ADAPTOFLUX: A TYPED GRAPH REPRESENTATION SUBSTRATE FOR STATEFUL SYMBOLIC PROGRAM SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Discrete program synthesis searches a combinatorial space of typed, executable structures, where invalid operator compositions and state-dependent control logic make even well-formed programs hard to discover. We present **AdaptoFlux**, a representation substrate that couples a metadata-driven **Method Pool**—registering heterogeneous, typed operations, including side-effectful actions—with **Generalized Data Flow Graphs (GDFGs)**, typed DAGs whose edges are statically checked for type compatibility. State-dependent behavior is accommodated without abandoning the DAG formulation: through time-step unrolling, nodes read from and write to an implicit external state, decoupling mutable state from graph topology. AdaptoFlux deliberately separates the representation space from the search procedure: type constraints prune invalid compositions before search begins, and the same substrate can be paired with arbitrary optimizers. Using lightweight gradient-free search procedures as proof-of-concept probes, we validate the space's navigability, type safety, and interface completeness on canonical static tasks; on a minimal stateful synthesis task, the type-constrained space automatically synthesizes the target stateful graph in 8 of 10 runs, versus 0 of 10 when type routing is disabled. A hand-crafted GDFG controlling a Snake agent further demonstrates that pure computation and side-effecting actions can execute within a single unified graph. AdaptoFlux is a concept-and-feasibility contribution to representation design, not a new optimization algorithm, and is not intended to outperform specialized symbolic-regression systems (e.g., PySR, gplearn) on static benchmarks.

1 INTRODUCTION

The success of contemporary deep learning relies on gradient optimization over smooth, continuous parameter manifolds. Symbolic program synthesis instead searches a **discrete combinatorial space** in which invalid operator compositions and state-dependent control logic make even well-formed candidate programs hard to discover. The difficulty is structural rather than incidental: without type constraints to prune such compositions and metadata to guide traversal, the space is effectively unnavigable, as our open-loop generation probes show (Appendix J). Consistently, a fitness landscape analysis of tree-based GP search spaces reports many local optima and high deception, with mutation in deep nodes making the global optimum hard to reach He & Neri (2023).

The limitations of existing methods stem from **structural deficits in their expression spaces**. Traditional Symbolic Regression (SR), particularly tree-based approaches Cranmer (2023), inherently struggles with variable reuse and state dependencies. While DAG-based methods like **Cartesian Genetic Programming (CGP)** Miller & Turner (2015) address variable reuse, they lack a unified mechanism for heterogeneous semantic operations, and pure-functional frameworks such as **DreamCoder** Ellis et al. (2021) model stateful interactions only through explicit state threading. Neuro-symbolic approaches Garcez & Lamb (2023) likewise depend on static priors or post-hoc explanations. These paradigms separate pure computation from state interaction, so the perception–decision–action loop that RL treats as one interacting system Sutton et al. (1998) must be decomposed into separate, statically typed stages. AdaptoFlux instead employs a metadata-driven Method Pool

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.

that admits arbitrary side-effectful functions, enabling the graph to interact with an implicit state space while maintaining structural determinism.

To bridge this gap, we propose **AdaptoFlux**, a representation substrate that formalizes the discrete expression space of program synthesis. It couples a metadata-driven **Method Pool** of typed operations with **Generalized Data Flow Graphs (GDFG)**, and a **Time-step Unrolling** mechanism that maps static DAGs Cormen et al. (2022) into cross-step state machines supporting both static forward propagation and dynamic behavioral feedback.

This paper focuses on the **construction and navigability of the expression space**, rather than proposing a new optimization algorithm; we employ classic gradient-free strategies as *proof-of-concept probes* to validate structural integrity. Our main contributions are: (i) we formalize the AdaptoFlux hypothesis class as typed DAGs over a fixed method pool, whose type system defines the space by admitting only executable compositions and yields a navigable discrete expression space; (ii) we propose the Method Pool as a unified container in which pure functions and side-effectful actions coexist, providing a structured interface for cross-task knowledge reuse; (iii) we empirically validate the substrate with lightweight probes, demonstrating navigability and type safety on canonical static tasks and **automatic synthesis of simple stateful graphs**—8/10 runs under type constraints versus 0/10 without (Section 4.3); and (iv) we show via a hand-crafted Snake controller (Appendix F) that pure computation and side-effecting actions execute within one unified graph, an existence proof intended as a foundation for future state-aware search strategies.

An anonymized implementation of AdaptoFlux is publicly available for peer review at: `https://anonymous.4open.science/r/AdaptoFlux-CF54`. Upon acceptance, the complete source code will be publicly released on GitHub.

2 Positioning and Brief Related Work

Traditional Symbolic Regression (e.g., PySR Cranmer (2023)) and tree-based GP Koza (1994) excel at static mappings but lack native mechanisms for state-dependent operations. While DAG-based methods like CGP Miller & Turner (2015) address variable reuse, they lack unified mechanisms for heterogeneous side-effects. Pure functional frameworks like DreamCoder Ellis et al. (2021) require complex explicit state threading to model dynamic interactions. AdaptoFlux fundamentally differs by natively integrating side-effectful actions within a typed GDFG, decoupling static topology from dynamic state mutations.

Our type-constrained search shares with Inductive Logic Programming (ILP) systems such as Popper Cropper & Morel (2021) the pruning of hypothesis spaces, but AdaptoFlux operates on a broader hypothesis class of black-box executable functions, natively encompassing non-monotonic side-effects that cannot be reduced to Horn clauses; unlike LLM-guided search Wang et al. (2025), its strict type system also guarantees structural executability.

Detailed comparisons with SR, CGP, DreamCoder, ILP/Popper, and neuro-symbolic methods, and the capability absent from each, are discussed system by system in **Appendix A**: none of them offers a single type-checked container in which pure functions and side-effectful black-box actions coexist while the graph stays acyclic. AdaptoFlux does not claim competitiveness on static symbolic-regression benchmarks (e.g., PySR, gplearn); Appendix E is a structural sanity check under a deliberately restricted method pool.

3 System Architecture

As illustrated in the Figure 1, the core design of AdaptoFlux lies in constructing a discrete expression space synergistically defined by a metadata-driven Method Pool and Generalized Data Flow Graphs (GDFG). The Method Pool serves as the symbolic hub, encapsulating heterogeneous methods into structured tuples with a standardized interface. Meanwhile, the GDFG acts as the structural skeleton with adaptive execution semantics: for static tasks it operates as standard DAG forward propagation, while for temporal dependencies it uses Time-step Unrolling and Cross-step Feedback to realise dynamic semantic loops *at runtime*, leaving the stored topology strictly acyclic (Appendix H.6). This dual formalization of "symbols and structure" accommodates execution modes from single-pass inference to iterative cyclic execution within one architecture.

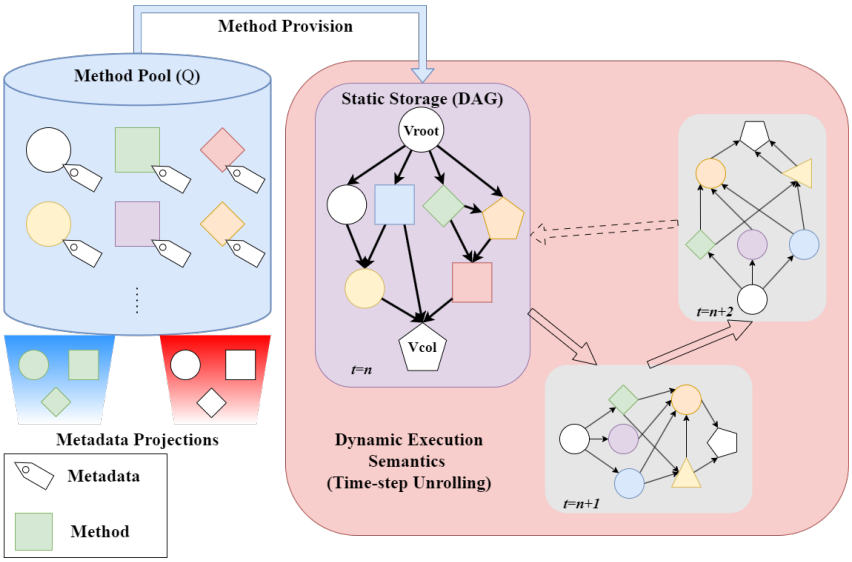


Figure 1: AdaptoFlux architecture. The stored GDFG (centre) is strictly acyclic: loops and state mutations live in the runtime state layer, so loop semantics and per-step parallel scheduling coexist (Appendix H.6).

3.1 Method Pool

The architectural cornerstone of AdaptoFlux is the **Method Pool** Q, the framework's **Symbolic Hub** for semantic heterogeneity and knowledge fragmentation in discrete spaces. Rather than rigidly isolating methods into separate sub-libraries, we adopt a **Single-Pool Multi-View** architecture: all atomic operations are registered in a single logical container whose classification, scheduling and probe-side view projection are achieved entirely through **metadata-driven feature partitioning**.

Formally, any method $m \in \mathrm{Q}$ in the pool is abstracted as a structured tuple $m = \langle \mathrm{impl}, \sigma_{\mathrm{in}}, \sigma_{\mathrm{out}}, \mathrm{M} \rangle$. Here, impl denotes the concrete implementation, $\sigma_{\mathrm{in/out}}$ defines strict type signatures to ensure data flow safety, and the extended metadata set M carries key information such as semantic tags and group identifiers, sampling weights, side-effect identifiers, and evolutionary history. Navigation probes (e.g., LayerGrow/GraphEvo) can dynamically project task-specific logical subsets $\mathrm{Q}_{\mathrm{view}} = \{m \in \mathrm{Q} \mid \phi(\mathrm{M}_m) = \mathrm{True}\}$ via a metadata filter $\phi(\mathrm{M})$ for on-demand scheduling. Metadata is not part of the hypothesis class, however: $\Omega_{\mathrm{Q},\Sigma}$ (Eq. 1) is fixed by the pool and the type system alone, and ϕ only restricts which of its elements a probe examines ($\Omega_{\mathrm{Q}_{\mathrm{view}},\Sigma} \subseteq \Omega_{\mathrm{Q},\Sigma}$). We accordingly separate *hard* constraints, whose violation yields a non-executable graph, from *soft* search-side ones (Appendix H.4).

Engineering realization (metadata injection, type-routing validation, integration interfaces) is detailed in Appendix H and the anonymous code repository.

3.2 Generalized Data Flow Graph (GDFG)

To establish structural order within the discrete combinatorial space, AdaptoFlux anchors executable structures to the **Generalized Data Flow Graph (GDFG)**. This architecture decouples static topological determinism from dynamic execution feedback by distinguishing between the framework's static storage structure and its dynamic semantics. Specifically, the underlying topology of any graph instance G is strictly a **Directed Acyclic Graph (DAG)** to ensure efficient parallel scheduling and avoid NP-hard subgraph isomorphism problems, while the "Generalized" aspect centers on the native support for **heterogeneous methods** via the method pool, complemented by a **Time-step Unrolling** mechanism that maps static DAGs into cross-step state machines to accommodate state-dependent logic and side effects. Let Q denote the method pool and let $\Sigma = \Theta$ be the global type system: a finite set of type atoms Θ (e.g. `scalar`, `vector`, `pos_type`). Every method declares, for each of its argument positions, a set of admitted atoms $\sigma_{\mathrm{in}}(v) \subseteq \Theta$, together with the atoms it may emit,

$\sigma_{\text{out}}(v) \subseteq \Theta$, and a fixed arity that is checked when the method is registered. An output carrying atom τ may feed a position that admits τ, i.e. the edge is admissible exactly when the two sets share an atom, $\sigma_{\text{out}}(u) \cap \sigma_{\text{in}}(v) \neq \emptyset$; routing is positional, so an output reaches only those argument positions whose declared set contains the atom it carries. The prototype decides admissibility by a membership test on Θ and performs no subtyping inference, so the check costs time linear in the number of incident edges. The search space consists of valid DAGs $G = \langle V, E \rangle$ satisfying: (1) The node set $V = V_{\text{root}} \cup V_{\text{proc}} \cup V_{\text{col}}$, where each processing node $v \in V_{\text{proc}}$ is bound to an atomic operation $m_v \in \mathrm{Q}$ with explicit type signatures; (2) The edge set E strictly enforces type compatibility $\sigma_{\text{out}}(u) \cap \sigma_{\text{in}}(v) \neq \emptyset$; and (3) G is acyclic. Consequently, the discrete expression space is defined as:

$$\Omega_{\mathrm{Q},\Sigma} = \Big\{ G = \langle V, E \rangle \;\Big|\; \begin{aligned} &G \text{ is a DAG}; \\ &\forall v \in V_{\text{proc}}, m_v \in \mathrm{Q}; \\ &\forall (u,v) \in E, \sigma_{\text{out}}(u) \cap \sigma_{\text{in}}(v) \neq \emptyset \Big\} \end{aligned} \tag{1}$$

The single-step execution of a GDFG integrates topological data flow with global state interaction. Let the global state space be S, the input space be X, and let $\Delta\mathrm{S}$ denote the space of state increments, with $\oplus : \mathrm{S} \times \Delta\mathrm{S} \to \mathrm{S}$ the operator that applies an increment to a state. Each processing node contributes a **local state influence** $\mathrm{I}_v : \mathrm{Data}_v \times \mathrm{S} \times \mathrm{X} \to \Delta\mathrm{S}$, where $\mathbf{in}_v$ denotes the data inputs aggregated from topological predecessors; pure mapping nodes contribute the neutral increment $\mathbf{0}_{\Delta\mathrm{S}}$, whereas side-effectful action nodes contribute the implicit state mutation. Because a node's data inputs are read along the acyclic edges, the nodes admit a topological order $v_1 \prec \cdots \prec v_m$, and one step of the graph is defined as the corresponding fold:

$$\mathbf{s}^{(0)} = \mathbf{s}_t, \qquad \mathbf{s}^{(k)} = \mathbf{s}^{(k-1)} \oplus \mathrm{I}_{v_k}\big(\mathbf{in}_{v_k}, \mathbf{s}^{(k-1)}, \mathbf{x}_t\big), \qquad \mathbf{s}_{t+1} = \mathbf{s}^{(m)}. \tag{2}$$

Node order therefore matters in general, which is the precise sense in which, as noted above, the topology fixes the order of the state changes. When the increments do not interact—formally, when every I_v may be evaluated on the same pre-step state and the increments it produces act on disjoint state coordinates, so that $\oplus$ is commutative over the reachable increments—the fold is order-independent and collapses to the data-parallel form

$$\mathbf{s}_{t+1} = \mathbf{s}_t \oplus \left(\bigoplus_{v \in V} \mathrm{I}_v(\mathbf{in}_v, \mathbf{s}_t, \mathbf{x}_t) \right), \tag{3}$$

which the scheduler exploits when several nodes are ready concurrently. Both regimes occur in this paper: the static probes use purely functional pools and the minimal stateful task requires a single side-effectful node, so they are covered by Eq. 3; the hand-crafted Snake controller instead chains three action nodes (`turn_left`, `turn_right`, `go_straight`) in one layer that write the same environment state, and therefore follows Eq. 2. Behaviour is determined by the semantic properties of $m_v \in \mathrm{Q}$, supporting a continuous spectrum of operations rather than rigid categories. For pure mapping nodes, I_v primarily propagates data through $\mathbf{in}_v$ with negligible state mutation; for side-effectful action nodes, I_v captures implicit state mutations triggered by environmental interactions, effectively creating an "implicit state space" accessible to the graph. Boundary nodes ($V_{\text{root}} \cup V_{\text{col}}$) are **semantic interfaces**: V_{root} distributes typed inputs via identity mapping and V_{col} projects internal states to the final interface, neither being mandatory unless explicitly configured. This formulation unifies two execution paradigms: in the *Pure Computational View*, execution degenerates into standard DAG forward propagation with $\mathbf{s}_{t+1} \approx \mathbf{s}_t$ carrying only final outputs; in the *State-Behavior View*, execution manifests as a sequence of implicit state mutations where the topology fixes the order of changes and the Method Pool their semantic impact. This design ensures strict DAG storage efficiency while natively accommodating state-dependent logic through the orthogonal decoupling of "static topology" and "dynamic side-effects."

Static acyclicity and runtime loops. Acyclicity constrains the *stored* topology, not the behaviour it induces: loops enter at runtime only—via time-step unrolling with the collapsed state fed back to the next step (Eq. 2), via methods whose opaque `impl` may self-invoke or execute a further registered graph, and via the V_{col} interface. Unrolling T steps yields T copies of G joined only by forward-in-t edges—still a DAG—so per-step parallel scheduling is preserved (Appendix H.6).

3.3 Validating the Navigability of the Expression Space

To validate the navigability and interface completeness of this space, we equip AdaptoFlux with three classic gradient-free search strategies. These are not algorithmic contributions but **Proof-of-Concept Probes** delineating the framework's capability boundaries under controlled conditions:

LayerGrow (Structural Initialization): This strategy admits only type-compatible compositions when growing layers and constructs the graph topological skeleton through incremental layering. This probe validates the type safety and compositional validity of the expression space during macro-structural growth.

GraphEvo (Topological Refinement): This strategy executes local greedy node replacement on a fixed graph skeleton. It aims to verify the loss descent potential within the discrete neighborhood of the expression space, demonstrating that performance optimization can be achieved through local semantic alignment given a fixed topological structure.

CombinedTrainer (Serial Composition and Pool Filtering Validation): This strategy composes LayerGrow's structural exploration with GraphEvo's local refinement, and adds an **experimental genetic algorithm-based method pool filtering module**. Through the two-stage "Grow $\rightarrow$ Refine" schedule it validates two hypotheses: (1) the feasibility of first constructing the macro-skeleton and then fine-tuning local semantics in the discrete space; and (2) the effectiveness of simple genetic filtering in compressing method-pool redundancy.

The complete algorithmic logic, complexity analysis, and engineering implementation details of these three probes are provided in Appendix B.3.

3.4 Inference Complexity and Parallel Scalability

The computational cost of executing synthetic GDFG is a fundamental property of the AdaptoFlux representation, independent of any specific optimization strategy. Let N denote the batch size, V the number of nodes, E the number of edges, and $\bar{C}_{\text{op}}$ the average computational cost per node. A single forward pass includes topology scheduling ($O(V+E)$), edge-level data routing ($O(E \cdot N \cdot C_{\text{copy}})$), and node-level computation ($O(N \cdot V \cdot \bar{C}_{\text{op}})$). The total inference complexity is given by:

$$T_{\text{infer}} = O\big(N \cdot (E \cdot C_{\text{copy}} + V \cdot \bar{C}_{\text{op}}) + V + E\big). \tag{4}$$

In typical deployment scenarios where $N \gg V, E$, the dominant term simplifies to $O(N \cdot V \cdot \bar{C}_{\text{op}})$: cost grows linearly in graph scale V without hidden quadratic interactions.

The explicit dependency metadata in GDFG supports efficient topology-aware parallel scheduling. For a graph with an average width of W (maximum number of concurrent nodes), the ideal speedup for P cores is bounded by $\min(P, W)$. Empirical results show that when methods release the GIL (e.g., via I/O or C extensions), wide and shallow graph structures reach a mean speedup of $10.25\times$ on 16 cores (64% of the ideal linear bound; Appendix C.3); for CPU-bound Python operators this gain disappears (Limitations).

Detailed derivations of inference overhead, lock contention, and GIL impact analysis are provided in Appendix B.

3.5 Search-Space Structure and Probe Complexity

Beyond inference over a fixed graph, the cost of navigating $\Omega_{\text{Q},\Sigma}$ is governed by the size of the candidate space. With a layer budget L and k nodes per layer, the number of syntactically valid graphs grows exponentially in depth, since each of the kL slots independently admits any method of the pool it is drawn from. Type routing acts on the space itself: an edge is admitted only if $\sigma_{\text{out}}(u) \cap \sigma_{\text{in}}(v) \neq \emptyset$, so type-incompatible compositions are absent from $\Omega_{\text{Q},\Sigma}$ and never reach evaluation. Metadata acts only on the probe: a filter ϕ draws candidates from Q_{view}, restricting the explored subset ($\Omega_{\text{Q}_{\text{view}},\Sigma} \subseteq \Omega_{\text{Q},\Sigma}$) without changing what the engine can execute. The probes inherit this structure: LayerGrow costs $T_{\text{LG}} = O(AN\bar{C}_{\text{op}}kL^2)$, quadratic in the layer budget; GraphEvo refinement costs $T_{\text{GE}} = O(CRNV_{\text{init}}\bar{C}_{\text{op}})$, linear in cycles C and steps R precisely because the topology is frozen, plus a sparsely triggered pool-evolution term $O(CSV^3/f)$; and CombinedTrainer composes them as $O(C_{\text{outer}}N\bar{C}_{\text{op}}(AkL^2 + RV_{\max}))$ without genetic filtering. Three consequences

follow: evaluation, not candidate generation, dominates a single attempt; type constraints remove non-executable candidates from the space, whereas metadata and pool composition shift the probes' expected branching factor without changing the asymptotic form of navigation; and fixed-topology refinement keeps the knowledge-reuse loop linear rather than combinatorial. Derivations are in Appendix B.3.

4 EXPERIMENTAL VALIDATION

This section validates the navigability and interface specifications of the AdaptoFlux representation space (GDFG + Method Pool) rather than performance upper bounds. LayerGrow and GraphEvo are positioned as Proof-of-Concept Probes under three principles: (1) isolating variables to the combinatorial capabilities of graph structures; (2) validating probes independently or combined on demand; and (3) separating representational expressiveness from navigation limits. Code, logs, and reproduction scripts: `https://anonymous.4open.science/r/AdaptoFlux-CF54`.

4.1 VALIDATION OF STRUCTURAL EXPLORATION BOUNDARIES FOR LAYERGROW

To evaluate the structural exploration capability of the `LayerGrow` probe in discrete space, we conducted controlled experiments on the Titanic datasetCukierski (2012). We employed a simplified method pool (9 scalar methods, see Appendix C.1) and varied the maximum number of layers, `max_layers` $\in [5, 50]$, to reveal the performance saturation points and failure boundaries of the search strategy. Each configuration was run independently 10 times, with a maximum limit of 2,500 attempts per run. Reported accuracies are means over the 10 runs of each run's best model, with standard deviation, *including* runs that reached the attempt cap (failures).

Performance Saturation and Strategy Failure. As shown in Fig. 2, model performance exhibits a distinct two-phase "saturation-failure" characteristic as depth increases:

1. **Performance Saturation Region** (`max_layers` ≤ 20): Mean best accuracy grows with the number of layers up to 0.763 ± 0.041 at 20 layers (0% failure rate); the strongest single run reached $\sim$82.7%. Within this region search costs remain controllable.
2. **Strategy Failure Region** (`max_layers` ≥ 25): Although the theoretical expression space expands, mean best accuracy no longer improves beyond statistical noise (0.769–0.783 at 25–50 layers), while the failure rate rises from 20% (25–30 layers) to 100% (45+ layers). This indicates that the simple "greedy + backtracking" strategy falls into local optima traps in deep combinatorial spaces, failing to effectively explore more distant topological structures.

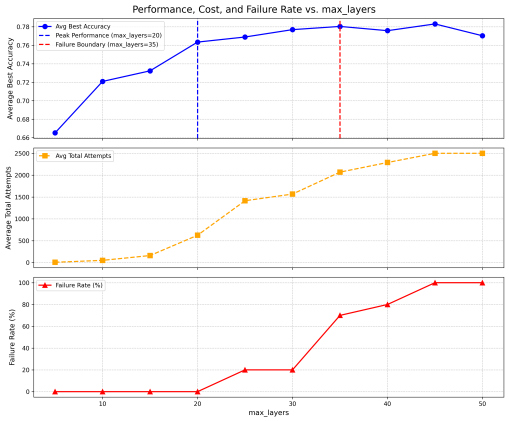
Path Dependence and Search Instability. Even within the performance saturation region, the search process exhibits strong path dependence. As shown in the right panel of Fig. 2, under `max_layers`=20 the 10 runs scatter widely around the mean, the strongest single run reaching $\sim$82.7%. The node structures selected randomly in the early stages have a decisive impact on final performance, confirming the extreme sensitivity to initial conditions in discrete combinatorial spaces. This finding suggests that search efficiency is a bottleneck that emerges earlier than theoretical expressive power; within 20 layers, the framework can effectively navigate the discrete space, while beyond this boundary, more advanced global search strategies are required to break through the combinatorial explosion. Per-configuration statistics are in Table 8 (complete logs in Appendix C.1).

4.2 VALIDATION OF GRAPHEVO PROBE EFFECTIVENESS AND TOPOLOGY-INVARIANT KNOWLEDGE EXTRACTION

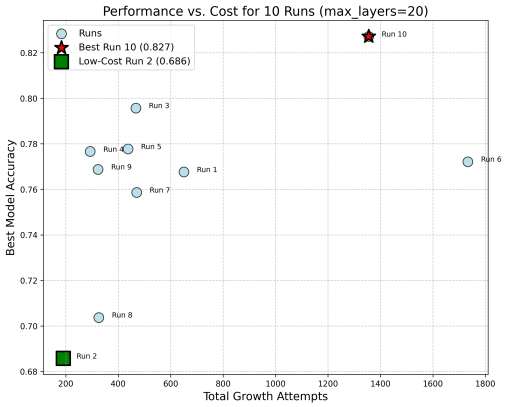
To validate the feasibility of **GraphEvo** as a node-replacement-based local optimization probe in discrete space, and to verify the effectiveness of its derived **topology-invariant knowledge extraction mechanism**, we designed a three-stage mathematical task chain with strong structural dependencies:

$$f_1(x) = 2(x+1) \overset{\text{depend}}{\longrightarrow} f_2(x) = f_1(x) + 3 \overset{\text{depend}}{\longrightarrow} f_3(x) = f_2(x) \times 0.5 \tag{5}$$

This experiment aims to address two core questions regarding the probe mechanism:



Left: Structural exploration boundaries of LayerGrow. Average best accuracy saturates after 20 layers, while the failure rate rises sharply after 35 layers; dashed lines mark the "performance peak" and the "strategy failure boundary".



Right: Scatter plot of runs under `max_layers`=20 (x-axis: attempts, y-axis: accuracy). Significant path dependence leads to substantial performance differences under identical configurations.

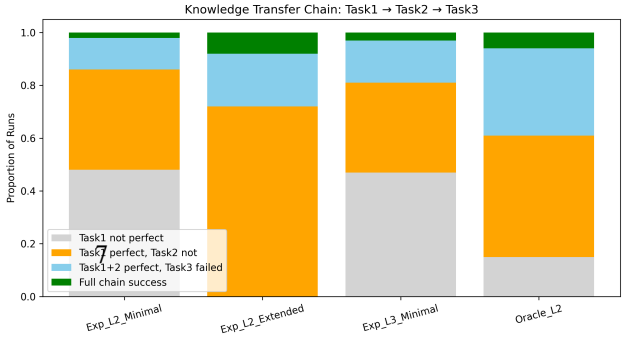
Figure 2: Performance, cost, and failure rate under different maximum layer counts (**left**) and run variability at `max_layers`=20 (**right**).

1. **Effectiveness of Node Replacement Optimization**: Under fixed topological constraints, can GraphEvo effectively reduce loss through greedy replacement, achieving convergence from random initialization to viable solutions?
2. **Feasibility of Topology-Invariant Knowledge Extraction**: Leveraging the fixed topology during the refinement phase, can the system automatically precipitate reusable intermediate modules, thereby supporting zero-shot synthesis across tasks?

Control Variable Design: Initialization Constraints and Knowledge Necessity. To strictly isolate the contribution of "knowledge extraction" to task success rates, we introduced an **initialization layer constraint** in the experimental configuration. This design ensures that: **if knowledge is not successfully extracted and precipitated from preceding tasks, the theoretical success rate for Task 3 is 0**. Furthermore, the framework supports dynamic switching of base method pools at different stages, further enhancing the controllability of knowledge migration paths (see Appendix C.2 for details).

Probe Optimization Performance Analysis. Table 11 (see Appendix C.2) shows that the control group with evolution disabled (Ctrl) reaches 0% success on all tasks, whereas enabling GraphEvo evolution (Exp) raises Task 3 success to $13/100$ runs (mean accuracy 0.131 ± 0.336), a level comparable to the hand-supplied static prior Oracle ($12/100$). Every configuration is executed with the same seed sequence ($42+$rep), so this comparison against the $0/100$ control is paired as well: with thirteen discordant pairs and none in the opposite direction, the exact McNemar test gives one-sided $p = 1.2 \times 10^{-4}$, confirming that **dynamic method-pool precipitation is the interface through which GraphEvo achieves cross-task reuse**. Minimal and Extended pools behave identically at 2 layers (13%), while increasing depth to 3 layers drops success to 9%, indicating that excessively deep spaces dilute the probe's signal-to-noise ratio.

Validation of Topology-Invariant Knowledge Extraction Mechanism. As shown in Fig. 3, the success of Task 3 depends on the structural mastery of the preceding tasks. This is consistent with topology-invariant alignment: with the topology frozen during refinement, execution snap-



shots from different time steps align at the node level, enabling structured reuse. Under the current probe this remains a controlled feasibility result rather than evidence of general knowledge sedimentation.

4.3 Validation of Automatic Stateful Synthesis

A central review question was whether AdaptoFlux can automatically synthesize stateful graphs rather than merely execute hand-crafted ones (e.g., the Snake controller, Appendix F). We therefore designed a **Minimal Stateful Attractor** task to isolate the representation layer's contribution.

Experimental Design. The task is the smallest non-trivial benchmark requiring the discovery of a stateful graph. We injected five irrelevant mathematical methods as distractors to test whether the representation alone constrains the search space. We compared (1) an **Unconstrained Baseline** treating all methods as generic scalar transformations (flat pool, no type routing) with (2) **Type-Constrained AdaptoFlux** using full type signatures. Each setting comprised 10 independent runs—one search per run—with distinct random seeds; a run counts as successful if the target state is reached within 4 steps in $\geq$95% of its 100 evaluation episodes. Both configurations register the same nine methods with byte-identical implementations and differ only in the declared type signatures, so any difference in outcome is attributable to type routing alone; full per-run statistics and reproduction paths are given in Appendix C.4.

Results and Analysis. Fig. 4 gives proof-of-concept evidence for the navigability of the substrate.

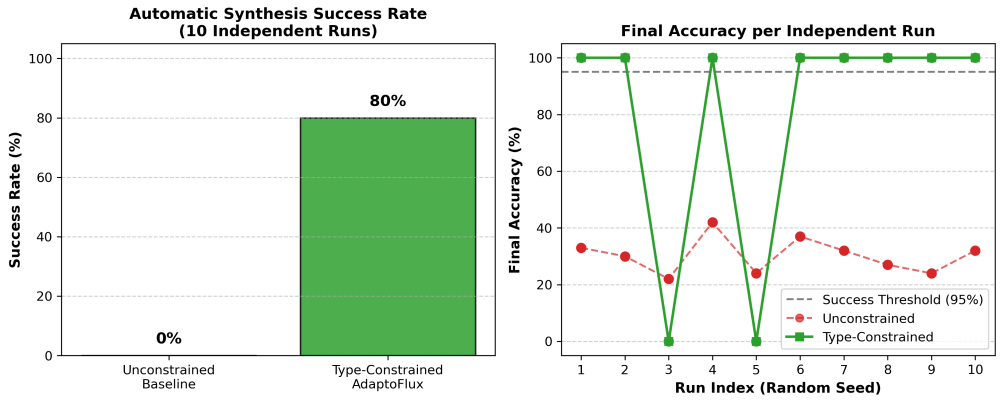


Figure 4: Comparison of automatic stateful synthesis between the Unconstrained Baseline and Type-Constrained AdaptoFlux over 10 independent runs. **Left:** Success rate (0% vs. 80%). **Right:** Final accuracy distribution per run. The dashed line marks the 95% success threshold.

Under the **Unconstrained Baseline**, the probe consistently failed (0%): accuracy stayed between 22% and 42% (mean 0.303 ± 0.060), never crossing the 95% success threshold (Fig. 4, right), as invalid compositions and distractors dominated the space.

With **Type-Constrained AdaptoFlux**, invalid compositions were rejected before search, so the probe synthesized the required stateful graph in 8 of 10 runs (80%), with accuracy reaching 100% in those runs; the two failures end on graphs that omit the side-effecting action node. In every successful run the evaluated graph contains the functional core `root` $\rightarrow$ `calc_direction` $\rightarrow$ `move_towards_target`: a pure node computes the signed distance, and a side-effectful node mutates the implicit state. Because both conditions are run with the same ten seeds, this comparison is paired: the type-constrained search succeeds while the baseline fails in 8 of the 10 pairs, and

the reverse never occurs (Table 17; exact McNemar test, one-sided $p = 3.9 \times 10^{-3}$, two-sided $p = 7.8 \times 10^{-3}$). Sharing the seed sequence controls for initialisation randomness, so the difference is attributable to type routing rather than to a favourable draw of search trajectories.

Discussion. This is a **mechanism validation**: type constraints supply structural priors for navigability, not a claim that the current optimizer solves long-horizon control (see Limitations).

LIMITATIONS

Our claims are bounded as follows. Automatic synthesis is established only on a minimal stateful task; zero-prior synthesis still fails on long-horizon environments, and the Snake controller is handcrafted, an existence proof rather than a learned policy. Under a deliberately restricted primitive set, the static probes recover multiplication-dominated Feynman formulas but fail on Coulomb, IdealGas, and Acceleration, where gplearn and PySR succeed; we claim no competitiveness on static symbolic regression. Knowledge extraction and the navigation probes remain prototype-stage, with low success rates and high variance in deep searches; the Python prototype is GIL-bound and lacks vectorization, so parallel gains hold only in GIL-releasable regimes. These are limitations of navigation and engineering, not of the GDFG representation: loop and recursion support is a representational property established by construction (Appendix H.6), and the probes do not search over loop structure or recursion depth.

5 CONCLUSION

We presented AdaptoFlux, a representation substrate that formalizes the discrete expression space of program synthesis as typed GDFGs over a fixed method pool whose type system defines the admissible compositions, accommodating side-effectful actions alongside pure computation. Proof-of-concept probes indicate that the substrate is navigable, type-safe, and able to synthesize a minimal stateful graph automatically, suggesting value in decoupling representation design from optimization heuristics for future state-aware search; complexity bounds and engineering constraints are detailed in Appendix B.

AI USE STATEMENT

In this work, we used generative AI tools to aid and polish the writing of the manuscript. Concretely, AI tools were used to proofread and edit the manuscript text and to improve its readability; this falls under the tasks with recommended disclosure and did not affect the scientific content, the methodology, or the originality of the research. We have not used generative AI tools for any of the tasks that require disclosure: the research ideas and hypotheses, the conceptual framework and its formalisation, the mathematical definitions and claims, the design of the experiments and of the search probes, the implementation of AdaptoFlux and of the probes, the generation and processing of data, and the interpretation of the results were all produced by the authors themselves. No reported number, table, or figure was generated or altered by an AI tool, and the remaining tasks with required disclosure (for example synthetic data generation, translation, and qualitative analysis of human data) are not applicable to this work. Every AI-assisted passage was reviewed by the authors, checked against our own implementations, logs, and results, and rewritten where necessary before inclusion. We take responsibility for the final content of this work, including all text, claims, and artifacts. The LLMs used as baselines in Appendix J were experimental subjects of this study—we measured their failure modes at direct graph construction—rather than authoring tools.

ETHICS STATEMENT

The research reported here conforms to the ICLR Code of Ethics. Automated program synthesis is a dual-use technology, and we address its societal implications explicitly in the Broader Impacts discussion (Appendix L): we identify the positive impacts (interpretability and transparency of automated decision-making, and the low compute footprint of a CPU-only framework) together with the negative risks (misuse of automatically generated executable code, and amplification of bias carried by the method pool or by the data), and we advocate concrete mitigations—sandboxed execution of synthesized programs, code verification before deployment, and filtering of harmful or biased operators from the method pool—which we also discuss in Appendix M.3. No specific release safeguards are required for this work: the paper presents a general-purpose symbolic program-synthesis substrate and does not release high-risk assets such as large generative models or scraped datasets. We use only standard public benchmarks (e.g., the Titanic dataset, hosted on Kaggle and cited with its URL) together with synthetic data generated by our own scripts, and we respect the corresponding terms of use, which permit academic and non-commercial use. This research involved no crowdsourcing, no human subjects, and no personally identifiable information, so no IRB approval or participant compensation was required. No new datasets or pre-trained weights are released; the accompanying code is provided anonymously for review.

REPRODUCIBILITY STATEMENT

All experiments reported in this paper can be reproduced from the artefacts referenced below. The implementation of AdaptoFlux, the gradient-free navigation probes (LayerGrow, GraphEvo, and CombinedTrainer), the method-pool definitions, the random-seed strategies, and the raw logs are released in the anonymous code repository accompanying this submission (`https://anonymous.4open.science/r/AdaptoFlux-CF54`). All experimental settings—data splits, method-pool configurations, hyperparameters, and attempt limits—are specified in Appendix C for the five experiment categories and are additionally hard-coded in the released code; Appendix G maps each experiment to its directory and output files, lists the software dependencies and the compute resources used (multi-core CPUs with 60 GB RAM and no GPU, with a 32-core server for the parallel-scalability study, and under one week of sequential wall-clock time to reproduce the main results), and provides a step-by-step reproduction guide. Statistical details supporting the main claims are reported as means and standard deviations over 10–100 independent runs in the appendix tables, and the controlled comparison between type-constrained and unconstrained search is analyzed per run in Appendix C.4. For the theoretical results, the derivations of the navigation probes and of the inference pipeline are given in Appendix B.3, the complexity analysis in Appendix B, and the loop–DAG correspondence (Proposition 1) in Appendix H.6.

REFERENCES

W. Ross Ashby. *Design for a Brain: The Origin of Adaptive Behaviour*. Springer Dordrecht, 2nd edition, 1960. doi: 10.1007/978-94-015-1320-3.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

Miles Cranmer. Interpretable machine learning for science with pysr and symbolicregression. jl. *arXiv preprint arXiv:2305.01582*, 2023.

Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *Machine Learning*, 110(4):801–856, 2021.

Will Cukierski. Titanic - machine learning from disaster. https://kaggle.com/competitions/titanic, 2012. Kaggle.

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 *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pp. 835–850, 2021. doi: 10.1145/3453483.3454080. URL https://doi.org/10.1145/3453483.3454080.

Artur d'Avila Garcez and Luis C Lamb. Neurosymbolic ai: The 3 rd wave. *Artificial Intelligence Review*, 56(11):12387–12406, 2023.

Yifan He and Ferrante Neri. Fitness landscape analysis of genetic programming search spaces with local optima networks. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, GECCO '23 Companion, pp. 2056–2063, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701207. doi: 10.1145/3583133.3596305. URL https://doi.org/10.1145/3583133.3596305.

John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

Mikel Landajuela, Brenden K Petersen, Sookyung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. Discovering symbolic policies with deep reinforcement learning. In *International Conference on Machine Learning*, pp. 5979–5989. PMLR, 2021.

Julian Miller and Andrew Turner. Cartesian genetic programming. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pp. 179–198, 2015. doi: 10.1145/2739482.2756571. URL https://dl.acm.org/doi/10.1145/2739482.2756571.

David J Montana. Strongly typed genetic programming. *Evolutionary computation*, 3(2):199–230, 1995.

Brenden K Petersen, Mikel Landajuela, T Nathan Mundhenk, Claudio P Santiago, Soo K Kim, and Joanne T Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. *arXiv preprint arXiv:1912.04871*, 2019.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Richard S Sutton, Andrew G Barto, and Andrew Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020. doi: 10.1126/sciadv.aay2631.

Haishuai Wang, Yang Gao, Xin Zheng, Peng Zhang, Jiajun Bu, and Philip S. Yu. Graph neural architecture search with large language models. *Science China Information Sciences*, 68(12), 2025. URL https://doi.org/10.1007/s11432-024-4539-1.

Norbert Wiener. Cybernetics. *Scientific American*, 179(5):14–19, 1948.

Xiao-Wen Yang, Jie-Jing Shao, Lan-Zhe Guo, Bo-Wen Zhang, Zhi Zhou, Lin-Han Jia, Wang-Zhou Dai, and Yu-Feng Li. Neuro-symbolic artificial intelligence: Towards improving the reasoning abilities of large language models, 2025. URL https://arxiv.org/abs/2508.13678.

Filippo Ziliotto, Tommaso Campari, Luciano Serafini, and Lamberto Ballan. Tango: training-free embodied ai agents for open-world tasks. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 24603–24613, 2025.

APPENDIX

A DETAILED RELATED WORK AND POSITIONING

Traditional Symbolic Regression (SR) methods (e.g., GP Koza (1994), PySR Cranmer (2023), AI Feynman Udrescu & Tegmark (2020)) excel at recovering static mappings ($y = f(x)$) but lack native mechanisms for **system state** or **state-dependent operations**, limiting their ability to model dynamic processes ($s_{t+1} = \Phi(s_t, x_t)$). Graph-based encodings such as Cartesian Genetic Programming (CGP) Miller & Turner (2015) partially address this by letting a node reuse the outputs of earlier nodes, but their function set is typically untyped and homogeneous, and state must be carried either by dedicated register or memory nodes with explicit feedback connections or by exposing it as additional graph inputs, rather than being kept in an implicit state space. Strongly typed genetic programming Montana (1995) addresses the typing half of this problem by restricting which node types may exchange data, but it inherits the same tree-level state routing and likewise provides no container for side-effectful black-box operators. AdaptoFlux addresses this by generalizing representations to **Generalized Data Flow Graphs (GDFG)**. Through explicitly typed edges and **Time-step Unrolling**, it natively supports state memory, filling the expression space gap for dynamic systems.

While methods like DreamCoder Ellis et al. (2021) mitigate combinatorial explosion via higher-order function abstraction and strict type systems, they primarily operate within pure functional paradigms where programs are represented as tree-structured lambda expressions, limiting native support for explicit stateful graph topologies. Modeling stateful interactions in such frameworks often requires complex explicit state threading. In contrast, AdaptoFlux constructs an expression space based on Generalized Data Flow Graphs (GDFGs) and a Method Pool. By natively accommodating side-effectful methods within the graph execution model, it allows state-dependent logic to emerge through implicit state mutations triggered by specific nodes, rather than requiring explicit state threading through topological edges. This architectural flexibility enables the framework to model dynamic interactions where state is maintained in an external or global context ('implicit space') accessed by specialized action functions. We note, however, that DreamCoder's principal contribution—the automatic discovery of reusable abstractions—is not matched by the present work: our knowledge-extraction probe remains prototype-stage (Section 4.2), so this comparison concerns representational affordances rather than demonstrated library-learning capability.

Inductive Logic Programming (ILP) is the closest symbolic relative of our constraint-based formulation. Popper Cropper & Morel (2021) learns definite programs from background knowledge and positive/negative examples under an explicitly declared constraint language, following a generate–test–constrain loop in which each failed hypothesis yields new constraints that prune the remaining hypothesis space. AdaptoFlux shares the underlying principle: background operators play the role of background knowledge, type signatures and topological rules play the role of mode declarations and language bias, and type-incompatible compositions are rejected statically, before evaluation, mirroring constraint-based pre-pruning. The two systems differ in what a hypothesis *is* and in how state is represented. Popper's hypothesis class consists of logical clauses with declarative semantics, so modeling a dynamic system requires threading state explicitly through additional arguments (e.g., a `next_state(S, A, S')` predicate): every mutation must be written down as a logical

relation, and all of it remains subject to logical inference. AdaptoFlux instead binds nodes to arbitrary black-box executable functions—including non-deterministic environment calls—and keeps mutable state in an implicit external state space that action nodes read from and write to as a side effect. State therefore need not appear as an argument on any edge, a single state variable can be consumed by several downstream action nodes through ordinary DAG fan-out without duplicating predicates, and the hypothesis class natively contains non-monotonic side effects that are not reducible to Horn clauses without loss of operational semantics. The trade-off is symmetric: Popper *learns* constraints from failed hypotheses, whereas AdaptoFlux's filters are currently declared rather than derived, and integrating failure-driven constraint generation into the metadata layer is a concrete direction for future work. The parallel is exact on both sides: type signatures and topological rules are space-defining, like an ILP language bias, whereas metadata filters act on the search side, like Popper's learned hypothesis constraints (Appendix H.4).

Neuro-symbolic AI Yang et al. (2025) and LLM-guided search Wang et al. (2025) often assume static I/O mappings and lack global validity checks for open-loop graph generation. AdaptoFlux provides a **deterministic, constraint-satisfying** backend. Its strict type system and topological rules ensure executability, offering a verifiable safety foundation missing in pure semantic models.

A further line of work optimises the search while leaving the representation fixed. Risk-seeking policy gradients over a discrete expression grammar learn a distribution over expressions Petersen et al. (2019), and the same machinery has been used to discover symbolic policies for control Landajuela et al. (2021); LLM-guided discovery instead spends its effort on proposal quality, pairing a pretrained language model with a systematic evaluator Romera-Paredes et al. (2024). In all three cases the operator set remains a homogeneous, purely functional grammar: the grammar constrains syntax, but there is no type system over heterogeneous operators, no place to register a side-effecting primitive, and state must enter the expression as an explicit argument. AdaptoFlux is therefore complementary rather than competing—it is the substrate such searches lack—and the probes used in this work are deliberately weak so that the representation's own contribution remains visible.

Cybernetics established feedback as a precondition for adaptive behaviour: Ashby's homeostat demonstrated that a system adapts through the organization of its feedback loops rather than through any property of its individual parts Wiener (1948); Ashby (1960). Symbolic program representations have no native counterpart to this loop. Tree- and DAG-based encodings are feed-forward by construction, and closed-loop behaviour must be reconstructed by threading state explicitly as an additional argument Koza (1994); Ellis et al. (2021). AdaptoFlux bridges this by employing **Time-step Unrolling** to transform the **static GDFG topology** into a **cross-step state machine**. This allows the system to formalize closed-loop logic where outputs influence subsequent inputs. Unlike LLM-based frameworks like TANGO Ziliotto et al. (2025) that lack evolutionary capabilities, AdaptoFlux uses metadata-driven mechanisms to explicitly encode system boundaries. This supports traceable, type-constrained state transitions and dynamic primitive evolution, with the loss function acting as a "negative feedback signal" for error-driven regulation.

In summary, AdaptoFlux is not a specific control solver but fills the expression space gap for the structured modeling of discrete dynamic systems. Its GDFG architecture provides a unified foundation for building interpretable, evolvable, and state-feedback-capable intelligent systems. For concreteness, the capability we claim as absent from the cited systems is a single statically type-checked container in which pure functions and side-effectful black-box actions coexist while the execution graph remains acyclic. CGP already supplies the acyclic graph and native variable reuse, but its function set is untyped; strongly typed GP Montana (1995) adds the missing type discipline, but only over program trees; neither admits side-effectful black-box operators as first-class nodes. DreamCoder remains within pure functional abstraction, and ILP requires each state mutation to be expressed as a logical relation.

B Complete Derivation of AdaptoFlux Inference and Training Complexity

This appendix provides a detailed derivation of the time and space complexity of the AdaptoFlux architecture during single inference, task-parallel inference, and under three main training strategies (LayerGrow, GraphEvo, CombinedTrainer).

B.1 Symbol Definitions and Basic Assumptions

To ensure consistency in the derivation, we first uniformly define general symbols, complexity constants, and basic assumptions.

B.1.1 General Symbol Table

Table 1: General Symbols

Symbol	Meaning
N	Inference batch size (number of samples)
V	Number of computation graph nodes (excluding `root` and `collapse`)
E	Number of computation graph edges (explicit data dependencies)
D_{in}	Input feature dimension
D_{out}	Output feature dimension
W	Average graph width (maximum number of concurrent nodes)
L_{crit}	Critical path length (longest data dependency chain)
P	Number of parallel scheduling threads/processes

Note: The above parameters are highly dependent on specific task configurations, data scales, and graph topologies. The framework design does not make fixed prior assumptions about them.

B.1.2 Complexity Constant Dependency Characteristics

Table 2: Complexity Constants and Dependencies

Constant	Meaning and Dependency
C_{copy}	Python object reference extraction overhead. Fixed at $O(1)$ (nanosecond-level memory pointer operations), independent of graph scale.
C_{sync}	Global lock acquisition/release and queue synchronization overhead. Fixed at $O(1)$ (microsecond-level OS primitives), potentially becoming a serial bottleneck under high concurrency.
$\bar{C}_{\text{op}}$	**Average method computational complexity** (where op denotes the underlying **operation**). **Strongly depends on method pool composition**: Pure Python loops are constrained by interpreter overhead; C/Numba extensions can release the GIL to achieve true parallelism. This constant directly determines the absolute performance boundary of the framework.
ρ_{lock}	Global lock contention factor. Dynamic range $[1/P, 1]$, actual value depends on graph width W, data flow topology, and thread scheduling strategy.

B.1.3 Basic Assumptions

Assumption 1 [Typical Deployment Scenario]
In typical deployment scenarios, the following conditions are met:

1. $N \gg V, E$ (Batch size is much larger than graph scale)
2. $D_{\text{in}}, D_{\text{out}}$ are constants or independent of N
3. $C_{\text{copy}}, C_{\text{sync}}$ are $O(1)$ constants
4. Method library size M is a bounded constant
5. Search hyperparameters (L, A, C, R, S, f) are bounded constants

Assumption 2 [Operator Implementation]
Operator implementations satisfy:

1. Pure Python operators: $\bar{C}_{\text{op}}$ includes significant interpreter overhead
2. I/O or C extension operators: Can release GIL, supporting true parallelism

B.2 Inference Phase Complexity Analysis

B.2.1 Single Inference: Four-Stage Decomposition

Definition [Single Inference]
Single inference refers to the process where the model, in deployment state, performs forward propagation on a given input batch to generate prediction results. In the AdaptoFlux architecture, the inference process is modeled as data flow execution on a Directed Acyclic Graph (DAG).

Stage 1: Input Normalization and Preprocessing The system first converts input data into an internal sample list structure. The complexity of this step depends on the input data type:

- **Dense Numerical Matrix**: Involves memory copying and type checking, with complexity $O(N \cdot D_{\text{in}})$
- **Complex Objects (dtype=object)**: Main overhead lies in reference counting and list construction, still maintaining a linear relationship $O(N \cdot D_{\text{in}})$

Stage 2: Graph Topological Sorting To ensure correct node execution order dependencies, the system performs topological sorting on the computation graph. This operation depends only on the graph structure and is independent of the sample size N:

$$T_{\text{topo}} = O(V + E) \tag{6}$$

Stage 3: Node Layer-by-Layer Execution This is the core computational part of the inference process. The system traverses each node $v \in \mathcal{V}$ in topological order. For each node, the following sub-operations are performed:

1. **Edge Data Extraction**: Based on incoming edge configurations, feature data for N samples is extracted from predecessor node outputs. The total number of incoming edges in the graph is E, and the data extraction complexity is:
$$T_{\text{extract}} = O(E \cdot N \cdot C_{\text{copy}}) \tag{7}$$
2. **Operator Computation**: Executes the computation method bound to the node. Let the single-sample computational complexity of node v be $C_{\text{op}}(v)$. If the method supports vectorization, $C_{\text{op}}(v)$ approaches the efficiency of underlying kernel optimization; if it degrades to sample-by-sample loops, $C_{\text{op}}(v)$ includes significant Python interpreter overhead.
The total complexity of the node execution phase is:
$$T_{\text{compute}} = O\left(N \cdot \sum_{v \in \mathcal{V}} C_{\text{op}}(v)\right) = O(N \cdot V \cdot \bar{C}_{\text{op}}) \tag{8}$$

Stage 4: Result Aggregation and Collapse The final stage of inference maps the outputs of specified nodes to global coordinates and performs collapse processing. This process requires traversing the final feature vectors of N samples:

$$T_{\text{collapse}} = O(N \cdot D_{\text{out}}) \tag{9}$$

Total Complexity Derivation Combining the four stages, the total time complexity of AdaptoFlux single inference is:

$$\begin{aligned} T_{\text{infer}}(N, V, E) &= T_{\text{input}} + T_{\text{topo}} + T_{\text{extract}} + T_{\text{compute}} + T_{\text{collapse}} \\ &= O(N \cdot D_{\text{in}}) + O(V + E) + O(E \cdot N \cdot C_{\text{copy}}) \\ &\quad + O(N \cdot V \cdot \bar{C}_{\text{op}}) + O(N \cdot D_{\text{out}}) \\ &= O\left(N \cdot (D_{\text{in}} + E \cdot C_{\text{copy}} + V \cdot \bar{C}_{\text{op}} + D_{\text{out}}) + V + E\right) \end{aligned} \tag{10}$$

Dominant Term Simplification
In actual deployment, typically $N \gg V, E$ and $D_{\text{in}}, D_{\text{out}}$ are constants. At this point, inference time is primarily dominated by operator computation:

$$T_{\text{infer}} \approx O(N \cdot V \cdot \bar{C}_{\text{op}}) \tag{11}$$

The framework scheduling overhead of AdaptoFlux grows **linearly** with model structural complexity V, but the total time strongly depends on the implementation efficiency of specific operators (e.g., whether vectorized, whether GIL is released).

Space Complexity and Implementation Residue *Remark [Full Intermediate Result Cache]* The current implementation retains a full intermediate result caching strategy (`node_outputs` stores all node outputs), with space complexity:

$$S_{\text{cache}} = O(N \cdot V \cdot S_{\text{data}}) \tag{12}$$

where S_{data} is the average size of intermediate features.

This mechanism is a functional residue from early architecture design: initially intended to support optimization algorithms requiring access to intermediate activations (such as architecture search), but in the current pure inference version, these intermediate data are not actually utilized nor returned externally. This leads to unnecessary memory occupation. Future plans include introducing a release strategy, i.e., releasing a node's output after all its successor computations are completed.

B.2.2 Task-Parallel Inference: Six-Step Execution Flow Decomposition

In task-parallel scenarios, the following extended symbols are introduced:

- P: Number of thread pool worker threads (`num_workers`)
- C_{sync}: Synchronization overhead for global lock acquisition/release, dictionary updates, and queue operations
- $L_{\text{crit}} = \sum_{v \in \mathcal{P}_{\text{crit}}} C_{\text{op}}(v)$: Cumulative computational cost of the Critical Path
- $\rho_{\text{lock}} \in [1, P]$: Global lock contention factor
  - Low load: $\rho_{\text{lock}} \approx 1/P$ (ideal parallelism)
  - High concurrency contention: $\rho_{\text{lock}} \to 1$ (fully serial)

Based on the actual execution trajectory of `infer_with_task_parallel`, the time complexity can be derived step-by-step:

Step 1: Input Normalization and Graph Initialization (Main Thread) Data format conversion and construction of `in_degree_remaining` and `ready_queue` depend only on N and graph structure:

$$T_1 = O(N \cdot D_{\text{in}} + V + E) \tag{13}$$

Step 2: Main Thread Dynamic Scheduler The `while` loop executes V times `queue.get` and `executor.submit`:

$$T_2 = O(V) \tag{14}$$

This part is completely serial.

Step 3: Lock-Protected Data Extraction (Serialization Bottleneck) The first `with lock:` block in the code wraps incoming edge traversal and feature extraction:

```
extracted = [sample_output[output_idx] for sample_output in src_output]
```

The total number of incoming edges in the graph is E, and each edge requires traversing N samples to generate a reference list. The total cost of this operation is $O(E \cdot N \cdot C_{\text{copy}})$. Since it is protected by a global mutex lock, **all threads execute this phase serially**. Considering lock contention, the actual time consumption is:

$$T_3 = O(E \cdot N \cdot C_{\text{copy}} \cdot \rho_{\text{lock}}) \tag{15}$$

Step 4: Lock-Free Operator Computation (Parallel Acceleration Zone) After data extraction, the lock is released, and threads independently execute `func` calls. The total computational workload is:

$$W_{\text{comp}} = N \cdot \sum_{v \in \mathcal{V}} C_{\text{op}}(v) \tag{16}$$

According to DAG scheduling theory (Brent's Theorem), the actual time consumption under P threads is limited by the critical path and computational share:

$$T_4 = O\left(N \cdot \max\left(L_{\text{crit}}, \frac{1}{P}\sum_{v \in \mathcal{V}} C_{\text{op}}(v)\right)\right) \tag{17}$$

Note: If an operator is marked `vectorized=True`, $C_{\text{op}}(v)$ is significantly reduced, but large N may trigger CPU cache and memory bandwidth contention. Moreover, due to framework characteristics, vectorization generally struggles to offset scheduling overhead.

Step 5: Lock-Protected Status Update and Successor Triggering The second `with lock:` block executes `node_outputs` writing and in-degree decrementing. The graph triggers V lock acquisitions and E queue pushes in total:

$$T_5 = O(V \cdot C_{\text{sync}} + E \cdot C_{\text{sync}}) \tag{18}$$

This part is also serialized, but $C_{\text{sync}} \ll C_{\text{copy}}$, so it is usually negligible.

Step 6: Result Aggregation (Collapse) (Main Thread) After `f.result()` blocking ends, the main thread sequentially collects `collapse` incoming edge data, sorts, and calls the collapse function:

$$T_6 = O(N \cdot D_{\text{out}}) \tag{19}$$

Total Complexity Formula Combining the six stages, the precise time complexity of task-parallel inference is:

$$\begin{aligned} T_{\text{parallel}}(N, V, E, P) &= \sum_{i=1}^{6} T_i \\ &= O\Big(N \cdot D_{\text{in}} + V + E \\ &\quad + E \cdot N \cdot C_{\text{copy}} \cdot \rho_{\text{lock}} \\ &\quad + N \cdot \max\left(L_{\text{crit}}, \frac{1}{P}\sum_{v \in \mathcal{V}} C_{\text{op}}(v)\right) \\ &\quad + V \cdot C_{\text{sync}} + N \cdot D_{\text{out}}\Big) \end{aligned} \tag{20}$$

Typical Scenario Simplification
In typical deployment scenarios ($N \gg V, E$, and $C_{\text{copy}}, C_{\text{sync}}$ are constants), the dominant terms simplify to:

$$T_{\text{parallel}} \approx O\left(N \cdot \max\left(L_{\text{crit}}, \frac{1}{P}\sum_{v \in \mathcal{V}} C_{\text{op}}(v)\right) + E \cdot N \cdot C_{\text{copy}} \cdot \rho_{\text{lock}} + N \cdot D_{\text{out}}\right) \tag{21}$$

B.2.3 Engineering Constraint Analysis

Remark [Lock Granularity Defect]
The current implementation places data extraction ($O(E \cdot N)$) within the global lock, causing this part to be unparallelizable. When E is large or graph width is high, $\rho_{\text{lock}} \to 1$, and increasing P may reduce throughput due to lock contention.

Remark [GIL Limitation]
`ThreadPoolExecutor` is constrained by the CPython Global Interpreter Lock. Pure Python intensive operators can only achieve concurrency, not true parallelism.

Amdahl's Law Analysis: Actual speedup satisfies

$$S(P) \leq \frac{1}{\alpha + (1-\alpha)/P} \tag{22}$$

where α is the proportion of serial execution within the lock.

Remark [Coupling of Vectorization and Thread Count]
If nodes have enabled vectorization, a single thread already approaches the hardware computational upper limit. At this point, $P > 1$ will cause resource oversubscription, leading to L1/L2 cache thrashing and memory bandwidth saturation.

Remark [Vectorization]
Due to the framework's weak support for vectorization and the need for significant scheduling overhead, operators with vectorization enabled often fail to bring performance improvements and may even degrade performance.

Optimal Thread Count Estimation:

$$P^* \approx \frac{\text{Hardware Physical Core Count}}{\text{Average Vectorization Parallelism}} \tag{23}$$

B.3 Detailed Analysis of Navigation Probe (Training Strategy) Complexity

Note: Main Text §3.5 states the final complexity results and their interpretation; this appendix provides the complete derivations, symbol tables, and constant-factor discussions.

B.3.1 LayerGrow Strategy: Layered Generate-Evaluate-Rollback

Table 3: LayerGrow Symbols

Symbol	Meaning
L	Maximum added layers (`max_layers`)
A	Maximum attempts per layer (`max_attempts`)
N	Number of input samples for evaluation
V_i	Number of graph nodes at iteration i, $V_i = V_0 + i \cdot k$
V_0	Initial number of nodes
k	Average new nodes per layer
$T_{\text{infer}}(N, V_i)$	Single inference complexity, $O(N \cdot V_i \cdot \bar{C}_{\text{op}})$
C_{gen}	Candidate generation complexity
C_{struct}	Graph structure modification (add/remove layer) complexity, $O(k)$
C_{copy}	Model snapshot deep copy complexity, $O(V_i + E_i + M_i)$
C_{eval}	Evaluation function computation complexity (loss/accuracy)
D	Number of edges pointing to `collapse` node (number of output slots)
M	Total number of methods registered in the method library
I_{avg}	Average number of input slots per method
$\bar{C}_{\text{op}}$	Average operator computational complexity (including vectorization efficiency factor)
p_{accept}	Candidate acceptance probability

Symbol Definition

Precise Derivation of Candidate Generation Complexity C_{gen} `process_random_method` is responsible for randomly matching compatible methods for each output slot and grouping them. Its core steps include:

1. **Grouping D slots by data type**: $O(D)$
2. **Constructing "Type → Method Input" reverse index**: Traversing I_{avg} inputs of M methods, $O(M \cdot I_{\text{avg}})$

3. **Matching compatible method inputs for each slot**: Worst-case traversal of all method inputs, $O(D \cdot M \cdot I_{\text{avg}})$
4. **Grouping by method call signature and post-processing**: $O(D)$

Therefore, the worst-case complexity of candidate generation is:

$$C_{\text{gen}} = O(D \cdot M \cdot I_{\text{avg}}) \tag{24}$$

Clearly, C_{gen} grows linearly with method library size M and input complexity I_{avg}, and is usually small compared to the inference complexity $T_{\text{infer}}(N, V_i)$ of the evaluation phase. In actual implementation, although the constant factor of C_{gen} is large, since T_{infer} involves complete forward propagation, typically:

$$C_{\text{gen}} \ll T_{\text{infer}}(N, V_i) \tag{25}$$

i.e., **candidate generation overhead is negligible, and the evaluation phase dominates the time consumption of a single attempt**.

Core Overhead of a Single Attempt Each candidate attempt (inside the `attempt` loop) contains the following sequential operations:

1. **Candidate Generation**: Call `process_random_method()`, complexity $O(C_{\text{gen}})$
2. **Graph Structure Modification**: Call `append_nx_layer()` to temporarily add a new layer, complexity $O(C_{\text{struct}})$
3. **Model Inference**: Execute one complete forward propagation to get model output, complexity $T_{\text{infer}}(N, V_i)$
4. **Evaluation Function Calculation**: Call loss/accuracy evaluator, complexity C_{eval}
5. **Decision and Rollback**: If candidate is rejected, call `remove_last_nx_layer()` to restore graph structure, complexity $O(C_{\text{struct}})$

Single Attempt Complexity:

$$\begin{aligned} T_{\text{attempt}}(i) &= O(C_{\text{gen}} + C_{\text{struct}} + T_{\text{infer}}(N, V_i) + C_{\text{eval}}) \\ &\approx O(T_{\text{infer}}(N, V_i)) \quad (\text{since } T_{\text{infer}} \gg C_{\text{gen}}, C_{\text{struct}}, C_{\text{eval}}) \\ &= O(N \cdot V_i \cdot \bar{C}_{\text{op}}) \end{aligned} \tag{26}$$

Configurability of Evaluation Function *Remark [Hardcoded Evaluation]*
In the current implementation, `_evaluate_loss` and `_evaluate_accuracy` are hardcoded to execute by default, which is technical debt from early design. Subsequent optimizers do not necessarily need to compute both loss and accuracy simultaneously, and some scenarios (such as reinforcement learning) may not require accuracy evaluation at all. Benefiting from AdaptoFlux's plugin-style evaluator design, theoretically supports:

1. **Skipping Redundant Evaluation**: If only loss signal is needed, accuracy calculation can be disabled, reducing C_{eval} overhead by 50%
2. **Environment Signal Reuse**: In scenarios like reinforcement learning, reward signals can be obtained directly from the environment, where $C_{\text{eval}} \approx O(1)$
3. **Delayed Evaluation**: Execute full evaluation only after candidate acceptance, calculating lightweight proxy metrics only during the attempt phase

Expected Overhead of Single Layer Growth For the i-th layer ($0 \leq i < L$), the algorithm attempts at most A times. Let the acceptance probability be p_{accept} (dependent on data and candidate space), then the expected number of attempts is $1/p_{\text{accept}}$. In the worst case ($p_{\text{accept}} \to 0$), full A attempts are executed:

$$T_{\text{layer}}(i) = O(A \cdot T_{\text{infer}}(N, V_i)) = O(A \cdot N \cdot (V_0 + i \cdot k) \cdot \bar{C}_{\text{op}}) \tag{27}$$

Total Training Complexity Derivation Summing over all L layers and considering graph scale grows linearly with layers:

$$\begin{aligned} T_{\text{LayerGrow}} &= \sum_{i=0}^{L-1} T_{\text{layer}}(i) \\ &= O\left(A \cdot N \cdot \bar{C}_{\text{op}} \cdot \sum_{i=0}^{L-1}(V_0 + i \cdot k)\right) \\ &= O\left(A \cdot N \cdot \bar{C}_{\text{op}} \cdot (L \cdot V_0 + k \cdot \frac{L(L-1)}{2})\right) \end{aligned} \tag{28}$$

Dominant Term Analysis: When the initial graph is small ($V_0 \ll k \cdot L$) and k is constant, the dominant term is quadratic:

$$T_{\text{LayerGrow}} = O(A \cdot N \cdot \bar{C}_{\text{op}} \cdot k \cdot L^2) \tag{29}$$

i.e., quadratic complexity with respect to maximum layers L, and linear with respect to attempt count A, evaluation batch N, and operator complexity $\bar{C}_{\text{op}}$.

Impact of Sparsely Triggered Operations

- **Best Model Snapshot**: Executed `copy.deepcopy()` only when a better model is found, frequency usually much lower than $L \cdot A$. Let trigger count be S ($S \ll L \cdot A$), then extra overhead is $O(S \cdot (V_{\text{best}} + E_{\text{best}} + M_{\text{best}}))$, usually negligible.
- **Rollback Operation**: When `on_retry_exhausted="rollback"`, multiple layers may be rolled back continuously. In the worst case, L layers are rolled back, but total rollback count is constrained by global attempt limit `max_total_attempts`, not changing the dominant complexity order.
- **Early Stopping Mechanism**: If `enable_early_stop=True` and model converges quickly, actual executed layers $L_{\text{actual}} \ll L$, significantly reducing actual runtime, but not changing worst-case complexity.

Space Complexity

- **Graph Structure Storage**: $O(V_L + E_L)$, where $V_L = V_0 + L \cdot k$, grows linearly with layers
- **Best Model Snapshot**: $O(V_{\text{best}} + E_{\text{best}} + M_{\text{best}})$, proportional to best model scale
- **Attempt History Record**: $O(L \cdot A)$ metadata records, each $O(1)$, total overhead $O(L \cdot A)$

Total Space Complexity:

$$S_{\text{LayerGrow}} = O(V_0 + L \cdot k + E_0 + L \cdot e + M_{\text{best}}) \tag{30}$$

where e is the average number of new edges per layer.

B.3.2 GraphEvo Strategy: Four-Stage Closed-Loop Optimization

Symbol Definition

Phase 1: Diverse Initialization This phase generates I random initial graphs and selects the best one:

1. For each candidate, randomly add L_{init} layers: each layer requires calling `process_random_method` ($O(D \cdot M \cdot I_{\text{avg}})$) and `append_nx_layer` ($O(k)$)
2. Perform one complete evaluation for each candidate: $T_{\text{infer}}(N, V_0 + L_{\text{init}} \cdot k)$
3. Select best candidate: $O(I)$

Initialization Phase Complexity:

$$\begin{aligned} T_{\text{init}} &= O\left(I \cdot [L_{\text{init}} \cdot (D \cdot M \cdot I_{\text{avg}} + k) + T_{\text{infer}}(N, V_0 + L_{\text{init}} \cdot k)]\right) \\ &\approx O(I \cdot L_{\text{init}} \cdot N \cdot V_{\text{init}} \cdot \bar{C}_{\text{op}}) \quad (\text{since } T_{\text{infer}} \text{ dominates}) \end{aligned} \tag{31}$$

where $V_{\text{init}} = V_0 + L_{\text{init}} \cdot k$.

Table 4: GraphEvo Symbols

Symbol	Meaning
N	Number of input samples for evaluation (mini-batch)
V_i, E_i	Graph nodes and edges at iteration i, $V_i = V_0 + i \cdot k$
C	Maximum evolution cycles (`max_evo_cycles`)
I	Number of diverse initialization candidate models (`num_initial_models`)
L_{init}	Random layers for each initialization candidate (`max_init_layers`)
R	Maximum refinement steps per cycle (`max_refinement_steps`)
S	Number of snapshots required to trigger evolution (`evolution_trigger_count`)
f	Snapshot recording frequency (`evolution_sampling_frequency`)
M	Total methods registered in method pool
I_{avg}	Average input slots per method
$\bar{C}_{\text{op}}$	Average operator computational complexity
V_{sub}	Average nodes of subgraph processed in evolution phase

Phase 2: Node-wise Refinement This phase performs local search optimization within C evolution cycles. It must be clarified that **GraphEvo's refinement process does not change the graph topology**, only greedily replacing methods bound to existing nodes, so graph scale V and E remain constant at initialized values V_{init} and E_{init} during refinement.

At most R refinement steps are executed per cycle

Each refinement step includes:

1. Select optimizable node: $O(V_{\text{init}})$ (traverse non-frozen nodes)
2. Get compatible method list: $O(M \cdot I_{\text{avg}})$ (type matching), can be reduced to $O(1)$ if hash index is enabled
3. Attempt method replacement: $O(1)$ (only update node method pointer, no add/delete edges)
4. Evaluate new structure: $2 \cdot T_{\text{infer}}(N, V_{\text{init}})$ (loss + accuracy)
5. Decision and rollback: $O(1)$ (restore method pointer)

Single Step Refinement Complexity:

$$T_{\text{refine}}^{(\text{step})} = O(M \cdot I_{\text{avg}} + T_{\text{infer}}(N, V_{\text{init}})) \approx O(T_{\text{infer}}(N, V_{\text{init}})) \tag{32}$$

(since $T_{\text{infer}} \gg M \cdot I_{\text{avg}}$)

Single Cycle Refinement Complexity: Since topology is fixed, cumulative complexity of R refinement steps is:

$$\begin{aligned} T_{\text{refine}}^{(\text{cycle})} &= O\left(\sum_{j=1}^{R} T_{\text{infer}}(N, V_{\text{init}})\right) \\ &= O\left(R \cdot N \cdot V_{\text{init}} \cdot \bar{C}_{\text{op}}\right) \end{aligned} \tag{33}$$

Total Refinement Complexity: Summing over all C cycles, graph scale remains unchanged:

$$T_{\text{refine}}^{\text{total}} = O\left(C \cdot R \cdot N \cdot V_{\text{init}} \cdot \bar{C}_{\text{op}}\right) \tag{34}$$

i.e., refinement phase is linear with respect to cycle count C and step count R, reflecting the significant pruning effect of the topology-fixed strategy on the combinatorial search space.

Phase 3: Method Pool Evolution (Optional) This phase abstracts new methods based on historical snapshots, triggered sparsely:

1. **Snapshot Recording**: Record once every f cycles, total $\lfloor C/f \rfloor$ times, each deep copy $O(V + E + M)$

2. **Evolution Trigger**: Trigger after accumulating S snapshots, total trigger count $T_{\text{evo}} = \lfloor \frac{C}{f \cdot S} \rfloor$
3. **Single Evolution Execution**:
   - Subgraph extraction and alignment: $O(S \cdot V \cdot E)$
   - Graph isomorphism clustering (if enabled): $O(S^2 \cdot V^3)$ (worst case, using VF2 algorithm)
   - New method abstraction and registration: $O(V_{\text{sub}} \cdot M)$

Total Evolution Phase Complexity:

$$\begin{aligned} T_{\text{evolve}} &= O\left(\frac{C}{f} \cdot (V + E + M) + \frac{C}{f \cdot S} \cdot (S^2 \cdot V^3 + V_{\text{sub}} \cdot M)\right) \\ &\approx O\left(\frac{C \cdot S \cdot V^3}{f}\right) \quad \text{(dominant term)} \end{aligned} \tag{35}$$

Cubic complexity with respect to graph scale V, but low trigger frequency ($S \geq 3$, $f \geq 1$), actual overhead is controllable.

Phase 4: Modular Compression (Design Residue, Optional Post-processing) *Remark [Technical Debt Note]*
The modular compression function (`_phase_modular_compression`) in current code is a residue from early architecture design technical debt:

- **Not involved in main training loop**: Compression is executed only as a one-time post-processing after training ends
- **Function deprecated**: Disabled by default (`enable_compression=False`), and numerical compression mode marked as `UNFINISHED`
- **Configurable extension point**: Interface retained to allow users to customize compression algorithms

If user explicitly enables symbolic compression and matches K_{comp} subgraphs, its single execution complexity is:

$$T_{\text{compress}} = O\left(K_{\text{comp}} \cdot (V_{\text{sub}} \cdot E_{\text{sub}} + N \cdot V_{\text{sub}}) + V + E\right) \tag{36}$$

Since $K_{\text{comp}} \ll V$ and executed only once, T_{compress} impact on total training time is negligible.

Total Training Complexity Combining the first three phases (ignoring optional compression), the worst-case time complexity of GraphEvo is:

$$\begin{aligned} T_{\text{GraphEvo}} = O(&\underbrace{I \cdot L_{\text{init}} \cdot N \cdot V_{\text{init}} \cdot \bar{C}_{\text{op}}}_{\text{Initialization}} + \underbrace{C \cdot R \cdot N \cdot V_{\text{init}} \cdot \bar{C}_{\text{op}}}_{\text{Refinement}} \\ &+ \underbrace{\frac{C \cdot S \cdot V_{\text{init}}^3}{f}}_{\text{Evolution}}) \end{aligned} \tag{37}$$

Typical Configuration Simplification: In typical configurations ($I, L_{\text{init}}, R, S, f$ are bounded constants, $V \ll N$), the dominant term is $O(N \cdot V_{\text{init}} \cdot \bar{C}_{\text{op}} \cdot (I \cdot L_{\text{init}} + C \cdot R))$ from initialization and refinement phases. If evolution is enabled, graph scale is large, and graph isomorphism clustering is enabled, then $O(C \cdot V_{\text{init}}^3)$ may become a bottleneck.

Impact of Sparse Triggering and Early Stopping

- **Early Stopping Mechanism**: If `enable_early_stop=True` and model converges quickly, actual cycles $C_{\text{actual}} \ll C$, reducing C term to C_{actual}, significantly reducing runtime

- **Evolution Sparse Triggering**: Evolution phase triggers only after accumulating S snapshots, and $S \geq 3$, $f \geq 1$, so actual trigger count $T_{\text{evo}} \leq C/3$, avoiding high-overhead graph isomorphism clustering every cycle
- **Refinement Step Limit**: `max_total_refinement_attempts` can globally limit evaluation count, preventing computational explosion due to excessive R

Space Complexity

- **Graph Structure Storage**: $O(V + E)$, grows linearly with refinement
- **Method Library Storage**: $O(M \cdot I_{\text{avg}})$
- **Snapshot Cache**: $O(S \cdot (V + E + M))$, S is trigger threshold, usually $S \ll C$
- **Intermediate Result Cache**: $O(N \cdot V \cdot S_{\text{data}})$ (if enabled, technical debt)

Total Space Complexity:

$$S_{\text{GraphEvo}} = O(S \cdot (V + E + M) + N \cdot V \cdot S_{\text{data}}) \tag{38}$$

If intermediate cache is disabled and S is bounded, space overhead is linear with respect to V.

B.3.3 CombinedTrainer Strategy: Closed-Loop Combinatorial Optimization

Table 5: CombinedTrainer Extended Symbols

Symbol	Meaning
C_{outer}	Outer evolution cycles (`num_evolution_cycles`)
L, A	LayerGrow max layers and attempts per layer
C_{ge}, R	GraphEvo internal cycles and refinement steps per cycle
G, T_{ga}	Genetic algorithm population size and evolution generations (if enabled)
f_{gen}	Genetic screening trigger interval (`genetic_interval`)
V_{new}	New nodes added per layer growth, $V_{\text{new}} = L \cdot k$
V_{old}	Historical node count
V_{max}	Final graph scale, $V_{\text{max}} = V_0 + C_{\text{outer}} \cdot L \cdot k$

Extended Symbol Definition

Sub-Trainer Complexity Review

- **LayerGrow**: $T_{\text{LG}} = O(A \cdot N \cdot \bar{C}_{\text{op}} \cdot k \cdot L^2)$
- **GraphEvo**: $T_{\text{GE}} = O(C_{\text{ge}} \cdot R \cdot N \cdot V_{\text{init}} \cdot \bar{C}_{\text{op}} + \frac{C_{\text{ge}} \cdot S \cdot V_{\text{init}}^3}{f})$
- **Genetic Screening**: $T_{\text{GA}} = O(G \cdot T_{\text{ga}} \cdot L_{\text{ga}} \cdot A_{\text{ga}} \cdot N \cdot V \cdot \bar{C}_{\text{op}})$

CombinedTrainer Execution Flow and Complexity Decomposition **1. Initial Genetic Screening (Optional)**
If `genetic_mode` is `"once"` or `"periodic"`, execute genetic screening once before the loop:

$$T_{\text{init_ga}} = O(G \cdot T_{\text{ga}} \cdot L_{\text{ga}} \cdot A_{\text{ga}} \cdot N \cdot V_0 \cdot \bar{C}_{\text{op}}) \tag{39}$$

This step is executed only once, usually $G, T_{\text{ga}}, L_{\text{ga}}, A_{\text{ga}}$ are bounded constants, so $T_{\text{init_ga}} = O(N \cdot V_0 \cdot \bar{C}_{\text{op}})$, negligible.

2. Outer Evolution Loop (C_{outer} times)
Each loop serially executes LayerGrow and GraphEvo:

- **LayerGrow Phase**: Extend L layers starting from current graph, complexity $T_{\text{LG}}^{(i)} = O(A \cdot N \cdot \bar{C}_{\text{op}} \cdot k \cdot L^2)$

- **GraphEvo Phase**: Execute refinement starting from LayerGrow output. **Key Constraint**: This phase only performs node method replacement, graph topology $V_{\text{current}}^{(i)}$ remains unchanged. Therefore its complexity is linear with respect to cycles C_{ge} and steps R. If `refine_only_new_layers` is enabled, search space is further restricted to new nodes ($V_{\text{new}} = L \cdot k$):

$$T_{\text{GE}}^{(i)} = \begin{cases} O(C_{\text{ge}} \cdot R \cdot N \cdot V_{\text{current}}^{(i)} \cdot \bar{C}_{\text{op}} + \frac{C_{\text{ge}} \cdot S \cdot (V_{\text{current}}^{(i)})^3}{f}) & \text{Full Graph Refinement} \\ O(C_{\text{ge}} \cdot R \cdot N \cdot V_{\text{new}} \cdot \bar{C}_{\text{op}} + \frac{C_{\text{ge}} \cdot S \cdot V_{\text{new}}^3}{f}) & \text{New Nodes Only} \end{cases} \tag{40}$$

Single Cycle Complexity: $T_{\text{cycle}}^{(i)} = T_{\text{LG}}^{(i)} + T_{\text{GE}}^{(i)}$

3. Periodic Genetic Screening (Optional)
If `genetic_mode == "periodic"`, execute genetic screening every f_{gen} rounds, total trigger count $\lfloor C_{\text{outer}}/f_{\text{gen}} \rfloor$:

$$T_{\text{periodic_ga}} = O\left(\frac{C_{\text{outer}}}{f_{\text{gen}}} \cdot G \cdot T_{\text{ga}} \cdot L_{\text{ga}} \cdot A_{\text{ga}} \cdot N \cdot V_{\text{mid}} \cdot \bar{C}_{\text{op}}\right) \tag{41}$$

where V_{mid} is the average graph scale at trigger time.

Total Training Complexity Combining the above phases, the worst-case time complexity of CombinedTrainer is:

$$\begin{aligned} T_{\text{Combined}} = O(& \underbrace{C_{\text{outer}} \cdot A \cdot N \cdot \bar{C}_{\text{op}} \cdot k \cdot L^2}_{\text{LayerGrow Accumulation}} \\ & + \underbrace{C_{\text{outer}} \cdot \left(C_{\text{ge}} \cdot R \cdot N \cdot V_{\max} \cdot \bar{C}_{\text{op}} + \frac{C_{\text{ge}} \cdot S \cdot V_{\max}^3}{f}\right)}_{\text{GraphEvo Accumulation (Linear Refinement)}} \\ & + \underbrace{\frac{C_{\text{outer}}}{f_{\text{gen}}} \cdot G \cdot T_{\text{ga}} \cdot L_{\text{ga}} \cdot A_{\text{ga}} \cdot N \cdot V_{\text{mid}} \cdot \bar{C}_{\text{op}}}_{\text{Periodic Genetic}}) \end{aligned} \tag{42}$$

Typical Configuration Simplification: In typical configurations ($A, L, C_{\text{ge}}, R, S, f, G, T_{\text{ga}}, f_{\text{gen}}$ are bounded constants, $V \ll N$):

- If genetic screening is disabled: $T_{\text{Combined}} = O(C_{\text{outer}} \cdot N \cdot \bar{C}_{\text{op}} \cdot (A \cdot k \cdot L^2 + R \cdot V_{\max}))$
- If genetic is enabled and f_{gen} is large: Genetic term is lower-order, dominant terms remain growth and refinement phases
- If `refine_only_new_layers` is enabled: GraphEvo's V^3 term reduces to $(L \cdot k)^3$, $V_{\max}$ replaced by V_{new}, significantly reducing overhead at high graph scales

Impact of Sparse Triggering and Early Stopping

- **Global Early Stopping**: If `enable_early_stop=True` and model converges quickly, actual cycles $C_{\text{actual}} \ll C_{\text{outer}}$, reducing all C_{outer} related terms to C_{actual}, overall complexity decreases linearly
- **Genetic Sparse Triggering**: Genetic screening triggers only at f_{gen} intervals, and G, T_{ga} are usually small, so $T_{\text{periodic_ga}}$ usually does not dominate total complexity
- **New Node Refinement Optimization**: `refine_only_new_layers` reduces GraphEvo's search space from $O(V^3)$ to $O((L \cdot k)^3)$, and since topology is fixed, cycle complexity strictly reduces from $O(C_{\text{ge}}^2 R^2)$ to $O(C_{\text{ge}} R)$, significant effect when C_{outer} is large

Space Complexity

- **Graph Structure Storage**: $O(V_{\max} + E_{\max})$, grows linearly with outer loop
- **Method Library Storage**: $O(M \cdot I_{\text{avg}})$, genetic screening may reduce M
- **Genetic Population Cache**: $O(G \cdot (V + E + M))$ (if enabled), G usually bounded
- **Sub-Trainer Intermediate State**: $O(N \cdot V \cdot S_{\text{data}})$ (technical debt, recommended to disable during search phase)
- **Logs and Snapshots**: $O(C_{\text{outer}} \cdot (V + E))$ for recording results of each cycle

Total Space Complexity:

$$S_{\text{Combined}} = O(V_{\max} + E_{\max} + M \cdot I_{\text{avg}} + G \cdot (V + E + M) + N \cdot V \cdot S_{\text{data}}) \tag{43}$$

B.4 Engineering Constraints and Technical Debt Notes

Major Constraints of Current Implementation

1. **Full Intermediate Result Cache**: Leads to space complexity $O(N \cdot V \cdot S_{\text{data}})$
2. **Global Lock Granularity**: Data extraction placed within lock, causing parallel bottleneck, recommended to switch to fine-grained locks or lock-free structures
3. **GIL Limitation**: `ThreadPoolExecutor` cannot break through CPython Global Interpreter Lock, pure Python operators can only achieve concurrency, future migration to C++ or process pools may be needed for true parallelism
4. **Hardcoded Evaluation**: Loss and accuracy evaluation are hardcoded
5. **Modular Compression Residue**: `_phase_modular_compression` function is deprecated but retained in code

C Experimental Details

Compute Resources. All experiments were executed on standard workstations with multi-core CPUs and 60GB RAM. No GPU was used. Specifically, the parallel scalability experiments utilized a server with 32 CPU cores to evaluate multi-threading performance. The total wall-clock time to reproduce all main experimental results sequentially is estimated to be less than one week.

C.1 Experiment 1 Configuration and Full Statistical Results (LayerGrow on Titanic)

***Positioning**: This appendix summarizes the **minimal configuration set** and **full statistical results** required to reproduce Experiment 1. For experimental objectives, core findings, and framework-level conclusions, please refer to Main Text § 4.1.*

C.1.1 Dataset and Task

Table 6: Dataset and Task Configuration

Item	Configuration
Dataset	Titanic Dataset. Only the training set is used for validation, stratified by `Survived` into 80% training / 20% testing.
Task Type	Binary Classification. *Note:* This is a benchmark for structural exploration capability, **not** for generalization performance evaluation. It aims to reveal saturation and failure boundaries of search strategies in discrete space.
Evaluation Metric	Test set accuracy (`best_model_accuracy`).

C.1.2 METHOD POOL CONFIGURATION (9 SCALAR OPERATIONS)

```
# All methods have scalar input/output types, supporting type-safe routing
[
    "return_value",           # Identity mapping
    "return_two_values",      # Single input, dual output
    "add_values",             # Addition
    "calculate_difference",   # Subtraction
    "multiply_values",        # Multiplication
    "divide_values",          # Division (zero-protected)
    "increment",              # +1
    "decrement",              # -1
    "negate_value"            # Negation
]
```

***Design Principles**: Type consistency + Low-dimensional data flow + Determinism. Excluding semantic heterogeneity interference to focus on validating the navigation capability of the LayerGrow probe in pure structural combinatorial space.*

C.1.3 HYPERPARAMETER GRID

Table 7: Hyperparameter Settings

Parameter	**Value/Description**
`max_layers`	`[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]`
`max_attempts` (per layer)	10
Rollback layers	2
`decision_threshold`	0.0
Max Total Attempts	**2500** (Forced termination upon reaching limit is defined as "failure" and included in failure rate statistics)
Independent Repetitions	10 times/configuration
Random Seed	Base 42, expanded per run: seed $= 42r + \text{hash}(\texttt{max_layers}_L) \bmod 1000$ ($r \in [1, 10]$)

C.1.4 FULL STATISTICAL RESULTS

Table 8 reports detailed statistical metrics for each `max_layers` configuration. The data corresponds to the trend curves in Main Text Fig. 2.

- **Performance Saturation Point**: Mean best accuracy saturates at `max_layers=20` (0.763 ± 0.041, 0% failure); for larger budgets it fluctuates only within statistical noise (0.769–0.783 at 25–50 layers) while the failure rate climbs toward 100%.
- **Strategy Failure Boundary**: When `max_layers` ≥ 25, the failure rate begins to rise (20% at 25–30 layers) and reaches 100% at 45+ layers, indicating that the greedy strategy falls into local optima traps in deep spaces.
- **Path Dependence**: Even at the optimal configuration (20 layers), the standard deviation reaches 0.041, reflecting the sensitivity of discrete search to initial conditions.
- **Aggregation Rule**: All Avg Best Acc values are the mean, over the 10 independent runs, of the best model observed during each run—including runs terminated by the 2,500-attempt cap (failures)—consistent with the reporting convention in Main Text §4.1.

C.1.5 SUPPLEMENTARY NOTES AND REPRODUCTION GUIDELINES

- **Path Dependence Analysis Data**: The scatter plot in the right panel of Main Text Fig. 2 is generated from the per-run summary `experiments/LayerGrowTrainer_exp1/experiment_results_summary.csv` (shipped; columns `run`, `best_model_accuracy`, `total_growth_attempts`) by `generate_experiment_figures.py`; no separate plot-data CSV is maintained.
- **Complete Logs**: Candidate methods, rollback events and per-attempt accuracy are recorded for every run in `exp_max_layers_<L>/run_<r>/training_log.json` (100 files, ≈216 MB in total; keys `attempt_history` and `rollback_events`). They are regenerated by `main.py` and are not shipped, for size.

max_layers	Avg Best Acc	Std Dev	Avg Actual Layers	Avg Total Attempts	Failure Rate (%)
5	0.665	0.042	5.0	6.5	0.0
10	0.721	0.030	10.0	50.5	0.0
15	0.732	0.036	15.0	160.5	0.0
20	**0.763**	**0.041**	**20.0**	**624.5**	**0.0**
25	0.769	0.028	22.9	1414.9	20.0
30	0.777	0.025	26.3	1567.4	20.0
35	0.780	0.022	21.7	2069.2	70.0
40	0.776	0.020	23.3	2289.7	80.0
45	0.783	0.024	19.6	2500.0	100.0
50	0.770	0.032	18.7	2500.0	100.0

Table 8: Statistical results under different maximum layer settings. Failure rate is defined as the proportion of runs where `total_growth_attempts=2500`. The bolded row indicates the optimal balance point between performance and efficiency.

- **Reproduction Script**: Execute `experiments/LayerGrowTrainer_exp1/main.py` to reproduce all configurations and statistical results mentioned above.

C.2 Experiment 2 Configuration Summary (Knowledge Transfer Validation)

***Positioning**: This appendix summarizes only the **minimal configuration set** required to reproduce Experiment 2. For experimental objectives, core findings, and framework-level conclusions, please refer to Main Text § 4.2.*

C.2.1 Task and Evaluation Protocol

Table 9: Task and Evaluation Configuration

Item	Configuration
Task Chain	$f_1(x) = 2(x+1) \rightarrow f_2(x) = f_1(x) + 3 \rightarrow f_3(x) = f_2(x) \times 0.5$
Data Type	Scalar `scalar`, resampled for each repetition (to eliminate data bias)
Loss Function	MSE (Focuses on structural transfer validation, not numerical fitting upper bounds)
Success Criterion	The reported Success Rate counts runs with Task 3 accuracy > 0.95; the stricter "Complete Transfer Chain" criterion (Tasks 1/2 ≥ 0.999 *and* Task 3 > 0.95) applies only to the breakdown in Fig. 3
Evaluation Metrics	Task 3 Mean Accuracy $\pm$ Std Dev, Success Rate (Successful Runs / 100)

C.2.2 Method Pool Configuration (Three Variants, all operating on `scalar`)

```
# Minimal (5 atomic operations)
[
    "increment",         # x -> x+1
    "multiply_by_2",     # x -> 2x
    "add_3",             # x -> x+3
    "multiply_by_05",    # x -> 0.5x
    "identity"           # x -> x
]

# Extended (Minimal + 1 item)
[...above 5 items..., "add_2"]  # x -> x+2

# Oracle (Static Injection): Minimal pool + f1_direct (6 items)
[
    "increment", "f1_direct",   # x -> x+1  |  x -> 2(x+1)
    "multiply_by_2", "add_3", "multiply_by_05", "identity"
]
```

***Design Principle**: All methods have input/output types of `scalar`, excluding type-matching interference to focus on validating structural composition capabilities.*

Table 10: Hyperparameters and Control Variables

Parameter	Value/Description
`max_init_layers` (Tasks 1/2)	2 layers / 3 layers (two comparison groups)
Method Pool Evolution	Enabled (`Exp`) / Disabled (`Ctrl`) / Static Injection (`Oracle`)
Independent Repetitions	100 times/configuration
Random Seed Strategy	`42 + rep_id` (`rep_id` $\in [0, 99]$)
Data Sampling	Regenerated for each repetition to avoid data bias
Early Stopping Threshold	Task-level accuracy ≥ 0.999 considered "Perfect Mastery"

C.2.3 Hyperparameters and Control Variables

C.2.4 Supplementary Notes

- **Chain Analysis Definition**: "Complete Knowledge Transfer Chain" = Task 1 Perfect (Acc ≥ 0.999) $\wedge$ Task 2 Perfect (Acc ≥ 0.999) $\wedge$ Task 3 Success (Acc > 0.95), used to isolate structured transfer paths.
- **Accuracy Calculation Logic**: Based on *Relative Error Threshold*. A sample is considered correct if $\frac{|\hat{y}-y|}{|y|+10^{-8}} < 0.01$. Task accuracy is the proportion of correct samples in the dataset.
- **Method Pool Evolution Data**: The change in the total number of methods at each stage is computed from `exp3_results/<config>/rep<k>/metadata.json` by `plot_method_pool_evolution.py`; the rendered figure is shipped as `GraphEvo_exp3/method_pool_three_stage_evolution.png`.
- **Complete Statistics**: Accuracy for Tasks 1/2/3, method-pool snapshots and evolution trajectories are stored per run under `exp3_results/<config>/rep<k>/` (100 repetitions per configuration, ≈10 MB in total). They are regenerated by `exp3_main.py` and are not shipped; the configuration-level summary backing Table 11 is shipped as `exp3_results/summary_table.csv`.

Table 11 reports the end-to-end performance of Task 3 under various configurations (100 repetitions). Figure 3 further decomposes the success paths of the knowledge transfer chain.

Table 11: Performance comparison for Task 3 in Experiment 3. Evolution column: ✓= Method pool evolution enabled, ×= Disabled, — = Static injection baseline.

Group	Evolution	Max Layers	Method Pool	Avg Acc $\pm$ Std	Success Rate (%)
Ctrl_L2_Minimal	×	2	Minimal	0.0006 ± 0.0021	0
Ctrl_L2_Extended	×	2	Extended	0.0003 ± 0.0012	0
Ctrl_L3_Minimal	×	3	Minimal	0.0007 ± 0.0022	0
Exp_L2_Minimal	✓	2	Minimal	0.1308 ± 0.3360	**13**
Exp_L2_Extended	✓	2	Extended	0.1312 ± 0.3359	13
Exp_L3_Minimal	✓	3	Minimal	0.0908 ± 0.2859	9
Oracle_L2	—	2	Oracle	0.1206 ± 0.3247	12

C.3 Experiment 3 Configuration and Supplementary Data (Topology-Aware Task Parallelism)

C.3.1 Experimental Motivation and Core Findings

To validate the **structural-level parallel schedulability** of GDFG during the inference phase, we evaluated the scalability of a topology-aware dynamic task parallelism mechanism in a multi-core CPU environment. This experiment aims to address two framework-level questions:

1. Does the explicit dependency metadata in GDFG support **lock-free or low-conflict parallel scheduling**?
2. How do the topological characteristics of the graph constrain the **parallel efficiency boundaries** of the representation space on real hardware?

Experimental results show that under simulated I/O or non-blocking loads, shallow wide graphs achieved a mean speedup of $10.25\times$ on 16 cores, i.e. 64% of the ideal linear bound, with efficiency still near 90% at 8 cores. This confirms that the DAG structure of GDFG naturally supports fine-grained task parallelism. Further analysis indicates that parallel efficiency is strongly positively correlated with the graph's **Average Width** (Pearson $r \approx 0.99$ across the 30 graphs at 16 cores), rather than its total node count ($r \approx -0.16$) or depth ($r \approx -0.41$), suggesting that "wide and shallow" topological structures are more conducive to multi-core utilization. Additionally, explicit type constraints eliminate the overhead of runtime data race detection, enabling the scheduler to maintain low-conflict, efficient execution even when facing heterogeneous DAGs generated through evolution.

This experiment confirms that the AdaptoFlux representation space possesses **native parallel potential**. Its performance is primarily constrained by the graph's average width and the scheduling granularity of the backend execution model. This provides clear guidelines for subsequent optimization: structurally, evolution should encourage the generation of high-concurrency, wide-shallow topologies to maximize the utilization of modern multi-core hardware; engineering-wise, migrating to GIL-free backends (such as C++ extensions or process pools) can further unlock the theoretical parallel upper bound. Detailed experimental configurations, structural metric analyses, and flame graph evidence are provided in the following subsections.

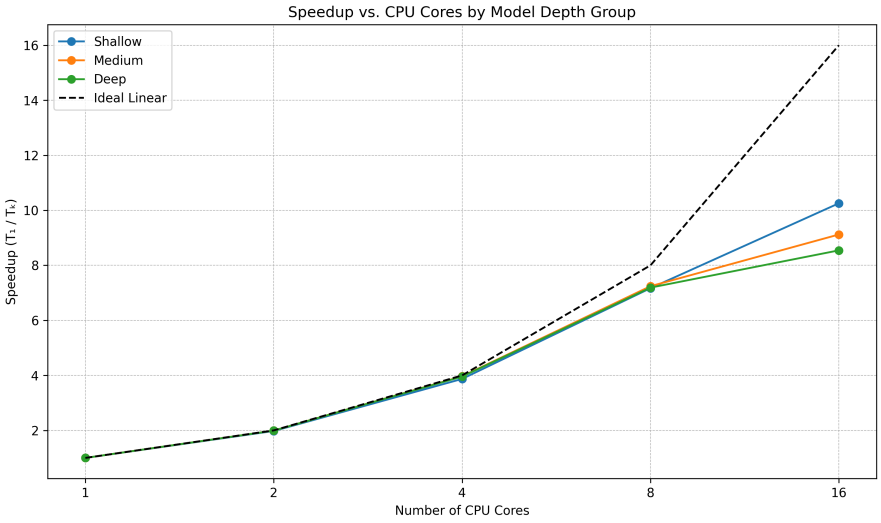


Figure 5: Average speedup under GIL-releasable scenarios. Different depth model groups under simulated non-blocking loads. The dashed line represents ideal linear speedup. Shallow wide graphs achieve a **10.25**$\times$ speedup on 16 cores.

C.3.2 Experimental Environment and Core Configuration

Hardware and Software Environment

- **Executor**: Dynamic task scheduler implemented based on Python `concurrent.futures.ThreadPoolExecutor`.
- **Hardware Platform**: Multi-core CPU environment on the same server, with other process interference isolated.
- **Parallel Granularity**: Number of threads $K \in \{1, 2, 4, 8, 16\}$, simulating concurrency capabilities of different core counts.
- **Load Simulation Modes**:
  - **GIL-Constrained Scenario (No-Sleep)**: Nodes execute pure Python lightweight computations, constrained by the Global Interpreter Lock, preventing true parallelism.

  - **GIL-Releasable Scenario (Sleep)**: every method in the pool calls `time.sleep(0.001)` inside its own implementation before returning, simulating a 1 ms I/O wait or C-extension call so that threads release the GIL and achieve CPU-level parallelism.

Control Variables

- **Fixed Variables**:
  - Dataset: Titanic (Titanic Survival Prediction).
  - Input Batch Size: Kept consistent.
  - Method Pool Configuration: Uses the simplified method pool from Experiment 1 (see Appendix C.1).
  - Random Seed: 42.
- **Variable Variables**:
  - Number of computation threads K.
  - Model topology structures (different evolution results from Experiment 1 training logs, see Appendix C.1).

C.3.3 Experimental Subjects: Detailed List of 30 DAG Models

In the main text, models are divided into shallow, medium, and deep groups based on depth. The table below lists the detailed source information for all 30 models participating in the experiment, including maximum allowed layers (`max_layers`), run sequence (`run`), model type (`model_type`), and actual constructed layers.

***Note**: Models are classified by actual layer count: Shallow (1–10 layers), Medium (11–20 layers), Deep (21–35 layers). Some models have actual layers less than `max_layers` due to early termination during evolution.*

C.3.4 Supplementary Analysis: Relationship Between Node Scale and Parallel Efficiency

The main text states that "total node count is not the core factor determining parallel efficiency." To verify this conclusion, we regrouped the 30 models by **total node count** and statistically analyzed their performance under different core counts.

Performance Statistics Grouped by Node Scale Analysis:
As seen in Table 13, although the large-scale group has a slightly higher absolute speedup under 16 cores ($9.03\times$ vs $8.92\times$), its efficiency fluctuation (standard deviation) is not significantly better than the small/medium groups. More importantly, the small-scale group achieved the highest average speedup ($10.19\times$) under 16 cores, directly refuting the intuitive assumption that "larger models are harder to parallelize."

Scatter Distribution of Node Count vs. Speedup Observation:
Fig. 6 shows that the vertical distribution of speedup has no obvious monotonic correlation with the horizontal axis of node count. Whether with dozens of small nodes or hundreds of large nodes, speedups are distributed within the $8\times$–$10\times$ range. This further confirms that **the total node count of the graph structure is not the core factor determining pipeline parallel efficiency**.

C.3.5 Supplementary Analysis: Impact of Structural Heterogeneity Metrics

Besides depth and width, local connectivity characteristics of the graph (such as branching rate, out-degree) may also affect scheduling complexity. We extracted the following structural metrics for analysis:

- **Branching Rate**: Proportion of nodes with out-degree ≥ 2.
- **Average Out-Degree**: Average number of output edges for all nodes in the graph.

Table 12: Source and Structural Information of the 30 Models Used in Experiment 3

ID	max_layers	run	model_type	Actual Layers	Group
1	5	1	final	5	Shallow
2	5	5	final	5	Shallow
3	5	6	final	5	Shallow
4	10	1	best	9	Shallow
5	10	2	best	2	Shallow
6	10	4	best	8	Shallow
7	10	5	best	5	Shallow
8	10	8	best	6	Shallow
9	5	7	final	5	Shallow
10	5	8	final	5	Shallow
11	15	1	best	12	Medium
12	15	4	final	15	Medium
13	15	8	best	11	Medium
14	15	9	final	15	Medium
15	20	1	final	20	Medium
16	20	3	best	9	Shallow
17	20	4	final	20	Medium
18	20	5	final	20	Medium
19	20	7	best	17	Medium
20	20	9	final	20	Medium
21	25	6	final	25	Deep
22	25	7	final	25	Deep
23	25	10	best	23	Deep
24	30	2	best	22	Deep
25	30	3	final	30	Deep
26	30	7	best	15	Medium
27	30	8	final	30	Deep
28	35	2	best	27	Deep
29	35	5	final	35	Deep
30	30	1	best	11	Medium

Table 13: Average Performance of Models with Different Node Scales on Multi-Core CPUs (No-GIL Valid Data Only)

Node Scale Range	CPU Cores	Avg Speedup (S_K)	Avg Efficiency (E_K)	Efficiency Std Dev
Small (≤93)	1	1.00×	100.0%	—
(10 models)	4	3.86×	96.6%	2.7%
	8	7.13×	89.1%	6.1%
	16	10.19×	63.7%	8.1%
Medium (94–190)	1	1.00×	100.0%	—
(10 models)	4	3.96×	99.0%	0.7%
	8	7.11×	88.9%	8.7%
	16	8.92×	55.8%	11.5%
Large (≥191)	1	1.00×	100.0%	—
(10 models)	4	3.97×	99.1%	0.6%
	8	7.36×	92.0%	5.4%
	16	9.03×	56.5%	8.3%

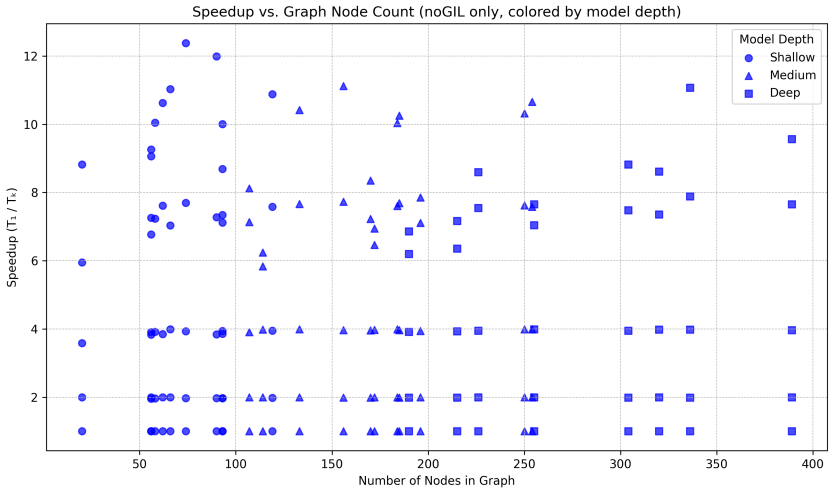


Figure 6: Scatter plot of node count vs. speedup (No-GIL data only). X-axis: Total nodes in graph; Y-axis: Speedup $S_K = T_1/T_K$. Different marker shapes represent model depth (Circle: Shallow, Triangle: Medium, Square: Deep).

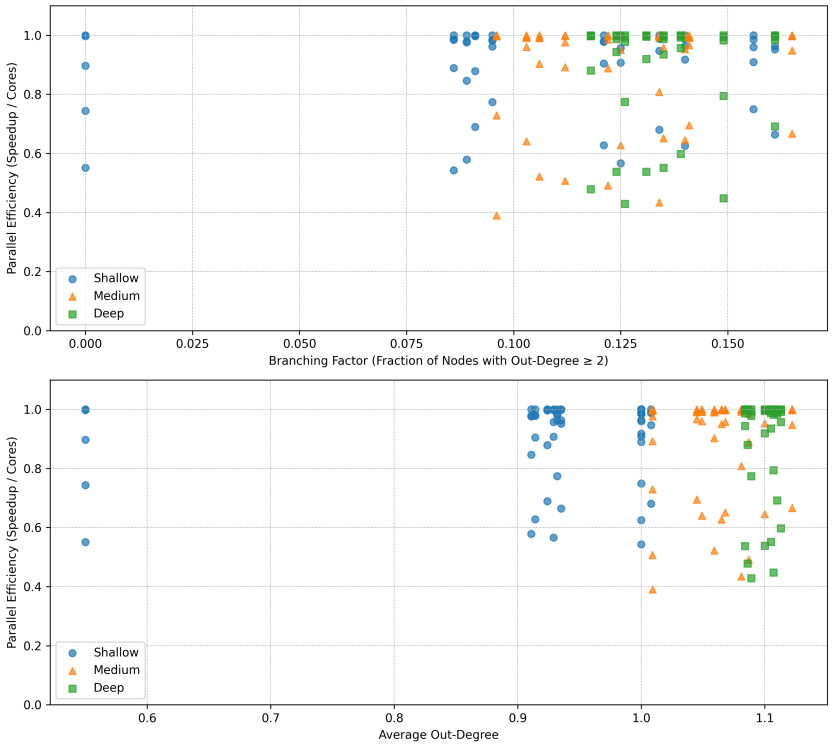


Figure 7: Scatter plots of structural heterogeneity metrics vs. speedup (No-GIL data only). (a) Branching Rate vs. Speedup; (b) Average Out-Degree vs. Speedup.

Analysis:

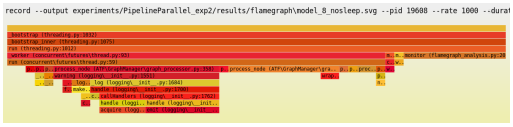
- **No Significant Correlation**: Fig. 7 shows that high branching rate or high average out-degree did not lead to a systematic decrease in speedup.
- **Coupling of Depth and Heterogeneity**: Notably, deep models (square markers) tend to cluster in regions of high branching rate and high out-degree. This indicates that as evolution layers deepen, the graph structure naturally tends towards more complex connection patterns. However, this increase in complexity did not impose a significant burden on AdaptoFlux's topology-aware scheduling, demonstrating that the in-degree-based dynamic scheduling algorithm has good robustness to local heterogeneity.

C.3.6 Micro-Performance Profiling: Flame Graph Evidence

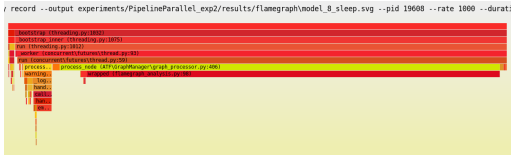
To deeply understand the impact mechanism of GIL on task parallelism, we selected three representative models: Shallow (6 layers), Medium (15 layers), and Deep (25 layers), and collected execution flame graphs for both GIL-constrained and GIL-releasable scenarios.

Flame Graph Comparison Overview Micro-Profiling Conclusion:

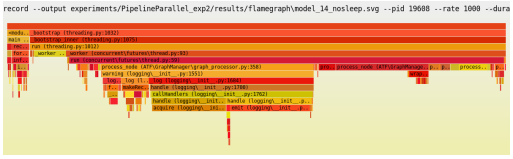
- **Left Column (GIL-Constrained)**: The `wrapped` node (computation logic) is extremely narrow in the flame graph, almost drowned out by scheduler overhead (dependency tracking, lock contention, logging). This indicates that multi-threading cannot execute computations concurrently, only incurring additional synchronization costs.
- **Right Column (GIL-Releasable)**: The width of the `wrapped` function increases significantly, showing overlapping execution across different threads. This confirms that when node computation can release the GIL, AdaptoFlux's topology-aware scheduling mechanism effectively exposes node-level parallel potential, and the coarse-grained tasks introduced by `time.sleep` further reduce the relative overhead of context switching.



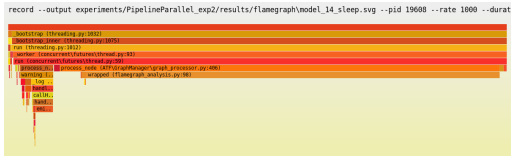
(a) Shallow model (6 layers), pure Python computation



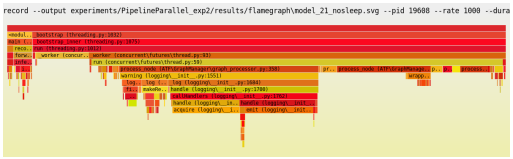
(b) Shallow model (6 layers), injected `time.sleep`



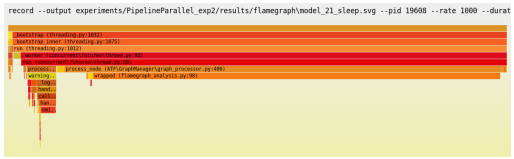
(c) Medium model (15 layers), pure Python computation



(d) Medium model (15 layers), injected `time.sleep`



(e) Deep model (25 layers), pure Python computation



(f) Deep model (25 layers), injected `time.sleep`

Figure 8: Flame graph comparison for representative models under GIL-constrained (Left) and GIL-releasable (Right) scenarios.

***Note**: In the flame graphs, the `wrapped` function may appear as `wrap..` or nested under `process_node` due to sampling rate or namespace folding; its true meaning is the node computation proxy function.*

Detailed Interpretation

1. **GIL-Constrained Scenario (Left Column)**:
   - **Phenomenon**: The `wrapped` node (i.e., user-defined computation logic) is extremely narrow in the flame graph, almost invisible.
   - **Dominant Overhead**: CPU time is mainly consumed by internal scheduler logic, including:
     - Dependency checking and in-degree updating (`decrement_indegree`).
     - Thread lock contention (`threading.Lock.acquire`).
     - Logging and status synchronization.
   - **Conclusion**: In pure Python computation, multi-threading not only fails to parallelize computation but also leads to "negative acceleration" due to frequent context switching and lock contention.
2. **GIL-Releasable Scenario (Right Column)**:
   - **Phenomenon**: The width of the `wrapped` function increases significantly, occupying the main part of the flame graph.
   - **Parallel Exposure**: Multiple `wrapped` blocks overlap on the timeline, indicating that different threads are executing different node computations simultaneously.
   - **Reduced Scheduler Overhead Ratio**: Although scheduling logic still exists, its relative proportion shrinks significantly because the coarse-grained tasks simulated by `time.sleep` make computation/wait time much larger than scheduling overhead.
   - **Conclusion**: Once node computation can release the GIL (e.g., I/O, C extensions, or explicit sleep), AdaptoFlux's topology-aware scheduling mechanism efficiently distributes tasks to idle cores, fully exposing parallel potential.

C.3.7 Scheduling Algorithm Pseudocode

AdaptoFlux employs a topological sorting-based dynamic task scheduling algorithm, as shown below:

```python
def topological_dynamic_schedule(graph, thread_pool):
    # 1. Initialization: Calculate remaining in-degree for each node
    remaining_in_degree = {node: len(node.predecessors) for node in graph
        .nodes}

    # 2. Ready Queue: Find all nodes with initial in-degree 0 (excluding
        root)
    ready_queue = [node for node in graph.nodes if remaining_in_degree[
        node] == 0 and node != root]

    futures = []

    while ready_queue or futures:
        # 3. Submit Tasks: Take nodes from ready queue and submit to
            thread pool
        while ready_queue:
            node = ready_queue.pop(0)
            future = thread_pool.submit(execute_node, node)
            future.add_done_callback(lambda f, n=node: on_node_complete(n
                , f))
            futures.append(future)

        # 4. Wait and Callback Processing
        for future in as_completed(futures):
            node = future_to_node_map[future]
            futures.remove(future)

            # 5. Update Successor Node Status
            for successor in node.successors:
                remaining_in_degree[successor] -= 1
                if remaining_in_degree[successor] == 0:
                    ready_queue.append(successor)

def execute_node(node):
    # Execute the method bound to the node
    result = node.method(*node.inputs)
    return result
```

Listing 1: Topological Dynamic Schedule

C.4 Experiment 4 Configuration and Full Statistical Results (Minimal Stateful Attractor)

***Positioning**: This appendix summarizes only the **minimal configuration set** required to reproduce the automatic stateful synthesis experiment of Main Text § 4.3. For the experimental objective and the framework-level interpretation, please refer to that section.*

C.4.1 Task and Evaluation Protocol

Table 14: Minimal Stateful Attractor: task and evaluation configuration

Item	Configuration
Environment (`MinimalState`)	Discrete position `pos` $\in \{-2,\dots,2\}$ and target `target` $\in \{-2,\dots,2\}$, resampled at every reset subject to `pos` $\neq$ `target`
Observation	`[pos, target]` at every step; the task goal is the state round(`pos`) = round(`target`)
Episode	One reset followed by at most 4 graph executions; after each execution the graph re-reads the updated observation (closed loop)
Runtime evaluation	100 episodes per run; a run counts as successful iff $\geq 95\%$ of its episodes reach the target state
Training-time signal	GraphEvo loss/accuracy probes use 20 episodes $\times \leq 4$ steps, a reduced budget that keeps refinement cycles cheap
Independent runs	10 per configuration, seeds 42 + run_id (42–51); both `random` and `numpy` are seeded per run

Protocol note (search vs. evaluation). Each run performs exactly one search (one refinement trajectory of the probe) and reports the graph it ends with; the 100 episodes are evaluation rollouts of that single graph, not 100 independent searches. "Success rate" throughout therefore counts runs—i.e. searches—whose final graph solves the task in at least 95% of the evaluation episodes.

This task is deliberately the smallest non-trivial instance that *requires* a stateful graph: since $\texttt{pos} \neq \texttt{target}$ at reset and the state can only change through a side-effecting action, no static input–output mapping can satisfy it, and the maximum initial distance ($|\texttt{pos} - \texttt{target}| = 4$) exactly saturates the 4-step budget.

C.4.2 Method Pool Configuration (Two Variants, 9 Methods Each)

Both variants register the same nine methods with byte-identical implementations; they differ only in the declared type signatures (and hence in the root input types), so every functional edge is typed in the constrained variant and uniformly `scalar` in the baseline.

```
# Type-Constrained AdaptoFlux: signatures enforce the functional chain
read_current_pos    : scalar                -> pos_type     (state_read)
read_target         : scalar                -> target_type  (state_read)
calc_direction      : pos_type, target_type -> scalar       (pure_computation)
move_towards_target : scalar                -> scalar       (stateful_action)
# Five irrelevant distractors, all scalar -> scalar
add(x,y), multiply(x,y), sin_op(x), cos_op(x), identity(x)

# Unconstrained Baseline: the same nine methods with identical bodies, but
# every signature is scalar -> scalar and the root tokens are plain scalars,
# so any method may be wired into any slot.
```

***Design Principle**: Both variants register the same nine methods with byte-identical implementations; only the declared type signatures differ (together with the root input tokens they imply). In the constrained variant the typed edges force the functional chain `read_current_pos`/`read_target` → `calc_direction` → `move_towards_target`, whereas in the baseline every method may be wired into any slot. Any change in success rate is therefore attributable to type routing alone, not to a different operator inventory or to different operator semantics. The root node injects the observation as `pos_type`/`target_type` tokens in the constrained variant and as two `scalar` tokens in the baseline, while `move_towards_target` mutates the implicit environment state $\mathbf{s}_t$ as a side effect (Eq. 2) and returns the updated position.*

C.4.3 Hyperparameters and Control Variables

Table 15: Minimal Stateful Attractor: hyperparameters and controlled variables

Parameter	Value/Description
Probe	`GraphEvoTrainer` used in isolation
`init_mode` / `max_init_layers`	`fixed` / 3 layers
`num_initial_models`	3 initial graph skeletons per run
`max_refinement_steps`	30 local node replacements per cycle
`max_evo_cycles`	5
`candidate_pool_mode`	`all`
`enable_evolution`	`False` (method-pool filtering disabled, so the comparison isolates the type system)
`early_stop_eps`	0.0
Reported graph	The graph each run *ends with*, saved as `results/<config>/run_i/final/graph.gexf`; intermediate snapshots are kept in `best/`

C.4.4 Full Statistical Results

Table 16 reports the final accuracy of every run of both configurations. Accuracy is the proportion of the 100 evaluation episodes in which the target state is reached within 4 steps.

- **Type-constrained configuration**: 8 of 10 runs synthesized a graph whose evaluated form contains the functional core `root` → `calc_direction` → `move_towards_target`, i.e. the observation is read through the typed root tokens, the signed distance is computed by a pure node, and the resulting scalar drives a side-effecting action. Accuracy in these runs is exactly 1.0 rather than merely above threshold: with $|\texttt{pos} - \texttt{target}| \leq 4$ and a unit step per execution, the synthesized policy is a complete closed-loop solution of the task. Note that the action node is frequently *off* the collapse path (e.g. `move_towards_target` → `read_target` → `collapse`); execution is triggered by node presence in topological order, not by output connectivity, which is precisely the side-effect semantics of Eq. 2.

Table 16: Per-run final accuracy of the Minimal Stateful Attractor (10 independent runs per configuration). A run is successful iff final accuracy $\geq 95\%$.

Run	Seed	Unconstrained (Baseline)	Type-Constrained AdaptoFlux
1	42	33.0%	100.0%
2	43	30.0%	100.0%
3	44	22.0%	0.0%
4	45	42.0%	100.0%
5	46	24.0%	0.0%
6	47	37.0%	100.0%
7	48	32.0%	100.0%
8	49	27.0%	100.0%
9	50	24.0%	100.0%
10	51	32.0%	100.0%
Success Rate ($\geq 95\%$)		0/10 (0.0%)	8/10 (80.0%)
Mean Final Accuracy $\pm$ Std Dev		0.303 ± 0.060	0.800 ± 0.400

- **The two failures (seeds 44 and 46)** terminate on graphs that contain the pure computation node but not `move_towards_target`, so the graph never leaves its initial state and scores 0%. This is a search failure of the greedy refinement probe, not a type-routing rejection: a compatible completion exists in the pool.
- **Unconstrained baseline**: 0 of 10 runs reached the threshold; per-run accuracy ranges from 22% to 42% (mean 0.303 ± 0.060). The final graph of every run contains the action node but never the subtraction node: the scalar driving `move_towards_target` comes from distractor nodes (`sin_op`, `cos_op`, `identity`), and in three runs (seeds 42, 43, 48) the graph contains no state-reading node at all, so it cannot condition on the observation. Without type routing the five distractors and the two state-reading nodes are interchangeable with the two functional nodes, so greedy replacement settles on compositions that are only partially correct or entirely static.
- **Reproducibility**: the comparison was executed twice on 2026-09-20—once in a clean checkout and once through `run_comparison.py` in the repository—and produced identical per-run accuracies, confirming that the reported outcome is determined by the seeds rather than by run-to-run noise.
- **Interpretation boundary**: the two configurations share the same search budget and the same nine methods. The result therefore isolates the contribution of the representation layer (type signatures plus the resulting routing) to the navigability of the space, and does not constitute evidence that the current probe scales to long-horizon control (Appendix I).

C.4.5 Paired Analysis and Exact Test

The two configurations are executed with the *same* ten seeds (42–51, Table 16), so the runs form matched pairs rather than two independent samples: the seed fixes initialisation and data order, and the declared type signatures are the only factor that differs. Each pair falls into one of four cells (Table 17), and only the *discordant* pairs carry information about the effect, because pairs in which both conditions succeed or both fail occur equally often whether or not type routing matters.

Table 17: Paired outcomes of the minimal stateful synthesis task over the ten shared seeds. Rows: type-constrained search; columns: unconstrained baseline. Only the discordant cells b and c enter the McNemar test; the concordant cells a and d carry no information about the difference between the two conditions.

	Baseline: fail	Baseline: success
Constrained: success	$b = 8$	$a = 0$
Constrained: fail	$d = 2$	$c = 0$

With $b = 8$ and $c = 0$, the null hypothesis that a discordant pair is equally likely to favour either condition gives $B \sim \mathrm{Binomial}(8, \frac{1}{2})$, and the exact McNemar probability of the observed and more extreme outcomes is

$$p_{\text{one-sided}} = \sum_{k=8}^{8} \binom{8}{k} 2^{-8} = 2^{-8} = 3.9 \times 10^{-3}, \qquad p_{\text{two-sided}} = 2^{-7} = 7.8 \times 10^{-3}.$$

The paired test is the conservative choice: the independent-samples reading of the same counts gives Fisher's exact test, $\binom{10}{8}/\binom{20}{8} = 45/125970 = 3.6 \times 10^{-4}$, but it is not applicable here, because the baseline's ten failures are not ten additional independent observations—they are the partners of the ten matched runs. We therefore report the exact version (`scipy.stats.binomtest`, cross-checked against `statsmodels`' exact McNemar); the asymptotic χ^2 form with continuity correction, $(|b-c|-1)^2/(b+c) = 6.125$, returns $p = 1.3 \times 10^{-2}$ and is unreliable at $b+c = 8$. Ten pairs also bound the design from below: even if all ten discordant pairs favoured the constrained arm, the exact one-sided test could only reach $2^{-10} = 9.8 \times 10^{-4}$, which is why the knowledge-transfer experiment of Appendix C.2 uses 100 repetitions.

C.4.6 Supplementary Notes and Reproduction Guidelines

- **Execution Script**: `experiments/minimal_stateful_comparison/run_comparison.py` reproduces both configurations and writes the aggregate report.
- **Environment and Method Pools**: `env.py` (task state), `methods_constrained.py` and `methods_unconstrained.py` (the two 9-method pools).
- **Complete Logs**: `logs/experiment_log_*.log` contain the per-run accuracy and success flag; the log of the reported execution is `experiment_log_20260920_124456.log`.
- **Per-Run Graphs**: `results/<config>/run_i/{best,final}/` store the saved graph (`graph.gexf`) and the method-pool copy of every run; `results/stateful_synthesis_comparison.png` is the figure used in Main Text Fig. 4.

D Ablation Study on Structural Composition and Method Pool Adaptability in Geometrically Sensitive Tasks

This appendix systematically evaluates the structural composition capabilities of the AdaptoFlux framework and its adaptability to different method pools when handling nonlinear classification tasks with complex geometric decision boundaries. By designing strict controlled-variable experiments, we deeply explore the actual utility of different training strategies (single strategy vs. combined strategy) and genetic screening mechanisms in symbolic program synthesis.

D.1 Experimental Design and Task Definitions

To comprehensively examine the framework's performance under varying semantic complexities, we designed three classic nonlinear geometric classification tasks. These tasks share the same training framework (`CombinedTrainer`) and hyperparameter configurations, with the only variable being the **composition of the method pool**. Each experiment was independently repeated 100 times, with results reported as Test Accuracy $\pm$ Standard Deviation.

Task Descriptions

1. **Concentric Circles**:
   - *Challenge*: Data is distributed in concentric circles, which are linearly inseparable. The model needs to implicitly construct radial distance features, i.e., approximate $x_1^2 + x_2^2$.
   - *Method Pool Configuration*: Uses the **Restricted Method Pool**, containing 15 basic operations without trigonometric functions, forcing the model to utilize algebraic operations (such as square, addition) to construct the solution space.
2. **Two Spirals – Full Method Pool**:

- *Challenge*: Data is distributed in intertwined double spirals, exhibiting strong rotational symmetry. The ideal solution typically involves polar coordinate transformations.
- *Method Pool Configuration*: Uses the **Full Method Pool**, containing 20 operations, explicitly providing trigonometric functions ($\sin, \cos, \arctan 2$) to verify the upper-bound performance of the framework when equipped with suitable atomic operators.

3. **Two Spirals – Restricted Method Pool**:
   - *Challenge*: Same as above, but with trigonometric function support removed.
   - *Method Pool Configuration*: Uses the **Restricted Method Pool**, aiming to test whether the model can fit spiral boundaries through Taylor expansion approximations or complex algebraic combinations in the absence of direct geometric operators.

Detailed Method Pool Composition

The specific definitions of the two method pools used in the experiments are listed below.

Table 18: Restricted Method Pool: 15 basic scalar operations, excluding trigonometric functions.

Method Name	Input Type	Function Description
`return_value`	[scalar]	Identity mapping
`add_values`	[scalar, scalar]	Addition
`calculate_difference`	[scalar, scalar]	Subtraction
`multiply_values`	[scalar, scalar]	Multiplication
`divide_values`	[scalar, scalar]	Division (with zero protection)
`return_two_values`	[scalar]	Copy input value (supports branching)
`increment`	[scalar]	Add 1
`decrement`	[scalar]	Subtract 1
`negate_value`	[scalar]	Negation
`square`	[scalar]	**Square** (Key nonlinear operation)
`relu`	[scalar]	ReLU activation: $\max(0, x)$
`absolute`	[scalar]	Absolute value
`mean_three`	[scalar, scalar, scalar]	Arithmetic mean of three numbers
`max_two`	[scalar, scalar]	Maximum of two numbers
`min_two`	[scalar, scalar]	Minimum of two numbers

Table 19: Full Method Pool: 20 operations, adding geometric and trigonometric functions to the restricted pool.

Method Name	Input Type	Function Description
(First 15 items same as Table 18)		
`sine`	[scalar]	Sine function
`cosine`	[scalar]	Cosine function
`atan2`	[scalar, scalar]	Polar angle calculation $\arctan 2(y, x)$
`sqrt_safe`	[scalar]	Safe square root (returns 0 if $x < 0$)
`scale`	[scalar, scalar]	Scaling: $x \times$ factor

Ablation Experiment Configurations

To systematically compare the effects of different training strategies, we defined the following six ablation configurations. All experiments fixed the random seed to 42 (incremented for each repetition), standardized input features using StandardScaler, with a training set size of 800 and a test set size of 200.

- **Group A: `LayerGrow_Only`**

  - Uses only `LayerGrowTrainer` to build the model. Graph Evolution (GraphEvo) and genetic screening are disabled. Aimed at evaluating the capability of pure structural growth strategies.
- **Group B: `GraphEvo_Only`**
  - Runs only `GraphEvoTrainer`. Performs local node replacement optimization on a fixed initial shallow structure. Aimed at evaluating the capability of pure local search strategies.
- **Group C: `LayerGrow_Then_GraphEvo`**
  - Serially executes `LayerGrow` once to build a deep structure, followed by `GraphEvo` once for optimization. This serves as the single-round combination baseline.
- **Group D: `CombinedTrainer` (Three Variants)**
  - Uses the complete `CombinedTrainer` framework, executing 3 evolution cycles. The difference lies in the enabling strategy of Genetic Screening:
    1. `disabled`: Genetic screening is disabled. The full method pool is retained in each round. Equivalent to executing 3 independent rounds of "LayerGrow → GraphEvo" processes, allowing continuous structural deepening.
    2. `once`: Genetic screening is executed only once before the first round; subsequent rounds use the compressed sub-pool.
    3. `periodic`: Genetic screening is executed after each round (total 3 times), aiming to test the impact of continuous method pool compression on search efficiency and final performance.

Unified Hyperparameter Settings:

- **LayerGrow**: Max attempts 250. Group A allows up to 15 layers; other groups are limited to 10 layers (to prevent search stagnation caused by excessive depth in a single growth phase).
- **GraphEvo**: Group B runs up to 10 rounds; Groups C/D run 8 rounds per cycle.
- **CombinedTrainer**: Fixed 3 evolution cycles. Target sub-pool size is 8.
- **Genetic Algorithm Config**: Population size 20, generations 4 (reduced to 2 in `periodic` mode to save time), elite ratio 30%, mutation rate 15%, data sampling ratio 30%.
- **Early Stopping**: Enabled for all training, with loss change threshold set to 1×10^{-6}.

D.2 Experimental Results

Concentric Circles Task (Restricted Method Pool)

The Concentric Circles task primarily tests the model's ability to combine quadratic nonlinear features (x^2). Since the method pool includes the `square` operation, an exact solution theoretically exists.

Table 20: Performance Comparison on Concentric Circles Task (100 repetitions)

Strategy	Test Acc ($\pm$ Std)	Pool Size	Time (s)	# $\geq$0.99*	Layers	Nodes
CombinedTrainer_disabled	**0.8909 $\pm$ 0.0837**	20.1	633.1	**24**	7.03	25.80
LayerGrow_Then_GraphEvo	0.7927 $\pm$ 0.1354	16.9	21.6	0	3.74	11.63
LayerGrow_Only	0.7725 $\pm$ 0.0622	15.0	17.6	0	3.81	11.90
CombinedTrainer_once	0.7472 $\pm$ 0.1547	14.8	720.1	9	5.33	20.87
GraphEvo_Only	0.6985 $\pm$ 0.1080	17.7	3.4	2	3.03	8.57
CombinedTrainer_periodic	0.6292 $\pm$ 0.1424	9.8	68.6	0	3.06	9.85

* **High-Precision Success Count**: Number of runs with Test Accuracy $\geq$ 99%.

Observations:

- `CombinedTrainer_disabled` achieved the highest average accuracy (89.1%) and the most high-precision successful instances (24/100). This indicates that under unconstrained multi-round growth, the model has sufficient opportunity to explore deep structures containing correct square combinations.

- `CombinedTrainer_periodic` performed the worst (62.9%), with the pool size compressed to 9.8. This suggests that excessive genetic screening may have eliminated key intermediate combinations or base operators required to construct concentric circle boundaries.
- Single strategies (Groups A/B) struggled to achieve high precision due to limitations in structural depth or local optima.

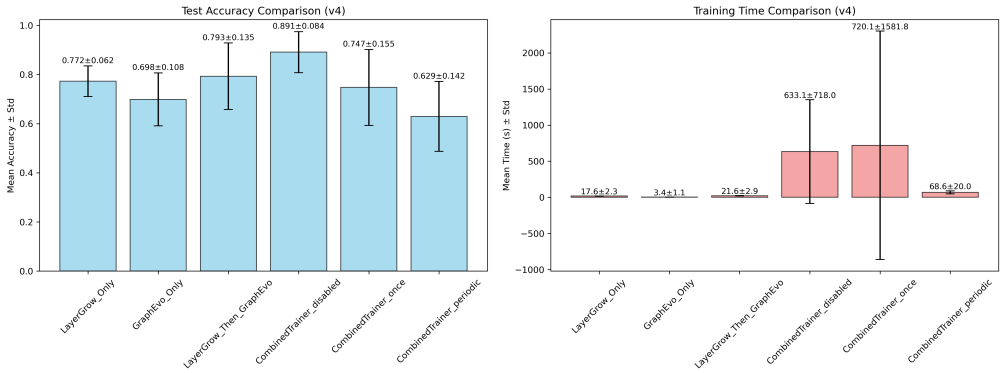
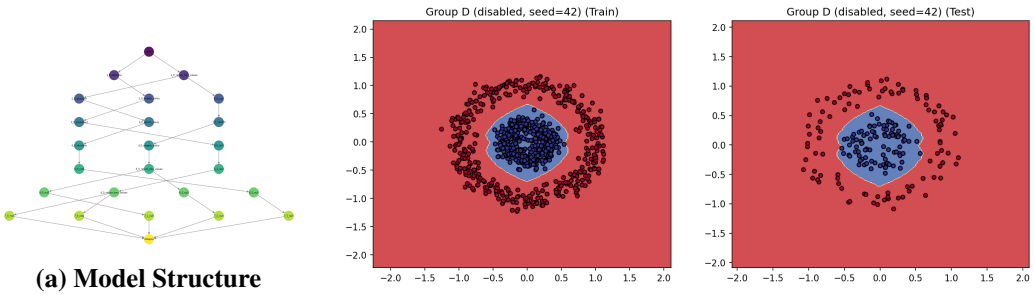


Figure 9: Comparison of Test Accuracy and Training Time for various strategies on the Concentric Circles task (error bars represent standard deviation).



(a) Model Structure **(b) Train Set Boundary** **(c) Test Set Boundary**

Figure 10: Structure and decision boundary visualization of the best model from `CombinedTrainer_disabled` on the Concentric Circles task. Note the clear square and addition combination logic visible in the structure graph.

Two Spirals Task (Full Method Pool)

This task introduces trigonometric functions to verify the framework's ability to utilize explicit geometric operators.

Table 21: Performance Comparison on Two Spirals Task (Full Pool, 100 repetitions)

Strategy	Test Acc ($\pm$ Std)	Pool Size	Time (s)	# $\geq$0.99*	Layers	Nodes
CombinedTrainer_disabled	**0.8493 $\pm$ 0.0338**	26.0	416.4	**1**	6.45	20.30
LayerGrow_Then_GraphEvo	0.8148 $\pm$ 0.0556	22.0	26.5	1	4.63	14.75
LayerGrow_Only	0.8053 $\pm$ 0.0389	20.0	27.4	0	4.77	15.67
CombinedTrainer_once	0.7823 $\pm$ 0.0793	15.9	742.6	0	6.15	20.75
GraphEvo_Only	0.7624 $\pm$ 0.0798	22.8	3.8	0	3.20	8.59
CombinedTrainer_periodic	0.7209 $\pm$ 0.1094	9.9	80.6	0	4.67	16.23

* **High-Precision Success Count**: Number of runs with Test Accuracy $\geq$ 99%.

Observations:

- Even with powerful operators like `atan2`, the Two Spirals task remains highly challenging. `CombinedTrainer_disabled` still maintained the lead, with an average accuracy of 84.9%.
- Notably, `CombinedTrainer_once`, despite having a moderate pool size, had the longest runtime (742.6s) and lower performance than the `disabled` mode. This may be because after the initial screening, the remaining sub-pool, although streamlined, limited the exploration diversity of new structures in subsequent multi-round evolutions.
- `GraphEvo_Only` performed better in the Full Pool than in the Restricted Pool (see Table 22), indicating that local optimization can more effectively fine-tune node semantics when rich semantic operators are available.

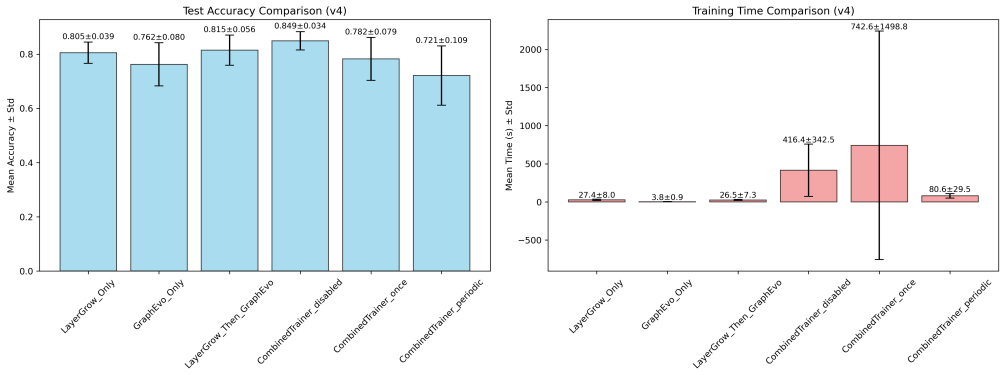
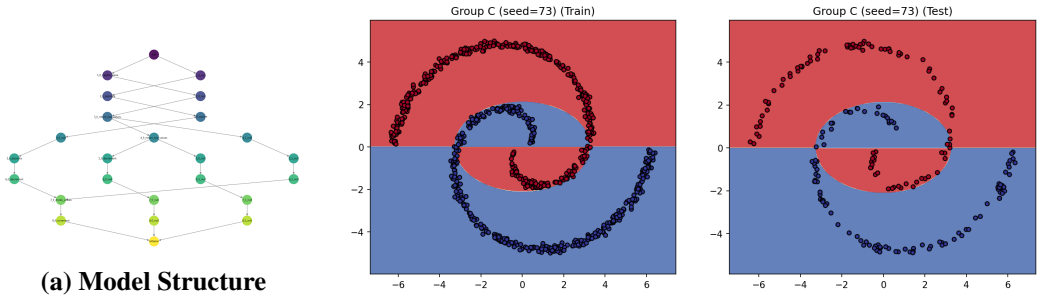


Figure 11: Performance comparison of various strategies on the Two Spirals task (Full Method Pool).



(a) Model Structure **(b) Train Set Boundary** **(c) Test Set Boundary**

Figure 12: Structure and decision boundary visualization of the best model from `CombinedTrainer_disabled` on the Two Spirals task (Full Pool). Direct calls to `atan2` nodes are visible in the structure.

Two Spirals Task (Restricted Method Pool)

This task removes trigonometric functions, forcing the model to approximate spiral boundaries using algebraic operations.

Observations:

- Surprisingly, `CombinedTrainer_disabled` in the Restricted Pool (83.9%) showed a minimal performance gap compared to the Full Pool (84.9%). This demonstrates that AdaptoFlux's **GDFG structural composition capability** possesses strong expressive redundancy, effectively approximating complex geometric functions through deep algebraic structures.
- However, `GraphEvo_Only` performance dropped significantly in the Restricted Pool (73.3% vs 76.2%), indicating that without high-level semantic operators, relying solely on

Table 22: Performance Comparison on Two Spirals Task (Restricted Pool, 100 repetitions)

Strategy	Test Acc ($\pm$ Std)	Pool Size	Time (s)	# $\geq$0.99*	Layers	Nodes
CombinedTrainer_disabled	**0.8386 $\pm$ 0.0390**	21.0	589.3	**1**	6.40	22.09
LayerGrow_Only	0.8019 $\pm$ 0.0355	15.0	26.0	0	5.62	20.69
LayerGrow_Then_GraphEvo	0.7989 $\pm$ 0.0421	17.0	25.2	0	4.91	16.61
CombinedTrainer_once	0.7791 $\pm$ 0.0644	15.9	505.3	0	5.64	20.06
CombinedTrainer_periodic	0.7381 $\pm$ 0.0762	10.0	78.6	0	4.50	14.89
GraphEvo_Only	0.7326 $\pm$ 0.0992	17.9	3.6	0	3.20	8.80

* **High-Precision Success Count**: Number of runs with Test Accuracy $\geq$ 99%.

local node replacement makes it difficult to escape local optima; it must rely on the structural diversity provided by `LayerGrow`.

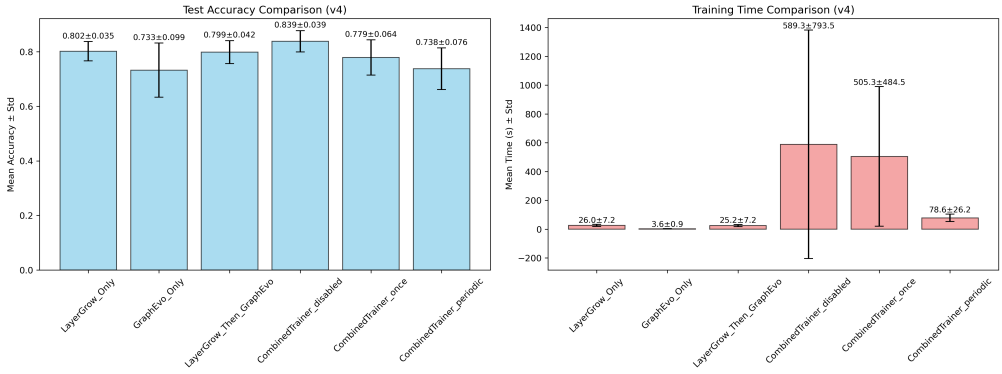
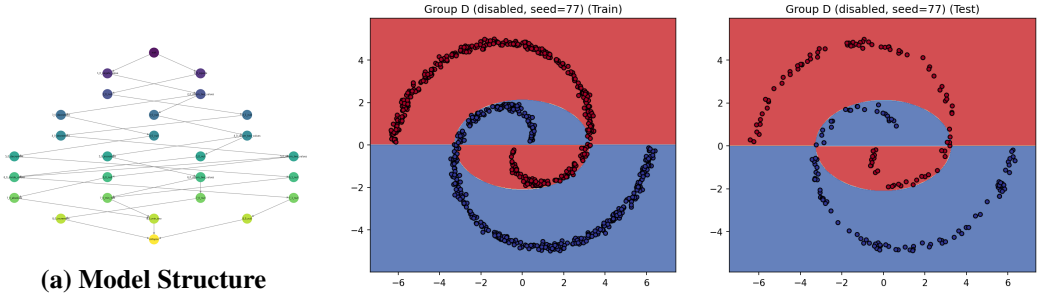


Figure 13: Performance comparison of various strategies on the Two Spirals task (Restricted Method Pool).



(a) Model Structure **(b) Train Set Boundary** **(c) Test Set Boundary**

Figure 14: Structure and decision boundary visualization of the best model from `CombinedTrainer_disabled` on the Two Spirals task (Restricted Pool). Note that its structure is more complex than the Full Pool model, approximating the spiral shape through multi-layer algebraic operations.

D.3 Analysis and Discussion

Based on the experimental results above, we draw the following core conclusions:

- **Method Pool Composition Determines Performance Upper Bound, but Structural Composition Can Compensate for Semantic Deficits**
  - In the Two Spirals task, the Full Pool slightly outperformed the Restricted Pool, verifying the natural advantage of geometric operators like `atan2`/`sin`/`cos` for polar coordinate tasks.

  - However, the gap was not substantial ($\sim$1%), and high-precision solutions were still obtained in the Restricted Pool. This proves that AdaptoFlux's GDFG structural composition capability has strong expressive redundancy, capable of approximating complex geometric functions through deep algebraic structures.

- **Multi-Round Combination (Disabled) Outperforms Single-Round and Over-Screening**
  - `CombinedTrainer_disabled` achieved the best average performance and high-precision success rate in all tasks. Through 3 rounds of "LayerGrow $\rightarrow$ GraphEvo" cycles, the model could gradually deepen the structure and refine semantics, proving the effectiveness of **deep structural exploration**.
  - In contrast, `LayerGrow_Then_GraphEvo` (single-round) was significantly inferior to the multi-round mode due to the depth limit of single growth.
- **Genetic Screening is a Double-Edged Sword: Diversity vs. Efficiency**
  - `CombinedTrainer_periodic` performed the worst in all tasks. Although it significantly reduced the method pool size (to $\sim$10) and shortened single-round time, **over-compression led to the loss of key nonlinear operations (such as `square`, `atan2`) or critical intermediate combinations**.
  - In geometrically sensitive tasks, **preserving method pool diversity is more important than streamlining the search space**. Genetic screening mechanisms should be used with caution; it is recommended to prioritize `disabled` or `once` modes when tasks are highly sensitive to specific atomic operations to avoid premature pruning of the search space.
- **Complementarity of Layered Growth and Graph Evolution**
  - `LayerGrow_Then_GraphEvo` outperformed separate `LayerGrow_Only` or `GraphEvo_Only` in all tasks.
  - `LayerGrow` is responsible for expanding the **macro-topological depth** of the structure, solving the problem of "whether it can be represented"; `GraphEvo` is responsible for **micro-semantic alignment**, solving the problem of "how accurately it is represented". The synergy of the two is the core driving force for AdaptoFlux to navigate discrete spaces.
- **Strong Interpretability and Symbolic Nature**
  - As shown in Figures 10, 12, and 14, the generated GDFG structures are clear and sparse. The mathematical expression of the decision logic can be directly deduced from the model structure graph (e.g., the $x_1^2 + x_2^2$ combination in the Concentric Circles task, or the `atan2` call in the Two Spirals task).
  - This **white-box characteristic** is the essential advantage of AdaptoFlux as a symbolic-structural hybrid paradigm, giving it potential application value in scientific computing and control fields requiring high credibility.

D.4 Conclusion

The experiments in this appendix demonstrate that AdaptoFlux exhibits powerful structural composition capabilities in complex geometric tasks. Its performance depends not only on the **completeness** of the method pool but also on the balance between **structural depth** and **semantic diversity** maintained by the training strategy.

For practical applications, we recommend:

- When exploring unknown complex tasks, prioritize using **`CombinedTrainer_disabled`** to maximize the exploration space.
- If computational resources are limited and the task semantics are known, try **`CombinedTrainer_once`** for moderate compression.
- Avoid using high-frequency periodic genetic screening (`periodic`) in the early stages to prevent destroying the connectivity of the solution space.

E PROOF-OF-CONCEPT FOR STRUCTURAL FEASIBILITY

Consistent with the Concept-and-Feasibility positioning in the main text (Section 2), this appendix is a **structural sanity check** rather than a competitive evaluation. As committed during the review process, AdaptoFlux does not aim to outperform mature, task-specialized symbolic-regression systems (e.g., PySR, gplearn) on static formula discovery; to make this boundary explicit, we deliberately reduce AdaptoFlux to a pure symbolic synthesizer over an identical, minimal primitive set, and ask two questions. Its distinctive capability—native support for side-effecting actions and stateful interactions—is disabled in this section and is validated separately by the automatic stateful synthesis experiment (Section 4.3) and the Snake case study (Appendix F); results here therefore bound only the **static expressiveness of the substrate under the current probes**.

1. Under **identical function primitive sets**, does the GDFG representation retain basic symbolic expressiveness—i.e., can the current probes still recover exact elementary formulas on canonical static tasks?
2. Where do the current probes fail under this restricted setting, and what does this reveal about the boundary between *representation* capacity and *search* capability?

E.1 TASKS AND RESTRICTED EXPERIMENTAL SETTING

As a structural sanity check of basic symbolic expressiveness, this experiment employs eight classic Feynman formulas from physics:

$$f_1(m, c) = m \cdot c^2 \qquad \text{(Energy)}$$
$$f_2(m, a) = m \cdot a \qquad \text{(Newton's Second Law)}$$
$$f_3(m, v) = 0.5 \cdot m \cdot v^2 \qquad \text{(Kinetic Energy)}$$
$$f_4(q_1, q_2, r) = \frac{q_1 q_2}{r^2} \qquad \text{(Coulomb's Law)}$$
$$f_5(n, T, V) = \frac{nT}{V} \qquad \text{(Ideal Gas Law)}$$
$$f_6(I, R) = I \cdot R \qquad \text{(Ohm's Law)}$$
$$f_7(v_1, v_2, t) = \frac{v_2 - v_1}{t} \qquad \text{(Acceleration Definition)}$$
$$f_8(V, I) = V \cdot I \qquad \text{(Electric Power)}$$

To keep the comparison meaningful at the level of pure expressiveness, all systems are strictly limited to the same four basic mathematical operators: `add`, `sub`, `mul`, `div`. AdaptoFlux additionally introduces two structural auxiliary operations: `fanout` (variable copying) and `identity` (passthrough). These are dictated by its underlying Directed Acyclic Graph (DAG) dataflow model—when an input variable is consumed by multiple nodes, it must be explicitly copied. In contrast, GP or PySR, based on trees or expressions, naturally allow multiple references to the same variable without such operations. Therefore, these two auxiliary methods do not expand functional expressiveness but merely maintain graph structure validity, preserving the fairness of the symbolic regression task.

Deliberately Restricted Setting (a sanity check, not a race): Because gplearn and PySR are pure-symbolic tools that cannot exercise AdaptoFlux's method imports, we **actively disabled ATF's method-import capability**, reducing it to a pure symbolic synthesizer using only 5 basic operators. This isolates the expressiveness of the representation itself (graph structure vs. tree/expression) from differences in function sets, rather than staging a head-to-head comparison of search engines. The experiment therefore probes a lower bound on what the GDFG substrate can express under the current probes; it is not the operating mode for which AdaptoFlux is designed.

E.2 EXPERIMENTAL SETUP

All methods were run independently 10 times on each task with different random seeds. This experiment reports the **best-of-10 MSE, Exact Match status (Exact?), runtime (seconds), and final method pool size**. AdaptoFlux was tested with three different `collapse_mode` strategies:

`first` First valid output;

`prod` Product aggregation;

`sum` Sum aggregation.

Baseline methods include:

gplearn Classic Genetic Programming implementation, using the same 5-function set;

PySR Modern symbolic regression tool based on sparse regression and symbolic enumeration;

XGBoost Black-box gradient boosting tree model, used to contrast pure predictive performance with interpretability trade-offs.

To more accurately assess **program equivalence** in symbolic regression tasks, this experiment adopts a **numerical-symbolic joint discrimination strategy**. Specifically, we first determine if the model output is numerically highly consistent with the ground truth using the following function:

```
def is_numerical_exact_match(y_pred, y_true, tol=1e-5):
    return np.all(np.abs(y_pred - y_true) < tol)
```

This function requires the prediction error for all test samples to be strictly less than 10^{-5}. However, relying solely on MSE or numerical matching may yield **false positives** (e.g., the model fits an expression that looks similar but is structurally incorrect) or **false negatives** (e.g., high MSE due to constant offset but correct structure). Therefore, in addition to automatic judgment, we further performed **manual symbolic parsing verification** on the **best expressions** output by ATF, gplearn, and PySR to confirm whether they are **algebraically equivalent** to the target formula (e.g., $x_0 \cdot x_1 \cdot x_1$ is equivalent to $m \cdot c^2$ under variable mapping).

Based on this dual verification mechanism, we constructed Table 28, which specifically records whether each method **truly recovered the target physical formula at the symbolic level**.

E.3 Results Overview

Tables 23–25 present the best-of-10 performance of AdaptoFlux under the three collapse modes; Table 26 summarizes the corresponding results for gplearn, PySR, and XGBoost; and Table 28 reports the symbolic-level exact match analysis. Best-of-10 tables are retained only because this is how the reference systems are conventionally reported on this suite; stability-oriented conclusions rely on the mean $\pm$ std statistics in Tables 29–31 and Figures 16–17.

Key findings are as follows:

- **gplearn and PySR achieved exact matches (Exact?=Yes) on most tasks**, with MSE typically below 10^{-12}, performing robustly especially in binary multiplication or simple division tasks (e.g., F_ma, OhmsLaw, Acceleration, Power, Coulomb)—as expected for systems specialized in exactly this static setting.
- **XGBoost failed to recover analytical expressions**, with MSE generally 2–4 orders of magnitude higher, highlighting its fundamental limitations in symbolic recovery tasks.
- **AdaptoFlux's performance is highly dependent on the collapse mode**:
  - The `prod` mode performed best in multiplication-dominated tasks (F_ma, OhmsLaw, Power, E_mc2, KE), achieving exact matches (MSE $\sim 10^{-13}$) for F_ma, OhmsLaw, and Power, comparable to gplearn/PySR;
  - The `first` mode succeeded in some tasks (e.g., F_ma) but exhibited poorer overall stability;
  - The `sum` mode generally performed worst, barely reaching high precision only in OhmsLaw and Power.
- **Complex tasks remain a bottleneck for the current probes**: for tasks involving explicit constants (e.g., 0.5 in KE) or three-variable nonlinear combinations (IdealGas, Coulomb), the restricted ATF failed to recover the exact formula, with MSE significantly higher than the specialized reference systems. As analyzed below, this reflects the limits of the lightweight local-search probes under a fixed uniform configuration rather than a representational deficit.

- **Native action support is not exercised here**: the "Supports actions" markers in the tables record only that the execution backend accepts side-effecting methods; that capability is deliberately disabled in this section and is demonstrated separately by the automatic stateful synthesis experiment (Section 4.3) and the Snake case study (Appendix F).

We tested three collapse modes, with `prod` performing best in multiplication-dominated tasks; thus, subsequent analysis focuses primarily on this mode.

Table 23: Parameter-free symbolic regression with collapse=first (best of 10 runs)

Task	Exact?	MSE	Runtime (s)	Method Pool	Notes
E_mc2	No	2e+03	4.0	5	Supports actions
F_ma	**Yes**	3e-12	0.3	5	Supports actions
KE	No	4e+03	42.0	8	Supports actions
Coulomb	No	1e+01	13.1	7	Supports actions
IdealGas	No	3e+04	11.0	7	Supports actions
OhmsLaw	**Yes**	5e-13	0.4	5	Supports actions
Acceleration	No	7e+00	10.7	7	Supports actions
Power	**Yes**	1e-12	0.4	5	Supports actions

Table 24: AdaptoFlux performance with `collapse=prod` (best of 10 runs). The "Exact?" column indicates **numerical exact match**: all absolute errors on the test set are $< 10^{-5}$. This is stricter than MSE $< 10^{-5}$ and does not imply symbolic equivalence.

Task	Exact?	MSE	Runtime (s)	Method Pool	Notes
E_mc2	No	2e-11	0.1	5	Supports actions
F_ma	**Yes**	2e-13	0.0	5	Supports actions
KE	No	2e-11	0.1	5	Supports actions
Coulomb	No	9e+00	6.0	5	Supports actions
IdealGas	No	2e+04	19.8	8	Supports actions
OhmsLaw	**Yes**	2e-13	0.0	5	Supports actions
Acceleration	No	5e+00	11.2	7	Supports actions
Power	**Yes**	8e-13	0.0	5	Supports actions

Table 25: Parameter-free symbolic regression with collapse=sum (best of 10 runs)

Task	Exact?	MSE	Runtime (s)	Method Pool	Notes
E_mc2	No	8e+00	532.1	8	Supports actions
F_ma	No	4e+01	6.4	7	Supports actions
KE	No	8e+02	39.9	8	Supports actions
Coulomb	No	1e+01	8.6	6	Supports actions
IdealGas	No	3e+04	2.2	5	Supports actions
OhmsLaw	**Yes**	1e-12	0.5	5	Supports actions
Acceleration	No	1e+01	9.7	6	Supports actions
Power	**Yes**	2e-12	1.6	5	Supports actions

To provide a more comprehensive view of performance stability beyond the "best-of-10" metrics reported in Tables 23–28, we visualize the full distribution of MSE and exact match success rates across all 10 independent runs in Figure 16 and Figure 17.

Figure 16 highlights the vast difference in convergence quality between simple multiplication tasks (e.g., `F_ma`) and complex multi-variable tasks (e.g., `IdealGas`). Notably, the `prod` mode exhibits extremely low variance in multiplication-dominated tasks, confirming that its superior performance is consistent rather than an outlier. Conversely, for complex tasks like `IdealGas` and `Coulomb`, all modes exhibit high variance and elevated MSE, indicating that the current search strategy struggles to consistently converge to the correct symbolic structure in high-dimensional spaces.

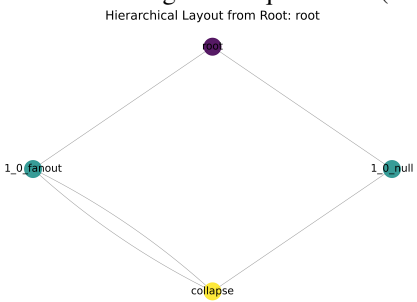
Table 26: Baseline performance of gplearn, PySR, and XGBoost (best of 10 runs)

Task	Method	Exact?	MSE	Runtime (s)
E_mc2	Gplearn	**Yes**	8.3e-12	17.86
	PySR	**Yes**	8.3e-12	20.29
	Xgboost	No	1.5e+02	0.04
F_ma	Gplearn	**Yes**	1.9e-13	16.40
	PySR	**Yes**	2.1e-12	20.49
	Xgboost	No	2.5e+01	724.38
KE	Gplearn	No	3.4e-11	72.71
	PySR	No	3.4e-11	20.31
	Xgboost	No	3.9e+02	0.03
Coulomb	Gplearn	**Yes**	6.8e-14	10.49
	PySR	**Yes**	2.7e-13	20.46
	Xgboost	No	6.4	0.02
IdealGas	Gplearn	No	1e-10	122.72
	PySR	No	2.2e-10	20.32
	Xgboost	No	4.5e+03	0.05
OhmsLaw	Gplearn	**Yes**	2.2e-13	9.24
	PySR	**Yes**	2.2e-13	20.40
	Xgboost	No	9.6	0.03
Acceleration	Gplearn	**Yes**	3.9e-14	28.84
	PySR	**Yes**	1.1e-13	20.46
	Xgboost	No	1.7	0.04
Power	Gplearn	**Yes**	8e-13	116.68
	PySR	**Yes**	1e-12	20.20
	Xgboost	No	3.1e+01	0.03

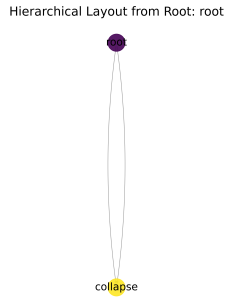
Table 27: Symbolic expressions recovered by gplearn and PySR

Task	Method	Expression	Exact?
E_mc2	gplearn	`mul(mul(X1, X0), X1)`	**Yes**
	PySR	`x0*x1*x1`	**Yes**
F_ma	gplearn	`mul(X0, X1)`	**Yes**
	PySR	`x0*x1 - 0.027047554 - 1*(-0.027047586)`	**No**
KE	gplearn	`mul(mul(X2, X1), mul(X0, X1))`	**Yes**
	PySR	`x0*x1*x1*x2`	**Yes**
Coulomb	gplearn	`div(mul(div(X1, X2), X0), X2)`	**Yes**
	PySR	`x0/((x2/((x1/x2))))`	**Yes**
IdealGas	gplearn	`div(X1, div(X2, X0))`	**Yes**
	PySR	`x0*x1/x2`	**Yes**
OhmsLaw	gplearn	`mul(X0, X1)`	**Yes**
	PySR	`x0*x1 - (-x0 + x0 - x1 + x1)`	**Yes**
Acceleration	gplearn	`div(sub(X1, X0), X2)`	**Yes**
	PySR	`(-x0 + x1 - 6.078859e-6/(x2/((x2/(-16.520508)))))/x2`	**No**
Power	gplearn	`mul(X1, X0)`	**Yes**
	PySR	`x1*(x0 - 1*0.99999976) + x1`	**No**

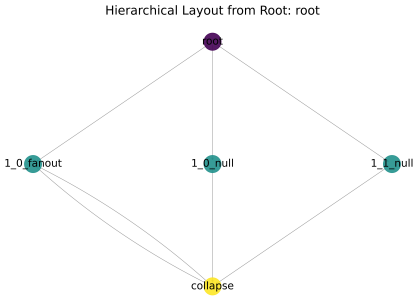
Figure 15: Best computation graphs generated by AdaptoFlux on eight Feynman formula tasks. Yes/No indicates algebraic equivalence (see Table 28).



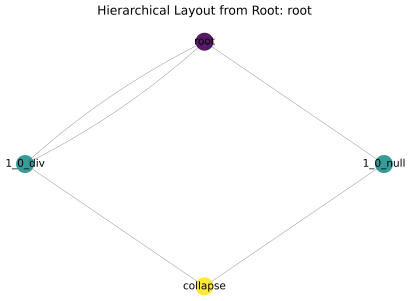
(a) E_mc2 (Yes)



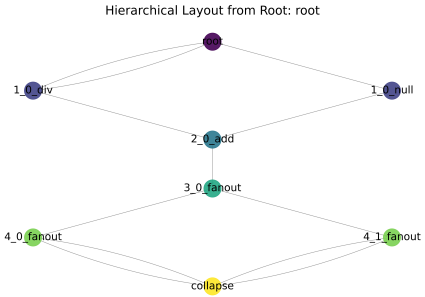
(b) F_ma (Yes)



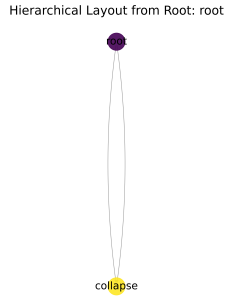
(c) KE (Yes)



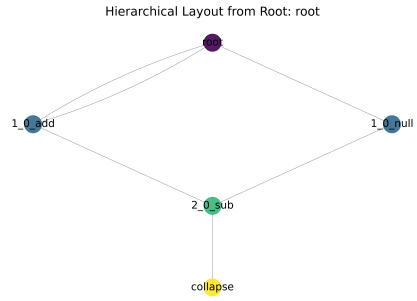
(d) Coulomb (No)



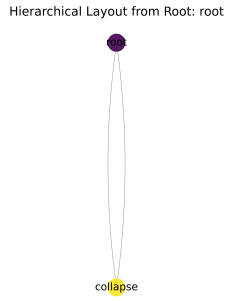
(e) Ideal Gas (No)



(f) Ohm's Law (Yes)



(g) Acceleration (No)



(h) Power (Yes)

Table 28: Symbolic exact match analysis: whether the best expression is **algebraically equivalent** to the ground-truth formula, as determined by manual inspection. This is stricter than numerical matching.

Task	AdaptoFlux (prod)	gplearn	PySR
E_mc2	**Yes**	**Yes**	**Yes**
F_ma	**Yes**	**Yes**	**No**
KE	**Yes**	**Yes**	**Yes**
Coulomb	No	**Yes**	**Yes**
IdealGas	No	**Yes**	**Yes**
OhmsLaw	**Yes**	**Yes**	**Yes**
Acceleration	No	**Yes**	**No**
Power	**Yes**	**Yes**	**No**

Figure 17 quantifies the stability of symbolic recovery. It clearly demonstrates that the `prod` mode achieves a 100% exact match success rate on tasks like `F_ma`, `OhmsLaw`, and `Power`, significantly outperforming `first` and `sum` modes in terms of reliability. This visual evidence reinforces the conclusion that the choice of collapse mode is critical for aligning with the mathematical structure of the target task.

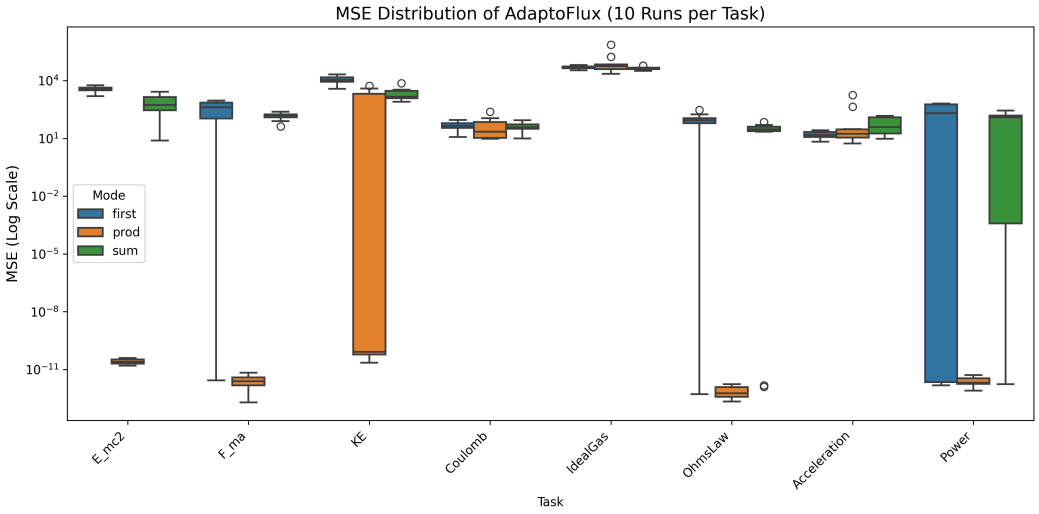


Figure 16: Distribution of Mean Squared Error (MSE) for AdaptoFlux across 8 Feynman tasks over 10 independent runs. The log-scale y-axis emphasizes the stability of the `prod` mode in multiplication tasks versus the high variance in complex tasks.

Table 29: AdaptoFlux performance with collapse=`first` over 10 independent runs. Values report Mean ± Std for MSE and Runtime, along with Median, Min MSE, and Exact Match Success Rate.

Task	MSE (Mean ± Std)	Median	Min	Success Rate (%)	Runtime (s) ± Std
E_mc2	$3.50e+03 \pm 1.21e+03$	$3.47e+03$	$1.55e+03$	0.0	3.10 ± 1.75
F_ma	$4.31e+02 \pm 3.28e+02$	$4.13e+02$	$2.77e-12$	20.0	3.39 ± 6.96
KE	$1.16e+04 \pm 5.28e+03$	$1.09e+04$	$3.71e+03$	0.0	10.14 ± 11.53
Coulomb	$4.78e+01 \pm 2.11e+01$	$4.49e+01$	$1.19e+01$	0.0	7.90 ± 8.58
IdealGas	$4.87e+04 \pm 8.04e+03$	$4.70e+04$	$3.47e+04$	0.0	7.44 ± 4.85
OhmsLaw	$1.04e+02 \pm 7.77e+01$	$8.71e+01$	$5.28e-13$	10.0	1.51 ± 1.72
Acceleration	$1.58e+01 \pm 6.16e+00$	$1.49e+01$	$6.68e+00$	0.0	2.41 ± 3.23
Power	$2.81e+02 \pm 2.88e+02$	$2.05e+02$	$1.49e-12$	50.0	0.54 ± 0.30

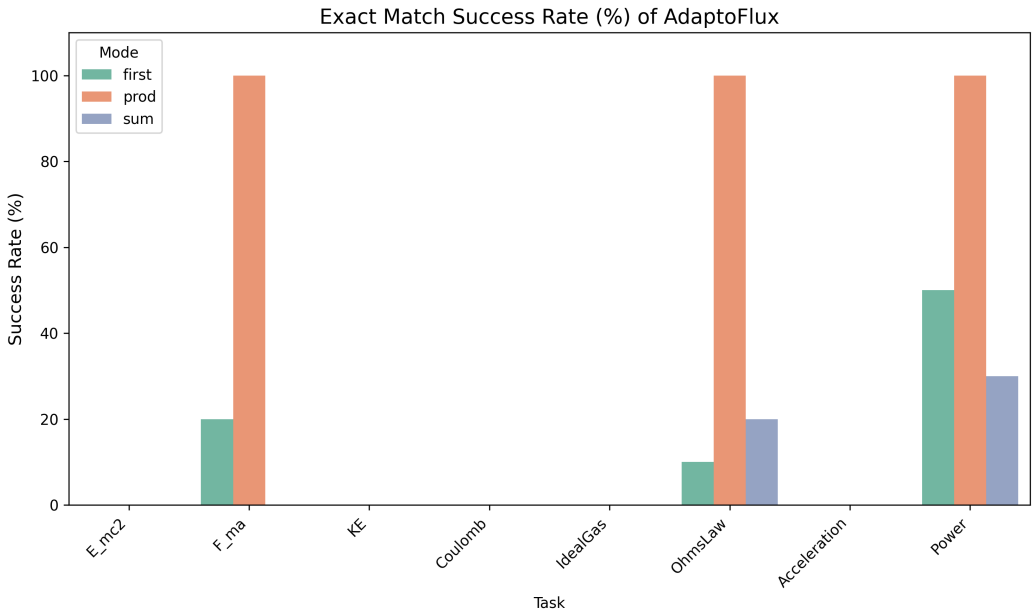


Figure 17: Exact Symbolic Match Success Rate (%) for AdaptoFlux across different collapse modes. The `prod` mode demonstrates robust stability (100% success) on multiplication-dominated tasks, whereas other modes show significant inconsistency.

Table 30: AdaptoFlux performance with collapse=`prod` over 10 independent runs. This mode demonstrates the highest stability and success rate in multiplication-dominated tasks (F_ma, OhmsLaw, Power).

Task	MSE (Mean ± Std)	Median	Min	Success Rate (%)	Runtime (s) ± Std
E_mc2	$2.64e-11 \pm 7.80e-12$	$2.53e-11$	$1.59e-11$	0.0	0.07 ± 0.01
F_ma	$2.88e-12 \pm 1.96e-12$	$2.42e-12$	$1.93e-13$	100.0	0.02 ± 0.00
KE	$1.31e+03 \pm 1.86e+03$	$8.29e-11$	$2.27e-11$	0.0	2.61 ± 3.14
Coulomb	$5.65e+01 \pm 6.85e+01$	$2.25e+01$	$9.40e+00$	0.0	9.20 ± 3.30
IdealGas	$1.28e+05 \pm 2.04e+05$	$5.55e+04$	$2.25e+04$	0.0	15.17 ± 7.40
OhmsLaw	$8.04e-13 \pm 5.15e-13$	$5.89e-13$	$2.21e-13$	100.0	0.02 ± 0.00
Acceleration	$2.36e+02 \pm 5.36e+02$	$1.73e+01$	$5.38e+00$	0.0	11.62 ± 4.05
Power	$2.65e-12 \pm 1.41e-12$	$2.07e-12$	$8.05e-13$	100.0	0.02 ± 0.00

Table 31: AdaptoFlux performance with collapse=`sum` over 10 independent runs. Generally exhibits higher variance and lower success rates compared to `prod`, particularly in complex tasks.

Task	MSE (Mean ± Std)	Median	Min	Success Rate (%)	Runtime (s) ± Std
E_mc2	$8.84e+02 \pm 7.94e+02$	$5.33e+02$	$7.73e+00$	0.0	154.79 ± 201.62
F_ma	$1.50e+02 \pm 5.70e+01$	$1.57e+02$	$4.11e+01$	0.0	12.78 ± 20.35
KE	$2.33e+03 \pm 1.88e+03$	$1.44e+03$	$7.84e+02$	0.0	159.76 ± 153.73
Coulomb	$4.27e+01 \pm 2.15e+01$	$3.81e+01$	$9.73e+00$	0.0	7.52 ± 5.12
IdealGas	$4.38e+04 \pm 9.34e+03$	$4.21e+04$	$3.14e+04$	0.0	11.50 ± 19.96
OhmsLaw	$3.06e+01 \pm 1.99e+01$	$2.98e+01$	$1.26e-12$	20.0	4.82 ± 3.52
Acceleration	$6.52e+01 \pm 5.45e+01$	$3.84e+01$	$9.58e+00$	0.0	19.94 ± 40.29
Power	$1.01e+02 \pm 9.30e+01$	$1.22e+02$	$1.76e-12$	30.0	25.39 ± 42.22

E.4 Analysis of Core Findings

1. **Under the restricted setting, the substrate recovered exact symbolic structures in multiplication-dominated tasks**: for F_ma, OhmsLaw, Power, E_mc2, and KE, the `prod` mode not only reached high numerical precision (MSE $\sim 10^{-13}$–10^{-11}) but also produced computation graphs that were **algebraically equivalent** to the target formulas under symbolic verification (Table 28), matching the specialized reference systems on these tasks. This demonstrates that the GDFG representation retains basic symbolic expressiveness—it can synthesize correct program structure, including formulas with explicit constants—even when stripped of method imports and driven by a lightweight probe.

2. **ATF can correctly combine input variables containing constants, but its performance is highly sensitive to search configuration**: In the Kinetic Energy (KE) task, where the constant 0.5 in the target formula $0.5 \cdot m \cdot v^2$ is provided as an independent input dimension, AdaptoFlux successfully constructed the correct multiplication chain ($x_2 \cdot x_0 \cdot x_1 \cdot x_1$) in `prod` mode, achieving algebraic equivalence (Table 28); whereas it failed completely in `sum` or `first` modes (Tables 25, 23). This contrast clearly indicates that **AdaptoFlux's search effectiveness strongly depends on configurations aligned with the mathematical structure of the task**. For three-variable nonlinear tasks like IdealGas and Coulomb, ATF has not yet recovered the correct symbolic structure under the current uniform `prod` mode setting; however, manual inspection of its best solutions reveals that the output programs often miss only one or two key operations (such as a division or variable copy). This suggests that success rates could be significantly improved by introducing more flexible collapse strategies, increasing graph depth, or expanding method pool capacity—but to maintain fair comparison with baselines, this experiment did not adjust hyperparameters or search strategies for specific tasks.

3. **The choice of Collapse mode is critical**: The `prod` mode performed excellently in multiplication-like tasks, while `first` and `sum` modes were generally poor. This reveals a key design principle of AdaptoFlux: the output aggregation strategy must align with the mathematical structure of the target task.

4. **Interpretability and capability boundaries**: even where the restricted ATF underperformed in MSE, its program graphs (Fig. 15) remained complete, traceable decision chains. The substrate is also *designed* so that pure computation and side-effecting actions coexist in one executable graph—a capability the pure-symbolic reference systems do not offer—but, again, that capability is outside the scope of this deliberately static section and is validated in Section 4.3 and Appendix F.

E.5 Conclusion

This experiment is a **proof-of-concept of structural feasibility**, not a benchmark ranking: it checks whether the GDFG substrate retains basic symbolic expressiveness under a deliberately restricted method pool and identical primitive sets, not whether AdaptoFlux outperforms specialized solvers on static formula discovery. Under this restricted setting, the substrate—driven by a lightweight probe with a `prod` collapse—exactly recovered the multiplication-dominated formulas (F_ma, OhmsLaw, Power, E_mc2, KE) with high stability and low variance over the 10 independent runs (Tables 29–31; Figures 16–17). By contrast, the harder multi-variable formulas (Coulomb, IdealGas, Acceleration) were not recovered under this restricted, uniformly configured setting, although the specialized reference systems did recover them (gplearn on all three; PySR on Coulomb and IdealGas). As stated in the main text (Section 2), this outcome is expected and consistent with our Concept-and-Feasibility positioning: AdaptoFlux is not intended to outperform mature, task-specialized symbolic-regression systems on static benchmarks, and these failures reflect the limits of the current lightweight probes rather than a representational deficit.

Scope of this section: because side-effecting imports are deliberately disabled, no conclusions about AdaptoFlux's stateful capability should be drawn from the static results above. The intended operating regime—mixing pure computation with side-effecting actions—is validated by the automatic stateful synthesis experiment (Section 4.3) and the hand-crafted Snake case study (Appendix F), which together establish the substrate's structural integrity as a foundation for future state-aware search strategies.

F ZERO-SHOT EMBODIED INTELLIGENCE: HAND-CRAFTED SNAKE CONTROLLER

To verify AdaptoFlux's native support for side-effectful actions, we manually constructed a Generalized Data Flow Graph (GDFG) to control an agent in the classic Snake game. This entire process required no training or parameter tuning.

As shown in Figure 18, the graph consists of four logical layers:

- **Input Extraction Layer**: Reads the current game state (e.g., snake head position, food coordinates, presence of hazards ahead);
- **Perception and Reasoning Layer**: Computes spatial relationships (e.g., "hazard ahead," "food to the right");
- **Decision Logic Layer**: Executes symbolic rules based on pure functions from the method pool (e.g., "if hazard ahead, then turn left");
- **Action Execution Layer**: Invokes methods with side effects (such as `turn_left()`, `move_forward()`), directly modifying the environment state via `SNAKE_STATE.set_action()`.

The key aspect is that all action nodes belong to the *environmental side-effect* category; their execution triggers real state changes within the game engine. The entire graph structure is static and interpretable, requiring only a single forward pass per time step to complete execution.

Although this controller is simple in structure and unoptimized, it operates stably for hundreds of steps in a 10×10 grid, effectively avoiding self-collision and continuously tracking food. This result validates the core claim of AdaptoFlux: **it provides a unified and executable underlying framework for both pure computation and interactive behavior**—a capability not possessed by pure symbolic regression systems such as gplearn or PySR. We have provided the running video of this model in the supplementary materials.

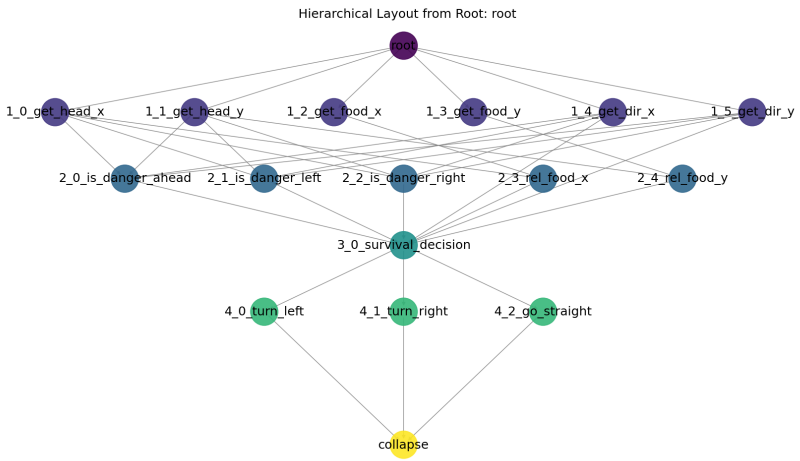


Figure 18: Hand-crafted GDFG for the Snake agent. Nodes with red borders in the action layer execute operations with side effects (such as `turn_left`), directly modifying the environment state. The figure clearly demonstrates AdaptoFlux's unique ability to uniformly model pure functions and interactive actions within an executable graph.

G IMPLEMENTATION DETAILS AND REPRODUCIBILITY GUIDE

Positioning: Main text Appendix C retains only core protocol parameters for the five categories of experiments. Complete hyperparameter grids, full method pool lists, random seed strategies, and raw logs are hard-coded in the anonymous code repository. This section provides repository path

mapping, complete engineering implementation of method registration interfaces, and a one-click reproduction guide for deep verification by reviewers.

G.1 Method Pool Interface and Decorator Implementation

The formal definition $m = \langle \text{impl}, \sigma_{\text{in}}, \sigma_{\text{out}}, \mathcal{M} \rangle$ from Main Text § 3.1 is bound in the prototype system via the `@method_profile` decorator. This mechanism transforms host language functions into atomic units parseable by the framework, automatically injecting type routing and metadata:

```python
def method_profile(
    output_count: int | None = None,
    input_types=None,
    output_types=None,
    group: str = "default",
    weight: float = 1.0,
    is_internal_decorator: bool = False,
    vectorized: bool = False
):
    def decorator(func):
        func.output_count = output_count if output_count else len(
            output_types or [])
        func.input_types = input_types or []
        func.output_types = output_types or []
        func.group = group
        func.weight = weight
        func.is_internal_decorator = is_internal_decorator
        func.vectorized = vectorized
        return func
    return decorator
method_profile.is_internal_decorator = True
```

Listing 2: Full Implementation of @method_profile Decorator

Mapping Relationship: Attributes injected by the decorator directly correspond to the formal tuple $\mathcal{M}$. When constructing graph edges, the framework enforces hard type-constrained routing by comparing `output_types` with downstream nodes' `input_types`; `group` and `weight` drive dynamic view projection and probe sampling, and neither participates in the definition of Ω (Appendix H.4).

G.2 Experiment Configuration and Repository Path Mapping

To ensure reproducibility transparency, all experiments adopt a self-contained directory structure. Table 32 establishes a precise index between main text experiments and repository assets.

Path Note: All scripts are located in the `experiments/` directory of the anonymous repository. The table below lists relative paths and core output files.

Note: All paths in the table are relative to the anonymous code repository root. Output artifact types vary by task, including: run logs and trajectories (`.log`/`.json`), statistical metric tables (`.csv`), execution demo videos (`.mp4`), and graph configuration files. The complete directory structure, environment dependencies, and one-click reproduction guides are detailed in the `README.md` file in the repository or supplementary materials.

Raw result tables—the LayerGrow configuration summary, the knowledge-transfer summary, the parallel-scheduling benchmark, the Feynman-baseline results and the minimal-stateful run logs—are shipped, so that the reported statistics can be recomputed independently. Per-run logs and model/graph dumps (`exp_max_layers_*/run_*/training_log.json`, `exp3_results/<config>/rep<k>/`, `models/`, `plots/`), together with the three `experiment_results/` directories of the geometric-classification ablation (≈0.5 GB each), are excluded for size and are regenerated by the corresponding scripts.

Table 32: Experiment Configuration and Repository Path Mapping

LayerGrow (Sec. 4.1, Appx. C.1)
`LayerGrowTrainer_exp1/main.py`
→ `experiment_results_summary.csv`

Knowledge Transfer (Sec. 4.2, Appx. C.2)
`GraphEvo_exp3/exp3_main.py`
→ `exp3_results/summary_table.csv`

Minimal Stateful Attractor (Sec. 4.3, Appx. C.4)
`minimal_stateful_comparison/run_comparison.py`
→ `logs/` + `results/stateful_synthesis_comparison.png`

Topology-Aware Parallel (Appx. C.3)
`PipelineParallel_exp2/scripts/benchmark.py`
→ `PipelineParallel_exp2/results/`

Geometric Classification (Appx. D)
- `Combined_exp4_circles/test_combined_trainer.py`
- `Combined_exp4_spiral/test_combined_trainer.py`
- `Combined_exp4_spiral_no_Trigonometric/test_combined_trainer.py`

→ `experiment_results/` (regenerated by the script; ≈0.5 GB per task, not shipped)

Feynman Regression (Appx. E)
`ExternalBaselines/run_baselines_*.py`
→ `ExternalBaselines/results/`

Snake Controller (Appx. F)
`embodied_snake/snake_main.py` + `snake.json`[†]
→ `snake_demo.mp4`

[†]Note: `snake.json` is a hand-crafted GDFG configuration file (not learned).

G.3 Reproduction Environment and Running Guide

Software Dependencies: AdaptoFlux is developed based on Python 3.12+. The complete dependency list is in `requirements.txt` at the repository root, installable via `pip install -r requirements.txt`. The framework executes entirely on CPU, with no GPU dependencies. **Compute Resources:** All experiments were executed on standard workstations with multi-core CPUs and 60GB RAM. No GPU was used. Specifically, the parallel scalability experiments utilized a server with 32 CPU cores to evaluate multi-threading performance. The total wall-clock time to reproduce all main experimental results sequentially is estimated to be less than one week. **Statistical Protocol Alignment:** The random seed allocation, repetition counts, and success determination thresholds for each experiment strictly follow the regulations outlined in the Main Text and Appendices. All raw logs have been fully exported, supporting independent recalculation of means, variances, and success rates.

H Semantic Taxonomy of the Method Pool

Main Text Section 3.1 introduced the engineering implementation of the Method Pool $\mathcal{Q}$ and its metadata-driven architecture. This section further provides two typical classification examples from a semantic perspective to demonstrate the flexibility and expressive potential of the method pool in handling heterogeneous operations. It is emphasized that the following classification systems serve only as **Illustrative Examples**, aiming to elucidate how the framework establishes structural order through metadata, rather than imposing hard constraints on the method pool. AdaptoFlux natively supports users defining arbitrary orthogonal classification views based on task requirements (e.g., by hardware affinity, computational complexity, or semantic role).

H.1 Example 1: Heterogeneous Operation Classification Based on Functional Attributes

To natively bridge pure computation and state interaction, AdaptoFlux's method pool $\mathcal{Q}$ supports semantically tagging methods from two orthogonal functional dimensions: **Mapping** and **Action**. It is emphasized that these two dimensions are not mutually exclusive sets but allow overlapping feature labels—a method can simultaneously possess deterministic input-output mapping relationships and side effects that change system state or environment. This flexibility allows the framework to uniformly model a wide range of operations from pure mathematical formulas to embodied control instructions.

Mapping Perspective ($\mathcal{Q}_F$) Focuses on the input-output transformation relationship f : Input $\rightarrow$ Output. Such methods provide deterministic computation under type safety guarantees, forming the basis for symbolic regression and feature extraction. Even if a method contains side effects, its mapping part can still be independently analyzed for data flow reasoning.

Action Perspective ($\mathcal{Q}_O$) Focuses on the effect produced on the system state $\mathbf{s}_{t+1} = \Phi(\mathbf{s}_t, \mathbf{x}_t)$ or the external environment after method execution. Such methods introduce time dependency and state evolution capabilities, forming the core of building dynamic systems and embodied agents.

Example Comparison and Overlap Cases

- **Pure Mapping Example**: $f(x) = x \times \frac{9}{5} + 32$ (Celsius to Fahrenheit). No state dependency, no side effects, only returns calculation results.
- **Pure Action Example**: `send_notification(msg)`. The primary purpose is to execute I/O operations; the return value is usually secondary or empty, with the core being the triggered external effect.
- **Mixed Case Example**: `update_and_return_score(state, action)`. This method both calculates a numerical score based on current state and action (mapping characteristic) and records the action in the historical trajectory while updating the environment state (action characteristic). AdaptoFlux allows such methods to simultaneously assume the dual roles of data transmission and state evolution in the graph.

H.1.1 Subdivision of Action Pool by Side-Effect Scope

The action pool $\mathcal{Q}_O$ can be further subdivided based on the object affected by side effects (see Table 33). This classification determines the state update strategy of the graph execution engine during time-step unrolling.

Table 33: [Example] Action Pool Classification by Side-Effect Scope

Type	Affects Self-State	Affects External Env	Typical Application
Self Side-Effect	✓	×	Internal counter updates, memory storage
Env Side-Effect	×	✓	API calls, robot movement commands
Dual Side-Effect	✓	✓	Perception-action loops in embodied intelligence

H.2 Example 2: Reversibility Classification Based on Information Flow Attributes

For the function pool $\mathcal{Q}_F$, we can classify methods based on the ability to restore inputs from outputs. This perspective demonstrates the theoretical potential of the method pool in supporting reverse reasoning or lossless encoding tasks.

Fully Reversible Function Pool Example Based on the Fundamental Theorem of Arithmetic, construct the function $f(a, b) = 2^a \cdot 3^b$ $(a, b \in \mathbb{N})$. Due to the uniqueness of prime factorization,

Table 34: Function Pool Classification by Reversibility

Type	Characteristic	Reverse Deduction Ability
Non-Reversible (N)	Many-to-One Mapping	Cannot uniquely restore; only probabilistic inference
Conditionally Reversible (CR)	Requires partial input or context	Can uniquely restore remaining inputs given partial inputs
Fully Reversible (FR)	One-to-One Mapping (Bijective)	Can uniquely restore original inputs solely from output

any output y can be uniquely inverted to (a, b). Such functions have theoretical value in scenarios requiring lossless state recovery or encoding/decoding, demonstrating AdaptoFlux's ability to handle complex mathematical properties.

H.3 Dynamic Construction and Task-Adaptive Filtering

The above classifications are not static labels but the foundation for dynamic filtering. AdaptoFlux allows dynamically constructing sub-pools through metadata projection. This is a search-side restriction that leaves Ω unchanged (Appendix H.4).

H.3.1 Method Registration and Global State

Methods are registered via the `@method_profile` decorator and can be associated with global state storage (such as memory dictionaries). This allows action functions to share information across time steps, achieving long-term memory without hard-coding state variables.

H.3.2 Task-Adaptive Sub-Pool Filtering Mechanism

Facing large-scale base method pools, the search space grows exponentially. To improve efficiency, subset filtering mechanisms can be introduced. Here we provide a simple example using a Genetic Algorithm (GA) to optimize method subsets:

1. **Initialization**: Randomly sample multiple fixed-size method subsets.
2. **Evaluation**: Run lightweight `LayerGrow` on each subset, using validation set accuracy as fitness.
3. **Evolution**: Iteratively optimize through selection, crossover, and mutation, eliminating redundant or interfering operators.

This process aims to find a "semantically complete and minimal" sub-pool. The performance fluctuation of `CombinedTrainer_periodic` in Appendix D reflects this trade-off: excessive compression may lead to the loss of key semantic primitives (such as trigonometric functions), thereby causing semantic collapse of the expression space. This further confirms that the composition of the method pool has a decisive impact on navigation efficiency.

H.4 Hypothesis Class and the Formal Role of Metadata

Hard constraints define the hypothesis class. Fix a registered pool Q and a type system Σ. The hypothesis class is exactly Eq. 1: acyclic graphs whose processing nodes bind methods of Q and whose edges satisfy $\sigma_{\text{out}}(u) \cap \sigma_{\text{in}}(v) \neq \emptyset$. The metadata set M does not occur in this definition. Type compatibility and acyclicity are therefore *hard*: a violation yields a graph the engine cannot execute, and the candidate is rejected before any evaluation.

Metadata defines a search-side view. For a filter ϕ, let $\mathrm{Q}_{\text{view}}(\phi) = \{m \in \mathrm{Q} \mid \phi(\mathrm{M}_m) = \text{True}\}$. A probe using ϕ explores $\Omega_{\mathrm{Q}_{\text{view}}(\phi),\Sigma}$, i.e. Eq. 1 with Q replaced by Q_{view}. Since $\mathrm{Q}_{\text{view}} \subseteq \mathrm{Q}$, we have $\Omega_{\mathrm{Q}_{\text{view}},\Sigma} \subseteq \Omega_{\mathrm{Q},\Sigma}$ for every ϕ, and $\phi \equiv \text{True}$ recovers the full space. Metadata can therefore only *remove* candidates from what a probe examines: it can never make a non-executable graph

executable, and a probe is free to ignore it entirely without leaving the formalism. This is the precise sense in which metadata is a *control interface* rather than part of the representation.

Operational discriminator. Removing a hard constraint yields graphs the engine cannot run, i.e. the space changes; removing a soft one merely widens the set of candidates a probe is willing to consider, i.e. the space is unchanged. Table 35 lists each mechanism, its formal position, and where it is exercised.

Table 35: Hard (space-defining) versus soft (search-side) constraints.

Mechanism	Formal position	Determines	If removed / violated	Exercised in
Type signatures $\sigma_{\text{in/out}}$	inside Ω	edge executability	candidate rejected before evaluation	§4.3 ($0/10 \to 8/10$)
Acyclicity	inside Ω	schedulability	candidate rejected before evaluation	Appendix B
Metadata `group` / tags (via ϕ)	outside Ω	which sub-pool a probe draws from	candidate simply not sampled	Appendix D (Full vs. Restricted; screening `disabled`/`once`/`periodic`)
Metadata `weight`	outside Ω	sampling distribution over methods	the bias is removed, not a candidate	probe design, §3.3
Metadata history / side-effect id	outside Ω	pool evolution and compression bookkeeping	affects reuse, not well-formedness	§4.2 (pool evolution on/off)

Scope of the evidence. A dedicated metadata-on/off ablation matching the type-routing ablation of §4.3 was *not* run. The soft-constraint column above is therefore supported mechanistically and by the pool-composition ablations of Appendix D; we claim no quantitative gain for ϕ alone.

H.5 Formalization of Boundary Nodes: Root and Collapse

To address the formal completeness of the GDFG definition (Equation 1 in Main Text), we explicitly define the behavior of boundary nodes V_{root} and V_{col}. Unlike processing nodes (V_{proc}) that draw operations from the Method Pool $\mathcal{Q}$, boundary nodes serve as structural interfaces with distinct configuration mechanisms.

Root Nodes (V_{root}): Typed Input Distributors. In the AdaptoFlux implementation, the Root node functions as a *Typed Input Distributor*. It does not perform computational logic but acts as a semantic anchor that injects external features into the graph while strictly enforcing type signatures. Formally, let $\mathbf{x}_t \in \mathbb{R}^{D_{in}}$ be the input vector at time step t. The Root node $v_r \in V_{root}$ decomposes $\mathbf{x}_t$ into a set of typed tokens $\{x_t^{(i)}\}_{i=1}^{D_{in}}$, where each token carries a type signature $\sigma_{out}(v_r^{(i)}) = \text{type}(x_t^{(i)})$ derived from the `input_types_list` configuration. The influence of the Root node is defined as an identity mapping for data injection:

$$I_{v_r}(s_t, \mathbf{x}_t) = \mathbf{x}_t \tag{44}$$

This design decouples data ingestion from semantic processing, allowing the same graph topology to accept heterogeneous input schemas without structural modification.

Collapse Nodes (V_{col}): Independent Output Aggregation. Crucially, the Collapse node operates **independently of the Method Pool** $\mathcal{Q}$. While internal nodes (V_{proc}) are bound to methods $m \in \mathcal{Q}$, the Collapse node is configured via a separate `CollapseFunctionManager` that handles global output projection. Formally, let $\mathbf{H}_t = \{I_u(s_t, \mathbf{x}_t) \mid u \in \text{pred}(v_c)\}$ be the set of inputs from all predecessor nodes connected to the collapse layer. The Collapse node $v_c \in V_{col}$ computes:

$$I_{v_c}(s_t, \mathbf{x}_t) = \mathcal{A}(\mathbf{H}_t) \tag{45}$$

where $\mathcal{A}$ is an aggregation operator selected from a predefined set of structural reducers (e.g., `SUM`, `PROD`, `IDENTITY`, `ENERGY`), as defined in the `CollapseMethod` enum. This separation ensures that:

- **Structural Consistency**: The aggregation logic (e.g., summing all outputs) remains stable regardless of the evolutionary changes in the internal method pool.
- **Dimensional Flexibility**: Operators like `IDENTITY` allow the graph to output raw intermediate representations without forcing a scalar reduction, supporting multi-output tasks.
- **Decoupled Optimization**: The search for optimal internal semantics ($\mathcal{Q}$) is orthogonal to the selection of the output aggregation strategy ($\mathcal{A}$).

Engineering Implementation Details. In our Python prototype, this separation is realized through distinct classes:

- **Method Pool**: Managed by `@method_profile` decorators, governing V_{proc} nodes.
- **Collapse Manager**: Implemented in `CollapseFunctionManager`, which dispatches to static aggregation functions (e.g., `_sum`, `_product`) based on the user-configured `collapse_method`. This manager is initialized independently of the graph's evolutionary state.

Detailed code snippets for the `CollapseFunctionManager` are provided in the anonymous code repository under `ATF/CollapseManager/collapse_functions.py`.

H.6 Static Acyclicity and Runtime Loop Semantics

Equation 1 constrains the *stored* topology of a GDFG to be acyclic, whereas the behaviour a GDFG induces may be closed-loop. This section makes that distinction explicit, since it is the basis on which the framework accommodates loop-like and recursive computation without giving up static scheduling.

Setup. Fix $G = \langle V, E \rangle \in \Omega_{\mathrm{Q},\Sigma}$, a state space S and an input space X. Write $\Phi_G : \mathrm{S} \times \mathrm{X} \to \mathrm{S}$ for the one-step operator of Eq. 2 and $\Pi_G : \mathrm{S} \to \mathrm{X}$ for the interface projection realised by the boundary nodes (V_{root} injects, V_{col} aggregates). An *execution trace* of horizon T is a sequence $\{(\mathbf{s}_t, \mathbf{x}_t)\}_{t=0}^{T}$ with

$$\mathbf{s}_{t+1} = \Phi_G(\mathbf{s}_t, \mathbf{x}_t), \qquad \mathbf{x}_{t+1} = \Pi_G(\mathbf{s}_{t+1}). \tag{46}$$

When Π_G forwards the collapsed state, Eq. 46 is exactly a closed perception–action loop; when Π_G discards it, the trace degenerates to feed-forward execution. Both readings use the same static G.

Proposition 1 (loop–DAG correspondence). For any $G \in \Omega_{\mathrm{Q},\Sigma}$ and any horizon T, the trace of Eq. 46 coincides with a single forward pass of the graph $G^{(T)}$ built from T copies $G_1, \ldots, G_T$ of G by connecting the interface output of G_t to the root nodes of G_{t+1}; and $G^{(T)}$ is acyclic.

Proof. Every edge of $G^{(T)}$ is either internal to some G_t—hence ordered by a topological order of G, which exists because $G \in \Omega_{\mathrm{Q},\Sigma}$—or runs from step t to step $t+1$. Ordering the nodes of $G^{(T)}$ by the pair (position in G_t, t) therefore orients every edge strictly forward, so no cycle exists. □

Three routes by which a loop enters.

- **Time-step unrolling** (Eq. 46): the loop is temporal and lies outside G; unrolling is precisely the construction of Proposition 1. Nothing in the definition of $\Omega_{\mathrm{Q},\Sigma}$ changes; the cost is that the horizon is not part of the searched object.
- **Self-invocation and sub-graph invocation**: a node binds an arbitrary callable impl, so a method may call itself or execute a further registered graph. The engine performs one opaque node evaluation, so type-checking and scheduling of the enclosing graph are unaffected; a nested graph built through the same API carries the same guarantees, and the argument applies inductively per nesting level. Recursion depth and termination are properties of impl, which the framework neither analyses nor constrains.
- **Collapse feedback**: V_{col} is the interface through which the state produced at step t becomes the input of step $t+1$ in Eq. 46. This is what makes the loop *semantic* rather than merely iterative: the graph's own output determines its next input.

Table 36: Where a loop lives, and what it changes.

Route	Formal position	Changes $\Omega_{\mathrm{Q},\Sigma}$?	Visible to the scheduler?
Time-step unrolling	outside G, in the trace	no	no (per-step schedule reused)
Self-invocation / sub-graph	inside impl (opaque)	no	no (one node evaluation)
Collapse feedback	boundary interface V_{col}	no	no (inter-step edge)

What is preserved, and what is not. Because each route above is either temporal or opaque to the engine, the stored topology remains a DAG at all times. Type-constrained routing, topological scheduling and per-step parallel execution (Section 3.4) therefore continue to hold without runtime cycle detection or fixed-point solving, and the unrolled computation is exactly the object analysed in Appendix B. The boundary is on the search side: the current probes search over G alone—they neither synthesise a recursion inside impl nor choose a loop horizon—and on long-horizon stateful tasks they converge to degenerate optima (Appendix I). Loop and recursion support is thus a representational property, established here by construction rather than by a learned example; the automatic stateful result of Section 4.3 concerns a single-predicate stateful graph, not a loop.

I Challenges in Automated Synthesis of Stateful Policies

Before presenting the hand-crafted Snake controller in Appendix F, we conducted a series of preliminary experiments attempting to use current navigation probes (LayerGrow/GraphEvo) to automatically synthesize policies with side effects on *Snake* and *Flappy Bird* tasks. The source code for these experiments can be found in the `experiments/embodied_snake` and `experiments/embodied_bird` directories of the anonymous repository.

Observations on Degenerate Local Optima. Experimental results indicate that without extremely complex type constraints or carefully designed sparse reward signals, the optimizer rapidly converges to **Degenerate Local Optima** that exploit simulation loopholes rather than solving task objectives:

- **Snake Task (Survival Bias):** The agent almost immediately converges to a "continuous turning" strategy. Since the agent does not consume food, its length remains constant, allowing it to survive indefinitely by circling within a safe local area. Although this behavior yields a high survival score, task progress is zero, vividly demonstrating how greedy probes exploit reward proxies without achieving semantic goals.
- **Bird Task (Immediate Failure):** The agent tends to converge to a "continuous jumping until death" strategy or remains stationary, failing to develop obstacle-avoidance behaviors with long-term vision.

In contrast, when GDFG is used purely as a numerical discriminator (i.e., outputting action probabilities rather than directly executing side effects), its performance is significantly more stable. This phenomenon indicates that current greedy replacement-based probes face significant limitations in navigating **Trajectory-dependent Loss Landscapes** introduced by stateful operations.

Explanation of Experimental Code Structure. To clearly demonstrate AdaptoFlux's **Representational Completeness**, we made specific adjustments to the provided codebase:

- **Snake Task (Demo Only):** Since the primary goal of the Snake example in Appendix F is to serve as an *Existence Proof* of the framework's native ability to execute stateful operations, we have **removed the training scripts** from the `experiments/embodied_snake` directory. The provided `snake_main.py` strictly executes the pre-defined JSON configuration to reproduce the behavior of the hand-crafted controller. This aims to prevent confusion between "hand-construction of graphs" (for validating the execution engine) and "automatic synthesis", which on these long-horizon tasks still converges to the degenerate optima above; the minimal stateful task of Section 4.3 demonstrates automatic synthesis in a controlled setting.
- **Bird Task (Training Reference):** For reviewers wishing to attempt automatic synthesis of stateful policies, we retained the complete training pipeline in `experiments/embodied_bird`.

  Code Version and Reproduction Note: Please note that the Bird experiment code was developed during an early iteration of the framework. Although the core logic of configuring side-effect method pools and running LayerGrow/GraphEvo probes remains valid and reference-worthy, some API interfaces in the main branch may have evolved. We recommend using this directory primarily as a structural reference for setting up embodied training loops, rather than expecting exact experimental logs to be reproduced with a one-click run without minor adaptations.

Parts that may require modification for reproduction include: method pool file definitions, loss function interfaces, disabling graphical display functions, and modifying variable access paths. Additionally, this directory contains an early hand-crafted model JSON file; since some methods referenced therein have been removed from the current method pool, **direct loading of this JSON file is not recommended**; instead, refer to the training logic in its Python scripts.

Theoretical Significance and Future Directions. The difficulty in automating these tasks empirically validates the core argument of Main Text § 3.3: **There is a significant Trade-off between the expressiveness of stateful operations and their optimization difficulty.** On such long-horizon tasks, current search strategies are insufficient to effectively explore non-monotonic paths introduced by state transitions, often falling into semantically hollow local optima (e.g., infinite survival without growth); the minimal stateful task of Section 4.3 shows that the same probes do succeed when the required stateful graph is small.

We emphasize that this limitation is not a flaw in the framework design but an inherent challenge of *Credit Assignment* in discrete program synthesis. Potential future directions to break this bottleneck include:

- **Hierarchical Search Strategies:** Introducing macro-action abstractions to encapsulate low-level side-effect operations into high-level semantic units, thereby reducing the dimensionality of the search space.
- **Fine-Grained Type Systems:** Extending metadata $\mathcal{M}$ to include pre/post-conditions of state transitions (e.g., requiring "consuming food" as a prerequisite for positive rewards), thereby pruning invalid state paths during candidate generation.
- **Trajectory-Aware Regularization:** Designing loss function smoothing terms targeted at long-range dependencies to guide the optimizer out of short-sighted local optima like "continuous turning."

This supplementary analysis clarifies the boundary between AdaptoFlux's *Representational Capacity* (as shown by the hand-crafted Snake, possessing completeness and native support) and its *Navigation Efficiency* in current stateful tasks (as shown by degenerate convergence in automated attempts, which remains an open research challenge).

J Limitations of LLM-based Direct Graph Construction

In addition to the gradient-free navigation probes discussed in Section I, we also explored the potential of using Large Language Models (LLMs) to directly synthesize AdaptoFlux graphs. Our initial hypothesis was: given the method pool definition and task description, LLMs could directly output valid GDFG configurations (JSON format) without iterative search. However, preliminary experiments revealed significant challenges, leading us to conclude that direct generation via LLMs is currently unsuitable for constructing stateful graphs in this framework.

Experimental Setup. We provided the following prompt content to state-of-the-art LLMs (such as GPT-4o, Claude-3.5-Sonnet):

1. Complete signatures and docstrings of all methods in the Method Pool $\mathcal{Q}$.
2. Natural language description of the target task (e.g., "Implement a Snake controller that avoids walls and seeks food").
3. JSON Schema of the GDFG structure, including node definitions and edge routing metadata.

The experimental goal was to generate a single, executable JSON file representing the graph's topological structure and method bindings.

Observed Failure Modes. Without extensive human intervention for correction or iterative self-correction frameworks (which essentially become a form of search), LLM-generated graphs consistently failed to execute or produce meaningful performance. We identified four major types of structural errors:

- **Type Mismatch and Interface Errors:** LLMs frequently connected nodes with incompatible data types (e.g., connecting `scalar` output to `vector` input) or ignored explicit `input_types`/`output_types` constraints defined in the method pool. Unlike continuous spaces where type errors are rare, the discrete nature of our method pool makes strict type adherence crucial.
- **Topological Invalidity:** Generated graphs often contained disconnected components, cycles in static DAGs (violating acyclic constraints), or missing links to `root` or `collapse` nodes, rendering the engine unable to execute.
- **Semantic Misalignment in Stateful Operations:** For tasks requiring side effects (like Snake), LLMs struggled to correctly order state-modifying actions. They often placed action nodes (like `turn_left`) in positions where subsequent state update logic could not correctly consume their outputs, or failed to establish necessary semantic loops via state variables. This confirms the structural origin of these failures noted in § 1: without static enforcement of type compatibility and grounding in the method pool, even small topological errors lead to complete functional collapse.
- **Method Hallucination:** Models occasionally invented methods not present in the provided method pool, indicating that their generation process was not grounded in the constrained expression space $\Omega_{\mathcal{Q},\Sigma}$.

Theoretical Implications. These failures reinforce the core argument of this paper: **Constructing effective and high-performance discrete structures requires not only semantic knowledge but also rigorous structured navigation.**

- **LLM as Semantic Prior, Not Structural Solver:** While LLMs excel at generating syntactically correct code snippets, they lack intrinsic mechanisms to verify global graph properties (such as long-distance dependency type consistency or DAG acyclicity) during generation.
- **Necessity of Constraint-Guided Search:** The success of our hand-crafted Snake example (Appendix F) and the partial success of gradient-free probes (Section I) indicate that *explicit constraints* (type systems, topological rules) and *iterative feedback* (evaluation signals) are indispensable. LLMs used in isolation as open-loop generators cannot effectively navigate the deceptive loss landscape of discrete program synthesis.

Conclusion and Limitations. Furthermore, we attempted to use LLMs as auxiliary evaluators for loss functions, aiming to leverage their semantic understanding to compensate for the shortcomings of traditional numerical metrics. However, this path faces two major bottlenecks: (1) **Huge Computational Overhead**, as each evaluation requires calling a large model, causing the time complexity of the training loop to increase by orders of magnitude; (2) **Signal Instability**, where the evaluation signals output by LLMs exhibit significant noise in consistency and task relevance, making them difficult to serve as stable gradient substitute signals for fine-grained optimization of discrete structures.

In summary, at the current technological stage, Large Language Models are not suitable for end-to-end construction of AdaptoFlux graphs. Their open-loop generation paradigm fundamentally mismatches the stringent structural rigor required by discrete combinatorial spaces. Therefore, the current focus of this research remains on improving the framework's own structured navigation mechanisms (such as type-constrained pruning and topology-invariant alignment), rather than relying on semantic heuristics from external black-box models. If LLMs are to be introduced in the future, they must be strictly limited to the role of local semantic suggestions and placed within the powerful structural verification closed-loop of the framework, which is beyond the scope of this paper.

K LIMITATIONS

While AdaptoFlux establishes a structured foundation for discrete program synthesis, several limitations remain. First, the current navigation probes (LayerGrow and GraphEvo) rely on local search heuristics, which may struggle with long-horizon dependency tasks or complex state trajectories where credit assignment is difficult. This reflects a limitation of the current optimization strategy rather than the expressive capacity of the GDFG space itself. Second, the topology-based knowledge

extraction mechanism is currently in a prototype stage. While it demonstrates feasibility in controlled settings, it requires further engineering refinement and debugging to ensure robustness in noisy, real-world environments. Finally, our current Python-based prototype faces engineering constraints. Specifically, it has limited support for vectorized operators (e.g., batched matrix operations), where fine-grained node-level scheduling overhead may outweigh computational benefits. Additionally, the Global Interpreter Lock (GIL) constrains fine-grained parallelism for CPU-bound tasks. These implementation bottlenecks suggest that migrating to system-level backends (e.g., C++ or Rust) and adopting hybrid execution models would be necessary for high-performance numerical tasks.

L BROADER IMPACTS

This work contributes to the field of neuro-symbolic AI and program synthesis, with several potential societal implications:

Positive Impacts. AdaptoFlux enhances the **interpretability** and **transparency** of automated decision-making systems by generating explicit, human-readable symbolic structures. This can help mitigate the "black box" risks associated with deep learning, particularly in safety-critical domains where understanding the logic behind a decision is paramount. Furthermore, our framework's **computational efficiency** (CPU-only, low memory footprint) lowers the barrier for entry, promoting more accessible and energy-conscious AI research compared to resource-intensive large-scale models.

Negative Impacts and Risks. We acknowledge that automated program synthesis technologies carry potential **dual-use risks**. The ability to automatically generate executable code could theoretically be misused for creating malicious scripts, automating cyber-attacks, or bypassing security controls. Additionally, if the method pool or training data contains biases, the synthesized programs might perpetuate or amplify these biases in downstream applications.

Mitigation Strategies. To address these risks, we advocate for the responsible use of AdaptoFlux. We encourage researchers and practitioners to implement appropriate safety checks, such as sandboxed execution environments and code verification steps, when deploying synthesized programs. Future work should also focus on developing robust filtering mechanisms for the method pool to prevent the inclusion of harmful or biased operators.

M DISCUSSION AND FUTURE WORK

This section discusses the theoretical implications of our current design choices regarding boundary nodes and outlines a concrete roadmap for systems engineering improvements, specifically focusing on parallel scalability.

M.1 REINTERPRETING BOUNDARY NODES: FROM AGGREGATION HEURISTICS TO SEMANTIC ALIGNMENT

In the current implementation and experiments of AdaptoFlux, we predominantly assign specific collapse functions (e.g., `SUM`, `PROD`) to V_{col} nodes to align internal representations with target outputs. However, we posit that this design choice is largely a **compromise necessitated by the limitations of current optimization probes**, rather than an intrinsic theoretical requirement of the GDFG formalism.

The "Collapse" as a Proxy for Optimization Constraints. Ideally, a sufficiently powerful optimizer should achieve semantic alignment directly through feature control during graph generation. In such an ideal scenario, the internal topology and operator selection would naturally structure the data flow such that the final nodes inherently encode the target semantics without requiring post-hoc aggregation heuristics.

Currently, the explicit collapse function acts as a *semantic corrector* that compensates for the optimizer's inability to precisely shape the internal feature manifold. For instance, if the optimizer could perfectly control the semantic flow, the output node would naturally compute the required

product or sum through its incoming edges and bound operators, rendering the external `PROD` or `SUM` collapse redundant. Thus, the current reliance on fixed collapse modes reflects the current ceiling of our navigation strategies in navigating the discrete combinatorial space.

Future Trajectory: Convergence to Pure Interfaces. As more advanced optimizers are developed—capable of finer-grained control over topological and semantic evolution—we anticipate a fundamental shift in the semantic role of boundary nodes:

- **From Aggregation to Identity:** The role of the V_{col} node will diminish from an active aggregator to a passive `IDENTITY` interface. It will serve merely as the "End" marker of the semantic flow, exposing the final computed state without altering it.
- **Symmetric Boundary Semantics:** Correspondingly, the V_{root} node serves as the "Start" marker, injecting raw data. In a fully optimized system, both V_{root} and V_{col} will converge to pure structural interfaces. The core logic of semantic transformation will reside entirely within the processing nodes (V_{proc}) and their topological connections, rather than being offloaded to boundary aggregation rules.

This perspective reinforces the decoupling principle of AdaptoFlux: the representation space (GDFG) is designed to be agnostic to the optimizer's sophistication. While current experiments require explicit collapse functions to bridge the gap between structural exploration and semantic precision, future work with stronger optimizers will likely render these functions obsolete, allowing the graph structure itself to bear the full burden of semantic alignment.

M.2 Systems Engineering: Migration to C++ for True Parallelism

A critical limitation of the current Python prototype is the Global Interpreter Lock (GIL), which constrains fine-grained parallelism for CPU-bound tasks. As noted in Appendix B.2.3 and C.3, while the GDFG topology theoretically supports lock-free parallel scheduling, the Python implementation incurs significant synchronization overhead and cannot achieve true parallel execution for pure computational nodes.

To unlock the full performance potential of AdaptoFlux, we plan to **migrate the core execution engine to C++**:

- **True Multi-Core Parallelism:** A C++ backend will eliminate the GIL bottleneck, enabling true parallel execution of independent nodes across multiple CPU cores. This is expected to yield near-linear speedup for wide and shallow graphs, as predicted by our theoretical complexity analysis (Section 3.4).
- **Reduced Scheduling Overhead:** By implementing a lock-free, topology-aware scheduler in C++, we can minimize the overhead of dependency tracking and data routing, which currently dominates runtime in fine-grained graphs.
- **Memory Efficiency:** Migrating to C++ will allow for manual memory management and the elimination of the current "full intermediate result cache" technical debt (Appendix B.2.1), significantly reducing memory footprint and improving cache locality for large-scale batch inference.

This engineering migration is not merely an optimization but a prerequisite for scaling AdaptoFlux to real-time embodied control tasks and large-scale symbolic regression benchmarks, where computational efficiency is paramount. Concurrently, we will focus on developing more robust optimization strategies to better exploit this enhanced computational substrate.

M.3 Intended Use and Enabling Scenarios

The experiments in this paper validate the substrate on a small set of tasks, so it is fair to ask what the representation enables in practice that a purely functional program space does not. This subsection answers that question at the level of design intent: the three scenarios below follow from the mechanisms already described, but *none of them is evaluated in this paper*, and building task-specific search procedures for them is future work.

Registering heterogeneous agent primitives. Physical agents—different robots, or industrial controllers—expose callable operations that differ per embodiment and frequently carry side effects. Because the Method Pool accepts typed black-box callables (Section 3.1) and edge compatibility is checked statically, such primitives can be registered as they are and interleave with pure computation as *structured anchors*. Under this reading, imitation becomes fitting an action sequence inside a constrained pool rather than training a monolithic policy from scratch, which lowers the amount of task-specific data a new embodiment requires.

Reusing validated domain modules. In many deployments the building blocks already exist as validated artefacts: control logic, external APIs, or physical models. Side-effectful modules of this kind can be composed directly into the expression space instead of being relearned, so that the search operates over their composition while the space itself remains explorable—an alternative to blind data-driven exploration.

Symbolic skeleton, then continuous parameters. Because the stored topology is a typed DAG, macro structure and numeric parameters can be optimized in two stages: search the sequence of actions first, then fine-tune each method's parameters with the topology frozen. This differs from neural architecture search, which returns a static architecture: the object searched here is a dynamic action chain in which heterogeneous, stateful methods determine the execution flow, and the substrate supplies only the type-safe interface for the second stage rather than a joint end-to-end training algorithm.

Future work. Turning these scenarios into evaluated capabilities requires state-aware search strategies—hierarchical decomposition, or failure-driven constraint learning—that address the credit-assignment difficulty documented in Appendix I, together with the systems work of Appendix M.2. The minimal stateful result of Section 4.3 is a first step in that direction; extending it to long-horizon control is the central open problem that this substrate is intended to make tractable.