Improve the Process That Proposes Improvements

A controlled study of inherited mutation policies in self-improving research agents

Abstract

An agent can accumulate better task routines while its ability to discover the next improvement stays fixed. We propose a lineage experiment that allows research agents to modify a bounded policy for proposing and testing their own harness edits. At each generation we will transplant the evolved mutation policy into a common fresh executor to measure its ability to discover improvements without inherited task routines. The goal is to determine whether the improvement process itself becomes more sample-efficient under a fixed resource budget.

Keywords / recursive self-improvement, research agents, lineage analysis, automated experimentation

Problem and prior work

STOP makes an improver modify its own code [1]. Recent research-agent RSI work reports gains in research efficiency on held-out tasks [2], and the Darwin Godel Machine shows the value of open-ended archives [3]. We focus on causal attribution. Improvement in the incumbent can arise from a better executor, a better archive, or a better policy for generating informative edits. A transplant experiment can separate these sources more directly than a rising lineage score.

Hypothesis and method

Represent each agent as an executor harness and a mutation policy. The mutation policy allocates proposals across failure categories, chooses parent configurations, selects a diagnostic experiment, and sets a local search budget. Use an allowlisted configuration language with bounded operations. Each generation uses its own mutation policy to propose executor edits and mutations to the policy itself. To score a mutation-policy candidate, launch it from a common parent executor on several short development searches and evaluate the resulting executor on separate validation tasks. Carry forward both selected components. Use fixed model weights and independent evaluation.

Transplant protocol

At predefined generations copy the incumbent mutation policy into a fresh executor with empty memory and the same initial diagnostics. Pair it with the seed mutation policy across common task packs and seeds. Cross evolved and seed executors with evolved and seed mutation policies to quantify interactions. A separate frozen-policy lineage may evolve only its executor. Match the full search cost, including the nested policy evaluations, across arms.

Validation plan

Experimental design

Use small, runnable ML-engineering tasks with deterministic success tests and resource measurements, such as correcting data preprocessing, fitting a compact model, or reducing batch-processing time within an accuracy floor. Partition by failure mechanism and task family. Start with six lineages and six generations, with a capped inner search of eight candidate executor edits per policy evaluation. Include random policy mutations and a fixed adaptive scheduler as controls. Save evaluator feedback and count both model tokens and tool wall time. Reserve novel task families for transplantation only.

Outcomes and rejection criteria

Primary endpoint is the number of externally verified improvements per fixed budget on the common fresh executor. Secondary endpoints are time to first accepted edit, held-out performance of descendants, and gains after charging all meta-search costs. Bootstrap by task family and lineage. A stronger incumbent paired with a flat transplant score supports executor adaptation alone. If policy gains disappear under full cost accounting or only appear on familiar task families, reject the claim of efficient transferable improvement.

Feasibility and deliverables

Weeks 1-2 define the configuration grammar and task splits. Weeks 3-4 run a pilot to calibrate inner-search noise. Weeks 5-7 execute matched lineages and transplants. If nested variance prevents reliable selection, freeze executor checkpoints for longer and report policy effects at fewer generations. Release configurations, budgets, lineage snapshots, and replayable evaluation traces.

References

[1] Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation. arXiv:2310.02304.

[2] Recursive self-improvement of AI research agents. arXiv:2609.26457.

[3] Darwin Godel Machine: Open-Ended Evolution of Self-Improving Agents. arXiv:2505.22954.