} 0, & h(v) = 1 \text{ or } b(v) = 0, \\ 1 + \max\limits_{\substack{u \in P(v) \\ b(u)=1}} \ell(u), & \text{otherwise.} \end{cases} \tag{2}$$

The world's contribution to Growth is

$$\Delta_t = \max\big(\{0\} \cup \{\ell(v) : v \in V_t\}\big). \tag{3}$$

The value Δ_t is the one-step Growth of world t. An earlier result starts at zero, and each subsequent new computation adds one step. For example, two new computations applied in sequence to an earlier result contribute a computation length of two. Computation length resets to zero at each earlier result. The maximum is taken over all nodes, so a long new path anywhere in the world counts. New computations that do not depend on an earlier result contribute zero. Such computations, the nodes with $b(v) = 0$, are grouped by their dependencies on one another. Each connected group is a *new computation tree*, and their number measures how much separate computation a world adds.

Bare variables and literals are not computation nodes. Reached branches and loop bodies contribute source-level dependencies without unrolling runtime iterations. Calls that cannot be resolved or expanded remain single nodes. Growth accumulates the one-step Growth across successor worlds,

$$\mathrm{Growth}_t = \sum_{\tau=1}^{t} \Delta_\tau. \tag{4}$$

The seed provides the initial reference library and has zero Growth.

B.3 Diversity

Function inventory. For each world, we collect the latest source definition of each function, together with earlier aliased definitions observed during generation. This includes functions retained from preceding worlds. We count identical normalized function bodies once. For this deduplication, we normalize local identifiers and remove documentation, type annotations, decorators, and validation-only statements while preserving literal values, external names, and computation structure.

Code embeddings. We encode one representative source implementation per normalized body using the pretrained 137M-parameter CodeRankEmbed model (Suresh et al., 2025). We remove comments and docstrings. Let $\phi(f_i) \in \mathbb{R}^{768}$ be the CLS embedding of this code. We normalize each embedding and collect them as rows:

$$e_i = \frac{\phi(f_i)}{\|\phi(f_i)\|_2}, \qquad E_t = [e_1, \ldots, e_{n_t}]^\mathsf{T} \in \mathbb{R}^{n_t \times 768}. \tag{5}$$

Each function is encoded separately, with calls retained in its source.

Within-world averaging. Pairwise code-embedding similarity has been used to assess the diversity of generated programs (Chon et al., 2024). We apply it to the functions within each world through the cosine similarity matrix

$$K_t = E_t E_t^\mathsf{T} \in \mathbb{R}^{n_t \times n_t}, \qquad (K_t)_{ij} = e_i^\mathsf{T} e_j. \tag{6}$$

Diversity is the mean pairwise cosine distance,

$$\text{Diversity}_t = \frac{2}{n_t(n_t-1)} \sum_{1 \le i < j \le n_t} \left(1 - e_i^{\mathsf{T}} e_j\right), \tag{7}$$

which averages $1 - (K_t)_{ij}$ over the off-diagonal pairs with equal weight. The score is the expected distance between two different functions sampled uniformly from the current library. Section 4.1 reports the change from the seed world, $100\,(\text{Diversity}_t - \text{Diversity}_0)$, as Diversity gain.

B.4 Control Sweep

The sweep contains one worldline for each of nine α_d–α_b settings. At each age, the curves in Figure 3a–b average three worldlines with the same value of the plotted control and different values of the other control. Figure 7 reports each worldline separately. In the final world, high α_b exceeds low α_b in Diversity at every fixed α_d.

Growth rises with α_d whenever breadth adds new operations for deeper candidates to build on. At the lowest breadth, few such operations are added. Growth also rises with α_b at every fixed α_d, since a new operation usually takes earlier results as inputs.



Figure 7: Growth and Diversity in the final world for all nine α_d–α_b settings. (a) Growth sums the one-step Growth of every world over the worldline. (b) Diversity is the mean pairwise cosine distance between function-code embeddings in the final available library.

B.5 Generated Worldlines in the Sweep

Figures 8–16 show the worldline of each of the nine settings in the control sweep. Each pair shows an input scene and its output, and ages increase from left to right and top to bottom.

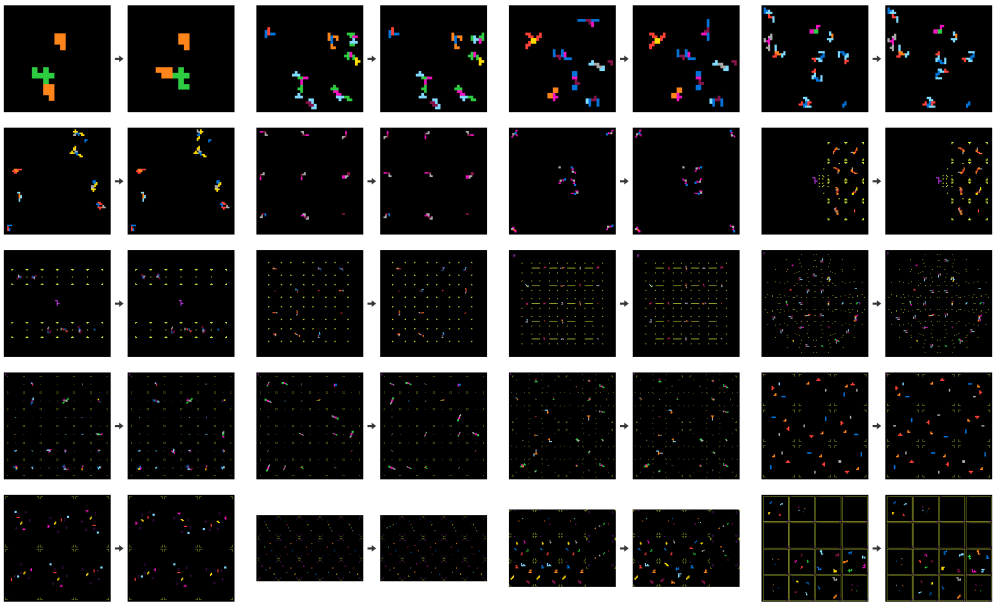


Figure 8: $\alpha_d = 0.1, \alpha_b = 0.1$.

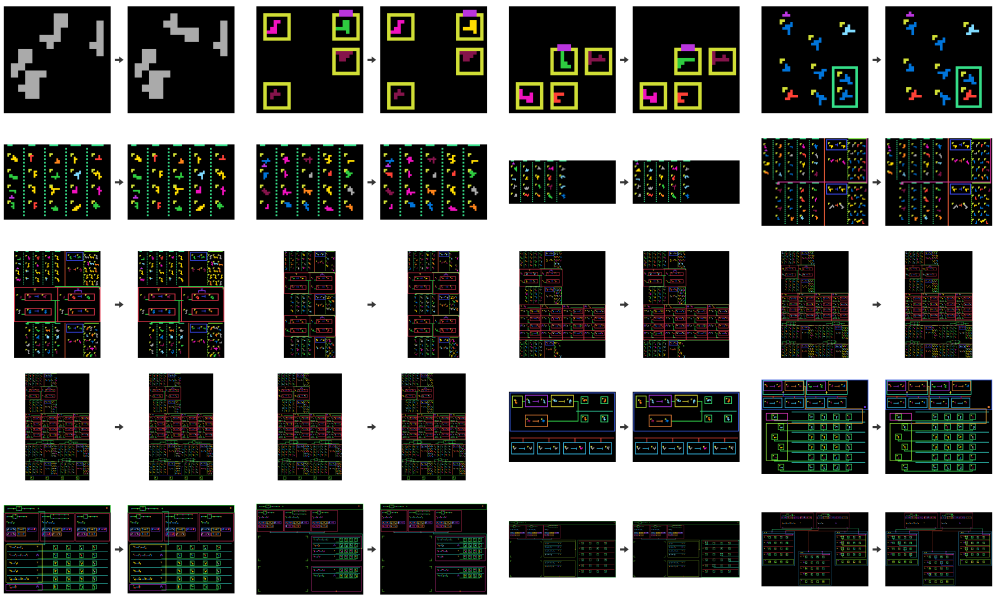


Figure 9: $\alpha_d = 0.5, \alpha_b = 0.1$.

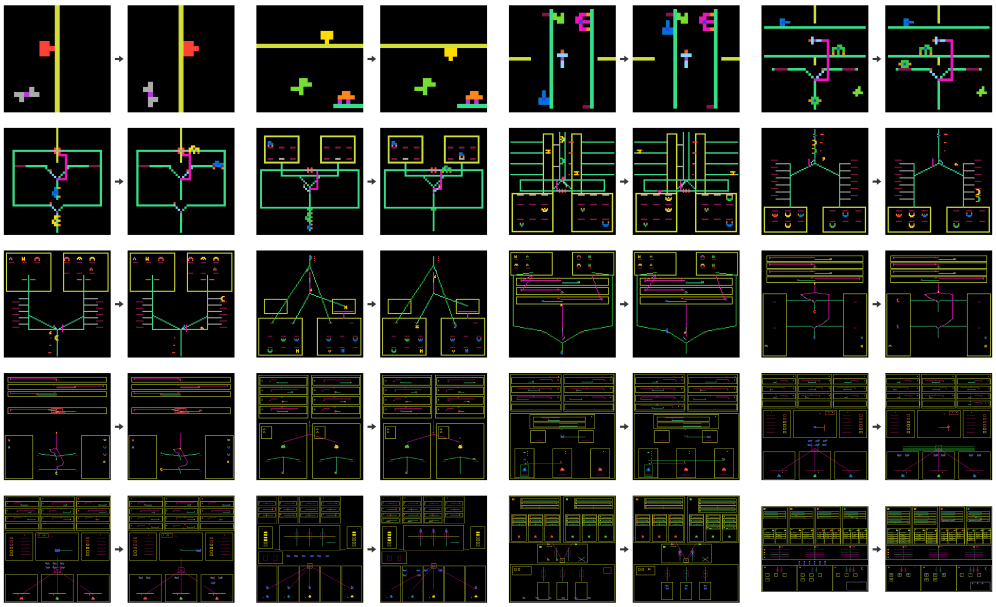


Figure 10: $\alpha_d = 0.9$, $\alpha_b = 0.1$.

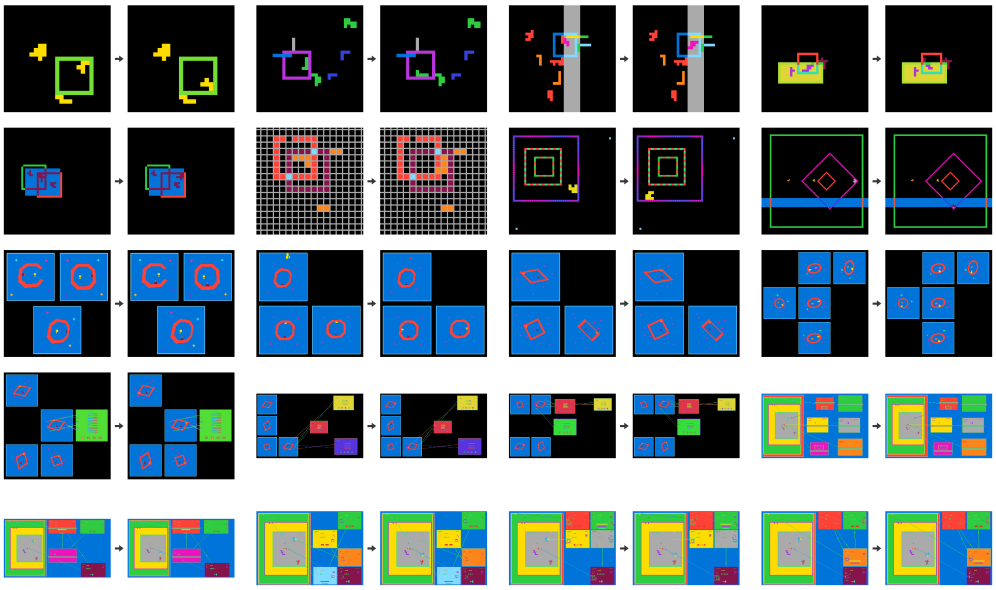


Figure 11: $\alpha_d = 0.1$, $\alpha_b = 0.5$.

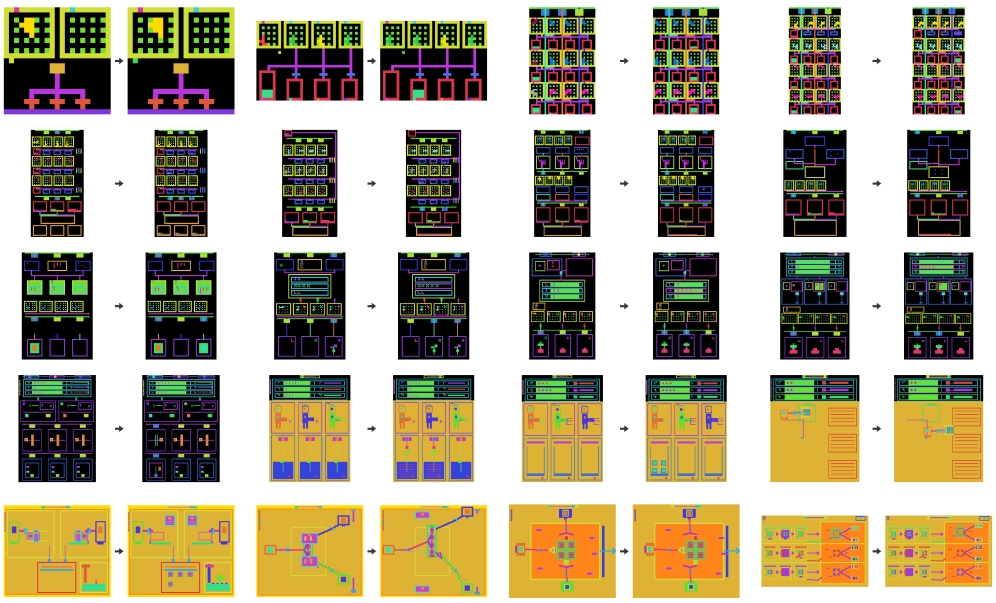


Figure 12: $\alpha_d = 0.5$, $\alpha_b = 0.5$.

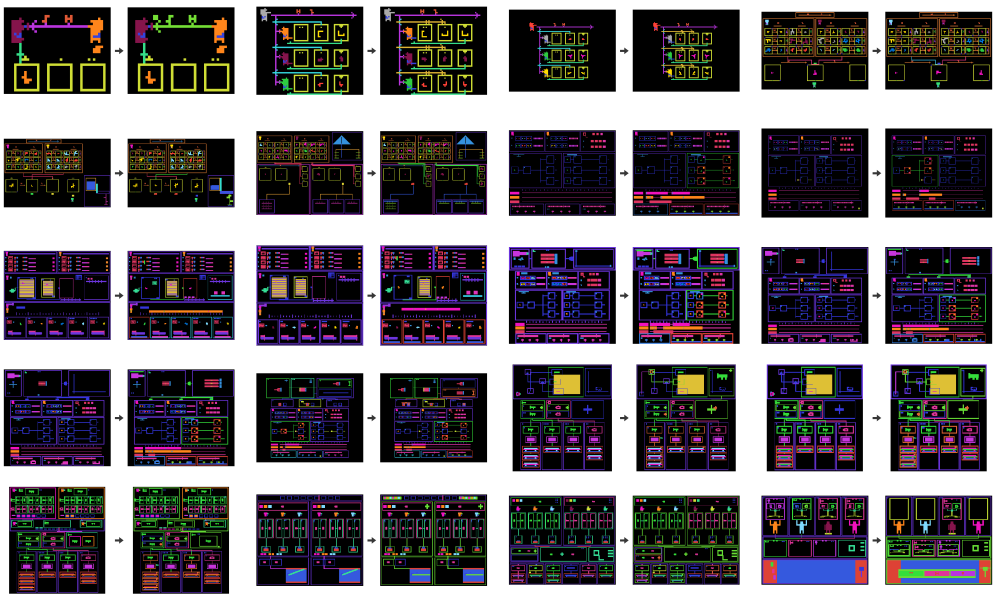


Figure 13: $\alpha_d = 0.9$, $\alpha_b = 0.5$.

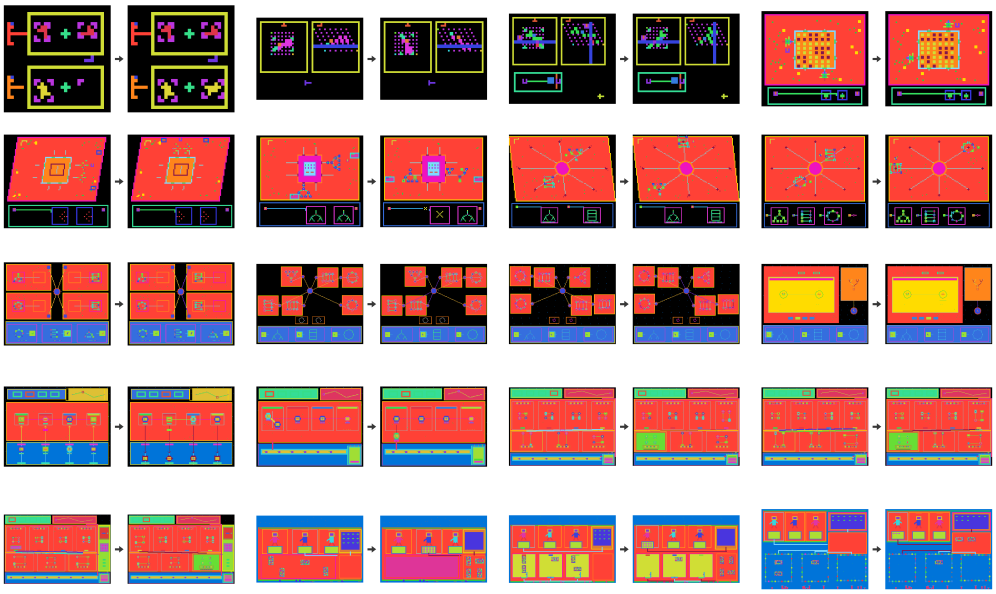


Figure 14: $\alpha_d = 0.1$, $\alpha_b = 0.9$.

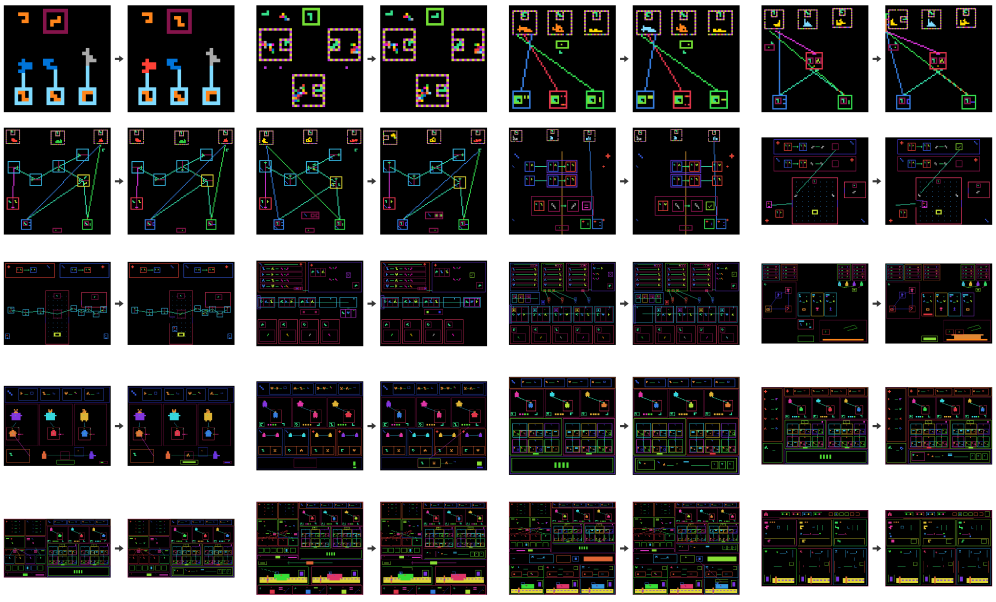


Figure 15: $\alpha_d = 0.5$, $\alpha_b = 0.9$.

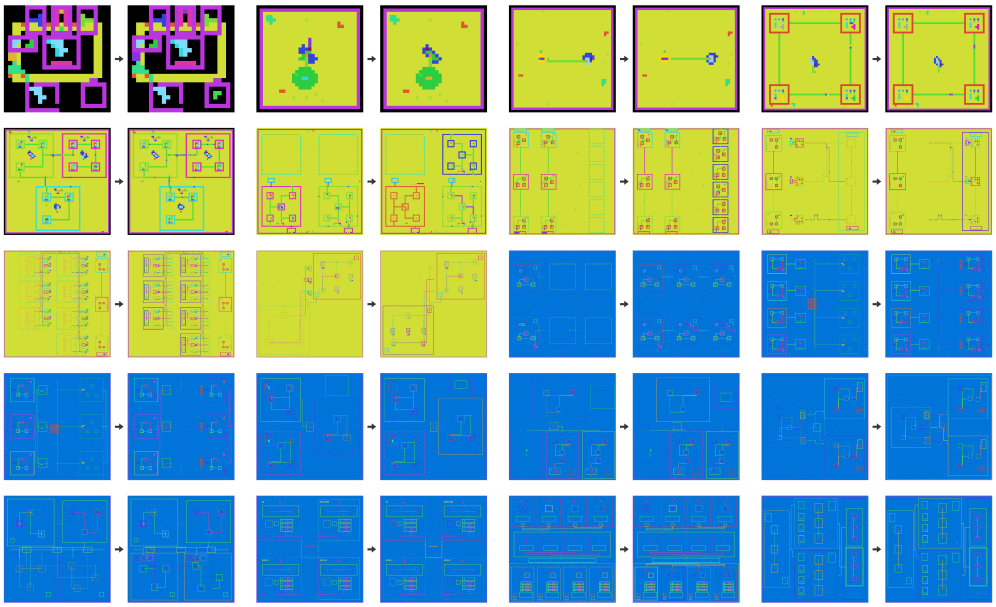


Figure 16: $\alpha_d = 0.9$, $\alpha_b = 0.9$.

B.6 RANK EXPERIMENT

The depth and breadth controls act by choosing a rank in the Director's candidate list. To test whether the rank itself changes the implemented world, we hold the parent world fixed and vary only the selected rank. We take three accepted worlds of the same early age from different worldlines as parents. For each parent, the Director produces one depth candidate list, and the Creator implements ranks 1, 10, and 20 with the breadth rank fixed at 10. The Director then produces one breadth candidate list for the rank-10 depth intent, and the Creator implements breadth ranks 1, 10, and 20. Depth rank 10 with breadth rank 10 appears in both lists, so each parent has five successors.

For each successor we measure one-step Growth, the number of new computation trees, and the change in Diversity from the parent. Table 3 lists all fifteen successors, and Figure 3c–d report the means. Growth and the number of new computation trees rise from rank 1 to rank 20 for every parent on the breadth list, and for two of the three parents on the depth list.

Parent	Depth rank	Breadth rank	One-step Growth	New computation trees	Diversity gain
A	1	10	21	31	$+0.50$
A	10	1	10	15	$+0.18$
A	10	10	12	25	$+0.68$
A	10	20	22	37	$+0.31$
A	20	10	63	33	$+0.21$
B	1	10	11	7	-0.15
B	10	1	12	7	-1.69
B	10	10	13	15	$+0.27$
B	10	20	16	27	$+0.65$
B	20	10	10	13	$+0.28$
C	1	10	9	9	$+0.19$
C	10	1	13	9	-1.18
C	10	10	20	19	$+0.61$
C	10	20	33	37	-0.06
C	20	10	28	28	$+0.09$

Table 3: Successors implemented from the same parent at selected candidate ranks. Diversity gain is the change from the parent world.

Table 4 shows the selected candidates for parent C, whose established concept is a quarter-turn applied to an attached form. Along the depth list, the turn is first repeated in new layouts, then applied to several composites independently, and finally used to decide whether another participant can cross a boundary. Along the breadth list, the same depth intent is realized first with the existing composites, then under partial occlusion, and finally through several agents coordinating their turns.

Depth candidates, breadth rank 10	
Rank 1	Recur with the established single attached-form quarter-turn across new positions and incidental layouts, preserving the same binding and outcome.
Rank 10	Establish that several spatially separate valid carrier–attachment composites each undergo the inherited turn independently within one transition.
Rank 20	Make the opening produced by the attachment-governed turn determine whether a separate visible participant can cross the carrier boundary, so the inherited turn becomes a constituent of an operational closure relation.
Breadth candidates, depth rank 10	
Rank 1	Several clean, spatially separate valid carrier–attachment composites in one scene each execute the inherited clockwise quarter-turn in their own carrier-centered frame.
Rank 10	Independent carrier–attachment composites persist through partial occlusion, so their visible fragments predict separate turns even when parts of carriers or attachments are temporarily inaccessible.
Rank 20	Several agents use public references to coordinate independent carrier-local turns whose combined state realizes a shared outcome unavailable from any one turn alone.

Table 4: Selected candidates for parent C. Each row quotes the Director's candidate at that rank.

B.7 Ablation Conditions

Each variant keeps the seed world, the model, the automatic validation of generated worlds, and the review of the generated training and test sets, and changes one part of the Designer. Appendix A.2 lists what each role sees in `Full`.

`w/o Candidate search.` The Director receives the same history but writes one growth intent for the next world. There are no depth or breadth candidate lists, no ranking, and no sampling. The Creator selects operations for, proposes, and implements this intent as in `Full`.

`w/ Code access.` The Director additionally reads, without editing, the parent world's code, the accepted history of code changes, and the execution records that the Creator sees. The candidate lists, sampling, Creator, and reviews are unchanged.

`w/o Director.` One agent receives the history and the parent code and constructs the next world directly. It decides what to develop while implementing it and delivers the code and the training and test generators. There is no growth intent, candidate list, or proposal stage. A separate review checks the generated world before acceptance.

`w/o Full history.` The Director and the Creator see only the immediately preceding world: its training set and selected intents, and, for the Creator, its code. Records, intents, and code snapshots of older worlds are not accessible. The parent code still contains the operations accumulated before it.

B.8 Reuse and Accumulation

We follow operation implementations through the final world of each worldline. Two implementations are the same operation when their normalized bodies match (Appendix B.3). A renamed function keeps its identity, and a changed body is a new implementation. We execute the first 30 training pairs of each world and record the operations used and their caller–callee connections.

For a training pair, an operation is *reused* if its implementation appeared in an earlier world. A current-age operation is a *new composition* if its executed call graph reaches a reused operation, directly or through other new operations. We average these counts over the 30 pairs. To measure accumulation, we collect the distinct new compositions of each age and check, at every later age, how many of them are still in that world's library. Table 1 reports the number collected along the worldline as Introduced, the number still present in the final world as Retained, and their ratio as Retention. Figure 17 plots these quantities by age, and Table 5 lists the accumulation counts.

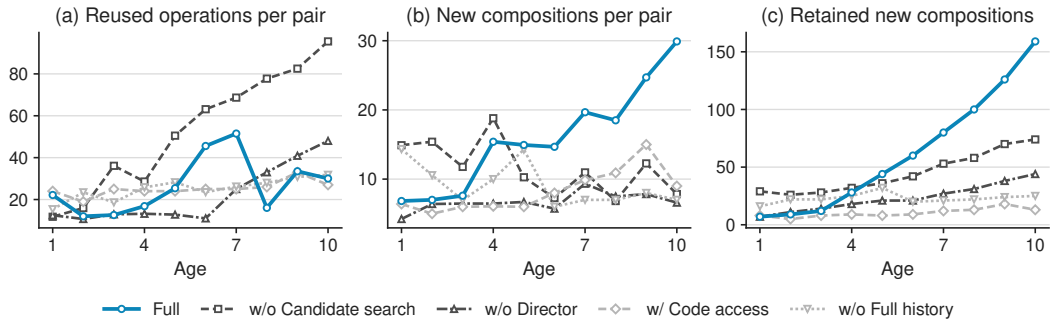


Figure 17: Operation use and accumulation by age. (a) Reused operations and (b) new compositions executed per training pair at each age. (c) Earlier new compositions still in the library at each age.

The two groups of Section 4.1.2 are visible over time. Without candidate search, reused operations per pair rise steadily while new compositions stay low. The worlds keep executing more of what already exists. With code access or parent-only history, new compositions are introduced at every age but the retained count stays flat, since each age replaces most of what the previous age introduced. `Full`'s retained count rises with its introductions from age 4 onward.

Method / Age	1	2	3	4	5	6	7	8	9	10
Distinct new compositions introduced through each age										
`Full`	7	14	22	42	58	74	94	114	140	173
`w/o Candidate search`	29	52	66	90	102	111	125	133	149	158
`w/o Director`	7	18	28	38	49	57	68	77	86	94
`w/ Code access`	8	13	19	26	32	40	50	61	76	85
`w/o Full history`	16	29	41	51	67	73	80	87	95	102
Of these, still present in the library at that age										
`Full`	7	9	12	28	44	60	80	100	126	159
`w/o Candidate search`	29	26	28	32	36	42	53	58	70	74
`w/o Director`	7	11	14	18	21	21	27	31	38	44
`w/ Code access`	8	5	8	9	8	9	12	13	18	13
`w/o Full history`	16	22	22	25	32	20	21	22	24	25

Table 5: New compositions by age. The last column equals Introduced and Retained in Table 1.

B.9 Generation Resources

Table 6 reports the mean input and output tokens, including reasoning tokens, and the summed call time per successor world. `Full`'s planning stage adds Director calls, but its total output is close to that of the variants that keep two roles, because a selected intent reduces the Creator's work. `w/o Director`, whose single agent plans and implements, uses far fewer tokens and less time, and produces less Growth than `Full` and the smallest Diversity gain. The cost of a world does not grow with age over the ten steps, although the history the Director reads does.

Method	Input (M tokens)	Output (k tokens)	Time (min)	Unrevised
`Full`	10.5	93.2	43.7	9/10
`w/o Candidate search`	14.2	96.5	39.2	7/10
`w/o Director`	2.5	38.6	15.9	9/10
`w/ Code access`	7.7	86.8	36.4	7/10
`w/o Full history`	7.1	80.3	34.4	8/10

Table 6: Resources used to generate successor worlds. Resources are means per successor world. Unrevised counts worlds that passed reviews without a revision request.

B.10 ONE-SHOT GENERATION

`One-shot` uses `Full`'s planning, proposal, and reviews to build one world from the seed. It receives the grid and library size limits of `Full`'s final world and the total time budget of all its generation steps. Table 7 compares the resulting worlds.

	`One-shot`	`Full` (final)
Growth	59	355
Diversity gain	+0.20	+3.27
Operations in the library	50	333
Operations executed per training pair	31.2	66.0
Reused operations per training pair	6.0	30.0
New compositions per training pair	2.0	29.9

Table 7: `One-shot` and gradual generation.

`One-shot` adds one layer of operations to the seed. `Full`'s final training pairs execute twice as many operations, drawing on building blocks developed in intermediate worlds.

C Analysis and Explanation of Generated Worldlines

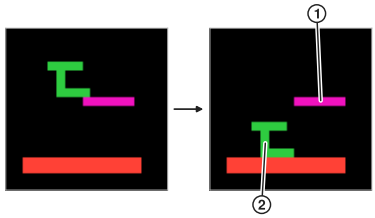
We first follow worlds whose rules build on the rules of earlier worlds, then show how the controls set the pace of this growth, and finally examine worlds where generation goes wrong. In each pair of grids, the input is on the left and the answer on the right, and numbers follow the steps of the rule. Code boxes show each rule in simplified form.

C.1 Conceptual Bootstrapping across Worlds

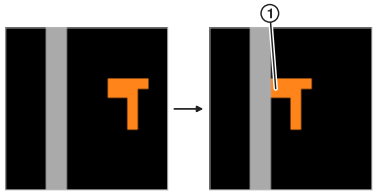
C.1.1 Shelves and Loads, Ages 1 to 14

This worldline grows from the seed world of Appendix B.1 with $\alpha_d = 0.33$ and $\alpha_b = 0.5$. Each world of this worldline adds one or two physical relations, such as support, contact, or a gate that lets a cart pass. We follow it from age 1 to age 14. Along the way, each new world uses the rules of earlier worlds as steps of a new rule.

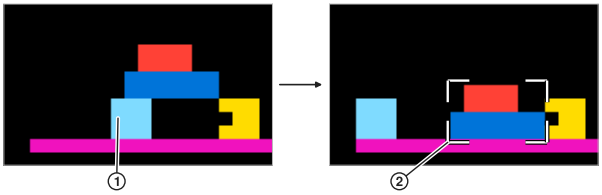
Age 1, support. In the pair on the right, a green piece rests on a magenta shelf above a red floor. ① The shelf slides to the right, away from the piece. ② Nothing is under the piece any more, so it falls straight down until it lands on the floor. So a piece stays where it is while something is right under it, and falls when nothing is.



Age 2, contact. Here an orange piece is pushed to the left, toward a gray post. ① The piece moves until it touches the post and stops there. So a pushed piece moves only until it touches something.

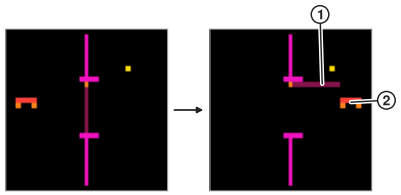


Age 3, a stack of blocks. Age 3 joins these two rules. Each scene shows blocks on a magenta bench. In the pair below, the light-blue block and the yellow block stand on the bench, and a blue plate with a red block on top rests on the light-blue block. The learner must predict where every block ends up after one block is pulled out.

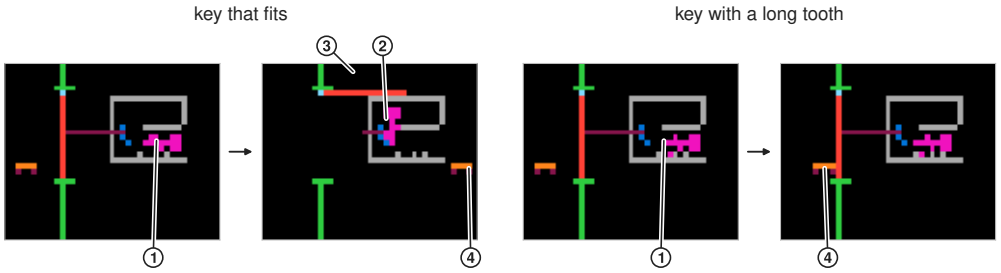


① The light-blue block is pulled out to the left by the push of age 2. ② The plate has lost the only block under it, so the plate and the red block on it fall together until they land on the bench, like the piece at age 1. The yellow block stands on the bench itself, so it stays. The new idea is that support forms a chain. A block stays when a chain of blocks leads from it down to the bench, and it falls when that chain is broken. A learner that has not learned age 1 may guess that whatever sits right above the pulled block falls. That guess fails as soon as another block still holds the plate.

Age 5, a gate. Ages 4 and 5 add walls and gates. At age 5, a dark red gate hangs from a magenta post and closes a doorway, and an orange cart waits on the left. ① The gate swings open about its hinge at the top. ② The cart rolls through the doorway to the right. When the gate stays closed, the cart stops at the gate, because the cart moves by the push of age 2.



Age 6, a key and a gate. Age 6 locks the gate of age 5. Each scene shows a doorway in a green post, closed by a red gate. A dark red bar runs from the gate into a gray lock box and keeps the gate shut. A magenta key lies in the lock, and an orange cart waits on the left. The learner must predict where the cart ends up. The left pair below uses a key that fits the lock, and the right pair uses a key with one tooth that is one cell too long.



① The key is pushed into the lock, toward the gate, until it touches something. ② If every tooth passes over the gray ridges at the bottom of the lock, the key reaches the end of the lock and turns, and the turn pulls the dark red bar back into the box. ③ With the bar gone, the gate swings open. ④ The cart rolls through the doorway to the far right. In the right pair, the long tooth catches on a ridge, so the key does not move at all. The bar stays, the gate stays shut, and ④ the cart stops at the gate. The difference between the two keys is a single cell, which the enlarged keys below show.

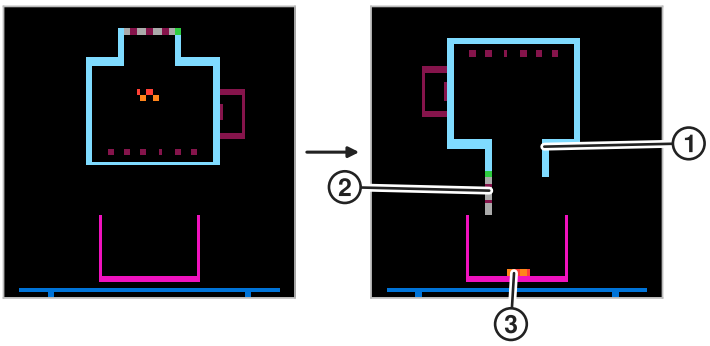


The code of this world writes these steps with the operations of earlier worlds. The key and the cart move by the push of age 2, and the gate turns by the swing of age 5. A key that does not fit therefore stops for the same reason that a pushed piece stops at a post at age 2. A learner that kept the push of age 2 and the swing of age 5 only has to learn how the key turns and pulls the bar.

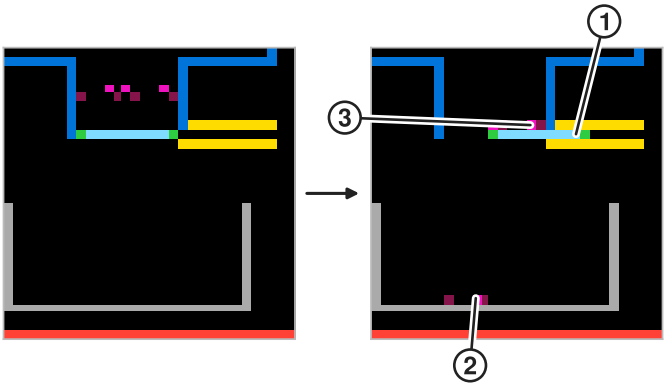
```
# age 6: open the gate with a key, then push the cart
def open_and_pass(scene):
    scene = push_until_contact(scene, scene.key)  # ① age 2
    if scene.key_reached_end:
        scene = turn_and_pull_bar(scene)          # ② age 6
        scene = swing(scene, scene.gate)          # ③ age 5
    return push_until_contact(scene, scene.cart)  # ④ age 2
```

Age 12, pouring. Ages 7 to 11 add a tipping beam, cages, a locked cage, a one-way gate, and carts that are coupled together. From age 12, the worldline turns to small parts that fall. At age 12, a light-blue can holds two bolts, and a stopper closes its mouth at the top. ① The can turns upside down, so its mouth faces down. ② The stopper slides out of the mouth. ③ The bolts fall through the

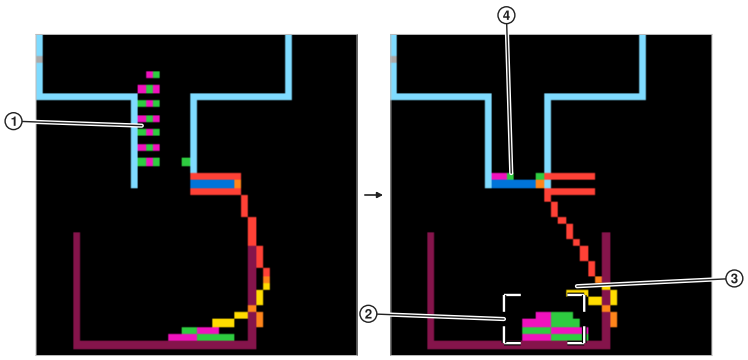
open mouth into the magenta tray. The falling of age 1 now applies to parts inside a container, and the mouth decides whether they can leave.



Age 13, a gap. At age 13, small parts sit in a blue container above an opening at its bottom, and a light-blue bar under the opening closes it. ① The bar slides to the right and opens part of the opening. ② The parts above the gap fall into the gray tray. ③ The parts above the rest of the bar stay on the bar. The position of the gap decides which parts fall.



Age 14, a bin that closes the opening. Age 14 adds feedback to age 13. Each scene shows a light-blue container that holds screws above an opening at its bottom. A red bar with a blue stripe lies beside the opening, ready to slide under it. A red chain links this bar to a yellow lever inside a dark red bin below. The learner must predict how many screws reach the bin and where the bar ends up.



① Screws fall one after another through the open part of the opening. ② They pile up in the bin. ③ When the pile reaches the yellow lever, the lever tips, and the chain pulls the bar under the opening.

④ Once the opening is closed, the screws that are left stay in the container, on top of the bar. Step ① is the rule of age 13, and the code of age 14 calls the function of age 13 for each screw. What age 14 adds is feedback. The screws that have fallen move the bar, and the bar then stops the rest of the screws. A learner that has not learned which parts pass a gap at age 13 cannot tell how many screws reach the bin before the bar closes the opening.

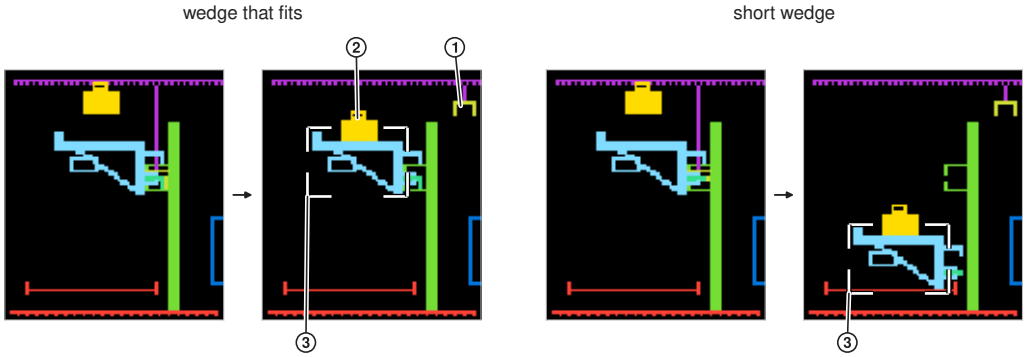
```
# age 14: screws fall until the pile closes the opening
def fill(scene):
    for screw in scene.screws:
        if screw.column in open_columns(scene.bar):  # ① age 13
            scene = drop_into_bin(scene, screw)       # ② age 12
            if pile_touches(scene.bin, scene.lever):  # ③ age 14
                scene = pull_bar_closed(scene)        # ③ age 14
    return scene                                      # ④
```

What this worldline shows. In this worldline, a rule learned at one age returns as one step of a larger rule. Support becomes the end of a chain at age 3. Pushing becomes the motion of a key at age 6. At age 14, the parts that fall through the gap close the gap.

C.1.2 Choosing the Tool, Ages 36 to 38

Later in the shelf worldline, the worlds turn from falling parts to loads, repairs, and a robot that grips and carries parts. At age 38, the robot must choose the right tool to repair a shelf. The rule of age 38 is made of the rules of ages 36 and 37, so we explain those two worlds first.

Age 36, replacing a wedge. Each scene shows a light-blue shelf on a green post. The corner of the shelf sits in a slot of the post, and a wedge in the slot holds the shelf up. The old wedge is missing, and a robot hand that hangs from a purple rail has just put a new wedge into the slot. A yellow crate waits above the shelf. The learner must predict whether the shelf holds the crate. The left pair below uses a wedge that fits the slot, and the right pair uses a wedge that is too short.

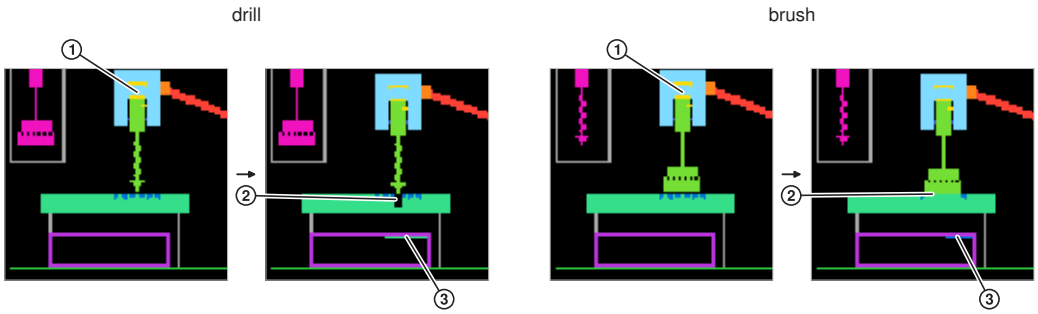


① The hand lets go of the wedge and rises to the top right. ② The crate is lowered onto the shelf. ③ If the wedge fills the slot and touches the post and the shelf on every side, the shelf holds the crate. If the wedge is too short or sits crooked, the shelf and the crate drop until they rest on the red safety bar. The short wedge lacks only the two cells at its lower end, which the enlarged slots below show.



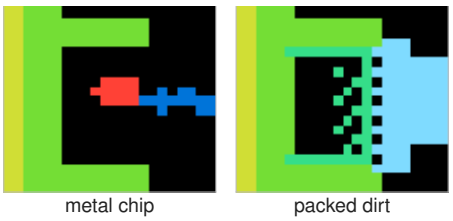
The rule is the support of age 1 on a larger scale. The wedge supports the shelf in the same way that a shelf supports a piece, and a shelf without full support falls until something stops it.

Age 37, drill and brush. Each scene shows a robot arm that holds a green tool head in a light-blue holder. The head is a drill or a brush, and the other head hangs in a gray rack on the left. Below the head lies a flat plate with blue dirt on its top, and a purple tray under the plate catches what the tool removes. The learner must predict what the head does to the plate. The left pair below uses the drill, and the right pair uses the brush.

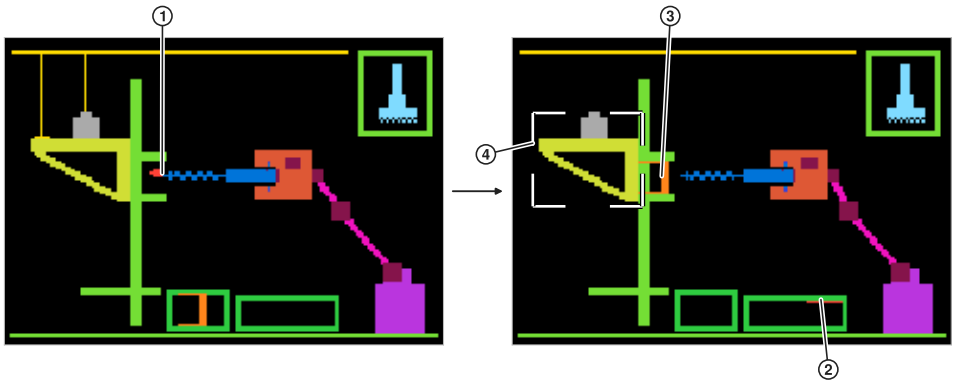


① The head works only when it sits fully in the holder and both yellow latches close. ② The head moves down onto the plate. The drill bores a narrow hole through the plate, and the brush sweeps the dirt off a wide strip of the top. ③ What the head removes falls into the tray. Other scenes vary the conditions. A head that stops short of the plate or is not latched changes nothing.

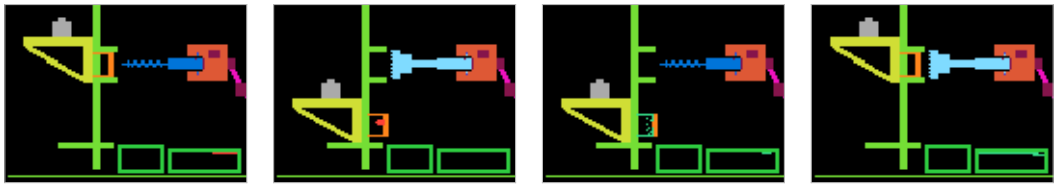
Age 38, choosing the tool. Each scene shows a repair stand. A yellow-green shelf with a gray crate on it hangs by two yellow cords next to a green post. The slot in the post has no wedge, and something blocks it. The blockage is either a small red metal chip or packed dirt on the walls of the slot, which the enlarged slots below show. A robot on the right holds a drill or a brush, and the other head waits in a rack at the top right. Below the stand, the left tray holds an orange spare wedge, and the right tray collects what the tool removes. The learner must predict whether the shelf stays up once the cords are released.



The pair below has a metal chip in the slot and a drill on the robot.



① The robot pushes its head into the slot. ② What the head removes falls into the right tray. The drill removes the whole metal chip but cuts only a narrow channel through the dirt. The brush sweeps away all the dirt but cannot remove the chip. ③ The robot moves the spare wedge into the slot, and the cords are released. ④ If the slot is clear, the wedge seats and the shelf keeps holding the crate. If anything is left in the slot, the wedge cannot seat, and the shelf drops with the crate until it rests on the safety bar. The answers below show every pairing of blockage and head. Only the drill clears the chip, and only the brush clears the dirt. Other scenes vary the conditions. When the slot is clean, either head works, and a head that is not latched removes nothing.



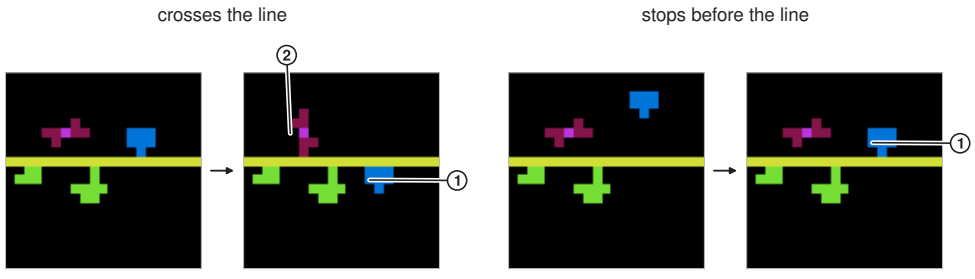
Age 38 joins the two worlds before it. The drill and the brush of age 37 decide what is left in the slot, and the wedge test of age 36 decides whether the shelf holds with what is left. The code below writes the rule of age 38 as these steps, with the age at which each step first appears. A learner that kept ages 36 and 37 only has to learn which head clears which blockage.

```
# age 38: clear the slot, then test the shelf
def service_and_test(scene):
    left = clear(scene.slot, scene.head)        # ① ② ages 37, 38
    scene = insert(scene, scene.spare_wedge)    # ③ age 36
    if seat(scene.wedge, scene.slot, left):     # ④ age 36
        return scene                            # the shelf holds
    return fall_until_support(scene.shelf)      # ④ age 1
```

C.1.3 Switches and Routing, Ages 1 to 5

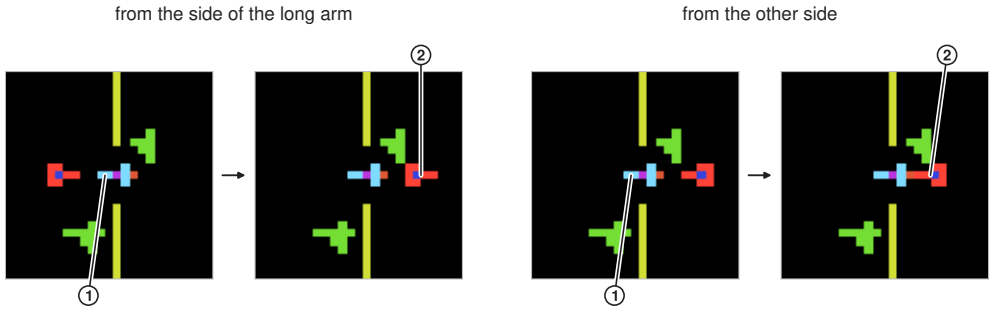
This worldline of the control sweep (Figure 10) grows from the seed world of Appendix B.1 with $\alpha_d = 0.9$ and $\alpha_b = 0.1$. Its first five worlds build a rail yard in which one tram decides where the next tram goes. At age 1, a piece that crosses a line changes another piece. By age 5, a tram that crosses a bridge sets the switch that decides the route of the next tram.

Age 1, crossing a line. Each scene shows a yellow line, a blue piece that moves toward it, and a dark red piece elsewhere. The green pieces never change. The learner must predict the scene after the blue piece moves. The left pair below has a blue piece that crosses the line, and in the right pair it stops before the line.



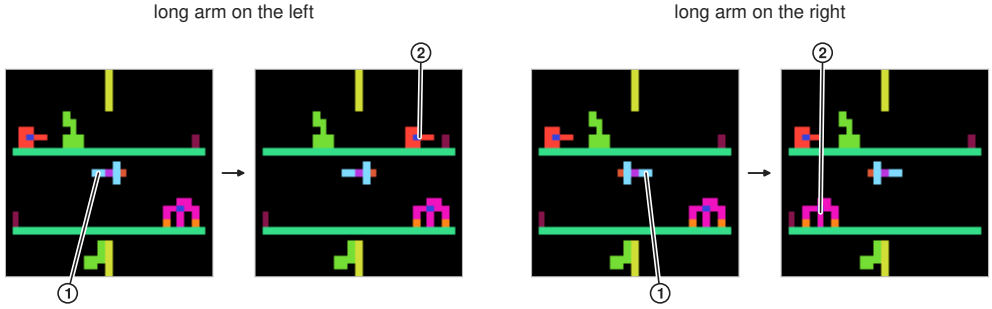
① The blue piece moves a short step toward the line. ② If it crosses the line, the dark red piece turns a quarter turn. In other scenes it flips or slides instead. If the blue piece stops before the line, the dark red piece stays as it is. Crossing the line is what triggers the change.

Age 2, a one-way gate. Each scene shows a yellow wall with a gap and a light-blue gate that sits in the gap. The gate has a long arm on one side and a short orange arm on the other. A red cart moves toward the gap. The learner must predict where the cart ends up. The left pair below has a cart that comes from the side of the long arm, and the right pair a cart that comes from the other side.



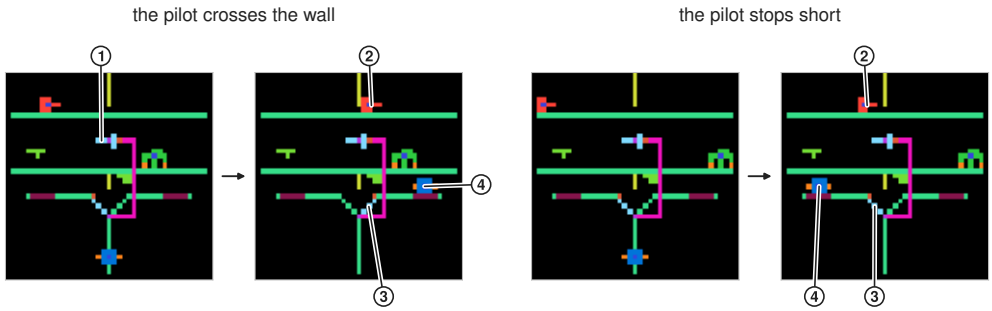
① The long arm of the gate points to the left. ② A cart that comes from the side of the long arm passes through the gap to the far side. A cart that comes from the other side stops when it touches the gate. So the gate lets carts through in one direction only.

Age 3, a signal. Age 3 moves the gate of age 2 between two tracks, where it works as a signal. A red cart waits at the left end of the upper track and a magenta cart at the right end of the lower track. The learner must predict which cart moves. The pairs below differ only in the side of the long arm.



① The long arm of the signal points to one side. ② The cart on that side runs to the other end of its track, and the other cart stays. The direction that the gate of age 2 lets through now chooses the cart.

Age 4, a crossing sets a switch. Each scene has two horizontal tracks and a short yellow wall between them. A red pilot waits on the upper track and a green pilot on the lower track, and the signal of age 3 sits between them. A magenta rod links the signal to a fork below. At the fork, a switch with a light-blue arm lies on either the left or the right branch. A blue cargo tram waits on the stem of the fork. The learner must predict where the pilot and the cargo tram end up. Both pairs below have the same scene, and only the distance that the pilot travels differs.

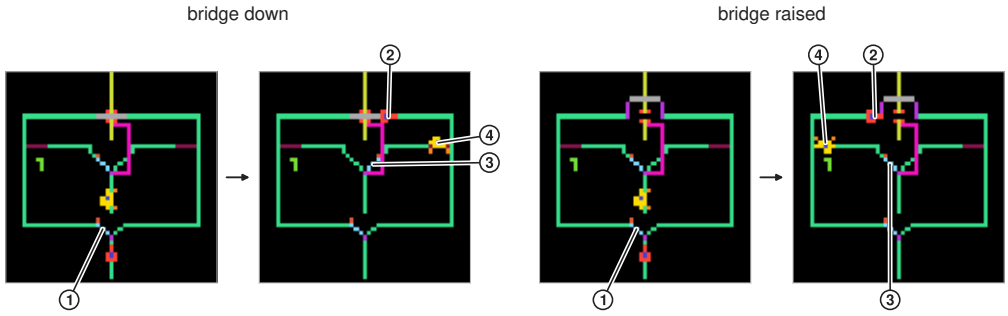


① The long arm of the signal is on the left, so the red pilot on the left moves, as at age 3. ② The pilot moves along its track. In the left pair, it crosses the yellow wall. ③ The crossing sets the switch to the direction in which the pilot crossed. The pilot crossed to the right, so the arm of the switch moves onto the right branch, as the enlarged switch below shows. ④ The cargo tram climbs the stem and takes the branch on which the arm lies. In the right pair, the pilot stops short of the wall. The switch keeps its arm on the left branch, and the cargo tram goes left. When the switch already lies on the side of the crossing, the crossing leaves it as it is, so a crossing sets the switch and does not flip it.



The crossing is the trigger of age 1, and the choice of pilot is the signal of age 3. What age 4 adds is that the trigger writes a direction into a switch, and the switch then routes another tram.

Age 5, a return tram sets the switch for the next tram. Each scene shows a loop of track. At the bottom, a red tram waits below a lower switch, which sends it up the left or the right side of the loop. At the top, both sides of the loop meet at a gray bridge across the yellow line. Inside the loop, an upper switch sits on the stem of a second fork, and a yellow tram waits below it. A magenta rod links the bridge to the upper switch. The learner must predict where the two trams end up and on which branch the arm of the upper switch lies. Both pairs below have the same scene, except that the bridge is down in the left pair and raised in the right pair.



① The arm of the lower switch lies on the left branch, so the red tram goes up the left side of the loop to the left end of the bridge. ② If the bridge is down, the tram crosses it to the right. ③ The crossing sets the upper switch to the direction of the crossing, here the right, as the enlarged switch below shows. ④ The yellow tram climbs its stem and takes the branch on which the arm of the upper switch lies, here the right platform. In the right pair, the bridge is raised. The red tram stops at the left end of the bridge, the upper switch keeps its arm on the left branch, and the yellow tram goes left.



Age 5 uses the rule of age 4 twice. A switch routes a tram, and a crossing sets a switch. One switch now routes a tram whose crossing sets another switch. The lower switch decides from which side the red tram crosses the bridge, and that crossing sets the upper switch for the yellow tram. The code of age 5 keeps the function of age 4 that sets a switch and calls it after the crossing. Without age 4,

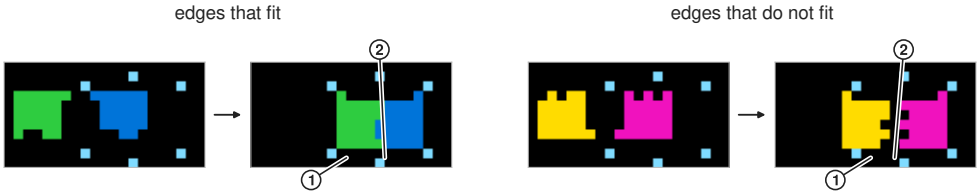
the learner would have to find this from a few pairs in which the red tram, the upper switch, and the yellow tram all change at once.

```
# age 5: a return tram sets the switch for the next tram
def return_and_route(scene):
    route(scene.red_tram, scene.lower_switch)          # ① age 4
    if scene.bridge_down:
        cross(scene.red_tram, scene.bridge)            # ② age 1
        set_switch(scene.upper_switch, scene.red_tram) # ③ age 4
    route(scene.yellow_tram, scene.upper_switch)       # ④ age 4
```

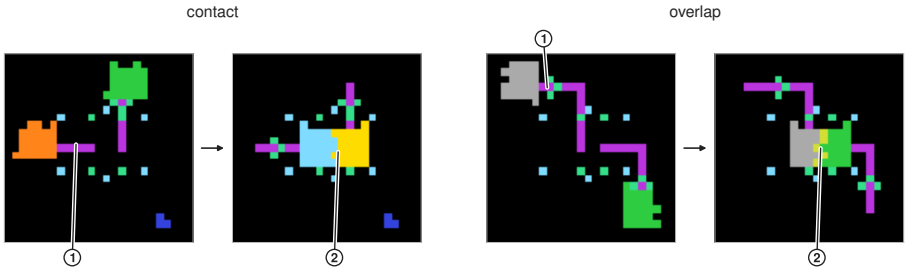
C.1.4 The Evaluation Worldline, Ages 1 to 9

This is the worldline on which we evaluate the learners (Section 4.2). It grows from the same seed world with $\alpha_d = 0.8$ and $\alpha_b = 0.5$. Its scenes look more abstract than those of the shelf worldline, but they are built like a small factory. A piece enters, a route card decides where it goes, and stations along the way run small programs on it. Each world adds one level to this factory, and at age 9 the whole factory of age 8 becomes a single tool that the new world calls twice.

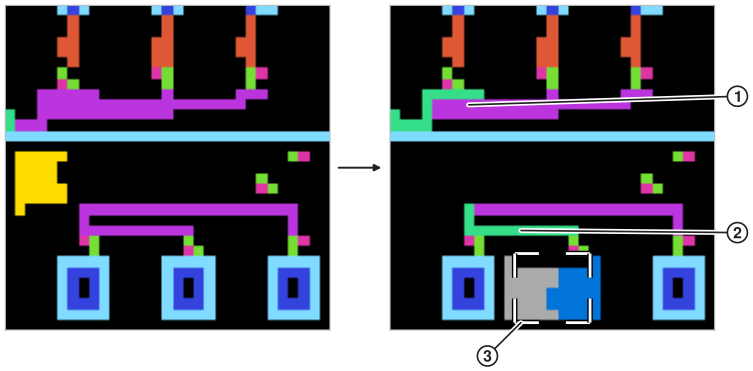
Age 1, plates that dock. Each scene shows two plates with jagged edges and light-blue marks around the middle. ① Each plate turns and flips until its jagged edge faces the other plate, and slides next to the marks. ② If the two jagged edges are exact complements, the plates meet with no gap. If they are not, the plates stay apart. The left pair below has edges that fit, and the edges of the right pair do not fit.



Age 2, routes and contact. At age 2, each plate first travels along its purple route to the middle and then turns into its standard pose. ① The route decides where the plate arrives. ② How the two plates meet decides the rest of the answer. When their edges touch cleanly, both plates change color. When they overlap, the overlapping cells turn yellow-green. When a gap remains, nothing more changes. The left pair below ends in contact and the right pair in overlap.



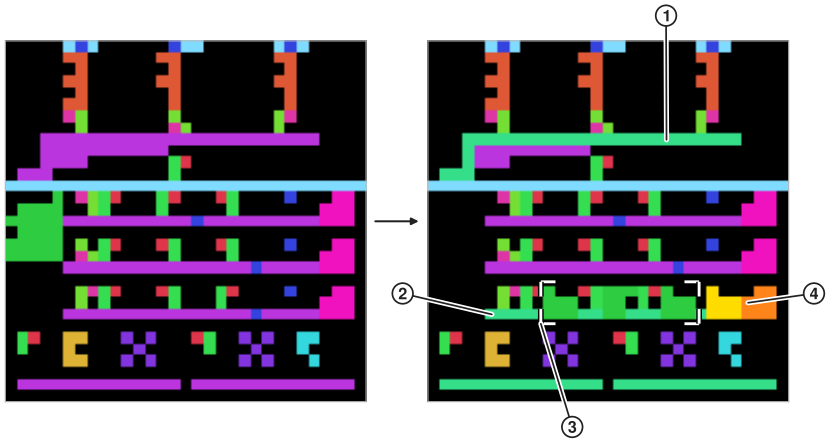
Age 3, a route card. Age 3 separates choosing a route from traveling it. The top half of each scene is a route card. It has three endpoints at the top, each with a short orange edge and a small badge below it, and a purple track runs from each endpoint to the left. The bottom half holds a yellow piece, three covered docks, and a purple corridor to each dock, and each corridor carries a badge. The learner must predict which dock the piece reaches and what happens there.



① The piece turns into its standard pose, and the endpoint whose edge fits the piece, as at age 1, is chosen. Its track on the card turns green. ② The badge of that endpoint names one corridor below, and that corridor turns green. ③ The piece travels along the corridor to its dock. The cover of the dock opens, the piece meets the socket inside, and when they touch cleanly both change color, as at age 2. Part of this answer, the color of the socket, never appears in the input, which is the failure we examine in Appendix C.3.2.

Age 4, stations on the lane. The chosen corridor becomes a lane with stations, and each station turns, flips, or checks the piece on the way.

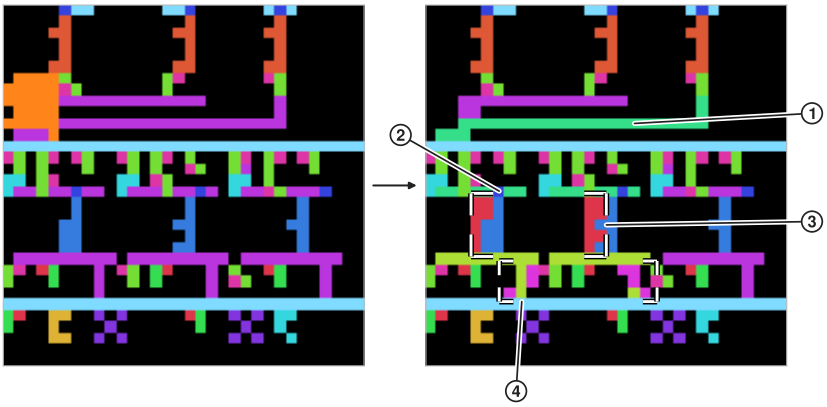
Age 5, programs written once. Age 5 adds programs that are written once and used many times. The bottom of each scene holds two short programs, marked A and B. Each program has two stages, a turn or a flip and a reset to the original copy, in a fixed order. Above them are three rows. Each row starts with a badge, holds three call marks, each an A or a B, and ends with a magenta socket.



① As at age 3, the piece chooses an endpoint on the card, and its track turns green. ② The badge of that endpoint chooses the row with the same badge, and the track of that row turns green. ③ Each call mark in the row runs the program with the same mark on a small copy of the piece, and the three results appear along the row. The programs that were used turn green at the bottom. ④ One marked result moves to the socket at the right end, and when they touch cleanly both change color. A call only names a program, so changing a program at the bottom changes every call to it.

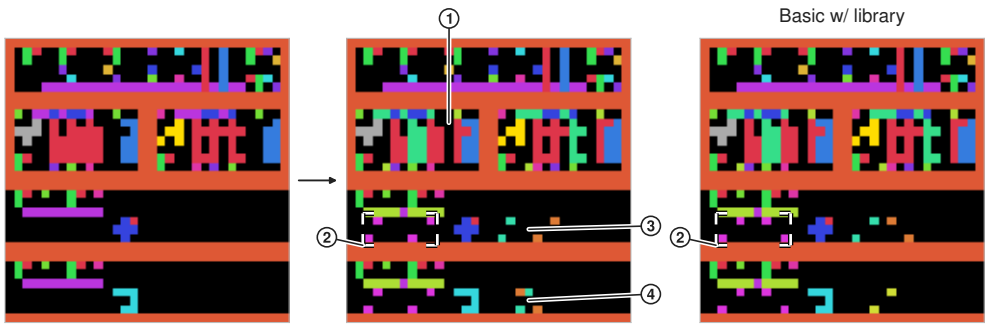
Ages 6 and 7. Age 6 adds a level of calls. A top caller calls two middle callers, each middle caller runs one of the programs and then turns, flips, or repaints its result, and the top caller joins the two results side by side. Age 7 lets one badge start several top callers at once, and they reuse the same middle results in different joins.

Age 8, two rounds. Age 8 connects two rounds of this factory. Each scene has four bands. The top band is the route card of age 3. The second band holds three first-round stations, each of which joins two results as at ages 6 and 7. The third band holds three receivers, each with a blue socket and a badge that names one first-round station. The bottom band holds the programs. The learner must predict which receivers run and what they produce.



① The piece chooses an endpoint on the card, and its track turns green. ② The first-round stations whose badge matches that endpoint run, and their tracks turn green. ③ The joined result of a station is not drawn. Only its edge is, as a red column next to the blue socket of each receiver that names that station. ④ A receiver whose blue socket is the exact complement of the red edge runs its own program on the edge and draws a small result below, and its track turns yellow-green. A receiver whose socket does not fit stays idle. The new idea is that the edge of one result decides whether the next round runs. This is the fitting of age 1, now applied to the output of a whole procedure.

Age 9, the factory as a tool. Age 9 turns the whole factory of age 8 into one tool. The top band draws the procedure of age 8 once, in compact form. The middle band holds two small inputs, A and B. Each is a small scene of age 8, with its own piece, endpoints, and socket. The bottom band holds two bays, and each bay calls the procedure once on A and once on B. The learner must predict what each bay produces. The right panel below shows the answer of `Basic w/ library`, which we discuss after the rule.



① In each input, the procedure runs as at age 8, and the answer marks the chosen tracks, the red edge, and the receiver that fits. ② Each call returns the small result of its receiver, and the returned results appear at the call marks of both bays. ③ The upper bay joins the two returned results side by side. ④ The lower bay turns one returned result and stamps it onto the other. The returned results never appear in the input, and no support returns the same pair as its query. A learner can find them only by running the procedure of age 8 again on A and B.

Every harness drops sharply at age 9 (Table 10), and the reason lies in what this world asks. At age 8, the two-round procedure solves a whole problem, and its result is the answer. At age 9, the same procedure must produce an intermediate result that another rule uses. Each bay needs the outputs of the procedure for A and for B. These outputs do not appear in the input, and no support provides the same pair, so a learner must run the procedure of age 8 again on inputs it has never seen. Every other step of the rule comes from an earlier world, and the answer is fully determined by the input. A learner that kept age 8 as a procedure it can run on new inputs would therefore have one new step to learn, the call. This is also the deepest step of the worldline, and the Director chose its highest-ranked depth candidate at no other age.

The learners in Section 4.2 do not take this step. They often reproduce the part of the answer above the bays but rarely get the returned results right (Table 8). `Basic w/ library` solves most problems of age 8 and reproduces the upper part at age 9 in almost every problem, yet it does not compute the returned results with its procedure of age 8. It records them in a table keyed by the whole layout of the scene (Appendix E.4), and almost no test layout appears in training, so it gets step ② wrong in the panel above. At age 10, the input shows the two results in a reference panel, and `Basic w/ library` recovers the most, because it kept earlier operations as functions that it can apply to them. Age 9 thus asks for the bootstrapping that the worldline is built to measure, and it exposes where accumulated knowledge remains stored as cases instead of becoming a procedure for later problems.

The code below writes the rule of age 9 with the age at which each step first appears.

```
# age 9: each bay calls the procedure of age 8 on A and on B
def bay(scene, combine):                    # join, or turn and stamp
    a = run(scene.procedure, scene.A)       # ① ②
    b = run(scene.procedure, scene.B)       # ① ②
    if a is None or b is None:
        return scene
    return combine(a, b)                    # ③ ④

# the procedure of age 8
def run(proc, x):
    piece = normalize(x.piece)              # age 1
    route = badge_of(end_fitting(piece, x)) # ages 1, 3
    part = join_parts(proc, piece, route)   # ages 5, 6, 7
    if not fits(edge_of(part), x.socket):   # ages 1, 8
        return None
    return run_module(proc, edge_of(part))  # ages 5, 8
```

	Age 8	Age 9	
Harness	Correct	Above the bays correct	Step ② correct
`Basic`	13	7	3
`Basic w/o memory`	6	10	0
`Schema`	4	6	0
`Tycho`	8	11	2
`Basic w/ library`	15	18	0

Table 8: Where the harnesses fail at age 9. Age 8 counts 20 test problems. Age 9 counts the 19 test problems in which the bays call the procedure.

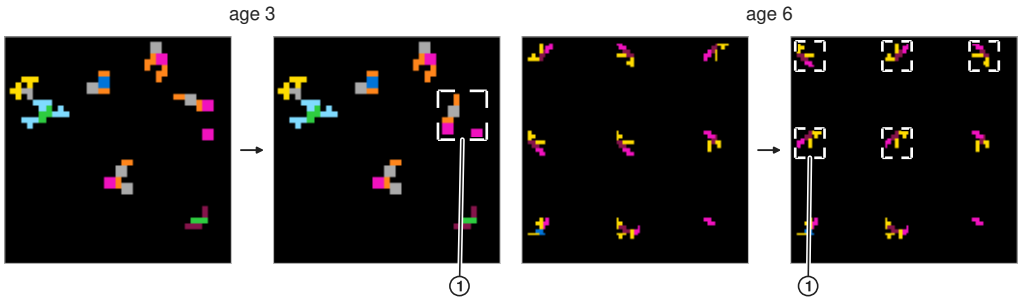
C.2 How the Controls Set the Pace of Bootstrapping

The controls α_d and α_b set how far each new world moves from the worlds before it. At their extremes, a worldline can grow so slowly that little is new at each age, or so steeply that many new relations arrive in one world.

C.2.1 Slow Growth at the Lowest Controls

This worldline of the control sweep (Figure 8) has $\alpha_d = 0.1$ and $\alpha_b = 0.1$, so the Director favors shallow candidates that reuse familiar concepts and introduce few new relations. At age 2, each scene holds several small mosaics, each made of a few colored plates. The training pairs show one signature, a set of colors, a set of plate shapes, and a number of plates. ① Every mosaic whose colors, plate shapes, and plate count all match this signature turns a quarter turn clockwise. Mosaics that match only part of the signature stay put, and so do loose plates that copy it.

The worlds that follow keep this rule and change only the arrangement. At age 3, the mosaic appears in all four poses. At ages 4 and 5, a scene holds none, one, or up to eight such mosaics. Age 6 packs them closely, age 7 changes their neighbors, age 8 swaps their poses, and age 9 sets them in a yard of sockets. In the pairs below, the brackets mark every mosaic that turns.

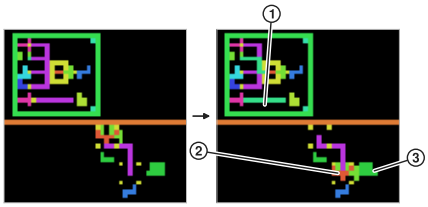


After age 2, the worlds add almost nothing to learn until age 10, which mixes three signatures in one scene. Each world stays close to its parent, as the lowest controls ask, so the worldline confirms one concept again and again instead of building on it. The worldline remains correct and learnable, but after age 2 it asks for little new bootstrapping.

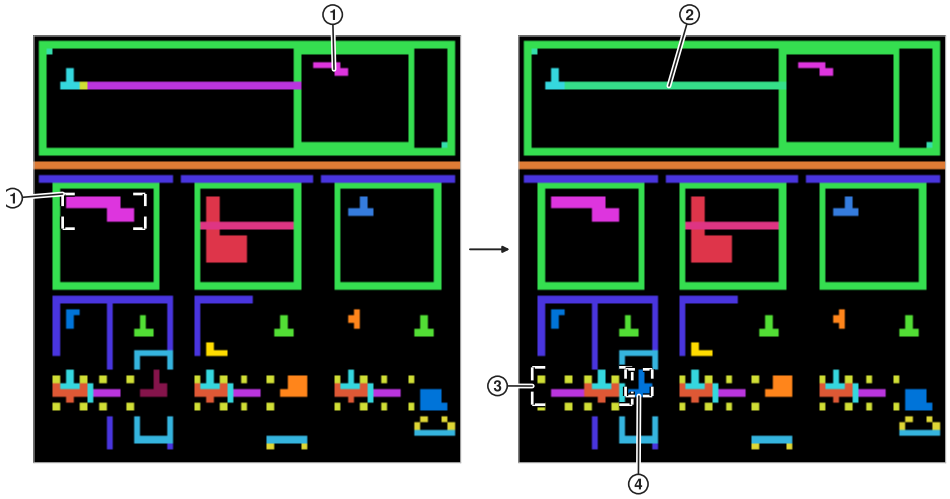
C.2.2 Steep Growth at the Largest Breadth

This worldline has $\alpha_d = 0.2$ and $\alpha_b = 1.0$, the largest breadth. At that breadth, the Director favors candidates that introduce a wider range of concepts and interactions. Age 3 still grows by one step.

Age 3, a board and a rail. The green frame at the top is a board with lines drawn on it. Below the orange bar lies the rail itself, with a cart on it. ① If the line on the board is unbroken and its key matches the key of the cart, the line changes color. ② The cart then runs along the rail to its end. ③ The cart flips the green piece that waits there. Age 2 already had carts, keys, and rails, so age 3 adds one thing, the board that allows the cart to start.



Age 4, a warehouse. Age 4 moves the scene into a warehouse and adds three checks at once. The frame at the top holds the board of age 3 and, on its right, a small magenta shape that serves as an order. Three bays below each hold a shape, and a rail with a cart runs under each bay.



① The order is compared with the shapes of the three bays, allowing a change of size and a turn, and the one bay whose shape it matches is chosen. Here it is the left bay. ② The line on the board changes color when it is unbroken and its key matches, as at age 3. ③ The cart of the chosen bay runs along its rail, but only when the door of the bay is open and the collar of the cart fits the dock at the end of the rail. ④ At the end of the rail, the cart paints the piece that waits there.

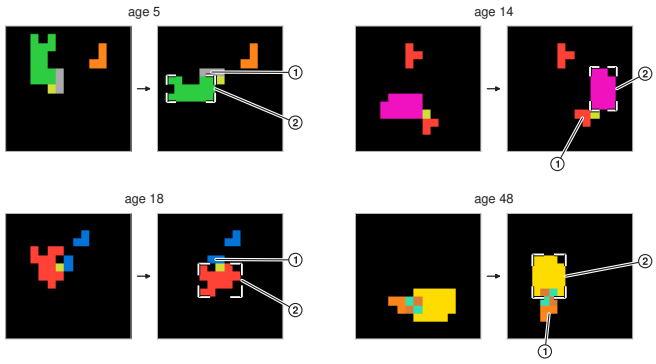
Steps ① and ③ bring three new conditions in one world, the matched order, the open door, and the fitting collar, together with the new scene. For each condition, its training pairs include cases that differ from a successful delivery in that condition alone, and the cart stops at a different place for each, before it starts, at the door, or at the dock. Still, the learner must learn all three together from the pairs of one world, where earlier ages added one relation at a time. This is what the largest breadth asks for, and it makes the step from age 3 to age 4 much steeper than the steps before it.

C.3 Failure Analysis

A world can also break bootstrapping because the Designer makes a mistake. The next world may only teach again what earlier worlds already taught, or part of its answer may depend on a value that the input never shows.

C.3.1 The Same Rule Again and Again

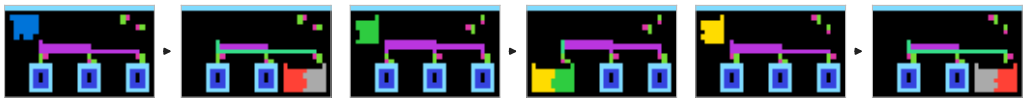
This worldline has $\alpha_d = 0.25$ and $\alpha_b = 0.5$. Its world at age 5 teaches a simple rule. A large piece has a small handle attached to it, and a loose piece lies nearby. ① The handle turns about the cell where it meets the large piece. ② The large piece turns with the handle because it is attached, and the loose piece stays where it is. The same rule returns at ages 7, 9, 14, 16, 18, and 48. The pairs below come from four of these ages. The pieces change shape and color, but the rule does not change, and ① and ② mark the same two steps in every pair.



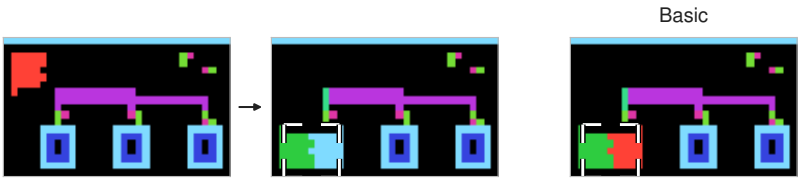
A learner that has solved age 5 has almost nothing new to learn at the six later ages. The code shows the same repetition. After age 5, the handle worlds add no new rule, and two of them only rename the functions written at age 5. The repetition comes from how the next world is planned. Each plan starts from what the training examples of earlier worlds show. A rule that seldom appears in those examples looks unlearned, so the next plan proposes it again.

C.3.2 An Answer That Depends on a Hidden Value

Every harness drops sharply at age 3 of the evaluation worldline (Table 10), and most of this drop comes from a flaw of the world. Age 3 asks the learner to follow the route card to a covered dock (Appendix C.1.4). When the piece reaches its dock, the cover opens and the answer shows the socket inside. The cover is drawn in the same colors in every scene, but the world's code draws the color of the socket at random for each scene. Part of the answer therefore depends on a value that the input never shows. The three support pairs of one test problem below show sockets of different colors behind covers that look the same.



The query of the same problem is below, with its answer and the answer of `Basic`. `Basic` follows the card, chooses the right dock, and places the piece correctly. Only the color of the socket is wrong, and no support could have told it the right one.



In 17 of the 20 test problems, changing only this color leaves the input unchanged and changes part of the answer. The Creator is meant to make everything a new operation changes visible (Section 3.2), and the generated-world review tests for such values by holding the input fixed and varying them (Appendix A.6). At age 3, the review requested other revisions but did not catch this one. The same test finds no such value in any other world, apart from two problems of age 4 that inherit this rule. Many errors at age 3 lie only in the hidden part, and ignoring it removes much of the drop (Table 9). The errors that remain come from choosing the endpoint on the card or finding the corridor by its badge, which is the new relation of this world. We still use this worldline for the evaluation, because it offers many informative cases and clear takeaways, and we confirmed that age 3 does not affect them. Unlike the repeated world, this world asks for something new. It fails because part of the answer depends on a value the learner cannot see.

Harness	Correct	Wrong only in the hidden part	Correct ignoring it
`Basic`	3	7	10
`Basic w/o memory`	4	10	14
`Schema`	2	1	3
`Tycho`	2	7	9
`Basic w/ library`	6	7	13

Table 9: Answers at age 3 out of 20 test problems.

D AGENT EVALUATION SETUP

This section describes the worldline, the protocol, and the harnesses used in Section 4.2.

D.1 EVALUATION WORLDLINE

The agents learn along the worldline shown in Figure 18, generated from the seed world of Appendix B.1 with $\alpha_d = 0.8$ and $\alpha_b = 0.5$. Appendix C.1.4 explains its first nine worlds.

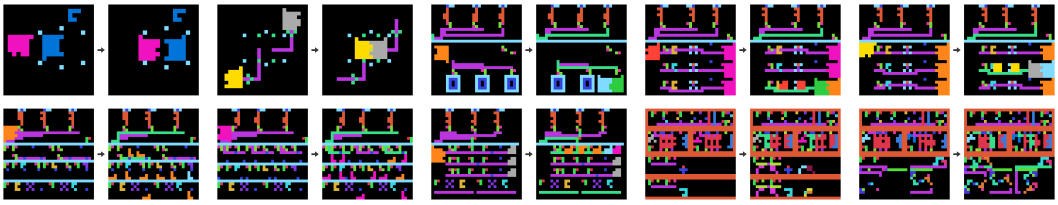


Figure 18: $\alpha_d = 0.8$, $\alpha_b = 0.5$.

D.2 AGENT EVALUATION PROTOCOL

Learning across worlds. Each harness starts a new run at age 0. At age t it receives the world's training set and updates its retained state M_{t-1} to M_t, which may hold code, notes, stored examples, and a continuing conversation, depending on the harness. Model weights are fixed. Learning happens in what the agent writes and revises. `Basic w/o memory` resets this memory before every age.

Solving a test problem. After learning, M_t is saved. Each test problem receives an independent copy of M_t, the current training set, its support pairs S, and the query input $x^\star$. The agent can reason, write and run code, and check candidate rules on the supports before submitting one predicted grid. Changes made while solving stay within that problem, and the next age starts from the saved M_t.

Data and scoring. All harnesses receive the same worlds in the same order, with 10 training pairs and 20 test problems of three supports and one query per world. The test problems are sampled from the world's test generator. Query inputs differ from the training inputs and from their own supports. A prediction is correct when the entire output grid matches the answer, and a missing or malformed prediction counts as incorrect.

Model and budget. All harnesses use GPT-5.6 Sol with high reasoning effort and the same Python and shell tools. We fix the model so that differences between harnesses come from their designs. `Basic` uses one learning session per age, `Schema` may use a second session after replay feedback, `Tycho` a second session for its builder, and `Basic w/ library` a second session for consolidation. Each test problem allows at most two sessions.

D.3 BASIC AND BASIC W/O MEMORY

`Basic` is a coding agent that keeps files across worlds, following arc-code (Berman, 2026). At each age, a fresh conversation receives the current training pairs and the retained files. The agent inspects the observations, writes and runs code, and saves what it wants to keep, such as predictors, helper functions, notes, and selected examples. These files are the learning state for the next age. Each test problem starts from a copy of the files and the supports. The agent infers the rule for that problem, checks candidates against the supports, and submits a grid within at most two sessions (Algorithm 2). `Basic w/o memory` runs the same procedure from an empty workspace at every age, so it learns from the current world alone.

Algorithm 2 `Basic` at one age

Require: Retained files M_{t-1}, training pairs D_t^{train}, test inputs $\{(S_j, x_j^\star)\}_{j=1}^N$
Ensure: Retained files M_t and predictions $\{\widehat{y}_j^\star\}_{j=1}^N$

Start a fresh learning conversation with M_{t-1} and D_t^{train}.
Inspect observations and retained explanations; propose reusable rules.
Write and test code; revise the rules and preserve useful code and notes.
$M_t \leftarrow$ files saved at the end of learning.
for $j = 1, \ldots, N$ **do**
    $\widetilde{M} \leftarrow \text{copy}(M_t)$; $\widehat{y}_j^\star \leftarrow \perp$
    for up to two problem sessions **do**
        Start a fresh conversation with $\widetilde{M}$, D_t^{train}, S_j, $x_j^\star$.
        Infer a shared rule from S_j, using and revising local code as needed.
        Test candidate rules on S_j and predict the output for $x_j^\star$.
        Preserve the session's work in $\widetilde{M}$.
        if an answer is submitted **then**
            $\widehat{y}_j^\star \leftarrow$ submitted output; **break**
        end if
    end for
end for
return $M_t, \{\widehat{y}_j^\star\}_{j=1}^N$

D.4 SCHEMA

In a continuing conversation, `Schema` maintains code that reproduces the observations (Zeng et al., 2026). The code consists of learned transformations F and an explanation procedure E. For an observed pair (x_i, y_i), E selects a program z_i over F that should reproduce y_i from x_i,

$$z_i = E(x_i, y_i; F), \qquad \widehat{y}_i = \mathrm{Execute}(F, z_i, x_i).$$

At each age the agent resumes its conversation with the retained code and the current training pairs and revises the code, and the evaluator then replays it on the training set. If some pair is not reproduced, a second session receives the replay feedback and revises again. Each test problem starts from the saved checkpoint, infers the query rule from the supports, and submits a grid (Algorithm 3).

Algorithm 3 `Schema` at one age

Require: Retained files and conversation (M_{t-1}, C_{t-1}), D_t^{train}, $\{(S_j, x_j^\star)\}_{j=1}^N$
Ensure: (M_t, C_t) and predictions $\{\widehat{y}_j^\star\}_{j=1}^N$
$(M, C) \leftarrow (M_{t-1}, C_{t-1})$; $R \leftarrow \varnothing$
for up to two learning sessions **do**
  Resume C with M, D_t^{train}, and replay feedback R.
  Revise the transformations F, explanation procedure E, and theory notes in M.
  $R \leftarrow \varnothing$
  for each $(x_i, y_i) \in D_t^{\mathrm{train}}$ **do**
    $z_i \leftarrow E(x_i, y_i; F)$; $\widehat{y}_i \leftarrow \mathrm{Execute}(F, z_i, x_i)$
    Add the explanation and its exact-match result to R.
  end for
  if all observations are reproduced **then**
    break
  end if
end for
Save $(M_t, C_t) \leftarrow (M, C)$.
for $j = 1, \ldots, N$ **do**
  $(\widetilde{M}, \widetilde{C}) \leftarrow \mathrm{copy}(M_t, C_t)$
  Continue $\widetilde{C}$ with $\widetilde{M}$, D_t^{train}, S_j, $x_j^\star$.
  Infer and check the query rule using the supports and the learned code.
  $\widehat{y}_j^\star \leftarrow$ output submitted within at most two problem sessions.
end for
return (M_t, C_t), $\{\widehat{y}_j^\star\}_{j=1}^N$

D.5 TYCHO

`Tycho` separates interpreting the evidence from constructing the model (Lehmann et al., 2026). An actor reads the training pairs and the retained files, records the supported relations and alternative explanations, and decides whether to call a builder. The builder turns this analysis into code and notes, checking it against the observations. Both roles share the retained files, and later ages start fresh conversations from them. For a test problem, the actor either answers directly or delegates to the builder, which checks the proposed rule on the supports and produces the query output (Algorithm 4).

Algorithm 4 `Tycho` at one age

Require: Retained files M_{t-1}, training pairs D_t^{train}, test inputs $\{(S_j, x_j^\star)\}_{j=1}^N$
Ensure: Retained files M_t and predictions $\{\widehat{y}_j^\star\}_{j=1}^N$

$M \leftarrow M_{t-1}$
Actor inspects M, D_t^{train} in a fresh conversation.
Record the supported relations and alternatives in a; choose builder decision b.
Save a and the decision's rationale in M.
if $b = \text{true}$ **then**
    Builder reads M, D_t^{train}, a in a fresh conversation.
    Construct and test reusable code; revise the model and notes in M.
end if
Save $M_t \leftarrow M$.
for $j = 1, \ldots, N$ **do**
    $\widetilde{M} \leftarrow \text{copy}(M_t)$
    Actor inspects $\widetilde{M}, D_t^{\text{train}}, S_j, x_j^\star$.
    Record the proposed relation and alternatives in a_j; choose builder decision b_j.
    if $b_j = \text{true}$ **then**
        Builder reads $\widetilde{M}, a_j, S_j, x_j^\star$ and checks the proposed rule.
        $\widehat{y}_j^\star \leftarrow$ builder's query prediction.
    else
        $\widehat{y}_j^\star \leftarrow$ actor's query prediction.
    end if
end for
return $M_t, \{\widehat{y}_j^\star\}_{j=1}^N$

D.6 BASIC W/ LIBRARY

`Basic w/ library` adds to each age of `Basic` a consolidation session that serves as a library learning step (Ellis et al., 2021; Grand et al., 2024). After the learning session, the consolidation session reads the kept code and notes and the training pairs of the current and earlier worlds. It first fills in missing or provisional computations that keep an explanation from producing its output. When an explanation takes an intermediate result as given for each example, the session infers how to compute that result from the input. It then extracts shared computations and recurring compositions into functions, rewrites their uses, and reruns the revised code on earlier observations to repair regressions. At test time, the agent writes one program that composes learned functions and applies the same program to every support and the query. The program is run on each support, and mismatches are returned so that a second session can revise it (Algorithm 5).

Algorithm 5 `Basic w/ library` at one age

Require: Retained files M_{t-1}, training pairs D_t^{train}, test inputs $\{(S_j, x_j^\star)\}_{j=1}^N$
Ensure: Retained files M_t and predictions $\{\widehat{y}_j^\star\}_{j=1}^N$

Start a fresh learning conversation with M_{t-1} and D_t^{train}.
Infer reusable computations; write and test code; save code and notes.
Start a consolidation conversation with the saved files and all training pairs so far.
Complete missing computations; compute per-example intermediate results from the input.
Extract shared computations into functions and rewrite their uses.
Rerun the revised code on current and earlier observations; repair regressions.
$M_t \leftarrow$ files saved at the end of consolidation.
for $j = 1, \ldots, N$ **do**
    $\widetilde{M} \leftarrow \text{copy}(M_t)$
    Write a program $\text{solve}(x)$ composing functions in $\widetilde{M}$ for S_j.
    Run solve on every support in S_j; if some mismatch, revise it in a second session.
    $\widehat{y}_j^\star \leftarrow \text{solve}(x_j^\star)$
end for
return $M_t, \{\widehat{y}_j^\star\}_{j=1}^N$

E Agent Evaluation Results and Analysis

E.1 Results by Age

Table 10 lists the number of correctly answered problems at each age. Across all 220 test problems, `Basic` solves 38 that `Basic w/o memory` misses, and `Basic w/o memory` solves 21 that `Basic` misses. `Basic w/ library` solves the most problems. The drop at age 3 comes from a flaw of that world (Appendix C.3.2), and the drop at age 9 from its depth (Appendix C.1.4).

Harness / Age	0	1	2	3	4	5	6	7	8	9	10
`Basic`	10	12	16	3	11	9	9	9	13	3	0
`Basic w/o memory`	13	13	15	4	7	6	7	3	6	0	4
`Schema`	14	13	4	2	1	1	4	1	4	0	0
`Tycho`	14	14	14	2	7	5	6	3	8	2	2
`Basic w/ library`	13	11	20	6	8	12	15	16	15	1	10

Table 10: Correct answers out of 20 test problems at each age.

Table 11 compares what each harness used over the whole worldline, summing learning and testing. `Basic w/ library` spends more on learning than `Basic` and much less on testing, and uses the fewest tokens and the least time overall.

	Output tokens (M)				
Harness	Learning	Testing	Total	Call time (h)	Correct answers
`Basic`	0.28	3.33	3.61	30.8	95
`Basic w/o memory`	0.33	3.71	4.04	36.6	78
`Schema`	0.12	3.82	3.94	41.5	44
`Tycho`	0.42	4.97	5.39	54.1	77
`Basic w/ library`	0.58	2.06	2.63	16.2	127

Table 11: Usage over the worldline. Output tokens include reasoning tokens, call time sums the duration of all calls, and correct answers are out of 220 test problems.

Table 12 compares `Basic` and `Basic w/o memory` at each age. From age 4 to 8, `Basic` answers more problems while using about a third fewer tokens, because it applies retained operations instead of recovering them from the supports. At ages 9 and 10, where its retained code no longer fits the new worlds (Appendix E.3), `Basic` uses more tokens and time.

Age	0	1	2	3	4	5	6	7	8	9	10
Output tokens (k)											
`Basic`	197	210	130	330	298	256	250	272	245	762	654
`Basic w/o memory`	159	220	185	313	358	431	388	501	403	624	456
Call time (h)											
`Basic`	1.4	1.3	0.9	2.3	2.4	2.2	1.8	1.9	1.7	8.0	7.0
`Basic w/o memory`	1.2	1.5	1.3	2.4	2.9	3.2	2.9	5.2	4.3	6.9	4.8
Correct answers											
`Basic`	10	12	16	3	11	9	9	9	13	3	0
`Basic w/o memory`	13	13	15	4	7	6	7	3	6	0	4

Table 12: `Basic` and `Basic w/o memory` at each age. Output tokens sum learning and testing and include reasoning tokens. Correct answers are out of 20.

E.2 Bootstrapping from Previous Worlds

We examine how earlier code reduces what an agent must learn in a new world, then follow how `Basic` uses that code to solve later problems.

Later worlds use earlier concepts as components of new ones. A task program of a later world applies operations from earlier worlds and uses their results as inputs or conditions of new operations (Section 3.2). To predict an output, an agent needs both the earlier operations and the new one.

Suppose an agent has kept code that performs the earlier operations on any input. In a new world, it can apply this code directly. It then has to learn only the new operation from the training pairs, and at test time it has to fit only that operation to the supports. It also uses fewer tokens, because it does not rediscover the earlier operations for every problem (Table 12).

An agent without such code has to infer the earlier operations and the new one together from ten training pairs and three supports. An earlier operation often chooses among several transformations depending on the input. Three supports seldom show every choice, so an agent that sees only the supports often misses the condition under which each transformation is chosen. The simplest program that matches the supports applies the transformation they show to every input. This program fails when the query needs a different transformation. Because the new operation uses the result of the earlier one, the error spreads to everything built on it, and whole objects in the answer are wrong.

The worldline of Section 4.2 develops one concept from age 2 to age 4. At age 2, a piece must be oriented so that its boundary fits a partner. At age 3, a route is chosen according to which endpoint fits the oriented piece. At age 4, the chosen endpoint decides how the pieces are assembled. `Basic` saved a function at age 2 that chooses the orientation of a piece from its shape, and a function at age 3 that builds its partner. Its age-4 predictor calls both.

```
# Basic, saved at age 2
def outward_orientation(piece):
    # try the 8 rotations and reflections of the piece
    return the orientation with the single-cell edge on top

# Basic, saved at age 3
def companion(piece):
    return the partner that interlocks with its notch rows

# Basic, saved at age 4
def predict_assembly(scene):
    left    = outward_orientation(payload)   # age-2 function
    right   = companion(left)                # age-3 function
    station = the header whose notch pattern matches left
    return scene with both pieces loaded at station
```

The age-2 function uses only the shape of the piece, so it applies to the new pieces of age 4. Every problem that only `Basic` solves at age 4 requires this choice of orientation. In many of them, the saved predictor alone already produces the pieces correctly, and the test session only corrects the route using the supports. `Basic w/o memory` instead takes the orientation shown in the supports and applies it to every input.

```
# Basic w/o memory, test program at age 4
# the same orientation for every input
ORIENTATION = orientation_shown_in(supports)
def solve(scene):
    return assemble(orient(payload(scene), ORIENTATION))
```

This program matches the supports. When the query piece has a different shape, it orients the piece incorrectly, and the piece overlaps its partner and keeps colors that should change.

E.3 Accumulation without Optimization

As the worlds grow, the advantage of `Basic` becomes smaller. We examine what each harness stores and how it uses that knowledge in later worlds.

Learning often leaves code that fits the current observations through stored cases. Stored cases do not provide a rule for transforming an unseen input. At test time, the agent then has to rebuild the operation from three supports for every problem, which costs tokens and often fails. In later worlds, new operations cannot use the results of this code, so the agent writes the code for each world separately instead of building it from earlier code. The saved models therefore reproduce their training pairs but rarely solve new queries on their own (Table 13), and the predictors of later worlds stop calling functions from earlier worlds (Table 14).

An agent can also keep the code that executes a transformation but not the rule that decides which transformation to apply. A single training pair shows how a transformation is executed, but the rule that selects it appears only when several pairs are compared. An agent that records only what the observations have settled therefore keeps the execution and leaves the rule open. Many new concepts of later worlds are decisions based on the results of earlier operations, so the rule left open is exactly what later worlds build on. Such an agent behaves like an agent without memory whenever a later world needs the rule.

Age	0	1	2	3	4	5	6	7	8	9	10
`Basic`'s saved predictor											
Queries it runs on	18	20	20	20	20	8	0	0	0	0	0
Queries exact	1	0	6	0	0	0	0	0	0	0	0
Cell accuracy on those queries	0.95	0.89	0.98	0.92	0.96	0.98	–	–	–	–	–
Cell accuracy of copying the input	0.97	0.87	0.86	0.89	0.88	0.88	0.92	0.91	0.92	0.92	0.91
`Schema`'s saved forward model											
Queries exact	–	–	0	0	0	0	0	0	0	0	0
Queries returned unchanged	–	–	18	19	19	20	19	20	19	20	20

Table 13: Saved models run without the harness. Each saved model is applied directly to the query inputs of its age. `Schema` has no forward model before age 2. The predictor's cell accuracy is averaged over the queries on which it returns a grid.

`Schema` records cases from the start. It revises its code until every training pair is reproduced, and from age 2 its forward model is a table from the objects seen in training to their changes.

```
# Schema, forward model saved at age 2
# one entry per object seen in training
CHANGES = {(color, shape): change, ...}
def advance_world(scene):
    for obj in scene:
        if key(obj) in CHANGES:
            apply(CHANGES[key(obj)], obj)
    return scene        # unseen objects stay unchanged
```

Run alone, this model returns almost every query unchanged, because the objects of the query are not in the table. The test programs call the table, and their wrong answers differ from the target in many cells, while `Basic w/o memory` solves most of the same problems. The accuracy of `Schema` falls at the age this table first appears and stays low.

`Basic` shows the shift from functions to recorded cases as the worlds grow. In the middle worlds, its code orients a piece on any input but records the routes between stations case by case.

```
# Basic, saved at age 4
def orientation(piece):
    return outward_orientation(piece)   # any piece
def route(source, piece):
    # recorded for seen cases only
    return ROUTES[(position(source), bitmap(piece))]
```

Many of its wrong answers in these worlds have the pieces right and only the routes wrong. In the later worlds, each new predictor records what changed in each training pair, keyed by the input configuration and the visible program symbols.

```
# Basic, predictor saved at age 6
CHANGES = {(configuration, symbols): changed_cells, ...}
def predict(scene):
    # fails on an unseen configuration
    return apply(CHANGES[key(scene)], scene)
```

This predictor reproduces every training pair exactly and answers no query on its own.

Age	0	1	2	3	4	5	6	7	8	9	10
Functions	7	11	15	20	27	33	38	41	43	45	47
Earlier ages called	–	–	–	2	2–3	2–4	2–5	6	–	–	–

Table 14: `Basic`'s retained code by age. The last row lists the earlier ages whose functions are reached, directly or through other calls, from the predictor written for that age.

`Tycho` keeps the execution and leaves the rule open. It keeps code that orients a piece for a given orientation, and records the rule that chooses the orientation from the shape of the piece as unresolved.

```
# Tycho, kept at age 2
def apply_orientation(piece, orientation):
    return oriented piece      # execution: kept
# choosing the orientation from the shape:
#   recorded as unresolved
```

When a later world requires this choice, `Tycho` takes the orientation shown in the supports, like `Basic w/o memory`, and its wrong answers differ from the target in as many cells as those of `Basic w/o memory`. It solves about as many problems as `Basic w/o memory` but not the same ones, succeeding where the execution code it kept is enough. Its code grows by functions written for the layout of each world, and these call few functions from earlier worlds.

Ages 9 and 10 combine both effects. A new layout first derives two intermediate values from the input and then places and transforms them with earlier operations. Deriving these intermediate values from unfamiliar layouts is a common source of failure. At age 10, the input itself shows candidate values, and most harnesses find them. At this age, `Basic` does worse than `Basic w/o memory`, because code that no longer matches the current world can hinder learning. `Basic` starts from its saved code, tries to fit the new world to it, and spends the problem undoing it, while `Basic w/o memory` starts from the current observations.

E.4 Optimizing Accumulated Knowledge

We trace how consolidation turns stored cases into reusable operations and how the revised code changes later predictions. The same analysis identifies decisions that remain stored as cases.

`Basic w/ library` adds a library learning step to `Basic` (Ellis et al., 2021; Grand et al., 2024). At each age, its consolidation session rewrites retained programs into shared functions that later programs can reuse (Appendix D.6).

The consolidation session has a different goal from the learning session. The learning session writes code that reproduces the current training pairs. The consolidation session rewrites the kept code so that it performs each operation on any input. It reads the code together with the training pairs of all previous worlds. Cases that were recorded separately then appear side by side, and a function that produces all of them becomes visible. The session replaces the recorded cases with this function, rewrites the code that used them, and reruns the revised code on earlier training pairs so that the earlier functions keep working.

This changes what later worlds find in the saved code. New operations can call the rewritten functions, so the code of a later world is built from the code of earlier worlds instead of written beside it. In later worlds, new functions of `Basic w/ library` call functions saved across many earlier worlds, while the new functions of `Basic` call almost no functions from earlier worlds. Test programs start from functions that already produce the needed intermediate results. Testing therefore costs far fewer tokens, which more than covers the tokens that consolidation adds to learning (Table 11). The same goal also changes how the learning session treats new evidence. When a training pair contradicts an earlier function, the session revises the function instead of recording the pair as an exception.

Some decisions remain stored as cases after consolidation. Test programs can then fit the supports while applying the wrong decision to a new query.

At age 5, an operation turns a larger source shape into two small masks. Before consolidation, the library records the masks for each source shape seen in training. Consolidation replaces the records with a function that folds the edge cells of any source shape into a mask and derives the second mask with an earlier rotation.

```
# before consolidation
def compact_payload_masks(source):
    return MASKS[shape_of(source)]   # seen shapes only

# after consolidation
def compact_payload_masks(source):
    first  = fold(edge_cells(source))   # any shape
    second = rotate(first)              # earlier function
    return first, second
```

This is the only change consolidation makes to this library, and the function keeps its interface. We rerun the test programs that the agent submitted at this age, once with the library before consolidation and once with the library after consolidation. The same programs solve 5 of 20 problems with the earlier library and 12 with the later one, and no problem is lost. The earlier library raises errors on source shapes absent from its records.

The learning session also revises an earlier function when new evidence contradicts it. At age 4, the library links each routing key to a transformation mode. At age 5, a training pair shows a known key with a different mode, and the session changes the function to read the mode from the lane's symbols.

```
# age 4
mode = MODE_OF_KEY[routing_key(scene)]
# age 5, after a known key appears with a different mode
mode = read_mode(symbols_in(selected_lane(scene)))
# the key still chooses the destination
destination = destination_of(routing_key(scene))
```

Age 9 shows where consolidation stops. Its new layout derives two intermediate masks from the input before earlier operations place and transform them. The library records the masks keyed by the full configuration, and for an unseen configuration it falls back to the nearest recorded one.

```
# Basic w/ library, age 9
MASKS = {(shape, wiring, markers, instruction):
            (mask_a, mask_b), ...}
def intermediate_masks(scene):
    if key(scene) in MASKS:
        return MASKS[key(scene)]
    # an unseen layout gets the nearest recorded masks
    return MASKS[nearest_recorded_key(scene)]
```

For queries using the new layout, the test programs match their supports but derive the wrong masks. In a failed problem, replacing only the two masks with the correct ones, and leaving the rest of the program unchanged, makes the whole output correct. The rest of the program, built from earlier functions, is therefore right. At age 10, the input itself shows candidate masks, and `Basic w/ library` solves half of the problems by applying its learned functions to them.

F Evaluation on Another Worldline

We repeat the evaluation of Section 4.2 on a worldline generated with $\alpha_d = 0.9$ and $\alpha_b = 0.9$, with the same protocol and harnesses (Figure 19). Tables 15 and 16 report the results.

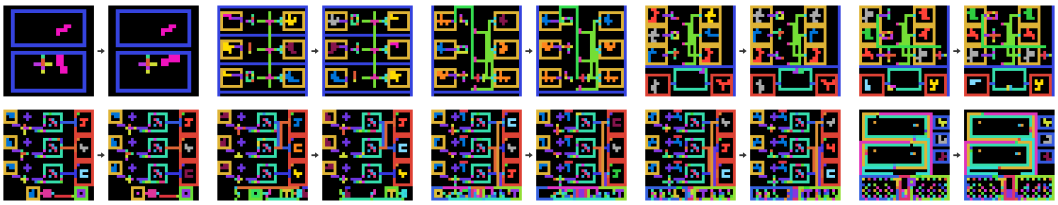


Figure 19: $\alpha_d = 0.9$, $\alpha_b = 0.9$.

Harness / Age	0	1	2	3	4	5	6	7	8	9	10	Total
`Basic`	14	18	19	16	7	3	6	4	2	2	0	91
`Basic w/o memory`	14	18	16	9	9	5	5	5	4	2	3	90
`Schema`	15	20	20	11	9	5	6	5	2	0	2	95
`Tycho`	17	19	20	18	8	7	2	4	4	0	2	101
`Basic w/ library`	13	19	17	13	10	9	3	5	3	0	5	97

Table 15: Correct answers out of 20 test problems at each age on another worldline.

Harness	Output tokens (M)	Call time (h)	Correct answers
`Basic`	3.50	20.7	91
`Basic w/o memory`	3.93	22.7	90
`Schema`	2.90	18.1	95
`Tycho`	4.94	29.5	101
`Basic w/ library`	3.38	20.7	97

Table 16: Usage over another worldline. Output tokens include reasoning tokens, call time sums the duration of all calls, and correct answers are out of 220 test problems.

Bootstrapping from previous worlds. In the early worlds, `Basic` solves clearly more problems than `Basic w/o memory`. In this worldline, symbols first select transformations, and later worlds turn them into routes that decide which objects move and exchange places. `Basic` applies the transformation and exchange code it saved in earlier worlds to the new routes, and only has to learn the routes. Over the whole worldline, the two harnesses solve about the same number of problems, but `Basic` uses fewer tokens, because its test sessions start from saved code (Appendix E.2).

Accumulation without optimization. From the middle worlds, the advantage of `Basic` disappears. In the later worlds, combinations of signals and control states decide which process runs. `Basic` records its predictions keyed by the full control configuration, so a new configuration matches no record. `Schema` and `Tycho` start stronger on this worldline than on the evaluation worldline, because in the early worlds they kept rules that apply to new inputs, such as code that reads the shape of a symbol to choose a transformation. In the later worlds, they also record control configurations case by case, and their accuracy falls in the same way. On both worldlines, a harness does well when what it keeps applies to new inputs (Appendix E.3).

Optimizing accumulated knowledge. `Basic w/ library` solves more problems than `Basic` with fewer tokens. As on the evaluation worldline, consolidation rewrites recorded cases into rules. At age 5, a signal update recorded for each observed position becomes a rule that moves the signal along the boundary of any frame.

```
# before consolidation
def advance_signal(scene):
    return UPDATE_AT[position(scene)]   # seen only

# after consolidation
def advance_signal(scene):
    head, tail = signal_ends(scene)
    return move_along(boundary(frame(scene)),
                      head, tail, direction(scene))
```

Before consolidation, this predictor reproduces none of the 60 test supports of that age, and after consolidation it reproduces 22. The remaining failures again come from decisions. Test programs match all their supports but choose the wrong route or target for the query. In one problem at age 3, the program infers from the supports a route map that misses a route the query uses.

```
# Basic w/ library, test program at age 3
# read off the supports;
# the query also needs 1 -> 0
ROUTES = {2: 2}
def solve(scene):
    return exchange(scene, ROUTES)   # earlier function
```

Replacing only this route map with the correct one, and leaving the rest of the program unchanged, makes the whole output correct (Appendix E.4).

G Extended Related Work

Conceptual bootstrapping and reusable programs. In cognitive science, conceptual bootstrapping describes how a learner uses concepts it already knows to learn new ones (Piantadosi et al., 2012; Zhao et al., 2024). Accounts of abstraction describe how learned procedures become reusable units for later reasoning (Sfard, 1991; Dehaene et al., 2022). Zhao et al. (2024) also show that the order in which people meet examples shapes the concepts they can learn next. Program-learning methods bring this idea to machines. DreamCoder and LILO compress solved tasks into a library of functions that can be composed to solve new tasks (Ellis et al., 2021; Grand et al., 2024). Learning to Theorize learns such functions from observations alone (Baek et al., 2026). Given pairs of observations before and after a change, it discovers a set of operations and explains each change as a program composed of them. Each world of ARC-∞ poses the same problem, since a learner sees only training observations and must infer the operations behind them. ARC-∞ provides tasks that reward reuse as a growing sequence, and the library learning step we add to `Basic` (Section 4.2.2) follows this line of work.

Intelligence evaluation. Intelligence evaluation asks how well a system can learn and apply skills beyond the tasks used to develop it (Hernández-Orallo & Dowe, 2010; Hernández-Orallo, 2017; Chollet, 2019; Hernández-Espinosa et al., 2026). Chollet (2019) measures intelligence as the efficiency of acquiring new skills, and ARC turns this view into a benchmark of few-shot rule induction (Chollet, 2019; Chollet et al., 2025). Procedural generators such as COGITAO scale this format by sampling input–output pairs from explicit transformation rules (Moffitt, 2025; Taoudi-Benchekroun et al., 2026). In these benchmarks, each task is an independent puzzle. The novel-games perspective proposes evaluating learners in games with unfamiliar objects, mechanics, and rules (Ying et al., 2025), and ARC-AGI-3 carries learning across the levels of a game (ARC Prize Foundation, 2026). Its games are still designed by hand, and recent systems already report near-perfect performance on its public games (Fox et al., 2026). Continual-learning benchmarks present tasks in sequence and measure transfer, forgetting, and the benefit of accumulated experience (Wołczyk et al., 2021; Zheng et al., 2025; Asawa et al., 2026). Their tasks are fixed in advance and are not built from one another. ARC-∞ keeps the ARC format for each problem and grows the worlds so that each builds its concepts from earlier ones.

Open-ended environment and benchmark design. Open-ended learning systems invent new problems as their learners improve. PowerPlay searches for a new task together with a solver that solves it (Schmidhuber, 2013), POET evolves environments together with the agents that solve them (Wang et al., 2019; 2020), and regret-based environment design builds curricula at the frontier of an agent's ability (Dennis et al., 2020; Parker-Holder et al., 2022). Foundation models now generate executable environments, games, and benchmarks (Faldor et al., 2025; Zhang et al., 2026; Verma et al., 2025; Ying et al., 2026), and LLM agents plan, generate, and refine benchmarks once people specify the ability to test (Butt et al., 2025; Dsouza et al., 2025). These systems generate tasks either to train an agent or to add independent tests. The Designer of ARC-∞ decides for itself how each world develops the concepts of earlier ones, so that the worlds form a single growing test. Once a worldline is generated, it stays fixed, so different learners can be compared on the same worlds.