# IMPROVING LLM-DRIVEN PROGRAM EVOLUTION WITH DYNAMIC SCOPE

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

LLM-driven program evolution typically searches over candidate programs while treating the extent that each mutation may change as a fixed generation interface. This creates a structural tension: narrow editing boundaries preserve locality and attribution but can make dependency-spanning improvements unreachable, whereas broad boundaries admit coordinated changes at the cost of a larger proposal space and weaker attribution. Because the appropriate side of this trade-off can change over a lineage, a fixed boundary can become either a reachability ceiling or an unnecessarily broad search interface. We introduce **DyScope**, which turns the permitted editing boundary into explicit, executable, and verifiable search state. A structural scope graph defines available regions, and each boundary action jointly specifies a target, dependency width, mutation regime, and transition guard. Trajectory evidence can retain productive local scope, widen to dependencies, redirect the target, contract after a broad transition, or invoke recovery. Before generation, the selected boundary changes the context exposed to the proposer and the set of reachable transformations. After generation, an executable transition contract compares the requested boundary with the realized parent–child diff, protects invariant regions, and records whether the transition is admissible. DyScope therefore adapts and verifies program transitions rather than only proposing programs. Feedback supplies evidence for updating boundary state; specific thresholds and rankings instantiate the selection policy rather than define the abstraction. DyScope attains 96.67%, 99.39%, and 98.83% on AIME 2026, HumanEval, and MBPP, respectively. In the matched boundary-policy comparison, dynamic scope reaches 90.18% versus 82.15% for fixed scope. These results identify the editing boundary as an adaptive dimension of program search rather than a static implementation choice. By making changes to an evolving system selectable, enforceable, and traceable, DyScope provides a concrete control interface toward auditable recursive self-improvement.

## 1 INTRODUCTION

Large language models (LLMs) increasingly serve as mutation operators over executable programs. By repeatedly generating, evaluating, and selecting code, LLM-driven evolution has improved algorithms, heuristics, and agent scaffolds (Romera-Paredes et al., 2023; Liu et al., 2024; Ye et al., 2024; Ma et al., 2024; Novikov et al., 2025; Sharma, 2025; Lange et al., 2025; Zhang et al., 2025). Most systems optimize *which candidate* to propose or retain under an editing interface chosen in advance (Liu et al., 2026; van Stein & Bäck, 2024; Assumpção et al., 2025; Yan et al., 2026). This leaves a consequential decision implicit: *what is the next program allowed to change?* Narrow boundaries preserve locality and attribution but can make dependency-spanning improvements unreachable; broad boundaries admit coordinated changes but enlarge the proposal space and weaken attribution. The missing capability is *dynamic scope*: changing the permitted edit region as the needs of the search change.

This tension changes over a lineage: local refinement may suffice for one transition, while a missing dependency may require a wider or redirected edit later. A static policy must choose one side for the run. Moreover, the region requested from a generator need not match the code it actually changes, so the boundary must govern both generation and admission.

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.

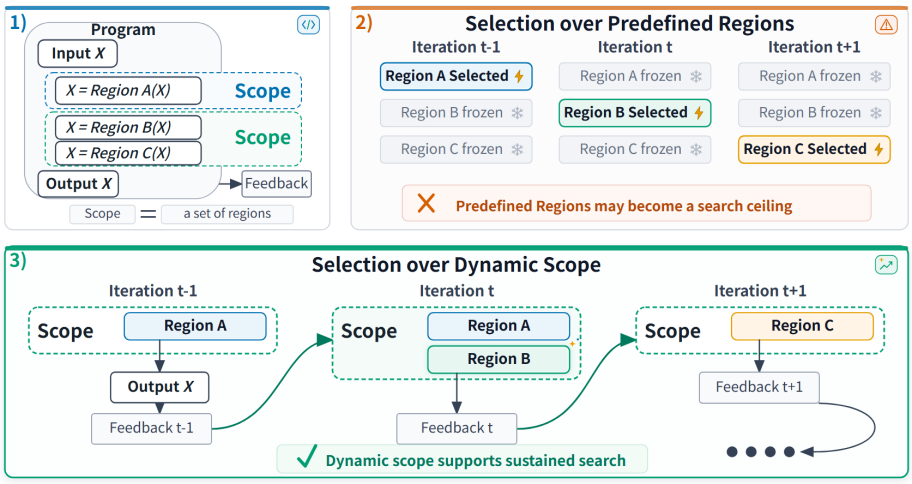


Figure 1: **Dynamic scope as search state.** A fixed policy operates within predefined regions (top right), whereas DyScope can retain, widen, or redirect the boundary as trajectory evidence changes (bottom).

We introduce **DyScope**, whose central move is to make scope a transition-level search decision rather than a run-level constant. At each iteration, DyScope can retain, widen, redirect, or contract the active region according to trajectory state. The selected boundary jointly specifies a target, dependency width, mutation regime, and guard, thereby changing the context and transformations available to the proposer. A transition contract then verifies the realized diff before admission (Figure 1). Verification supports dynamic scope by ensuring that the chosen boundary is executed; it is not a separate validity claim. DyScope therefore adapts how programs may change, not only which program is proposed next.

DyScope attains 96.67% accuracy on AIME 2026 and solution rates of 99.39% on HumanEval and 98.83% on MBPP. We use ScopeBench as a controlled diagnostic suite, rather than a universal software benchmark, to test boundary-sensitive transitions. In its matched comparison, both arms share the model, information boundary, initialization, evaluator, proposal budget, generation and admission pipeline, and transition contract. Fixed scope locks the initial legal editing region; DyScope can change the target, width, and guard with trajectory state. DyScope reaches 90.18% versus 82.15% for fixed scope, isolating the effect of dynamic boundary control under this protocol.

Our contributions are:

- We formulate *dynamic scope* as an explicit, transition-level state variable in LLM-driven program evolution. Factoring each boundary action into target region, dependency width, mutation regime, and guard lets the search move between local attribution and dependency-spanning reachability over a lineage.
- We operationalize dynamic scope with a structural scope graph and an executable transition contract. The selected boundary conditions proposer context and reachable transformations before generation; realized-diff verification then enforces that boundary and records transition-level lineage before evaluation and archive admission.
- We establish end-to-end performance on standard reasoning and code tasks and evaluate boundary policy under a strictly matched ScopeBench protocol. Dynamic scope reaches 90.18% versus 82.15% for fixed scope; component ablations and external-proposer integration further characterize the complete transition-control loop and its portability.

## 2 Related Work

**LLM-driven program and algorithm evolution.** LLM-based evolutionary methods couple program generation with executable feedback. Representative systems evolve executable algorithms,

heuristics, or programs (Romera-Paredes et al., 2023; Liu et al., 2024; Ye et al., 2024; van Stein & Bäck, 2024; Liu et al., 2025; Ma et al., 2024; Novikov et al., 2025; Sharma, 2025; Lange et al., 2025; Assumpção et al., 2025), while newer agentic formulations extend the object of evolution to self-modifying coding systems (Yin et al., 2024; Zhang et al., 2025; Wang et al., 2025; Gao et al., 2025). These methods primarily adapt candidate content, proposal policies, or population dynamics. DyScope addresses a different control variable: it treats the reachable editing boundary as trajectory state and updates it after each verified transition.

**Structured and localized editing.** SPARK is the closest prior work on where an LLM edits: its where-then-how procedure selects a functional factor and freezes the others during a patch (Liu et al., 2026). Program slicing and dependence analysis provide related structural notions of locality (Weiser, 1984; Horwitz et al., 1988). However, these approaches supply a factorization or patch region whose granularity is fixed for the relevant decision. DyScope instead uses progress, dependency, and boundary-violation evidence to retain, expand, redirect, or recover the editing boundary, and verifies the realized diff after each transition.

**Adaptive search control.** Adaptive parameter control and operator selection make search actions state-dependent (Eiben et al., 1999; Da Costa et al., 2008; Fialho et al., 2010; Maturana et al., 2012; Guo et al., 2025; Boldi et al., 2023; Jamieson & Talwalkar, 2016; Li et al., 2018); recent LLM-driven systems similarly adapt allocation, resampling, or model routing (Yang et al., 2026; Jiang et al., 2026; Tanveer, 2026). They nevertheless leave the admissible form of an individual transition fixed. DyScope applies feedback to that boundary itself: it stays local when refinement is productive, widens for dependency-spanning changes, and contracts after the transition. The resulting distinction is between adapting search within a fixed transition system and adapting the transition system as evidence changes.

## 3 METHOD

**Problem Formulation and Overview.** DyScope augments a standard program-evolution loop with an explicit decision over the support of each mutation. The semantic objective remains unchanged: the evaluator assigns utility to executable programs. The methodological change is that generation and admission are conditioned on a boundary action whose consequences are observed after the candidate is produced. One iteration has four stages: construct the current boundary state, select an admissible operator and boundary, generate a conditioned proposal, and verify the realized transition before updating the population and controller state.

Let $p_t \in \mathcal{P}$ be a parent program and let $q : \mathcal{P} \to \mathbb{R}$ be an evaluator. The evolution system selects $p_t$ from its population or archive and generates a candidate child $\widetilde{p}_t$. Let $\mathcal{S}_t$ be the named executable regions of $p_t$ and let $G_t = (\mathcal{S}_t, E_t)$ be a structural or dependency graph over those regions. The graph can be induced from annotations, syntax, symbol references, or a combination of these sources (Weiser, 1984; Horwitz et al., 1988; Bairi et al., 2024; Cheng et al., 2024). The graph is an operational decomposition for generation and verification, not an assertion that the named regions form a unique causal decomposition of program behavior.

The controller observes a state

$$x_t = (p_t, G_t, h_t, e_t), \tag{1}$$

where $h_t$ is transition history and $e_t$ contains evaluator, validity, novelty, stagnation, and failure evidence. It selects a boundary action

$$a_t = (o_t, \tau_t, w_t, g_t) \in \mathcal{O} \times \mathcal{S}_t \times \mathcal{W} \times \mathcal{G}, \tag{2}$$

where $o_t$ is a mutation operator, $\tau_t$ is a target region, $w_t$ is the editable width (for example, a single scope, a dependency cone, or a cross-scope composition), and $g_t$ is the transition guard. The generated proposal is sampled conditionally:

$$\widetilde{p}_t \sim M(\cdot \mid p_t, a_t, c_t), \tag{3}$$

where $c_t$ is the boundary-relevant context supplied to the proposal model. Thus, the boundary changes the proposal distribution before generation and the admissible transition set after generation; it is not a label attached to an already evaluated child.

The width component has executable semantics. Let $\mathcal{U}_t$ denote residual regions not named in $\mathcal{S}_t$, and let $\mathcal{N}_t(\tau_t)$ be the dependency neighborhood of the target in $G_t$. The controller compiles $w_t$ into an allowed region set

$$W_t = \begin{cases} \{\tau_t\}, & \text{single-region contract,} \\ \{\tau_t\} \cup \mathcal{N}_t(\tau_t), & \text{one-hop dependency contract,} \\ \mathcal{S}_t \cup \mathcal{U}_t, & \text{cross-scope or escape candidate.} \end{cases} \tag{4}$$

The guard $g_t$ maps a contract outcome to an admission decision, such as acceptance, scoped retry, projection, repair, or rejection. Prompt context therefore encourages a particular edit, whereas $W_t$ and $g_t$ determine whether the realized transition may enter semantic evaluation. This distinction is essential because an LLM can modify code outside the region named in its instruction.

**Stage-Gated Operator Ordering.** At each iteration, a state-dependent gate returns an admissible set

$$\Gamma(x_t) \subseteq \mathcal{O}. \tag{5}$$

The gate excludes actions that are incompatible with the current phase, remaining budget, or transition evidence. Its inputs include recent progress, stagnation, repeated violations, dependency coupling, and cycles in the active lineage. This prevents expensive or disruptive operators from competing with local refinement at every iteration (Eiben et al., 1999; Da Costa et al., 2008; Fialho et al., 2010; Maturana et al., 2012; Whitacre et al., 2009). For each $o \in \Gamma(x_t)$, the order model combines a history-slice rank estimate, delayed credit from descendants, an archive-coverage bonus, and penalties for risk and recorded cost:

$$V_t(o) = \widehat{F}_t(o) + \lambda_{\mathrm{d}} D_t(o) + \lambda_{\mathrm{c}} C_t(o) - \rho_t(o) - \kappa_t(o). \tag{6}$$

Here $\widehat{F}_t$ is a rank/CDF estimate from prior observations in the current state slice, $D_t$ is delayed lineage credit, $C_t$ rewards coverage of under-explored operators, and $\rho_t, \kappa_t$ encode observed risk and configured cost. An exploration floor keeps admissible operators available. The controller selects the highest-ranked admissible operator after combining these terms. Equation 6 is a within-state ordering rule rather than a universal utility shared across tasks or operators.

The final controller exposes a common vocabulary of DyScope, dependency cone, strict refinement, structured patching, particle-style routing, repair, and escape operators, but the gate does not open all of them simultaneously. Which operators are admissible is itself state dependent; boundary search then chooses the executable extent for the selected operator. Separating admission from ranking keeps a high historical reward from reactivating an operator that is unsafe or inappropriate in the current state.

The resulting routing profile is part of the method rather than an after-the- fact outcome summary. Figure 2 shows how local refinement remains available while structured, dependency-aware, and dynamic routes complement it when the current boundary is insufficient. Marker area denotes mean routing probability; the display summarizes controller behavior within the realized search traces.

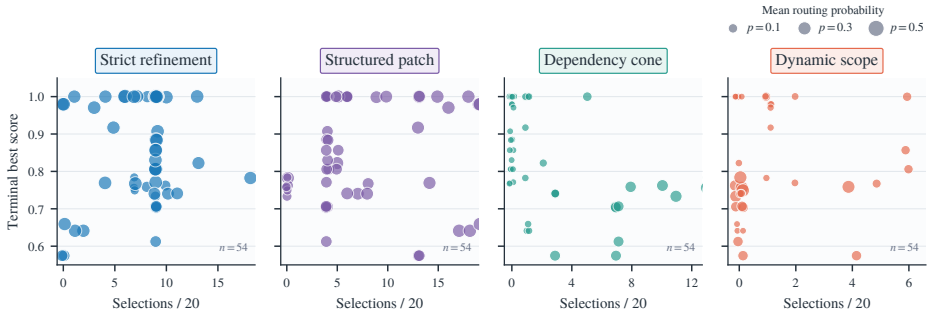


Figure 2: Operator routing under DyScope. Selection frequency and terminal quality show that broader scopes complement, rather than replace, local refinement; marker area denotes mean routing probability.

**Controller Regimes.** For execution and trace analysis, the controller's fine-grained actions are grouped into three mutually exclusive regimes; these are reporting categories within one DyScope controller, not three independently run models. *Boundary expansion* includes the initial `scope-lite` and dependency-aware `scope-cone` proposals, which enlarge the editable region when local progress is insufficient or a dependency-spanning change is indicated. *Strict refinement* keeps the active scope essentially fixed and performs constrained local improvement after a productive region has been identified. *Recovery* handles failed, invalid, collapsed, or stalled transitions by backtracking, repair, projection, or a guarded retry; it preserves the search state rather than serving as the primary source of quality gain.

Figure 3 shows how these regimes are allocated over the proposal horizon and how their realized best-score gains accumulate. In the complete 54-cell DyScope traces, the observed proposal shares are 31.5% (boundary expansion), 46.7% (strict refinement), and 21.9% (recovery), while their corresponding shares of the observed positive gain are 48.4%, 48.7%, and 2.9%. These are descriptive trace-level attributions, not causal estimates from regime-specific interventions.

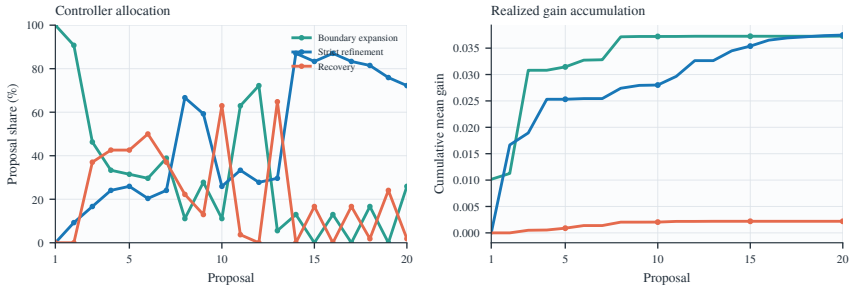


Figure 3: **Controller-regime allocation and realized gain.** Boundary expansion combines initial scope-lite and dependency-aware scope-cone proposals. The left panel shows the proposal share assigned to each regime at each step; the right panel shows the corresponding cumulative mean best-score gain over the 20-proposal horizon. The three curves are mutually exclusive trace labels within the complete DyScope controller, not separate models.

**Boundary Search.** Given the selected operator and the same state, boundary search ranks candidate targets and boundary profiles:

$$(\tau_t, w_t, g_t, c_t) \in \arg \max_{(\tau,w,g,c)\in\mathcal{B}(x_t)} J_t(\tau, w, g, c), \tag{7}$$

where $\mathcal{B}(x_t)$ contains only boundary hypotheses compatible with the scope graph and current gate. The score $J_t$ combines region-specific outcome history, dependency coupling, expected interface risk, novelty, and the option value of widening or leaving a repeated local cycle. When exploration is active, the controller materializes a small portfolio of boundary-conditioned hypotheses and prunes context to the selected boundary. Joint parent–boundary sampling can choose a parent whose lineage and target scope are compatible with the same hypothesis. These operations change what the model is asked to edit; they do not alter the evaluator's definition.

Target uncertainty and recent transition reward favor a single-region or one-hop profile, whereas sustained stagnation or dependency evidence can admit a cross-region candidate. The transition contract remains active after the profile is selected, so widening the search does not silently weaken interface or feasibility requirements. The resulting action space supports four qualitatively distinct responses—preserve, widen, switch, and recover—while holding the semantic objective fixed.

The boundary update is phase-aware but not phase-only. During stabilization the controller keeps the target local; in exploration, persistent local stagnation can open a one-hop dependency cone; repeated scope or interface violations strengthen the guard; and a plateaued incumbent can trigger a budget-gated escape or recovery proposal. Phase fractions, retry limits, and the escape budget are configured before search; observed transition evidence determines which eligible response is selected within those constraints.

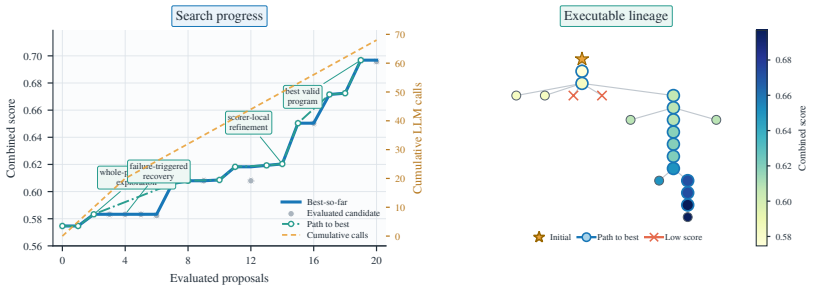


Figure 4: Representative search trace and executable lineage. Boundary decisions are linked to realized parent–child transitions and the subsequent lineage used by DyScope.

**Transition Verification and Recovery.** After generation, the controller extracts the changed regions $\widehat{\mathcal{C}}_t$ from the parent–child diff and records the attribution source. When authoritative attribution is available, a transition is admissible only when the realized change satisfies the requested contract, for example

$$\widehat{\mathcal{C}}_t \subseteq W_t, \qquad \mathrm{Frozen}(p_t, p_{t+1}) = \mathrm{true}, \tag{8}$$

where $W_t$ is the editable region set induced by $w_t$. The frozen-region predicate tests the invariants encoded by $g_t$. Feasibility checks additionally cover syntax, public interfaces, duplicate definitions, and undefined names before expensive semantic evaluation (Le Goues et al., 2012; Long & Rinard, 2016; Martinez & Monperrus, 2019; Madaan et al., 2023; Wu et al., 2024; Dai et al., 2025). If exact attribution is unavailable, the system records the fallback source explicitly rather than treating a requested scope as an observed one.

The guard first admits a valid transition or requests a scoped retry, projection, repair, or rejection. Recovery is invoked only when the recorded state indicates repeated failure, stagnation, or collapse. It may consult scope-specific change memory, backtrack to an earlier lineage state, or repair the proposal within the active contract. Successful and rejected transitions both update $h_{t+1}$ with the intended boundary, realized change, verifier outcome, and score response. A repaired child is therefore attributed to the repair transition that produced it rather than to the original invalid proposal.

## 4 Experiments and Results

**Evaluation protocol.** Our experiments establish DyScope's performance across established program-solving benchmarks and isolate the contribution of DyScope under a controlled protocol. DyScope performance across benchmarks provides the primary evidence that the controller transfers across reasoning and code-generation settings; ScopeBench then holds the surrounding search conditions fixed to identify why dynamic scope improves search. In all proposal-based experiments, the proposer receives the public task instruction, current incumbent, proposal budget, and declared black-box feedback only. Evaluator implementations, reference solutions, target values, exact optima, and evaluator artifacts remain outside the proposal context. Candidates execute in a cleared, network-isolated worker, while the host-side evaluator returns only the permitted scalar feedback.

The controlled study is a matched comparison on ScopeBench. DyScope and a fixed-scope control start from the same release programs and run for 20 proposal steps on 18 tasks with three seeds (45–47), yielding 54 paired task–seed cells. Both arms use DeepSeek-v4-Flash at temperature 0.7, disabled reasoning, and no completion-token cap. They differ only in boundary policy: DyScope selects an editable region from the current trajectory state and can admit dependency-connected scopes or controlled escape; fixed scope retains the initial legal region throughout search. The model, evaluator, initialization, and proposal budget are otherwise identical.

For each cell, the primary endpoint is the best combined score reached within the 20-step budget. We report its mean and sample standard deviation and estimate the policy effect from within-cell differences. For uncertainty, we average seed-level differences within each task and apply a percentile bootstrap over the 18 tasks; win, tie, and loss counts use a tolerance of $10^{-6}$.

Results on each benchmark use its prescribed metric and data split. Together with CLRS-10, these evaluations establish DyScope's performance across the target settings. ScopeBench provides the matched analysis of the boundary policy underlying that performance. Table 1 summarizes the program-solving benchmark and CLRS-10 results; Table 2 reports the ScopeBench analysis.

Table 1: **DyScope performance across benchmarks.** All methods share one header; each row uses its stated native metric. Scores are percentages on a 0–100 scale. For CLRS-10, the OpenEvolve, FunSearch, and SPARK entries reproduce the values reported in SPARK Table 1; CodeEvolve and ShinkaEvolve are our official-implementation runs under the same protocol as DyScope.

| | | **Ours** | **Reference methods** | | | | |
|---|---|---|---|---|---|---|---|
| **Benchmark / task** | **Metric** | **DyScope** | **OpenEvolve** | **CodeEvolve** | **ShinkaEvolve** | **FunSearch** | **SPARK** |
| *Mathematical reasoning* | | | | | | | |
| AIME 2026 | Accuracy | **96.67%** | 83.33% | 80.00% | 66.67% | 46.67% | 63.33% |
| AIME 2025 | Accuracy | **98.89%** | 70.00% | 66.67% | 56.67% | 33.33% | 63.33% |
| AIME 2024 | Accuracy | **97.78%** | 80.00% | 80.00% | 32.2% | 36.67% | 66.67% |
| *Code generation* | | | | | | | |
| HumanEval | Solved | **99.39%** | 98.78% | **99.39%** | 97.56% | 96.95% | 79.88% |
| MBPP | Solved | **98.83%** | 96.11% | 76.26% | 97.28% | 94.55% | 96.89% |
| *CLRS-10 OOD accuracy* | | | | | | | |
| DFS | OOD accuracy | **98.14%** | 20.17% | 18.54% | 16.13% | 17.25% | 83.74% |
| AP | OOD accuracy | 78.82% | 61.87% | 6.41% | 6.41% | 6.41% | **97.90%** |
| QS | OOD accuracy | **79.59%** | 36.59% | 8.50% | 8.50% | 12.08% | 10.69% |
| SCC | OOD accuracy | **84.01%** | 36.72% | 27.78% | 27.20% | 29.41% | 77.34% |
| AS | OOD accuracy | **99.53%** | 58.29% | 62.62% | 62.62% | 62.62% | 98.03% |
| FMS | OOD accuracy | **90.76%** | 30.73% | 27.64% | 27.64% | 28.79% | 85.64% |
| Dij | OOD accuracy | 81.42% | 74.92% | 74.04% | 74.04% | 74.10% | **99.22%** |
| LCS | OOD accuracy | **88.28%** | 81.15% | 56.34% | 56.34% | 61.33% | 87.60% |
| Min | OOD accuracy | 85.28% | 65.28% | 67.82% | 67.82% | 67.82% | **99.07%** |
| TS | OOD accuracy | 88.37% | 59.48% | 53.73% | 53.73% | 57.63% | **99.95%** |
| Average | OOD accuracy | **87.42%** | 52.52% | 40.34% | 40.04% | 41.74% | 83.92% |

Table 1 presents the primary performance evidence: DyScope attains strong results across mathematical reasoning, code generation, and CLRS-10 task clusters. The matched ScopeBench comparison then explains this performance advantage. DyScope improves terminal quality by +3.66% over fixed scope, with a 95% interval of $[+0.00185, +0.07414]$ (Table 2). Since all remaining search conditions are shared, this difference identifies the contribution of adapting the editing boundary.

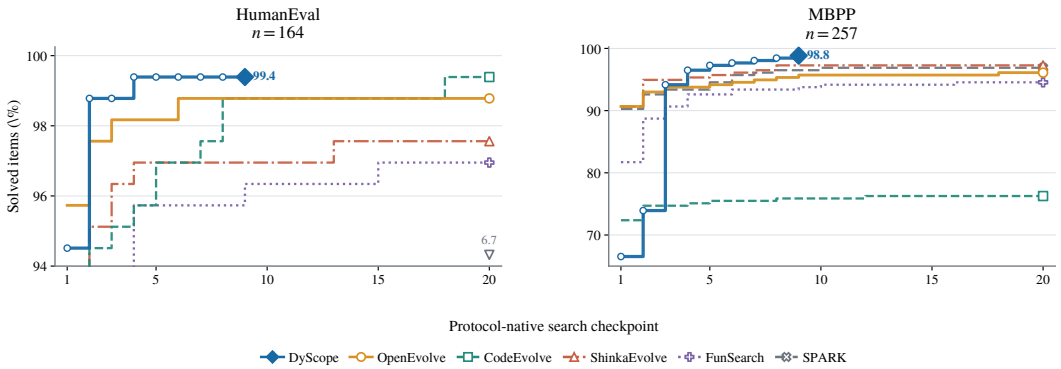


Figure 5: **DyScope performance across benchmarks: progress.** DyScope follows cumulative repair checkpoints over unresolved items; the comparison curves follow each method's native proposal, generation, attempt, or convergence checkpoints.

Table 2: **ScopeBench ablation results.** The first rows form one continuous ablation panel: the matched boundary-policy comparison contrasts DyScope with a fixed-scope control, while the adjacent mechanism rows remove structural routing, recovery, and boundary enforcement on the complete 18-task suite. The repository-native rows report final-controller performance by task family, and the final rows contrast two Codex configurations on the same task suite.

| **ScopeBench ablations** | | | | | |
|---|---|---|---|---|---|
| **Protocol** | **Cache (%) ↑** | **Graph routing (%) ↑** | **Knapsack (%) ↑** | **Scheduling (%) ↑** | **Overall (%) ↑** |
| Fixed scope | 69.72% | 81.34% | 75.39% | 99.43% | 82.15% ± 14.68% |
| **DyScope** | **71.27%** | **93.88%** | **99.85%** | **99.57%** | **90.18% ± 12.85%** |
| **DyScope w/o structural routing** | 68.50% | 91.54% | 99.47% | 99.55% | 88.69% ± 14.95% |
| **DyScope w/o boundary enforcement** | 68.99% | 88.75% | 82.09% | 99.55% | 85.15% ± 13.59% |
| **DyScope w/o recovery** | 72.53% | 93.34% | 94.06% | 99.58% | 89.41% ± 12.57% |
| **Multi-method ScopeBench comparison** | | | | | |
| **Method** | **Cache (%)** | **Graph routing (%)** | **Knapsack (%)** | **Scheduling (%)** | **Overall (%)** |
| **DyScope** | **71.27%** | 93.88% | 99.85% | 99.57% | **90.18%** |
| OpenEvolve | 59.10% | **100.00%** | **99.74%** | **99.64%** | 88.49% |
| CodeEvolve | 60.06% | 99.95% | 92.74% | 99.58% | 87.56% |
| ShinkaEvolve | 59.33% | 97.24% | 92.30% | 99.44% | 86.50% |
| FunSearch | 58.71% | 99.69% | 93.80% | 99.47% | 87.27% |
| SPARK | 58.82% | 70.49% | 80.25% | 99.41% | 76.91% |
| **Codex-based ScopeBench comparison** | | | | | |
| **System** | **Cache (%)** | **Graph routing (%)** | **Knapsack (%)** | **Scheduling (%)** | **Overall (%)** | **Time/run (s)** |
| Codex + DeepSeek-V4.1-Flash | 72.74% | 100.00% | 99.94% | 99.74% | 92.34% ± 12.73% | 4684.81 ± 1391.00 |
| **Codex + DeepSeek-V4.1-Flash + DyScope** | **68.79%** | **100.00%** | **99.94%** | **99.69%** | **91.23% ± 14.56%** | 269.28 ± 143.08 |

Table 2 reports DyScope at $90.18\% \pm 12.85\%$ on ScopeBench. Its strongest task-family results are $93.88\%$ on graph routing, $99.85\%$ on knapsack, and $99.57\%$ on scheduling; cache remains the most difficult family at $71.27\%$. The fixed-scope control reaches $82.15\% \pm 14.68\%$ in the matched 54-cell evaluation. The component controls further expose the role of the controller: removing structural routing, recovery, or boundary enforcement yields $88.69\%$, $89.41\%$, and $85.15\%$, respectively, on their 18-task evaluations.

We additionally integrate DyScope with Codex through the same ScopeBench protocol. Codex with DeepSeek-V4.1-Flash serves only as the candidate proposer: it receives the task instruction, current incumbent, and permitted feedback. DyScope then selects the editable scope for each proposal and applies its transition contract to the realized parent–child diff before the evaluator admits the candidate. The two Codex rows therefore use the same task suite, 20-step proposal budget, initialization, and evaluator interface, differing only in whether this DyScope boundary-control layer is active. This integration produces $92.34\% \pm 12.73\%$ for the Codex proposer and $91.23\% \pm 14.56\%$ when it is coupled with DyScope, demonstrating that the controller can be placed around an external proposal system without changing the benchmark or evaluation contract.

Figure 6 complements terminal scores with six views of the same matched study: task-family and difficulty transfer, gains over initialization, proposal trajectories, matched wall-time quality, and a representative boundary trace. Together they show the intended search pattern: selective widening makes otherwise inaccessible transitions available, and strict refinement consolidates the resulting improvement without adopting a permanently broad editing policy.

## 5 LIMITATIONS

DyScope relies on meaningful executable regions, approximate dependencies, and a mapping from requested scopes to realized edits. Dynamic dependencies or externally mediated behavior can

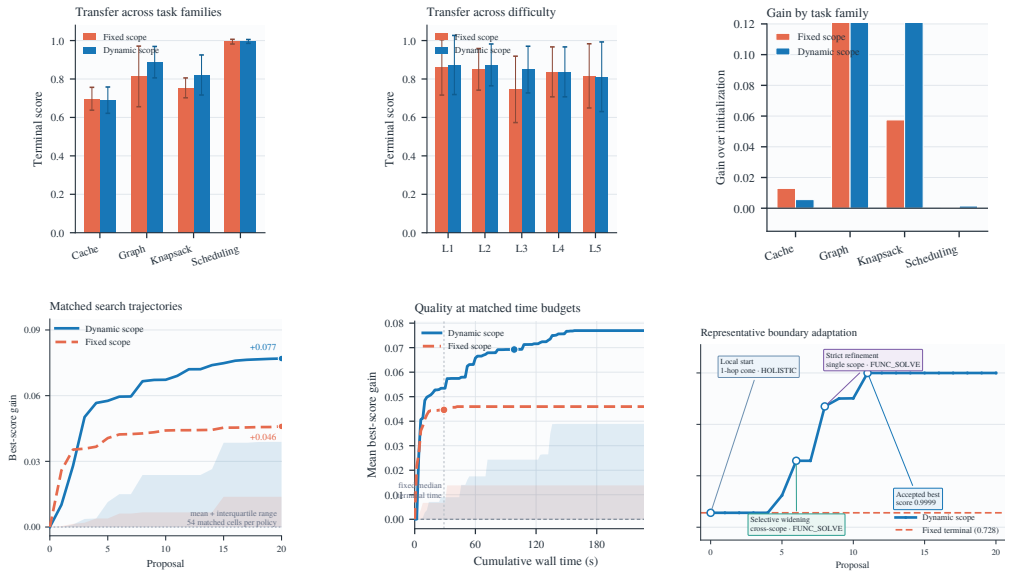


Figure 6: **Matched ScopeBench search and transfer.** Compact views of (a) transfer across task families, (b) transfer across difficulty levels, and (c) matched gain over initialization by task family (top row), followed by (d) aggregate proposal trajectories, (e) quality at matched wall-time budgets, and (f) a representative boundary trace (bottom row). The panels preserve the original data and estimands while using a reduced, single-display layout.

violate these assumptions, and the current fixed decomposition does not isolate the causal effect of every boundary decision (Wang et al., 2026; Iacob et al., 2026). Boundary search and verification also add controller cost; extending the decomposition itself and evaluating the resulting trade-offs under cost-matched protocols are important next steps.

## 6 Discussion and Conclusion

DyScope makes edit scope a transition-level search variable. By retaining, widening, redirecting, or contracting the boundary over a lineage, it balances the reachability of coordinated changes against the attribution afforded by local edits. The transition contract checks that the selected boundary matches the realized diff, making dynamic scope enforceable without implying that it necessarily improves program validity.

In the matched boundary-policy comparison, dynamic scope reaches 90.18% versus 82.15% for fixed scope with the surrounding search protocol held constant. Component ablations and the external-proposer integration further localize the mechanism and demonstrate portability.

Separating transition semantics from boundary selection also permits future policies to be learned or optimized over the same action space without changing what an admissible edit means. The selection policy can evolve while the contract and transition lineage remain comparable and auditable.

The contribution is the dynamic-scope abstraction—boundary state, factored actions, realized-diff verification, and transition lineage—not a particular set of heuristic coefficients. For recursively self-improving systems, it offers an explicit, enforceable, and traceable interface for changing the evolving system itself. DyScope thus provides a concrete foundation for auditable recursive self-improvement, while our evidence remains limited to bounded program-evolution trajectories rather than open-ended RSI.

## Ethics Statement

This work studies automated program modification on public benchmark tasks and does not involve human participants, personal data, or deployment in a safety-critical setting. Systems that generate and execute code can nevertheless be misused to produce faulty or harmful software. Our experiments therefore execute candidates in cleared, network-isolated workers and restrict the proposer to public task instructions, the incumbent program, and declared black-box feedback; evaluator implementations, reference solutions, and protected evaluation artifacts are withheld from the proposal context. These controls do not eliminate risks in downstream deployment, where generated code should remain subject to sandboxing, review, and domain-appropriate validation.

## Reproducibility Statement

The paper specifies the DyScope controller, transition contract, evaluators, and metrics needed to reproduce the reported studies. The main text reports the benchmark versions, task splits, proposal budgets, seeds, decoding settings, and matched comparison design; the supplement provides the complete reproduction protocol, policy settings, statistical procedure, and additional task-level results. All reported ScopeBench comparisons use the stated public release programs and fixed evaluator interfaces; each benchmark is evaluated with its prescribed metric and split. These details are intended to support independent reruns under the same information boundary.

## AI Use Statement

Large language models are studied as proposal models under the controlled information boundaries specified in this paper. AI-assisted tools were also used to support manuscript preparation. The authors designed the experimental protocols, verified the reported results, reviewed the final manuscript, and remain responsible for its scientific claims and conclusions.

## References

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. CodeEvolve: An open source evolutionary coding agent for algorithmic discovery and optimization, 2025.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-level coding using LLMs and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698, 2024. doi: 10.1145/3643757.

Ryan Boldi, Martin Briesch, Dominik Sobania, Alexander Lalejini, Thomas Helmuth, Franz Rothlauf, Charles Ofria, and Lee Spector. Informed down-sampled lexicase selection: Identifying productive training cases for efficient problem solving, 2023.

Wei Cheng, Yuhan Wu, and Wei Hu. Dataflow-guided retrieval augmentation for repository-level code completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7957–7977, 2024. doi: 10.18653/v1/2024.acl-long.431.

Luis Da Costa, Álvaro Fialho, Marc Schoenauer, and Michèle Sebag. Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, pp. 913–920, 2008.

Dekun Dai, MingWei Liu, Anji Li, Jialun Cao, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. FeedbackEval: A benchmark for evaluating large language models in feedback-driven code repair tasks, 2025.

A. E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999. doi: 10.1109/4235.771166.

Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. Comparison-based adaptive strategy selection with bandits in differential evolution. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, 2010.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Qihan Ren, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence, 2025.

Hongshu Guo, Zeyuan Ma, Jiacheng Chen, Yining Ma, Zhiguang Cao, Xinglin Zhang, and Yue-Jiao Gong. ConfigX: Modular configuration for evolutionary algorithms via multitask reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2025.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 35–46, 1988. doi: 10.1145/53990.53994.

Alex Iacob, Andrej Jovanović, William F. Shen, Daniel Burkhardt, Meghdad Kurmanji, Nurbek Tastan, Lorenzo Sani, Niccolò Alberto Elia Venanzi, Ambroise Odonnat, Zeyu Cao, Bill Marino, Xinchi Qiu, and Nicholas D. Lane. The red queen gödel machine: Co-evolving agents and their evaluators, 2026.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pp. 240–248, 2016.

Jiachen Jiang, Huminhao Zhu, and Zhihui Zhu. SMCEvolve: Principled scientific discovery via sequential monte carlo evolution, 2026.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution, 2025.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012. doi: 10.1109/TSE.2011.104.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185): 1–52, 2018.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2401.02051.

Yihong Liu, Junyi Li, Wayne Xin Zhao, Hongyu Lu, and Ji-Rong Wen. Experience-guided reflective co-evolution of prompts and heuristics for automatic algorithm design, 2025.

Zhen Liu, Yuhan Liu, Jinjun Wang, Wei Song, Jianyi Liu, and Jingwen Fu. Structured progressive knowledge activation for LLM-driven neural architecture search, 2026.

Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 298–312, 2016. doi: 10.1145/2837614.2837617.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations*, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36*, pp. 46534–46594, 2023. doi: 10.52202/075280-2019.

Matias Martinez and Martin Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *Journal of Systems and Software*, 151:65–80, 2019. doi: 10.1016/j.jss.2019.01.069.

Jorge Maturana, Álvaro Fialho, Frédéric Saubion, Marc Schoenauer, and Frédéric Lardeux. Adaptive operator selection and management in evolutionary algorithms. In *Autonomous Search*, pp. 161–189. Springer, 2012. doi: 10.1007/978-3-642-21434-9_7.

Alexander Novikov et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025.

Bernardino Romera-Paredes et al. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2023. doi: 10.1038/s41586-023-06924-6.

Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent. GitHub software, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve. OpenEvolve project repository.

Temoor Tanveer. LEVI: Stronger search architectures can substitute for larger llms in evolutionary search, 2026.

Niki van Stein and Thomas Bäck. LLaMEA: A large language model evolutionary algorithm for automatically generating metaheuristics, 2024.

Wenyi Wang, Piotr Piekos, Nanbo Li, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-gödel machine: Human-level coding agent development by an approximation of the optimal self-improving machine, 2025.

Xing Wang, Xiaoyi Zhang, and Jie Shao. Auditing harness tampering in self-improving agents, 2026.

Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984. doi: 10.1109/TSE.1984.5010248.

James M. Whitacre, Tuan Q. Pham, and Ruhul A. Sarker. Credit assignment in adaptive evolutionary algorithms, 2009.

Zhenyu Wu, Qingkai Zeng, Zhihan Zhang, Zhaoxuan Tan, Chao Shen, and Meng Jiang. Large language models can self-correct with key condition verification. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 12846–12867, 2024. doi: 10.18653/v1/2024.emnlp-main.714.

Minghao Yan et al. PACEvolve: Enabling long-horizon progress-aware consistent evolution, 2026.

Yang Yang, Zining Zhong, Jindong Li, Jiemin Wu, Kaishen Yuan, Wenshuo Chen, Menglin Yang, and Yutao Yue. TurboEvolve: Towards fast and robust llm-driven program evolution, 2026.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution, 2024.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursive self-improvement, 2024.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2025.