RESEARCH NOTE · 31 JUL 2026 · UPDATED 3 AUG 2026

Data-Driven RSI: Evaluation, Ladders, and Trajectories

RSI describes a recursive process of improvement: the current model helps create a stronger successor, and the successor carries the process into the next round. The article starts with the part that seems easiest to test—data.

Eval | Synthetic data | Harness-Evolve

What RSI is meant to achieve

RSI stands for Recursive Self-Improvement. A current model helps create a stronger successor. That successor inherits the ability to find problems, propose improvements, and train the next generation. Each round improves the model and the method used to produce it. Continued repetition gives the process its recursive character.

Foundation-model teams currently maintain this loop. Researchers at organizations such as OpenAI, Moonshot AI—the company behind Kimi—and Zhipu AI—the developer of GLM—locate capability boundaries, build evaluations, produce data, run training, and choose the next direction. Models complete individual tasks; people improve the process that produces the models.

RSI asks the model to do some of that second job too. Without a researcher tending every step, an LLM would find a capability gap, propose a testable change, prepare the next round of training material, and check whether the new model actually improved. The new model would then do the same again.

For this loop to run, the model has to find a useful problem and turn it into a training intervention. After training, it needs independent evidence of what changed and a decision about the next round. More samples are not enough. Scores on a fixed benchmark will eventually stop giving direction. The improvement process itself has to get better over time.

Why start with data

Data is an easy place to begin. It is hard to predict where the next algorithmic change will come from, or whether a small result will survive at scale. A data experiment is more concrete: give a model unfamiliar tasks, find failures that repeat, turn them into new tasks and trajectories, and train the next model on them.

Each round of failures should change the next batch of data. The model then starts to decide what it should learn next. Full RSI remains far away, but this small part can already be tested.

That leaves three questions:

- How can an open-ended goal such as “be better at mathematics” or “be better at coding” be evaluated well enough to decide what to train next?
- When a batch of synthetic data looks good, how can its absorption by the target model be measured, and how much should be produced?

- How can synthesis techniques accumulated across domains become a repeatable production line for useful trajectories?

The rest of the article stays with these three questions.

A few terms used below

The article looks at one segment of the longer loop. The current model helps find capability gaps and make training data for the next round. Its successor then has to improve reliably on fresh tasks outside production. Parameter rewriting during a single inference run is out of scope.

An **open-ended evaluation** can begin with a broad goal such as mathematical research ability. Every round still leaves concrete records: tasks, trajectories, failure causes, verifier results, and a plan for the next batch of data.

I use **Harness–Evolve** as a temporary name for the data process. The harness is the environment in which the model works. Evolve covers repeated attempts, revisions, and trajectory selection inside that environment. The selected trajectories go into SFT, where the target model's weights are updated.

Why this leads to synthetic data

Take the data-driven view one step further and synthetic data appears immediately.

Many SFT examples, preference pairs, critiques and rewrites, verifier-filtered rollouts, and tool-use trajectories are constructed through some combination of models, rules, tools, and people. Recent post-training gains often rely on this deliberately constructed experience.

Pretraining is moving in a similar direction. Natural data remains the foundation. As cleaning, deduplication, and filtering improve, more marginal data comes from rewriting, transformation, difficulty control, reasoning completion, and execution-based filtering. Existing material is also rebuilt into tasks that are easier to learn from. The line between pretraining and post-training data will keep fading.

Natural data remains the foundation. It gives a model knowledge, language, and contact with the world. It rarely grows around the failure modes of a particular checkpoint. When a model gets stuck on one step of one kind of problem, the web rarely supplies ten thousand exercises at exactly the right difficulty with reliable feedback attached.

The data wall does not mean that the internet suddenly runs out of new text. The shortage is narrower: material for the next model has to be useful, unfamiliar, and good enough to train on. Estimates of public text stocks and training demand suggest that high-quality human text may constrain further scaling (Villalobos et al., 2024). Nobody knows the date of impact. We already know that more tokens do not reliably buy more capability.

Will DePue calls the internet a "one-time subsidy" to deep learning (DePue, 2026). For decades, people wrote webpages, code, papers, and discussions with no model-training plan; together they happened to form a huge, inexpensive, cross-domain dataset. The next increment is unlikely to arrive through the same accident. Much of the valuable material outside the public web lives in internal workflows, tacit expert judgment, unrecorded failures, and processes visible only inside real environments.

Two shortages are mixed together. **Volume** asks how many good examples are still missing in known domains. **Coverage** asks which tasks, tools, edge cases, and long-running processes were never recorded at all. Synthetic data is good at volume. Coverage still needs an anchor from people or a real environment before a model can elaborate and recombine it.

> *The scarce resource is experience that is useful for the model's next step.*

If scaling is to continue, models will have to help make the next round of experience: turn failures into tasks, keep tool and verifier feedback in the trajectory, and train on the useful parts. This is where synthetic data meets RSI.

This can go wrong in simple ways. A model may generate millions of answers from abilities it already has. If one model produces, grades, and filters them, its old preferences grow stronger with every round. The evaluation, ladder, and trajectory pipeline below are attempts to deal with that problem.

1. Evaluation has to keep moving

Benchmarks have a shelf life. Models approach the ceiling, teams adapt to the format, and some examples may find their way into training. The test then loses resolution at the frontier. Different benchmarks age at different rates, but a study of 60 widely used text benchmarks found that nearly half were already highly saturated (Akhtar et al., 2026). Contamination adds another reason to doubt the score (Singh et al., 2024).

The usual response is to ask people for a harder test. That is getting expensive. Humanity's Last Exam selected 2,500 questions from more than 70,000 submissions, with nearly a thousand experts and several rounds of review (Center for AI Safety & Scale AI, 2025). If model cycles keep getting shorter, people will eventually fall behind on writing questions, maintaining answers, and refreshing validation sets.

Evaluation has to move with the model. A goal may begin broadly—better mathematical research, stronger coding, or longer-running work. After each round, the model uses the failure map to search for a new boundary, write tasks and counterexamples, adjust difficulty, and add to the benchmark and validation set. AutoBencher already lets models search for evaluation data that is salient, novel, and difficult (Li et al., 2025).

A model can help write the test, but it cannot be the only judge of improvement. An active benchmark used for diagnosis may change constantly. Private validation used to confirm progress stays outside the training pipeline and hidden from the evaluated checkpoint. Independent generators, environment evidence, and human audits still check the questions, answers, and possible leakage.

> *A model can write its next test. Independent evidence confirms the improvement.*

Self-evaluation keeps the capability boundary moving. Self-validation lets the model help design a test and leaves the final decision to evidence it could not prepare for.

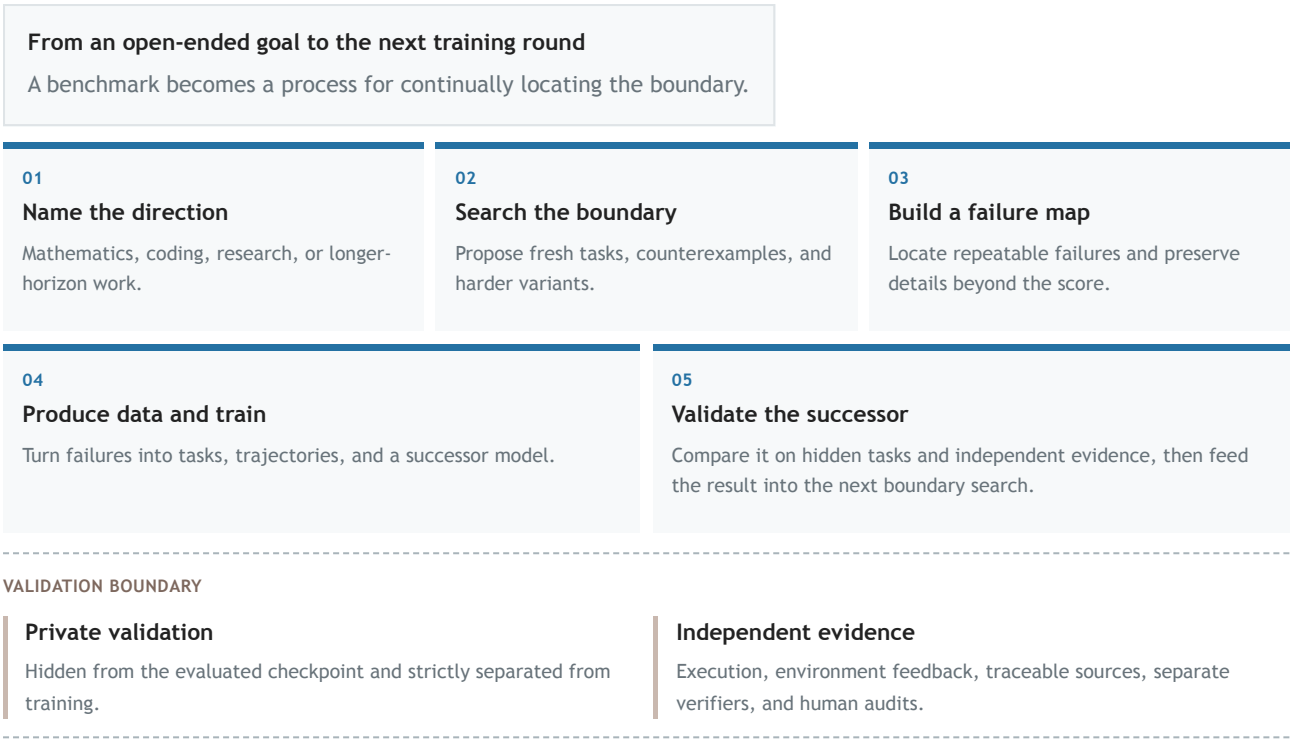

Figure 1. Steps 01–05 form one iteration. Validation feeds the next boundary search, while the validation boundary prevents the system from writing the test, training on it, and declaring victory.

The most useful output of an evaluation round is a **failure record**. Keep the score, but also record the task slice, terminal error, first point where the trajectory went wrong, likely cause, and proposed data fix. Two runs may both end in a timeout for completely different reasons: bad planning, a broken tool call, or failed context management.

Five layers of evidence matter:

1. **Outcome:** Was the task completed under an executable or explicit verifier?
2. **Process:** Where did the first causal deviation occur?
3. **Slice:** Does the mechanism repeat across a fresh task family? One bad case provides weak evidence.
4. **Prescription:** Does the failure imply a concrete task, feedback, or harness intervention?
5. **Transfer:** After training, does the gain appear on private validation that did not participate in data production?

2. Ladders: use small experiments to decide whether to climb

Imagine that a coding harness can produce one million trajectories in a week. The decision to run at full scale needs evidence. Synthetic data can look polished and be fully correct while doing almost nothing when trained into the target model.

A ladder starts with a cheap experiment and raises the cost one rung at a time. Each rung changes one main variable and holds the others steady. The next rung opens only after a repeatable gain over control. This gives a team a chance to see the return and saturation before full-scale spending.

Architecture ladders are often used to see whether a new design survives scale. A candidate and matched baseline are trained at several compute budgets. Data, objective, evaluation, and the rule for allocating parameters and tokens are fixed in advance. The question is simple: **does the candidate keep its advantage as compute grows?**

A data ladder scales the dataset. Keep the target model, training setup, and evaluation fixed; add more volume or coverage from one source; then measure the held-out gain from each batch. The curve shows where marginal return starts to disappear.

Synthetic data adds another complication: the dataset is made on demand. The producer, harness, verifier, sampling policy, and filter determine what the data looks like. The target model determines what can be absorbed. A comparable ladder has to keep the **production recipe and target model fixed** and vary only the number of accepted, deduplicated trajectories. Change the pipeline or target model and the old curve becomes a reference, not a guarantee; the cheaper rungs must be run again.

Scaling production can also change the data. Acceptance may fall, duplicates rise, and easy cases crowd out the tail. The same trajectories may teach planning to one model and only phrasing to another. SynthLLM also found that the saturation region changes with target-model scale (Qin et al., 2025). “How much can we generate?” and “how much should this model consume?” are separate questions.

All three ladders make small experiments precede large ones, but their experimental units and failure conditions differ:

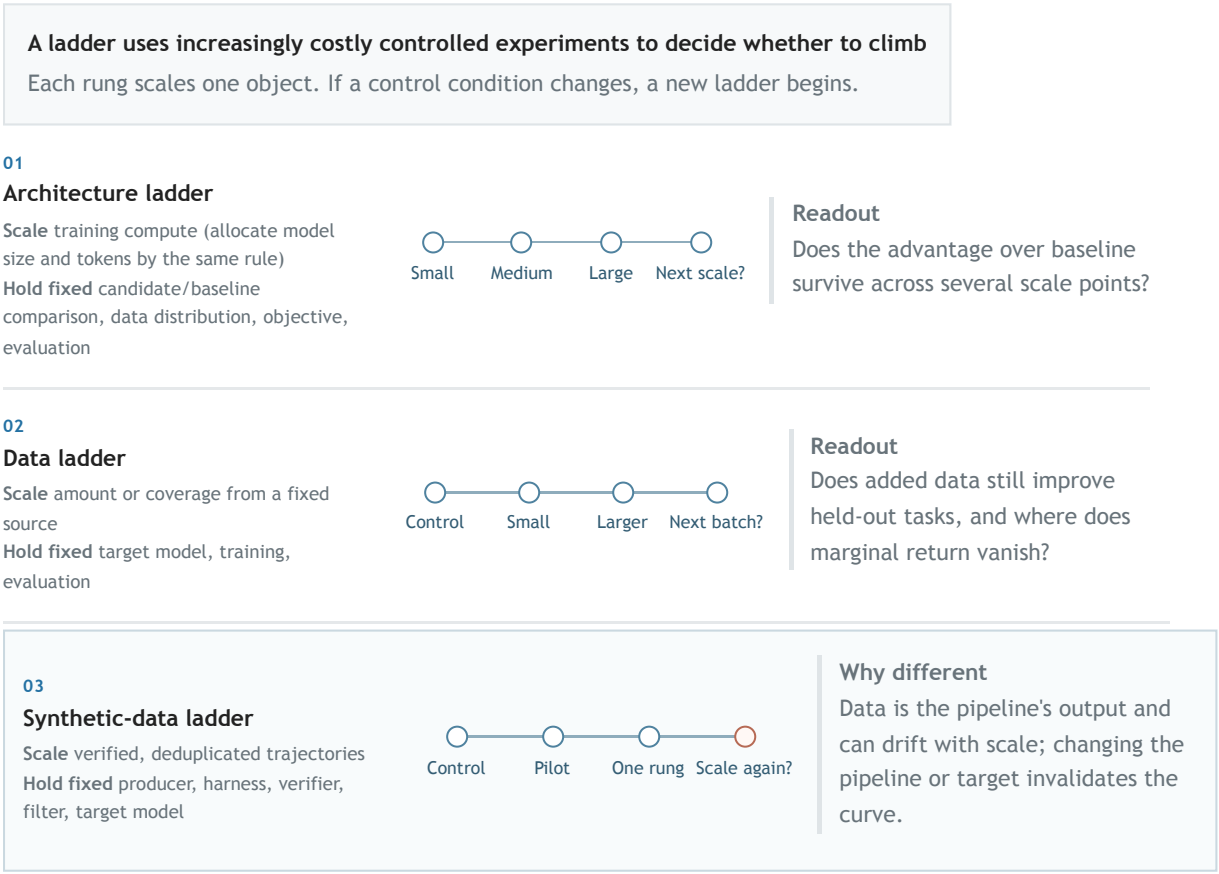

Figure 2. Each synthetic-data ladder belongs to one production recipe and one target model.

In practice, the ladder is a series of increasingly expensive bets. Audit a few tasks and verifiers. Run a pilot from the same base checkpoint. Add another rung when the held-out signal repeats. Repeat the experiment when the signal approaches the noise or cost threshold. If it falls below, stop and change the pipeline.

A minimum viable experiment

A minimum experiment fixes the base checkpoint, optimizer, number of epochs, and held-out evaluation. Only the number of accepted, deduplicated trajectories changes, so total training tokens grow with data volume. If the training-token budget is fixed, the experiment measures coverage or mixture under equal compute. That is a different question from pure data scaling and should be reported separately. Run each rung at least twice to estimate training noise.

Keep three ledgers for every point. Production records cost, acceptance, and deduplication. Training records effective tokens, stability, and behavior changes. Evaluation records target-slice gains, transfer, and regressions. Move to a more expensive rung only when adjacent rungs point in the same direction across repeats.

Until the result repeats across scales, target models, and pipelines, it is a **ladder experiment**, not a scaling law that can be extrapolated. Pacing investment and finding saturation are already useful outcomes.

At every rung, ask again: how much new capability did the next batch buy?

Each project sets its threshold from uncertainty across repeats, production cost, and regression risk.

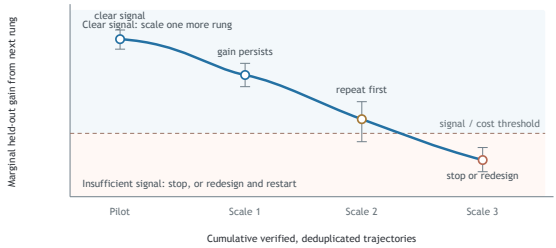

Figure 3. A schematic decision rule. Train every point from the same base checkpoint and repeat it on held-out tasks outside data production; a new production recipe or target model starts a new curve.

What to measure first

I would start with three numbers: the verification pass rate, the count after deduplication, and the gain on an independent evaluation from each new batch of distinct trajectories. The first two describe cost. The last describes return. Total generation volume says little on its own.

When a small run shows a signal, add accepted, non-duplicate trajectories in stages. Start from the same checkpoint, keep training and evaluation stable, and retain the previous rung as a control. If data volume multiplies while unfamiliar tasks stop improving, or the acceptance rate keeps falling, the pipeline is near saturation.

Stop before saturation. Without a ladder, teams often finish an expensive dataset and only then start explaining why it failed.

The ladder belongs to its pipeline and target model

The same synthetic dataset can behave very differently on two models. Every example carries habits from its generator and harness: how problems are decomposed, which tools are preferred, when the model backtracks, and which errors it tends to make. There is no guarantee that another model can absorb those habits.

Change the target model, producer, harness, verifier, sampling policy, or filter, and the ladder should return to its cheap rungs. The old curve offers experience, not a guarantee. This is what makes synthetic-data scaling laws so hard to establish.

3. Trajectories need a production line

People have always designed data. They choose where to look, what to keep, how to label it, and what should be learned first. Even internet data reaches a model through a human-built pipeline.

As public webpages stop supplying most of the marginal data, production will shift from collecting documents to recording work. An expert workflow may use private tools, repeated communication, local judgment, recovery from failure, and final review. Save only the input and answer, and much of the trainable process disappears. New data infrastructure has to record the environment, actions, feedback, and outcome together.

Many synthetic-data pipelines are still prompt + model + filter. They make questions and answers quickly, but record little process. How the model searched, tried, saw an error, revised the plan, and noticed a wrong turn may be more useful than the final answer.

From a prompt to a production line

Code needs tests. Mathematics uses proof checks, counterexamples, and gradual increases in difficulty. Agent tasks cannot lose environment state or tool output. These habits now live across prompts, scripts, and researchers' heads. A harness puts them into the environment the model meets every time it works.

Here, a harness includes prompts, tools, context management, workflow, persistent state, permissions, and verification logic. It controls what the model sees, what it can do, how it saves work, and how it checks itself. It also determines the trajectories the pipeline can collect (Weng, 2026).

Traceable
The task, generator, harness, tools, and verifier all have explicit versions.

Process is visible
Keep the necessary states, actions, tool feedback, failures, and revisions.

Verifiable
Prefer execution or independent judgment, and retain a reason when a trajectory is rejected.

Diverse
Control the mixture of harnesses and generators; remove duplicate paths and repeated phrasing.

Trainable and testable
Send every batch back through the ladder and an independent evaluation.

Harness–Evolve first, then SFT

Harness–Evolve is straightforward. Let the model make several attempts in an environment with tools and feedback. It may follow different paths, fail, backtrack, and revise. Afterwards, keep the complete processes that are correct, representative, and genuinely different.

Those trajectories go into SFT. The hope is that next time the model reaches reliable behavior more easily: it checks when needed, revises after failure, and uses tools when the task calls for them.

STaR provides a direct precedent: generate rationales, keep paths that recover the correct answer, and train them back into the model (Zelikman et al., 2022). Harness–Evolve expands the candidate object from a textual rationale to a stateful agent trajectory. Tool responses, file changes, execution errors, backtracking, and replanning become part of the data. DGM and Self-Harness optimize the agent or harness itself. Here the output is training data, and the target model changes in the final SFT stage (Zhang et al., 2025; Zhang et al., 2026).

Figure 4. Four stages of one data pipeline. Evolve improves candidate trajectories; the final SFT stage writes the selected processes back into the target model.

Why use more than one harness

One harness tends to grow one style of trajectory. A test-driven coding harness produces many run–error–revision paths. A critique-and-rewrite harness produces more self-checking. A search-oriented harness produces branches and comparisons. Each can be useful, and each can harden into a template when overused.

Run several harnesses and use the ladder to see which source the target model absorbs and which mixture works best. This costs more experiments, but makes it harder for one pattern to fill the training set.

A harness has a horizon too

Long-running tasks force the harness to change. Much of today's agent code assumes that a task ends in one run or context, tools return synchronously, verifier feedback comes quickly, and one terminal state marks success. That may work for a coding task measured in minutes. It breaks down when experiments run for days, data programs cross training rounds, or research feedback arrives late.

METR describes task-completion horizon using the time a human expert needs for the same task. This measures task difficulty, not the agent's literal wall-clock runtime. It captures a familiar problem: knowing many local skills does not mean a model can join them into a reliable long sequence (METR, 2026). As tasks get longer, lost state, growing context, accumulated errors, and failed recovery become bottlenecks.

An RSI-oriented harness still lacks some basic infrastructure:

- **Persistent state:** experiments, code, data versions, and unfinished work need storage beyond context; interrupted runs need recoverable checkpoints.
- **Hierarchical goals:** long tasks need verifiable stages that preserve dependencies across stages and resist the pull of the nearest local score.
- **Asynchronous execution:** training, evaluation, and data generation may run for hours or days; the harness must launch, monitor, cancel, and resume background work.
- **Delayed feedback:** when final reward arrives late, intermediate evidence must be retained and failure attributed to the step that caused it.
- **Evolvability with boundaries:** the model may revise workflows, context policy, and tool composition, while verifiers, permissions, and audit logs remain outside the editable loop.

Recent work on long-horizon agents emphasizes compact state, checkpoints, verifier-backed state transitions, and targeted recovery, reducing the amount of raw interaction history repeatedly stuffed into the prompt (Wu et al., 2026). Harness–Evolve faces the same constraint. As trajectories grow longer, the harness carrying them has to improve too. Otherwise the pipeline will only produce data inside its current horizon.

Synthesis still needs priors

Goal priors	Know roughly what “better at mathematics” contains while keeping the target broader than one frozen test.
Task priors	Know which tasks are worth producing, how difficulty should increase, and which failures are informative.
Verification priors	Execute what can be executed and check what can be checked; avoid asking the generator to grade itself.
Exploration priors	Allow trial, error, and backtracking while cutting off long trajectories that are going nowhere.
Diversity priors	Control task types, tools, solution paths, and generator sources; keep any one pattern from filling the dataset.

These priors come from the practical experience of making data: where to explore, which feedback to trust, and which results deserve suspicion. The model has to learn some of that judgment too. On a new topic or open-ended goal, it should borrow from old experience, design new tasks and environments, and find the next useful data.

Generators copy themselves. If one model acts, judges, and filters, Evolve may simply amplify its old patterns. Execution tools, independent verifiers, different generator families, real-data anchors, and small human audits can reduce the echo.

Selection creates shortcuts too. A verifier that checks only the answer may keep lucky short paths. A blanket preference for long traces rewards empty elaboration. Selection should look at outcome, process completeness, novelty, and cost together. Keep rejected traces and rejection reasons as well; they are useful clues for the next harness and task design.

Relationship to adjacent work

The nearest lines of work optimize different things:

- **Reasoning self-training** optimizes reasoning paths to be internalized by the model. STaR is the clearest example.
- **Synthetic-data scaling** measures the gain and saturation of a generation method as data and model scale change. SynthLLM belongs here.
- **Dynamic evaluation** continually searches for and refreshes benchmark items. AutoBencher is one example.
- **Harness optimization** changes prompts, tools, context, or workflow so that a frozen model performs better at deployment. Self-Harness closes a loop among failure mining, editing, and regression testing.
- **Open-ended agent evolution** retains a population of candidate agents and continues branching under executable feedback. DGM and AlphaEvolve show how archives and evaluators support this search (Novikov et al., 2025).

- **Harness–Evolve** connects these lines: the harness expands trajectory search, verifiers and diversity rules select the data, and SFT distills it into a target model. The idea still needs empirical tests.

Three tests matter: whether multiple harnesses make data that transfers better; whether a ladder predicts the return from large-scale production; and how much reusable process SFT learns compared with the surface habits it copies from the generator and verifier.

Putting the loop together

Put the pieces together and the workflow is ordinary: evaluation finds a problem, Harness–Evolve makes trajectories, SFT trains them back, and the ladder decides whether to scale the data method.

One model's failures determine the next model's data. As the model improves, old tasks lose value and old harnesses saturate. The tasks, environment, and data all have to move.

01
Name the direction
The goal can begin broad and still be clear enough to explore.

02
Build fresh tasks
Keep changing the test until it reaches failures that matter for the current model.

03
Design the harness
Put human domain knowledge into tools, feedback, verification, and exploration rules.

04
Produce trajectories
Let models try, fail, and revise; retain several kinds of complete process.

05
Filter, then SFT
Train only trajectories that are reliable and meaningfully different.

06
Run the ladder
Measure absorption, transfer, and saturation before deciding to produce more.

The final test is whether the next model becomes reliably stronger. Everything else is supporting evidence.

People still make most of the hard calls: what goal to pursue, how to write the harness, which evidence to trust, and when to stop. Making one segment measurable and repeatable is enough for a start.

Open questions

- How can an open-ended goal be covered well enough? How do we keep it from quietly turning into another fixed benchmark?
- How can model-generated benchmarks and validation sets remain novel without mistaking the generator's preferences for the capability boundary?
- How should trajectories from different harnesses be deduplicated and mixed without cancelling one another out?
- When should the data keep scaling, and when should the generator or harness be replaced instead?

- Can the model gradually participate in harness design without causing evaluation and production to collapse into the same loop?
- Does SFT learn a better process, or merely a subtler version of the generator's style?

These questions have no settled answers. The direct next step is to pick a narrow domain, run the whole loop a few times, and see what breaks first.

References

1. Villalobos et al. (2024), *Will we run out of data? Limits of LLM scaling based on human-generated data*.
2. Singh et al. (2024), *Evaluation data contamination in LLMs: how do we measure it and (when) does it matter?*.
3. Zelikman et al. (2022), *STaR: Bootstrapping Reasoning With Reasoning*.
4. Qin et al. (2025), *Scaling Laws of Synthetic Data for Language Models*.
5. Zhang et al. (2025), *Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents*.
6. Novikov et al. (2025), *AlphaEvolve: A coding agent for scientific and algorithmic discovery*.
7. Zhang et al. (2026), *Self-Harness: Harnesses That Improve Themselves*.
8. Weng (2026), *Harness Engineering for Self-Improvement*.
9. DePue (2026), *A Stargate for Data*.
10. METR (2026), *Task-Completion Time Horizons of Frontier AI Models*.
11. Wu et al. (2026), *StructAgent: Harness Long-horizon Digital Agents with Unified Causal Structure*.
12. Akhtar et al. (2026), *When AI Benchmarks Plateau*.
13. Center for AI Safety & Scale AI (2025), *Humanity's Last Exam*.
14. Li et al. (2025), *AutoBencher: Creating Salient, Novel, Difficult Datasets for Language Models*.