EvoTreeNAD: Genealogy-Guided Evolution for LLM-Driven Neural Architecture Discovery

Anonymous authors
Paper under double-blind review

Abstract

AI-driven scientific discovery accelerates research by autonomously developing solutions and designs. Large language model (LLM) agents support this process through iterative generation and evaluation. Yet these iterations alone do not ensure cumulative progress or establish which directions to pursue next. Costly evaluation further constrains the scope of exploration. Neural architecture discovery brings these challenges together, coupling open-ended design with resource-intensive experimentation. We introduce **EvoTreeNAD**, a genealogy-guided evolutionary algorithm that constructs trainable architectures without a supplied seed or a hand-specified search space. Starting from an empty root, it grows a persistent genealogy in which each new node represents a complete architecture. Top-percentile values computed from each node and its descendants guide lineage selection. Using the selected design history, an Idea Agent proposes a variant and a Code Agent implements it. Each evaluated variant becomes a child node, expanding the genealogy while providing evidence for subsequent lineage selection. Our theoretical analysis establishes the existence of stationary variation regimes as the genealogy grows. Under specified variation assumptions, sustained top-percentile family values quantify the probability of generating high-reward architectures in these regimes. EvoTreeNAD discovers architectures that outperform the compared NAS and NAD baselines, achieving CIFAR-10/100 test errors of $2.05\pm0.06\%$ and $15.09\pm0.22\%$. On all six MedMNIST-v2 tasks, the discovered architectures surpass the strongest listed baselines. A controlled CIFAR-10 study further shows that EvoTreeNAD outperforms direct generation, best-of-N greedy continuation, and full-family-mean routing.

1 Introduction

Large language model (LLM) agents are advancing AI-driven scientific discovery and automated research by generating, implementing, and evaluating candidate solutions with limited human intervention (Lu et al., 2024; Novikov et al., 2025). Yet the ability to repeat this cycle does not ensure that individual findings accumulate into sustained progress or establish which research directions to pursue next. Costly evaluation makes both challenges more consequential, limiting both the breadth of exploration and the depth to which promising directions can be developed (Jiang et al., 2025).

Existing approaches organize iterative design through repeated refinement, candidate selection, and evolutionary search over evaluated solutions (Novikov et al., 2025; Jiang et al., 2025). These strategies support continued improvement, but candidate quality alone does not fully reveal the potential for further development. Strong designs may yield weak variants, while initially less successful designs may become competitive through further modification. Sustained discovery therefore requires building on successful designs while using accumulated experimental evidence to revise earlier development choices.

Neural architecture discovery (NAD) makes this challenge concrete. Recent LLM-based methods move beyond predefined search spaces by generating and iteratively refining executable architectures (Chen et al., 2023; Yang et al., 2025). Yet empirical evaluation of newly generated architectures can require substantial computation, making it critical to decide not only what to generate next, but which directions remain worth pursuing.

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.

We introduce **EvoTreeNAD**, a genealogy-guided evolutionary algorithm that treats architecture discovery as the continued development and reassessment of design lineages. EvoTreeNAD preserves complete trainable architectures and their branching design histories in a persistent genealogy. Starting from an empty root, it constructs architectures without a supplied seed architecture or hand-specified search space. Top-percentile family values summarize the outcomes accumulated by each node and its descendants and guide the selection of a root-to-node lineage for further evolution. Using the selected design history, an Idea Agent proposes the next architectural design or modification, and a Code Agent implements it. Each evaluated variant extends the genealogy, contributing both a design for further evolution and new evidence for subsequent lineage selection. Successful architectures can thus support continued evolution while later evidence can redirect development toward retained alternatives.

Our theoretical analysis establishes the existence of stationary empirical variation regimes as the genealogy grows. Under specified assumptions on lineage-conditioned variation, sustained top-percentile family values exactly characterize the probability of generating high-reward architectures in these regimes. We further quantify how weak outcomes affect full-family means and establish a pathwise equivalence between maximum-family routing and best-of-N greedy continuation, a strategy used in automated design (Ma et al., 2024). Empirically, EvoTreeNAD discovers architectures that outperform the compared NAS and NAD baselines on CIFAR-10/100 under fixed full-fidelity training. The discovered architectures also surpass the strongest listed baseline on each of six MedMNIST-v2 tasks spanning diverse medical imaging modalities across 2D image and 3D volume classification. Controlled CIFAR-10 experiments further demonstrate its advantage over direct generation, best-of-N greedy continuation, and full-family-mean routing.

Our contributions are summarized as follows.

- **Genealogy-guided architecture evolution.** We introduce an evolutionary algorithm that preserves complete architectures and their branching histories, combining lineage-conditioned LLM generation with cumulative descendant evidence and reassessment of earlier design choices.
- **Theoretical characterization of genealogy-guided evolution.** We establish stationary empirical variation regimes for adaptive genealogy growth and, under specified lineage-conditioned variation assumptions, derive an exact relation between sustained top-percentile family values and the long-run probability of generating high-reward architectures in these regimes.
- **Architecture performance and controlled evaluation.** We discover architectures that outperform the compared baselines across CIFAR and six MedMNIST-v2 tasks and demonstrate the advantage of EvoTreeNAD through controlled comparisons with direct generation, greedy continuation, and full-family-mean routing.

2 Related Work

LLM agents for automated discovery. LLM agents extend research automation from proposing solutions to implementing experiments, evaluating outcomes, and developing subsequent proposals. OPRO uses evaluated solutions to inform new proposals, while the AI Scientist automates idea generation, experimentation, and reporting (Yang et al., 2024; Lu et al., 2024). Other systems explicitly organize multiple attempts and choose which to develop further. AIDE retains candidate programs in a solution tree and refines promising solutions using summaries of previous attempts (Jiang et al., 2025). AIRA examines the interaction between generation operators and greedy, evolutionary, and Monte Carlo tree-search policies (Toledo et al., 2025). MLEvolve combines graph-based search with descendant-reward propagation, cross-branch references, and retrospective memory (Du et al., 2026). EvoTreeNAD uses top-percentile family values to select a retained root-to-node lineage whose design history supplies the context for the next architectural variation.

Evolutionary discovery with LLMs. Evolution through Large Models uses LLMs as program mutation operators, linking pretrained generative capabilities to evolutionary optimization (Lehman et al., 2024). Eureka evolves reward programs for reinforcement learning through best-of-N refinement, using the best-performing candidate to inform the next generation (Ma et al., 2024). FunSearch and AlphaEvolve demonstrate mathematical and algorithmic discovery through LLM-generated programs, programmatic evaluation, and evolutionary selection (Romera-Paredes et al., 2024; Novikov et al., 2025). ShinkaEvolve develops sample-efficient program evolution, while the Darwin Gödel Machine

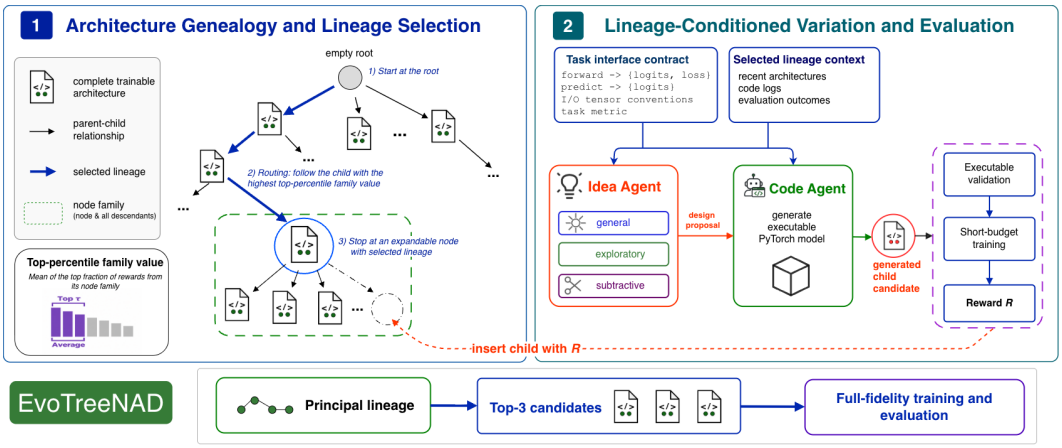


Figure 1: Overview of EvoTreeNAD.

maintains an expanding archive of self-modifying agents from which further variants are generated (Lange et al., 2026; Zhang et al., 2026). In EvoTreeNAD, successful designs can be developed further while subsequent descendant outcomes can redirect evolution toward retained alternatives.

LLM-based neural architecture discovery. Conventional NAS, including evolutionary approaches, typically optimizes architectures within a specified search space (Elsken et al., 2018; Real et al., 2019). LLM-based methods extend architecture generation and modification through executable code. EvoPrompting combines code-level variation with evolutionary search, and LLMatic integrates architectural variation with quality-diversity optimization (Chen et al., 2023; Nasir et al., 2024). LeMo-NADe generates architectures under user-defined requirements, while GENIUS, LM-Searcher, and RZ-NAS combine LLM proposals with structured architecture optimization or efficient evaluation (Rahman & Chakraborty, 2024; Zheng et al., 2023; Hu et al., 2025; Ji et al., 2025). NADER develops a supplied initial architecture through multi-agent modification and reflection; RevoNAD refines base architectures through reflective exploration and multi-objective evolutionary selection (Yang et al., 2025; Chang et al., 2025). Starting from an empty root, EvoTreeNAD unifies the construction of complete trainable architectures with their subsequent lineage-conditioned evolution in a persistent genealogy, without requiring a supplied seed architecture or a hand-specified architecture search space.

3 EvoTreeNAD

EvoTreeNAD organizes architecture discovery through a persistent genealogy of evaluated architectures and their parent–child relationships (Figure 1). At each iteration, top-percentile family values computed from the current genealogy guide the selection of a root-to-node lineage for further evolution. Using architectures and evaluation outcomes from the selected lineage, an Idea Agent proposes an architectural design or modification, and a Code Agent implements it as a complete trainable architecture. After evaluation, the resulting child is added to the genealogy, providing both a candidate for further variation and new evidence for subsequent lineage selection.

3.1 Task Setting

A task interface $\mathcal{S}$ fixes the model inputs and outputs, supervised target, data splits, discovery evaluator, and reporting protocol, while leaving the architecture's internal modules, connectivity, and forward computation to be determined through evolution. Here, an architecture denotes the complete trainable model specification, including its model-defined loss. The task, data, evaluator, and stage-specific training protocols remain fixed across candidates. Appendix C provides the model interface.

3.2 Architecture Genealogy and Family Values

EvoTreeNAD begins with an empty root ρ, which anchors routing but does not represent an architecture. Each non-root node s represents an architecture a_s, and each edge records a parent-to-child design change. After t evolution rounds, the genealogy is $\mathcal{G}_t = (\{\rho\} \cup \mathcal{V}_t, \mathcal{E}_t)$.

Node rewards. The discovery score $S(s)$ measures the performance of architecture a_s under a fixed short-budget training and validation protocol. We convert this score into a comparable node reward while preserving the metric direction:

$$R(s) = r(S(s)) + \lambda_{\text{size}} g(\text{Params}(a_s)), \qquad r(v) = \sigma\left(\frac{\kappa d\,(v - v_e)}{|v_l - v_e|}\right), \quad d = \text{sgn}(v_l - v_e). \tag{1}$$

Here σ is the logistic sigmoid, $\kappa > 0$ controls its slope, v_e and v_l are fixed task-level reference values, and g maps parameter count to a normalized compactness score in $[0, 1]$. The coefficient λ_{size} controls the compactness contribution. The node reward $R(s)$ evaluates one architecture; the family value defined below aggregates node rewards over a descendant family to guide routing.

Top-percentile family value. For a non-root node s, its *descendant family* contains s and every architecture already evolved below it:

$$\mathcal{D}_t(s) = \{u \in \mathcal{V}_t : s \preceq_t u\}, \qquad \mathcal{H}_t(s) = \biguplus_{u \in \mathcal{D}_t(s)} \{R(u)\}. \tag{2}$$

Here $s \preceq_t u$ means that s is an ancestor of u in $\mathcal{G}_t$, including $s = u$.

In our genealogy-based evolutionary framework, each non-root node is assigned a value for routing, termed its *family value*. A family-value rule Φ maps the rewards in $\mathcal{H}_t(s)$ to the value of node s, $V_t^{\Phi}(s) = \Phi(\mathcal{H}_t(s))$. EvoTreeNAD uses a top-percentile rule to emphasize evidence from high-reward family members while reducing sensitivity to weak variants. Fix $0 < \tau \leq 1$ for the top-percentile rule Φ_{top}. Let $n_t(s) = |\mathcal{D}_t(s)|$, $k_t(s) = \max\{1, \lceil \tau n_t(s) \rceil\}$, and let $x_{[1]}^{(s)} \geq \cdots \geq x_{[n_t(s)]}^{(s)}$ be the rewards in $\mathcal{H}_t(s)$. The top-percentile family value of node s is

$$V_t^{\text{top}}(s) = \frac{1}{k_t(s)} \sum_{i=1}^{k_t(s)} x_{[i]}^{(s)}. \tag{3}$$

3.3 Evolutionary Iteration

Each evolutionary iteration begins with lineage selection based on top-percentile family values computed from the current genealogy. Algorithm 1 summarizes the complete evolutionary process.

Lineage selection. Each node s has fixed child capacity $W(s)$; $\text{Ch}_t(s)$ denotes its children in $\mathcal{G}_t$. Routing starts at ρ. If the current node has remaining capacity, traversal stops. If it is saturated, traversal follows its highest-valued child:

$$s_t^{(j+1)} \in \arg\max_{u \in \text{Ch}_t(s_t^{(j)})} V_t^{\text{top}}(u), \quad \text{while } |\,\text{Ch}_t(s_t^{(j)})| \geq W(s_t^{(j)}). \tag{4}$$

The resulting root-to-node path $L_t = (\rho, s_t^{(1)}, \ldots, s_t^{(m)})$ is the *selected lineage*. Its endpoint is the first node with remaining capacity and becomes the parent of the next architecture. Because every iteration begins again at the root and uses the current genealogy, later descendant outcomes can redirect evolution toward a retained branch.

Architectural variation with Idea and Code Agents. The selected lineage provides the design and evaluation history for two agents with distinct roles. Agent prompts and context settings are provided in Appendices F and B.1.

1. **Idea Agent.** The Idea Agent uses the task requirements and recent architectures and evaluation outcomes from the selected lineage to propose distinct architectural designs or modifications. These span *general refinements* of existing structures, *exploratory modifications* that investigate alternative designs, and *subtractive simplifications* of low-contribution components.

2. **Code Agent.** The Code Agent implements a selected design proposal as a complete architecture for evaluation, using the task interface and design history from the same lineage.

Algorithm 1: EvoTreeNAD genealogy-guided evolution

Require: Task interface $\mathcal{S}$, evaluator, top-percentile fraction τ, child capacities W, iterations H
Initialize $\mathcal{G}_0$ with empty root ρ
for $t = 0, \ldots, H-1$ **do**
  Compute V_t^{top} from $\mathcal{G}_t$
  $s \leftarrow \rho$
  while $|\,\text{Ch}_t(s)| \geq W(s)$ **do**
    $s \leftarrow \arg\max_{u \in \text{Ch}_t(s)} V_t^{\text{top}}(u)$
  end while
  $L_t \leftarrow \text{Path}(\rho, s)$
  Retrieve unused proposals I_t for s; if none remain, set $I_t \leftarrow \textsc{IdeaAgent}(\mathcal{S}, L_t)$
  Select an unused proposal $i_t \in I_t$ for s
  $A_{t+1} \leftarrow \textsc{CodeAgent}(\mathcal{S}, L_t, i_t)$
  Evaluate A_{t+1}; compute R_{t+1}
  Add A_{t+1} with R_{t+1} under s to form $\mathcal{G}_{t+1}$
end for
Extract the principal lineage from $\mathcal{G}_H$

Architecture evaluation. Generated implementations are checked for executability and compatibility with the task interface before short-budget training and validation under the fixed task-specific protocol. Poorly performing candidates may be stopped early, with recorded validation results still used to compute discovery scores and node rewards. Attempts that yield no valid score trigger bounded Code-Agent retries within the selected lineage. Appendix B.2 provides details of these checks, early stopping, and failure handling.

Genealogy growth. After evaluation, the new architecture $a_u = A_{t+1}$ is added as a child of the selected parent together with its reward $R(u) = R_{t+1}$, forming $\mathcal{G}_{t+1}$. Both improvements and weaker variants remain available for further evolution. The new reward becomes part of the descendant-family records, providing additional evidence for subsequent lineage selection.

3.4 Final Architecture Selection

After discovery, EvoTreeNAD extracts the *principal lineage* by applying the same routing rule (Eq. (4)) to the final genealogy $\mathcal{G}_H$. The three architectures on this lineage with the highest discovery rewards form a fixed full-fidelity candidate set. Genealogy growth, principal-lineage extraction, and candidate-set formation use only the discovery training and validation data. After this set is fixed, its architectures are trained from scratch under the full-fidelity reporting protocol and evaluated on the official test split. Discovery and full-fidelity protocols are provided in Appendices D.1 and D.3.

4 Dynamics of Genealogy-Guided Evolution

In EvoTreeNAD, accumulated descendant outcomes guide lineage selection, which determines the design history used for subsequent variation. We establish the existence of stationary empirical variation regimes for this adaptive genealogy-growth process. Under specified assumptions on lineage-conditioned variation, we further show that sustained top-percentile family values quantify the probability of generating high-reward architectures in these regimes.

We consider EvoTreeNAD and its mean and maximum variants. Let Φ denote the corresponding family-value rule and Ω_t^Φ the process state after t genealogy expansions, including the genealogy, observed rewards, and retained information used for subsequent variation. Rewards lie in a bounded interval $[a, b]$. One step of the induced process is

$$\begin{aligned} L_t^\Phi &= \text{Route}_\Phi(\Omega_t^\Phi), \\ (A_{t+1}^\Phi, R_{t+1}^\Phi) &\sim K(\cdot \mid C_t^\Phi), \\ \Omega_{t+1}^\Phi &\sim \text{Grow}(\cdot \mid \Omega_t^\Phi, L_t^\Phi, A_{t+1}^\Phi, R_{t+1}^\Phi), \end{aligned} \tag{5}$$

where C_t^Φ is the variation context constructed from the process state Ω_t^Φ and selected lineage L_t^Φ. The kernel K specifies the conditional joint law of the next architecture and its reward, and Grow inserts the child and updates retained variation information. Context construction and the variation and state-update kernels are shared across rules, while the selected contexts and subsequent genealogies are policy-dependent.

Stationary selection-variation. A conditional reward law F describes the possible rewards of variation from a selected lineage context. We characterize ongoing variation through the joint

sequence of lineage-conditioned reward laws and realized rewards. Empirical occupation measures over time shifts of this sequence admit shift-stationary subsequential limits. We call these limits reachable stationary regimes of the run.

Theorem 1 (Stationary selection-variation)**.** *Every infinite run of Eq.* (5) *admits reachable stationary regimes. Almost surely, the following conclusions hold simultaneously for every such regime* Q*. Let* ν_{Q} *be the occupation distribution over lineage-conditioned reward laws and* Λ_{Q} *the reward distribution of generated architectures. Then*

$$\Lambda_{\mathsf{Q}} = \int F \, \nu_{\mathsf{Q}}(dF). \tag{6}$$

Along every subsequence realizing Q*, the empirical exceedance frequency at each threshold* q *satisfying* $\Lambda_{\mathsf{Q}}(\{q\}) = 0$ *converges to the corresponding occupation-weighted conditional probability.*

We next connect top-percentile family values to high-reward production within these regimes and characterize how mean and maximum rules differ.

Family values and recurrent development. The variation assumptions in Appendix A.4 keep the high- and weak-reward distributions and intermediate reward r_0 fixed, while their mixture weights vary with the selected lineage context. The highest-reward τ fraction of each conditional reward law contains all high outcomes and excludes weak ones. Let μ_+ denote the mean high reward. The mean $U_\tau(F)$ of the highest-reward τ fraction of F is the population counterpart of the empirical top-percentile family value.

Let q separate high rewards from intermediate and weak outcomes. A family is *recurrent* if selected lineages pass through its node infinitely often. For each recurrent family v, we assume that the conditional high- and weak-outcome probabilities have convergent averages along its sequence of selected contexts. Let $\rho_{\Phi,v}(q)$ denote the limiting average conditional probability of reward at least q. At the root, this average covers the complete run.

Theorem 2 (Top-percentile routing and recurrent high-reward production)**.** *Under the lineage-conditioned variation assumptions in Appendix A.4, the following comparisons hold almost surely in the genealogy generated by each rule.*

Top-percentile. *For every fixed recurrent non-root family* v *and every reachable stationary regime* Q *of the same run,*

$$\lim_{t\to\infty} V_t^{\mathrm{top}}(v) = U_\tau(\Lambda_{\mathsf{Q}}) = r_0 + \frac{\mu_+ - r_0}{\tau}\Lambda_{\mathsf{Q}}([q, b]). \tag{7}$$

The limiting family value therefore characterizes the regime's top-percentile reward mean and high-reward production probability. When Φ *is the top-percentile rule, the equality* $\rho_{\Phi,v}(q) = \Lambda_{\mathsf{Q}}([q, b])$ *holds for every recurrent family* v *in the same run, connecting the high-reward production rate of each recurrent family to that of the run.*

Full-family mean. *Weak-outcome frequency reduces the limiting family value. Recurrent child families attain equal limiting means but can have different high-reward frequencies. The gap between the maximal high-reward frequency among recurrent children and the parent frequency is proportional to the corresponding gap in weak-outcome frequency (Eq.* (31)*).*

Family maximum. *The resulting genealogy growth is pathwise equivalent to best-of-*N *greedy continuation. The current parent* v *produces* $N = W(v)$ *children, the child with the highest immediate reward becomes the next parent, and unchosen siblings receive no further development.*

For each rule, $\rho_{\Phi,\rho}(q)$ *equals the realized long-run frequency of generated architectures with reward at least* q*.*

Maintaining a high top-percentile family value as a family grows requires a nonvanishing fraction of high-reward descendants, rather than a few early successes (Proposition 2). Under mean routing, a recurrent child family can produce high-reward architectures more frequently than a recurrent sibling yet receive no higher limiting value because it also produces weak variants more frequently. The limiting top-percentile family value instead tracks high-reward production without this weak-outcome penalty. Unlike the maximum-family variant, EvoTreeNAD can develop successful architectures without making earlier branch choices permanent, allowing subsequent outcomes to redirect evolution toward retained alternatives. Full statements and proofs appear in Appendix A.

5 EXPERIMENTS

We evaluate EvoTreeNAD on CIFAR-10/100 and six MedMNIST-v2 tasks spanning diverse medical imaging modalities and both 2D image and 3D volume classification. We report the performance of the discovered architectures and use controlled CIFAR-10 comparisons to assess the benefit of genealogy-guided evolution. Code and artifacts are available anonymously at `https://github.com/anonymoussub541/anonymous-code-submission`.

5.1 PROTOCOL

CIFAR-10/100 (Krizhevsky et al., 2009) provide established architecture discovery benchmarks. We split each 50k training set into 42k/8k discovery-training and validation subsets. Final performance is evaluated on the official 10k test set. MedMNIST-v2 (Yang et al., 2021; 2023) uses its official train, validation, and test splits (dataset details in Appendix D.2). EvoTreeNAD begins from an empty root on every task. Main discovery uses three independent runs per task, with GPT-4.1 for architectural proposals and gpt-oss-20b (OSS20B) for code realization. We mask dataset identifiers in agent prompts to reduce LLM reliance on benchmark-specific prior knowledge. CIFAR-10/100 architecture discovery uses budgets of up to 300 iterations, with the corresponding discovery costs reported in Table 1. CIFAR-10 discovery-strategy and agent-configuration comparisons use 100 iterations per run. MedMNIST-v2 discovery runs use 80 iterations each.

Candidates receive short-budget evaluation during evolution. After each EvoTreeNAD run, the three architectures with the highest discovery rewards on its principal lineage form a fixed candidate set for full-fidelity evaluation. Genealogy growth and candidate-set formation use only discovery training and validation data. Each candidate is then trained from scratch under the fixed task-specific full-fidelity recipe, with no pretraining, external data, or architecture-specific tuning after selection. Reported task metrics use the official test splits. Discovery budgets, hyperparameters, and training recipes appear in Appendices D.1, B.1, and D.3.

5.2 ARCHITECTURE DISCOVERY ON CIFAR-10/100

Table 1: CIFAR-10/100 test performance. For each dataset, we report the best-performing discovered architecture and a smaller competitive architecture from a different discovery run. EvoTreeNAD architectures are trained from scratch under fixed full-fidelity recipes; mean±std is computed over four random seeds. Results for other methods are taken from their publications.

Approach	CIFAR-10			CIFAR-100			Method	Space
	Top-1 err. (%)↓	Params (M)	GD	Top-1 err. (%)↓	Params (M)	GD		
Classical NAS and differentiable NAS								
NASNet-A (Zoph et al., 2018)	2.65	3.3	2000	N/R	N/R	N/R	RL	NASNet
ENAS (Pham et al., 2018)	2.89	4.6	0.45	N/R	N/R	N/R	RL	NASNet
AmoebaNet-B (Real et al., 2019)	2.55±0.05	2.8	3150	N/R	N/R	N/R	EA	NASNet
Random Search (Li & Talwalkar, 2019)	2.85±0.08	4.3	9.7	N/R	N/R	N/R	Random	DARTS
DARTS (Liu et al., 2018)	2.76±0.09	3.3	4	N/R	N/R	N/R	GB	DARTS
P-DARTS (Chen et al., 2021)	2.50	3.4	0.3	15.92	3.6	0.3	GB	DARTS
β-DARTS (Ye et al., 2022)	2.51±0.07	3.78	0.4	16.24±0.22	3.8	0.4	GB	DARTS
Λ-DARTS (Movahedi et al., 2022)	2.35	3.8	0.8	15.8	3.8	0.8	GB	DARTS
EG-NAS (Cai et al., 2024)	2.53	3.2	0.1	16.22	3.2	0.1	GB	DARTS
ProxylessNAS-G (Cai et al., 2019)	2.08	5.7	N/R	N/R	N/R	N/R	GB	DARTS
LLM-guided NAS and open-ended NAD								
EvoPrompting (Chen et al., 2023)	6.89±0.90	N/R	N/R	29.61±0.58	N/R	N/R	LLM	NATS
GENIUS (Zheng et al., 2023)‡	6.21±0.09	N/R	N/R	29.09±0.72	N/R	N/R	LLM	NB201
LM-Searcher (Hu et al., 2025)	3.10	N/R	N/R	27.04	N/R	N/R	LLM	DARTS
RZ-NAS (Ji et al., 2025)	2.41±0.13	N/R	0.03	17.49±0.08	N/R	0.03	LLM	DARTS
RevoNAD (Chang et al., 2025)‡	4.78±0.2	N/R	N/R	23.62±0.32	N/R	N/R	LLM	None
NADER (Yang et al., 2025)‡	5.38	N/R	N/R	24.00	N/R	N/R	LLM	None
EvoTreeNAD (ours) with no predefined architecture space or supplied seed architecture								
Best discovered architecture	**2.05**±0.06	7.03	0.63	**15.09**±0.22	7.67	1.02	**LLM**	**None**
Smaller discovered architecture	2.38±0.06	**2.25**	**0.38**	15.18±0.10	5.66	1.14	**LLM**	**None**
Best training-seed result	2.00	N/R	N/R	14.82	N/R	N/R	LLM	None

RL, reinforcement learning; EA, evolutionary algorithm; GB, gradient-based search. NB201, NAS-Bench-201; NATS, NATS-Bench. GD denotes reported discovery cost in GPU-days, which may vary with hardware. For EvoTreeNAD, GD reports the discovery cost of the corresponding run. Space denotes the reported hand-specified search space, if any. N/R indicates not reported. ‡ marks results obtained under the official NAS-Bench-201 protocol.

Table 2: CIFAR-10 comparison of architecture-discovery strategies. Each strategy has three independent 100-iteration runs. The Run 1–3 columns report the highest full-fidelity accuracy (%) among three candidates selected by discovery reward within each run; Best and Mean summarize these three run-level accuracies.

Strategy	Lineage-conditioned variation	Branch reconsideration	Selection signal	Run 1	Run 2	Run 3	Best	Mean
Repeated direct generation	–	–	–	96.94	96.59	96.60	96.94	96.71
Best-of-N greedy continuation	✓	–	Immediate child reward	96.34	96.92	97.06	97.06	96.77
Full-family mean	✓	✓	Mean-based family value	96.81	95.66	96.49	96.81	96.32
EvoTreeNAD	✓	✓	**Top-percentile family value**	**97.39**	**97.26**	**97.11**	**97.39**	**97.25**

Table 3: MedMNIST-v2 2D results.

Method	**PathMNIST**	**OCTMNIST**	**TissueMNIST**
ResNet-18 (28)†	0.907	0.743	0.676
ResNet-18 (224)†	0.909	0.763	0.681
ResNet-50 (28)†	0.911	0.762	0.680
ResNet-50 (224)†	0.892	0.776	0.680
AutoML (AutoKeras)†	0.834	0.763	0.703
Google AutoML Vision†	0.728	0.771	0.673
MSTF-NAS	0.910	0.780	0.740
ZO-DARTS++ L	0.804±0.024	0.799±0.010	0.656±0.012
EvoNASMed	0.920	0.800	0.760
EvoTreeNAD (64)	**0.962**±0.003	**0.926**±0.009	**0.769**±0.001

Table 4: MedMNIST-v2 3D results.

Method *Metric*	**Vessel3D** *AUC, ACC*	**Synapse3D** *AUC, ACC*	**Organ3D** *ACC*
ResNet-18 + 3D†	0.874, 0.877	0.820, 0.745	0.907
ResNet-18 + ACS†	0.930, 0.928	0.705, 0.722	0.900
ResNet-50 + 3D†	0.907, 0.918	0.851, 0.795	0.883
ResNet-50 + ACS†	0.912, 0.858	0.719, 0.709	0.889
auto-sklearn†	0.910, 0.915	0.631, 0.730	0.814
AutoML (AutoKeras)†	0.773, 0.894	0.538, 0.724	0.804
EGNAS	0.918±0.002, 0.901±0.004	0.892±0.004, 0.868±0.004	0.943±0.003
EvoNASMed	0.940, 0.940	0.820, 0.846	0.908
EvoTreeNAD (64^3)	**0.975**±0.008**, 0.970**±0.002	**0.981**±0.002**, 0.952**±0.003	**0.956**±0.004

The 2D tasks and Organ3D report accuracy; Vessel3D and Synapse3D report AUC, accuracy. EvoTreeNAD results report mean±std over four random seeds. † marks official MedMNIST-v2 baselines from Yang et al. (2023). Other prior methods are MSTF-NAS (Wang et al., 2024), ZO-DARTS++ (Xie et al., 2025), EvoNASMed (Ali et al., 2024), and EGNAS (Benmeziane et al., 2025).

EvoTreeNAD discovers architectures that outperform the listed NAS and NAD methods, achieving test errors of $2.05\pm0.06\%$ on CIFAR-10 and $15.09\pm0.22\%$ on CIFAR-100 (Table 1). On CIFAR-10, it also discovers a 2.25M-parameter architecture with $2.38\pm0.06\%$ error, improving on DARTS and P-DARTS with fewer parameters. On CIFAR-100, both discovered architectures surpass the reported results of all listed baselines. These results demonstrate that EvoTreeNAD can discover multiple high-performing architectures that are competitive with established NAS and NAD methods (representative architectures in Appendix E.1).

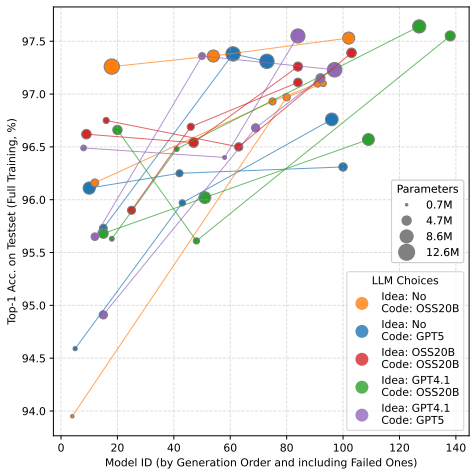
5.3 Comparison of Discovery Strategies

To assess the benefit of genealogy-guided evolution, we compare EvoTreeNAD with repeated direct generation, best-of-N greedy continuation, and full-family-mean routing on CIFAR-10 (strategy schematics in Appendix Figure 4). Each method uses three independent 100-iteration runs with OSS20B for both agents. Repeated direct generation produces independent architectures from the empty root. The maximum-family variant is exactly best-of-N greedy continuation (Proposition 1), generating siblings and continuing only from the child with the highest immediate reward. The full-family mean variant replaces the top-percentile family value of each node with the mean reward of its entire family. Both variants change the family-value rule within EvoTreeNAD's lineage-conditioned generation and evaluation pipeline.

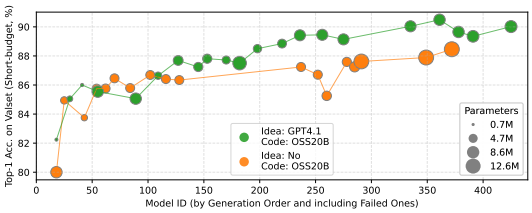
EvoTreeNAD consistently outperforms direct generation, best-of-N greedy continuation, and full-family-mean routing. Each of its three runs achieves higher accuracy than the best run of any alternative (Table 2). The advantage over full-family-mean routing shows the benefit of top-percentile selection over averaging all family rewards within the same genealogy-guided framework. The advantage over mean routing and greedy continuation is consistent with the distinctions established in Theorem 2, which characterizes the sensitivity of full-family means to weak descendants and the irreversible lineage choices of maximum-family routing.

5.4 Architecture Evolution and Agent Configurations

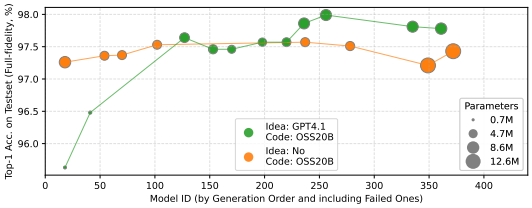
Architecture outcomes along principal lineages. To examine how architectures develop along a lineage, we evaluate three architectures spanning its early and later stages under the full-fidelity protocol (Figure 2a). Later architectures often outperform the initial designs despite intermediate performance declines. For these CIFAR-10 architectures, short-budget validation scores correlate positively with full-fidelity performance (Appendix Figure 3), supporting the use of short-budget



(a) Full-fidelity test accuracy of three architectures spanning the early and later stages of each principal lineage from independent 100-iteration runs.



(b) Short-budget validation accuracy along principal lineages from two 300-iteration runs.



(c) Full-fidelity test accuracy of architectures from the same principal lineages.

Figure 2: Architecture development along EvoTreeNAD principal lineages on CIFAR-10.

scores to guide evolution. The two 300-iteration runs provide a longer view of this development, showing further gains in short-budget validation and full-fidelity performance as evolution proceeds (Figures 2b and 2c). In the run with GPT-4.1 proposals, later evolution produces our best-performing CIFAR-10 architecture. Architecture-selection protocols and details of the 300-iteration runs appear in Appendices E.2 and E.3.

Idea and Code Agent controls. We evaluate EvoTreeNAD under five Idea and Code Agent configurations, including code-only variation. Each configuration uses three independent runs with genealogy-guided selection held fixed. EvoTreeNAD discovers competitive architectures across these configurations (Figure 2a). The best-performing architectures are discovered using GPT-4.1 as the Idea Agent and OSS20B as the Code Agent. Code realization accounts for most of the agent token usage in these runs. Locally deployed OSS20B handles this token-intensive role without API charges. Appendix Figure 5 summarizes the best architecture performance from each run, and Appendix Table 10 reports process statistics and costs.

5.5 Architecture Discovery across MedMNIST-v2 Tasks

EvoTreeNAD discovers architectures that outperform the strongest listed baseline on all six MedMNIST-v2 tasks (Tables 3 and 4). These gains cover diverse medical imaging modalities and both 2D image and 3D volume classification. Starting from an empty root on each task, the same genealogy-guided framework produces high-performing architectures across these settings, demonstrating its effectiveness beyond natural-image benchmarks.

6 Discussion and Conclusion

EvoTreeNAD combines lineage-conditioned variation with selection guided by top-percentile family values in a persistent architecture genealogy. Our theoretical analysis establishes stationary empirical variation regimes and, under specified lineage-conditioned variation assumptions, derives an exact relation between sustained top-percentile family values and the probability of generating high-reward architectures in these regimes. Experiments demonstrate strong architecture performance across natural-image and medical-image classification tasks, while controlled comparisons show the advantage of EvoTreeNAD over direct generation, greedy continuation, and full-family-mean routing.

Scaling EvoTreeNAD depends on obtaining informative evaluation feedback at an affordable computational cost. More broadly, EvoTreeNAD shows how LLM-driven discovery can build on successful designs without making earlier branch choices permanent, using accumulated descendant evidence to redirect development toward retained alternatives.

AI USE STATEMENT

The authors conceived and carried out this research and wrote the manuscript, using LLM-based tools (ChatGPT and OpenAI Codex) for assistance. The authors used LLMs to help improve the literature review by identifying additional relevant references and refining its organization. Within the theoretical framework developed by the authors, LLMs assisted the authors in developing proofs, exploring proof strategies, and refining and checking mathematical derivations. LLMs also assisted with drafting and revising portions of the manuscript, including the theoretical section and appendix. The authors reviewed all AI-assisted content, checked the mathematical arguments and cited sources, and take responsibility for the entire work.

REFERENCES

Mohamed S. Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D. Lane. Zero-cost proxies for lightweight NAS. *arXiv preprint arXiv:2101.08134*, 2021.

Muhammad Junaid Ali, Laurent Moalic, Mokhtar Essaid, and Lhassane Idoumghar. Evolutionary neural architecture search for 2D and 3D medical image classification. In *International Conference on Computational Science*, pp. 131-146. Springer, 2024.

Hadjer Benmeziane, Abderaouf Gacem, Kaoutar El Maghraoui, and Sara Benmeziane. Efficient graph neural architecture search for medical imaging in real-world clinical settings. In *4th Muslims in ML Workshop co-located with ICML 2025*, 2025.

Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2019.

Zicheng Cai, Lei Chen, Peng Liu, Tongtao Ling, and Yutao Lai. EG-NAS: Neural architecture search with fast evolutionary exploration. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(10):11159-11167, Mar. 2024. doi: 10.1609/aaai.v38i10.28993.

Gyusam Chang, Jeongyoon Yoon, Shin han yi, JaeHyeok Lee, Sujin Jang, and Sangpil Kim. RevoNAD: Reflective evolutionary exploration for neural architecture design. *arXiv preprint arXiv:2512.05403*, 2025.

Angelica Chen, David Dohan, and David So. EvoPrompting: Language models for code-level neural architecture search. *Advances in neural information processing systems*, 36:7787-7817, 2023.

Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive DARTS: Bridging the optimization gap for NAS in the wild. *International Journal of Computer Vision*, 129(3):638-655, 2021.

Shangheng Du, Xiangchao Yan, Jinxin Shi, Zongsheng Cao, Shiyang Feng, Zichen Liang, Boyuan Sun, Tianshuo Peng, Yifan Zhou, Xin Li, Jie Zhou, Liang He, Bo Zhang, and Lei Bai. MLEvolve: A self-evolving framework for automated machine learning algorithm discovery. *arXiv preprint arXiv:2606.06473*, 2026.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via Lamarckian evolution. *arXiv preprint arXiv:1804.09081*, 2018.

Yuxuan Hu, Jihao Liu, Ke Wang, Jinliang Zheng, Weikang Shi, Manyuan Zhang, Qi Dou, Rui Liu, Aojun Zhou, and Hongsheng Li. LM-Searcher: Cross-domain neural architecture search with LLMs via unified numerical encoding. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 9408-9421, 2025.

Zipeng Ji, Guanghui Zhu, Chunfeng Yuan, and Yihua Huang. RZ-NAS: Enhancing LLM-guided neural architecture search via reflective zero-cost strategy. In *Forty-second International Conference on Machine Learning*, 2025.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. AIDE: AI-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*, 2025.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Robert Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations*, pp. 74026-74078, 2026.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through large models. In Wolfgang Banzhaf, Penousal Machado, and Mengjie Zhang (eds.), *Handbook of Evolutionary Machine Learning*, pp. 331-366. Springer, Singapore, 2024. doi: 10.1007/978-981-99-3814-8_11.

Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI Scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

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

Sajad Movahedi, Melika Adabinejad, Ayyoob Imani, Arezou Keshavarz, Mostafa Dehghani, Azadeh Shakery, and Babak N Araabi. Λ-DARTS: Mitigating performance collapse by harmonizing operation selection among cells. *arXiv preprint arXiv:2210.07998*, 2022.

Muhammad Umair Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn. LLMatic: neural architecture search via large language models and quality diversity optimization. In *proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1110-1118, 2024.

Xuefei Ning, Changcheng Tang, Wenshuo Li, Zixuan Zhou, Shuang Liang, Huazhong Yang, and Yu Wang. Evaluating efficient performance estimators of neural architectures. In *Advances in Neural Information Processing Systems*, 2021.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

Md Hafizur Rahman and Prabuddha Chakraborty. LeMo-NADe: Multi-parameter neural architecture discovery with LLMs. *arXiv preprint arXiv:2402.18443*, 2024.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pp. 4780-4789, 2019.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468-475, 2024. doi: 10.1038/s41586-023-06924-6.

Binxin Ru, Clare Lyle, Lisa Schut, Miroslav Fil, Mark van der Wilk, and Yarin Gal. Speedy performance estimation for neural architecture search. In *Advances in Neural Information Processing Systems*, 2021.

Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Lupidi, Andrei Lupu, Roberta Raileanu, Tatiana Shavrina, Kelvin Niu, Jean-Christophe Gagnon-Audet, Michael Shvartsman, Shagun Sodhani, Alexander Miller, Abhishek Charnalia, Derek Dunfield, Carole-Jean Wu, Pontus Lars Erik Saito Stenetorp, Nicola Cancedda, Jakob Foerster, and Yoram Bachrach. AI

research agents for machine learning: Search, exploration, and generalization in MLE-bench. In *Advances in Neural Information Processing Systems*, volume 38, pp. 35309-35348, 2025. doi: 10.52202/085713-1186.

Yan Wang, Liangli Zhen, Jianwei Zhang, Miqing Li, Lei Zhang, Zizhou Wang, Yangqin Feng, Yu Xue, Xiao Wang, Zheng Chen, et al. MedNAS: Multiscale training-free neural architecture search for medical image analysis. *IEEE Transactions on Evolutionary Computation*, 28(3):668-681, 2024.

Colin White, Arber Zela, Binxin Ru, Yang Liu, and Frank Hutter. How powerful are performance predictors in neural architecture search? In *Advances in Neural Information Processing Systems*, 2021.

Lunchen Xie, Eugenio Lomurno, Matteo Gambella, Danilo Ardagna, Manual Roveri, Matteo Matteucci, and Qingjiang Shi. ZO-DARTS++: An efficient and size-variable zeroth-order neural architecture search algorithm. *arXiv preprint arXiv:2503.06092*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2024.

Jiancheng Yang, Rui Shi, and Bingbing Ni. MedMNIST classification decathlon: A lightweight AutoML benchmark for medical image analysis. In *IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pp. 191-195, 2021.

Jiancheng Yang, Rui Shi, Donglai Wei, Zequan Liu, Lin Zhao, Bilian Ke, Hanspeter Pfister, and Bingbing Ni. MedMNIST v2 - a large-scale lightweight benchmark for 2D and 3D biomedical image classification. *Scientific Data*, 10(1):41, 2023.

Zekang Yang, Wang Zeng, Sheng Jin, Chen Qian, Ping Luo, and Wentao Liu. NADER: Neural architecture design via multi-agent collaboration. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 4452-4461, 2025.

Peng Ye, Baopu Li, Yikang Li, Tao Chen, Jiayuan Fan, and Wanli Ouyang. β-DARTS: Beta-decay regularization for differentiable architecture search. In *proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10874-10883, 2022.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, 2026.

Mingkai Zheng, Xiu Su, Shan You, Fei Wang, Chen Qian, Chang Xu, and Samuel Albanie. Can GPT-4 perform neural architecture search? *arXiv preprint arXiv:2304.10970*, 2023.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2018.

A Dynamics of Genealogy-Guided Evolution

This appendix provides the full statements and proofs of the theoretical results in Section 4. We first formalize genealogy growth in EvoTreeNAD and establish the properties of the family-value rules. We then prove the existence of stationary empirical variation regimes and show how sustained top-percentile family values characterize high-reward production, alongside comparisons with mean and maximum routing. Table 5 lists the notation and corresponding descriptions.

Table 5: Notation Table

Notation	Description
Φ	Rule mapping a descendant-family reward multiset to a family value.
$\Omega_t^\Phi,\ \mathcal{G}_t^\Phi$	Process state and its genealogy after t expansions under rule Φ.
$\mathcal{H}_t^\Phi(v)$	Reward multiset of node v and all its descendants in the current genealogy.
$V_t^\Phi(v)$	Empirical family value $\Phi(\mathcal{H}_t^\Phi(v))$ used for routing.
$\tau,\ V_t^{\mathrm{top}}(v)$	Fixed top-percentile fraction and corresponding empirical family value.
L_t^Φ	Selected root-to-node lineage; its endpoint receives the next child.
C_t^Φ	Variation context constructed from the process state and selected lineage.
K	Lineage-conditioned kernel generating an architecture and its reward.
$A_{t+1}^\Phi,\ R_{t+1}^\Phi$	Generated architecture and its realized reward at expansion $t+1$ under rule Φ.
F_t^Φ	Conditional reward law of the next architecture, given the pre-iteration history.
$\mathsf{Q}_{\Phi,T},\ \mathsf{Q}$	Empirical occupation measure over time shifts of the sequence of conditional reward laws and realized rewards, and a reachable stationary regime given by a shift-invariant weak subsequential limit.
ν_{Q}	Marginal distribution of conditional reward laws under Q.
Λ_{Q}	Marginal distribution of generated rewards under Q.
$M_{k,n}(x)$	Mean of the k largest entries of $x \in [a,b]^n$.
$U_\alpha(F)$	Mean of the highest-reward α fraction of F, for $0 < \alpha \le 1$. For $\alpha = \tau$, the population counterpart of the empirical top-percentile family value.
$N_T^\Phi(v)$	Number of iterations $t < T$ whose selected lineage contains v.
$\mathcal{I}_\Phi,\ \mathcal{R}_\Phi(v)$	Nodes visited infinitely often by selected lineages, and children of v with this property, respectively.
$\rho_{\Phi,v}(q),\ \bar{\ell}_{\Phi,v}$	Limiting averages of conditional high- and weak-reward probabilities over iterations routed through v.

A.1 Formal Setup

We consider EvoTreeNAD with a family-value rule Φ and denote the resulting stochastic process by $(\Omega_t^\Phi)_{t\ge 0}$. The process state Ω_t^Φ includes the genealogy $\mathcal{G}_t^\Phi$ after t expansions and the information retained for subsequent variation. The initial genealogy consists only of the empty root ρ. Each non-root node v represents an architecture with reward $R^\Phi(v) \in [a,b]$, and every node has a fixed child capacity $W(v) \in \mathbb{N}_+$.

For a non-root node v, its descendant-family reward multiset is

$$\mathcal{H}_t^\Phi(v) = \biguplus_{u \in \{v\} \cup \mathrm{Desc}_t^\Phi(v)} \{R^\Phi(u)\}, \tag{8}$$

where $\mathrm{Desc}_t^\Phi(v)$ denotes the set of strict descendants of v in $\mathcal{G}_t^\Phi$. The family-value rule Φ assigns v the value $V_t^\Phi(v) = \Phi(\mathcal{H}_t^\Phi(v))$ used for routing. For a nonempty reward multiset $\mathcal{H} = \{r_1, \dots, r_n\}$ with rewards ordered as $r_{[1]} \ge \cdots \ge r_{[n]}$, set $k_\tau(n) = \max\{1, \lceil \tau n \rceil\}$ for $0 < \tau \le 1$ and define

$$\Phi_{\max}(\mathcal{H}) = r_{[1]}, \qquad \Phi_{\mathrm{mean}}(\mathcal{H}) = \frac{1}{n}\sum_{i=1}^{n} r_i, \qquad \Phi_{\mathrm{top}}(\mathcal{H}) = \frac{1}{k_\tau(n)}\sum_{i=1}^{k_\tau(n)} r_{[i]}. \tag{9}$$

EvoTreeNAD uses Φ_{top}, while Φ_{mean} and $\Phi_{\max}$ define its mean and maximum variants. We use top, mean, and $\max$ for the corresponding family-value rules.

Routing starts at ρ and stops at the first node with remaining child capacity. At each saturated node v, the next node is

$$\min_{\prec} \arg\max_{c \in \mathrm{Ch}_t^\Phi(v)} V_t^\Phi(c), \tag{10}$$

where $\mathrm{Ch}_t^\Phi(v)$ denotes the children of v and $\prec$ is a fixed order for resolving ties. Let L_t^Φ denote the resulting root-to-node lineage.

The variation context C_t^Φ is constructed from the process state Ω_t^Φ and selected lineage L_t^Φ. It determines the conditional law of the next architecture and reward. The process evolves according to

$$\begin{aligned}(A_{t+1}^\Phi, R_{t+1}^\Phi) &\sim K(\cdot \mid C_t^\Phi),\\ \Omega_{t+1}^\Phi &\sim \mathrm{Grow}(\cdot \mid \Omega_t^\Phi, L_t^\Phi, A_{t+1}^\Phi, R_{t+1}^\Phi),\end{aligned} \tag{11}$$

where A_{t+1}^Φ and R_{t+1}^Φ denote the generated architecture and its reward, respectively. The state-update kernel Grow adds the generated architecture as a child of the selected endpoint and updates the retained variation information. All three policies use the same context construction, variation and state-update kernels, capacity rule, and tie rule. Each policy's routing decisions determine the selected lineages and their variation contexts, thereby shaping the architectures generated and the genealogy that develops.

A.2 Properties of Family-Value Rules

Proposition 1 establishes that EvoTreeNAD with $\Phi = \Phi_{\max}$ is pathwise equivalent to best-of-N greedy continuation. Proposition 2 quantifies the sensitivity of top-percentile family values to individual rewards and shows that maintaining a high value as a family grows requires a nonvanishing fraction of high-reward descendants.

In best-of-N greedy continuation, the current parent v generates $N = W(v)$ children in succession. The child with the highest immediate reward becomes the next parent, and the process repeats without returning to unchosen siblings. We use the same context construction, variation and state-update kernels, child capacities, and tie rule as in the genealogy-growth process.

Proposition 1 (Family maximum and best-of-N greedy continuation)**.** *EvoTreeNAD with $\Phi = \Phi_{\max}$ is pathwise equivalent to best-of-N greedy continuation. Starting from the same initial state and using the same random draws for variation and state updates, the two processes select the same lineage and generate the same architecture and reward at every iteration.*

Proof. Consider EvoTreeNAD with $\Phi = \Phi_{\max}$. Before a node v becomes saturated, routing stops at v whenever it reaches that node. Its children therefore have no descendants when v first becomes saturated, so their family maxima equal their immediate rewards. Let $c^\star(v)$ be the highest-reward child, with ties resolved by $\prec$. An expansion below $c^\star(v)$ cannot decrease its family maximum and leaves the sibling family values unchanged. The next visit to v therefore selects $c^\star(v)$ again. Induction over visits shows that this choice persists. Thus EvoTreeNAD generates $W(v)$ children before continuing through the highest-reward child, without returning to unchosen siblings.

We now compare this process with best-of-N greedy continuation using the same random draws. Their states agree initially. Suppose they agree before an iteration. The selection property above gives the same root-to-node lineage in both processes. Applying the same context construction to these identical states and lineages gives the same variation context. The kernel K, evaluated at this context with the same random draw, produces the same architecture and reward. The shared state-update kernel Grow, using the same update draw, then gives identical next states. Induction over iterations establishes the claimed pathwise equivalence. □

Proposition 2 (Support and bounded influence)**.** *Fix $0 < \tau \le 1$. Let $x = (x_1, \dots, x_n) \in [a, b]^n$, with $x_{[1]} \ge \cdots \ge x_{[n]}$ denoting their descending order. Set $k = \max\{1, \lceil \tau n \rceil\}$. The top-$k$ average $M_{k,n}(x) = k^{-1} \sum_{i=1}^{k} x_{[i]}$ gives the top-percentile family value for these rewards and satisfies*

$$M_{k,n}(x) = \max_{\substack{w_i \ge 0,\ \sum_i w_i = 1 \\ \|w\|_\infty \le 1/k}} \sum_{i=1}^{n} w_i x_i. \tag{12}$$

For a fixed family size, changing one reward by δ changes $M_{k,n}(x)$ by at most $|\delta|/k$. Moreover, for $N_q = |\{i : x_i \geq q\}|$ and every $q < b$,

$$\frac{N_q}{n} \geq \frac{k}{n} \frac{(M_{k,n}(x) - q)_+}{b - q}. \tag{13}$$

Consequently, maintaining a family value a fixed amount above q as the family grows requires a nonvanishing fraction of rewards at least q. Finitely many early high rewards cannot maintain this gap if subsequent rewards are at most q.

Proof. For $1 \leq k \leq n$, let

$$\mathcal{W}_{n,k} = \left\{ w \in \mathbb{R}_+^n : \sum_i w_i = 1,\ \|w\|_\infty \leq \frac{1}{k} \right\}.$$

If positive weight is assigned below the top k rewards while some top- k coordinate has weight below $1/k$, transferring the available mass to the higher reward cannot decrease the objective. Repeating this exchange puts weight $1/k$ on the top k rewards and zero elsewhere, proving Eq. (12).

The variational form also gives

$$|M_{k,n}(x + \delta e_j) - M_{k,n}(x)| \leq \sup_{w \in \mathcal{W}_{n,k}} w_j |\delta| \leq \frac{|\delta|}{k}.$$

For the support bound, if $N_q < k$, the top k rewards include all N_q rewards at least q. Bounding these by b and the remaining $k - N_q$ by q gives

$$M_{k,n}(x) \leq q + (b - q)\frac{N_q}{k}.$$

If $N_q \geq k$, this bound follows directly from $M_{k,n}(x) \leq b$. Rearranging and using $N_q \geq 0$ gives Eq. (13).

Finally, suppose only $r < \infty$ rewards in a growing family exceed q, while all subsequent rewards are at most q. When the family has rewards $x_1, \ldots, x_N$,

$$M_{k_\tau(N),N}(x_1, \ldots, x_N) \leq q + \frac{\min\{r, k_\tau(N)\}}{k_\tau(N)}(b - q).$$

Hence $\limsup_{N\to\infty} M_{k_\tau(N),N}(x_1, \ldots, x_N) \leq q$, so the family value cannot maintain a fixed positive gap above q. The family value also eventually falls below that of any sibling with a fixed family value greater than q. This proves the proposition. □

A.3 Stationary Selection-Variation

We study the long-run behavior of the genealogy-growth process by jointly tracking the reward law induced by each selected lineage context and the realized reward. Theorem 1 establishes the existence of stationary empirical regimes for this joint sequence. Within each regime, the distribution of generated rewards is the mixture of the lineage-conditioned reward laws. The theorem also bounds high-reward probability from below using the average top-percentile mean across these conditional reward laws.

Let $\mathcal{F}_t^\Phi$ denote the σ-algebra generated by the history available before iteration t. Define F_t^Φ as the conditional law of the next reward given this history, and let Z_t^Φ denote the pair consisting of this law and the realized reward,

$$F_t^\Phi = \mathcal{L}(R_{t+1}^\Phi \mid \mathcal{F}_t^\Phi) \in \mathcal{P}([a, b]), \qquad Z_t^\Phi = (F_t^\Phi, R_{t+1}^\Phi). \tag{14}$$

F_t^Φ is the reward marginal of $K(\cdot \mid C_t^\Phi)$. Let $\mathsf{E} = \mathcal{P}([a, b]) \times [a, b]$ be the state space of Z_t^Φ. Equip $\mathcal{P}([a, b])$ with the weak topology and use the corresponding product topologies on E and $\mathsf{E}^{\mathbb{N}_0}$. Define the empirical occupation measure on the sequence space $\mathsf{E}^{\mathbb{N}_0}$ by

$$\mathsf{Q}_{\Phi,T} = \frac{1}{T} \sum_{t<T} \delta_{\sigma^t \mathbf{Z}^\Phi}, \qquad \mathbf{Z}^\Phi = (Z_0^\Phi, Z_1^\Phi, \ldots), \tag{15}$$

where σ is the left shift on $\mathsf{E}^{\mathbb{N}_0}$. This is the empirical distribution of the shifted sequences $\sigma^t \mathbf{Z}^\Phi$, $0 \le t < T$.

A weak subsequential limit Q of $(\mathsf{Q}_{\Phi,T})_{T\ge 1}$ is called a *reachable stationary regime*. Let ν_Q and Λ_Q denote the two marginals of the time-zero coordinate pair under Q, corresponding to the conditional reward law and realized reward, respectively. These are the weak limits of the respective empirical distributions along the same subsequence. For $0 < \alpha \le 1$, define the top-percentile mean and its average under ν_Q by

$$U_\alpha(F) = \frac{1}{\alpha}\int_{1-\alpha}^{1} F^{-1}(u)\,du, \qquad \overline{U}_{\alpha,\mathsf{Q}} = \int U_\alpha(F)\,\nu_\mathsf{Q}(dF),$$

where F^{-1} is the quantile function of F, defined as the generalized inverse of its cumulative distribution function.

Theorem 1 (Stationary selection-variation)**.** *Every infinite realization of the genealogy-growth process admits weak subsequential limits of $\mathsf{Q}_{\Phi,T}$, and every such limit is shift-stationary. Almost surely, every reachable stationary regime Q satisfies the following properties.*

1. Reward distributions and exceedance frequencies. *For every Borel set $A \subseteq [a,b]$,*

$$\Lambda_\mathsf{Q}(A) = \int_{\mathcal{P}([a,b])} F(A)\,\nu_\mathsf{Q}(dF). \tag{16}$$

If $\mathsf{Q}_{\Phi,T_j} \Rightarrow \mathsf{Q}$, then at every threshold q satisfying $\Lambda_\mathsf{Q}(\{q\}) = 0$,

$$\lim_{j\to\infty} \frac{1}{T_j}\sum_{t<T_j} \mathbf{1}\{R_{t+1}^\Phi \ge q\} = \Lambda_\mathsf{Q}([q,b]) = \int F([q,b])\,\nu_\mathsf{Q}(dF). \tag{17}$$

For each fixed $q < b$, the difference between the empirical exceedance frequency and the average conditional exceedance probability converges to zero almost surely along the full iteration sequence.

2. High-reward probability bound. *For every $0 < \alpha \le 1$ and $q < b$,*

$$\Lambda_\mathsf{Q}([q,b]) \ge \frac{\alpha}{b-q}\big(\overline{U}_{\alpha,\mathsf{Q}} - q\big)_+. \tag{18}$$

Proof. Since $[a,b]$ is compact, $\mathcal{P}([a,b])$ is compact metrizable under the weak topology. Thus E and its countable product $\mathsf{E}^{\mathbb{N}_0}$ are compact metrizable. By weak compactness of $\mathcal{P}(\mathsf{E}^{\mathbb{N}_0})$, the sequence $(\mathsf{Q}_{\Phi,T})_{T\ge 1}$ has a weakly convergent subsequence. For every bounded continuous f,

$$\mathsf{Q}_{\Phi,T}(f - f\circ\sigma) = \frac{f(\mathbf{Z}^\Phi) - f(\sigma^T\mathbf{Z}^\Phi)}{T} \longrightarrow 0.$$

Every subsequential limit is shift invariant.

For $h \in C([a,b])$, set

$$D_{t+1}^{\Phi,h} = h(R_{t+1}^\Phi) - \int h(r) F_t^\Phi(dr).$$

These are bounded martingale differences. The martingale strong law, applied simultaneously to a countable sup-norm-dense subset of $C([a,b])$, gives $T^{-1}\sum_{t<T} D_{t+1}^{\Phi,h} \to 0$ on one probability-one event. The map $(F,r) \mapsto h(r) - \int h\,dF$, composed with the first-coordinate projection, is bounded continuous on $\mathsf{E}^{\mathbb{N}_0}$. Passing to any weak subsequential limit of $\mathsf{Q}_{\Phi,T}$ and extending by uniform approximation yields

$$\int h(r)\Lambda_\mathsf{Q}(dr) = \int\left[\int h(r)F(dr)\right]\nu_\mathsf{Q}(dF)$$

for every continuous h. Equality of these probability measures proves Eq. (16) for every Borel set.

The generated-reward marginal of Q_{Φ,T_j} is $T_j^{-1}\sum_{t<T_j}\delta_{R_{t+1}^\Phi}$. Marginalization is continuous under weak convergence, so these empirical reward measures converge weakly to Λ_Q. If $\Lambda_\mathsf{Q}(\{q\}) = 0$, the set $[q,b]$ is a Λ_Q-continuity set. The Portmanteau theorem and Eq. (16) then give Eq. (17).

For each fixed $q < b$, the sequence $\mathbf{1}\{R^{\Phi}_{t+1} \geq q\} - F^{\Phi}_t([q, b])$ is also a bounded martingale-difference sequence. Its strong law gives

$$\frac{1}{T}\sum_{t<T}\left[\mathbf{1}\{R^{\Phi}_{t+1} \geq q\} - F^{\Phi}_t([q, b])\right] \longrightarrow 0 \quad \text{almost surely.} \tag{19}$$

For any law F, $0 < \alpha \leq 1$, and $q < b$, the mass at or above q within the highest-reward α fraction is at most $F([q, b])$. Bounding rewards at or above q by b and those below q by q gives

$$\alpha U_\alpha(F) \leq \alpha q + (b - q)F([q, b]).$$

Rearranging and using $F([q, b]) \geq 0$ yields

$$F([q, b]) \geq \frac{\alpha}{b - q}\big(U_\alpha(F) - q\big)_+.$$

Integrating under ν_{Q} and applying Jensen's inequality to the positive-part map proves Eq. (18). □

Theorem 1 connects the variation contexts explored by EvoTreeNAD to the architectures it produces during continued genealogy growth. In each reachable stationary regime, if the top-percentile means of the encountered conditional reward laws average above q, the theorem gives a positive lower bound on the probability of generating architectures with rewards at least q.

A.4 Top-Percentile Routing and High-Reward Production

Under the variation assumptions below, we connect sustained top-percentile family values to high-reward production in these regimes and characterize how mean and maximum routing differ.

Lineage-conditioned variation. We keep the component reward distributions fixed across lineage contexts to compare how family-value rules respond to changes in high- and weak-outcome probabilities. Fix $0 < \tau < 1$. Let r_0 be the intermediate reward and q the high-reward threshold, with $a \leq r_- < r_0 < q \leq r_+ \leq b$. Let F_- and F_+ be fixed weak- and high-reward distributions supported on $[a, r_-]$ and $[r_+, b]$, respectively. For a lineage context z, define the conditional reward law

$$F_z = \ell(z)F_- + \big(1 - p_q(z) - \ell(z)\big)\delta_{r_0} + p_q(z)F_+, \tag{20}$$

where $p_q(z)$ and $\ell(z)$ denote the probabilities of high- and weak-reward outcomes conditional on z, respectively. We assume $p_q(z) + \ell(z) \leq 1$, $0 < p_q(z) < \tau$, and $0 < \ell(z) < 1 - \tau$. The same conditional law is used by every routing rule at the same context. Let μ_- and μ_+ denote the means of F_- and F_+, respectively. The highest-reward τ fraction contains all high-reward outcomes and no weak outcomes. Therefore,

$$U_\tau(F_z) = r_0 + \frac{\mu_+ - r_0}{\tau}p_q(z), \qquad \int rF_z(dr) = r_0 + (\mu_+ - r_0)p_q(z) - (r_0 - \mu_-)\ell(z). \tag{21}$$

Averaging the reward laws of adaptively selected contexts changes only their mixture weights; the component distributions remain fixed. Hence the same identities hold for the averaged reward law. Write $F_{p,\ell}$ for the same mixture with high- and weak-outcome probabilities p and ℓ. At iteration t, the context is C^{Φ}_t, so $F^{\Phi}_t = F_{C^{\Phi}_t}$.

Recurrent families. Fix a family-value rule Φ and an infinite realization of its genealogy-growth process. For every created node v, let $N^{\Phi}_T(v)$ denote the number of iterations before T whose selected lineage contains v,

$$N^{\Phi}_T(v) = \sum_{t<T}\mathbf{1}\{v \in L^{\Phi}_t\}. \tag{22}$$

Let $\mathcal{I}_\Phi$ denote the set of recurrent nodes, which are visited infinitely often by selected lineages,

$$\mathcal{I}_\Phi = \{v : N^{\Phi}_T(v) \to \infty\}.$$

For a saturated node v, let $\mathrm{Ch}(v)$ denote its children. Its recurrent children form the set

$$\mathcal{R}_\Phi(v) = \{c \in \mathrm{Ch}(v) : N^{\Phi}_T(c) \to \infty\}. \tag{23}$$

For a recurrent node v, let $t_n^{\Phi,v}$ denote the n-th iteration whose selected lineage contains v,

$$t_n^{\Phi,v} = \inf\{t \geq 0 : N_{t+1}^{\Phi}(v) = n\}.$$

For every $v \in \mathcal{I}_\Phi$, including the root, we assume

$$\frac{1}{n}\sum_{i=1}^{n} p_q(C_{t_i^{\Phi,v}}^{\Phi}) \longrightarrow \rho_{\Phi,v}(q), \qquad \frac{1}{n}\sum_{i=1}^{n} \ell(C_{t_i^{\Phi,v}}^{\Phi}) \longrightarrow \overline{\ell}_{\Phi,v}. \tag{24}$$

The quantities $\rho_{\Phi,v}(q)$ and $\overline{\ell}_{\Phi,v}$ are the limiting average conditional probabilities of high- and weak-reward outcomes, respectively, over iterations whose selected lineages contain v. These limits may differ across policies because the policies generate different contexts.

For every created non-root node c and every T at or after its creation, let $\widehat{\Lambda}_{c,T}$ denote the empirical reward law of its descendant family. The family size and this law satisfy

$$|\mathcal{H}_T^{\Phi}(c)| = 1 + N_T^{\Phi}(c), \qquad \widehat{\Lambda}_{c,T} = \frac{\delta_{R^{\Phi}(c)} + \sum_{t<T:c\in L_t^{\Phi}} \delta_{R_{t+1}^{\Phi}}}{1 + N_T^{\Phi}(c)}. \tag{25}$$

On the iteration that creates c, routing stops at its parent. Every later iteration whose route contains c inserts exactly one architecture into the subtree rooted at c, and every later descendant is created by one such iteration. This proves Eq. (25).

Lemma 1 (Descendant-family law and family-value convergence)**.** *For every non-root node c, let $I_t^c = \mathbf{1}\{c \in L_t^{\Phi}\}$. On one probability-one event, simultaneously for every created non-root node c and every $h \in C([a,b])$,*

$$N_T^{\Phi}(c) \to \infty \quad \Longrightarrow \quad \frac{1}{N_T^{\Phi}(c)}\sum_{t<T} I_t^c \left(h(R_{t+1}^{\Phi}) - \int h\, dF_t^{\Phi}\right) \longrightarrow 0. \tag{26}$$

The same statement holds at the root with $I_t^{\rho} \equiv 1$ and $N_T^{\Phi}(\rho) = T$.

Under the stated conditions on the variation law in Eq. (20) *and the convergence assumption in Eq.* (24)*, the empirical family law of every recurrent non-root node c converges along the full sequence of iterations,*

$$\widehat{\Lambda}_{c,T} \Rightarrow F_{\rho_{\Phi,c}(q),\overline{\ell}_{\Phi,c}}. \tag{27}$$

The top-percentile and full-family mean values computed from the same reward multiset $\mathcal{H}_T^{\Phi}(c)$ satisfy

$$\lim_{T\to\infty} \Phi_{\mathrm{top}}\big(\mathcal{H}_T^{\Phi}(c)\big) = r_0 + \frac{\mu_+ - r_0}{\tau}\rho_{\Phi,c}(q), \tag{28}$$

$$\lim_{T\to\infty} \Phi_{\mathrm{mean}}\big(\mathcal{H}_T^{\Phi}(c)\big) = r_0 + (\mu_+ - r_0)\rho_{\Phi,c}(q) - (r_0 - \mu_-)\overline{\ell}_{\Phi,c}. \tag{29}$$

The empirical frequencies of $R_{t+1}^{\Phi} \geq q$ and $R_{t+1}^{\Phi} \leq r_-$ over the same routed iterations converge to $\rho_{\Phi,c}(q)$ and $\overline{\ell}_{\Phi,c}$, respectively.

Proof. Fix $h \in C([a,b])$ and a node address c, taking $I_t^c = 0$ before its creation, and write

$$D_{t+1} = h(R_{t+1}^{\Phi}) - \int h\, dF_t^{\Phi}, \qquad \mathcal{M}_T = \sum_{t<T} \frac{I_t^c D_{t+1}}{1 + N_t^{\Phi}(c)}.$$

The weights are measurable with respect to the history before each iteration, so $\mathcal{M}_T$ is a martingale with respect to $\mathcal{F}_T^{\Phi}$. Since $|D_{t+1}| \leq 2\|h\|_\infty$ and the denominator at the n-th selected iteration is n, orthogonality of martingale increments gives

$$\sup_T \mathbb{E}[\mathcal{M}_T^2] \leq 4\|h\|_\infty^2 \sum_{n\geq 1} n^{-2} < \infty.$$

Thus $\mathcal{M}_T$ converges almost surely. On paths with $N_T^{\Phi}(c) \to \infty$, the series $\sum_{n\geq 1} D_{t_n^{\Phi,c}+1}/n$ converges, and Kronecker's lemma gives $n^{-1}\sum_{i=1}^{n} D_{t_i^{\Phi,c}+1} \to 0$. Between successive visits to c, the numerator in Eq. (26) and the visit count $N_T^{\Phi}(c)$ remain unchanged. Applying this argument to a countable dense subset of the unit ball of $C([a,b])$, intersecting over the countable set of finite node

addresses, and extending by uniform approximation proves Eq. (26). The same argument applies at the root. For the two reward indicators, the conditional expectations are $p_q(C_t^\Phi)$ and $\ell(C_t^\Phi)$. The same argument, together with Eq. (24), yields both frequency limits.

The average conditional reward law over the first n iterations routed through c is

$$\frac{1}{n}\sum_{i=1}^{n} F_{C^\Phi_{t_i^{\Phi,c}}} = F_{\overline{p}_{\Phi,c,n},\overline{\ell}_{\Phi,c,n}},$$

where $\overline{p}_{\Phi,c,n}$ and $\overline{\ell}_{\Phi,c,n}$ denote the average conditional probabilities of high- and weak-reward outcomes over these n visits, respectively. By Eq. (24), this mixture converges to $F_{\rho_{\Phi,c}(q),\overline{\ell}_{\Phi,c}}$. The empirical family law in Eq. (25) has the same weak limit. For each continuous test function, Eq. (26) shows that the martingale average converges to zero. The reward of c has vanishing weight in the empirical family law and therefore does not affect its weak limit. This proves Eq. (27) along the full sequence of iterations.

Let $m_T = |\mathcal{H}_T^\Phi(c)|$ denote the family size and $\alpha_T = k_\tau(m_T)/m_T$ the fraction of family rewards included in the top-percentile average. Then $\alpha_T \geq \tau$ and $\alpha_T \to \tau$, and the empirical top-percentile family value is $U_{\alpha_T}(\widehat{\Lambda}_{c,T})$. Let $F_* = F_{\rho_{\Phi,c}(q),\overline{\ell}_{\Phi,c}}$. On the bounded reward interval, weak convergence gives $W_1(\widehat{\Lambda}_{c,T}, F_*) \to 0$, where W_1 denotes the Wasserstein–1 distance. Its quantile representation and $\alpha_T \geq \tau$ give

$$\left|U_{\alpha_T}(\widehat{\Lambda}_{c,T}) - U_\tau(F_*)\right| \leq \frac{W_1(\widehat{\Lambda}_{c,T}, F_*) + (b-a)(\alpha_T - \tau)}{\alpha_T} \longrightarrow 0.$$

The identity for U_τ in Eq. (21) also holds at the limiting mixture weights, giving Eq. (28). Weak convergence on the bounded reward interval also gives convergence of the means, proving Eq. (29). □

Lemma 1 establishes family-value convergence for adaptively accumulated descendant rewards. We now use the routing rule to relate these limits across recurrent families and connect top-percentile family values to the high-reward production rate of the run. Let $A = \mu_+ - r_0 > 0$ and $B = r_0 - \mu_- > 0$.

Theorem 2 (Top-percentile routing and recurrent high-reward production)**.** *Under the stated conditions on the variation law in Eq.* (20) *and the convergence assumption in Eq.* (24)*, the following conclusions hold almost surely.*

1. Top-percentile. *For every fixed non-root $v \in \mathcal{I}_{\mathrm{top}}$ and every reachable stationary regime Q of the same top-percentile run,*

$$\lim_{t\to\infty} V_t^{\mathrm{top}}(v) = U_\tau(\Lambda_{\mathsf{Q}}) = r_0 + \frac{\mu_+ - r_0}{\tau}\Lambda_{\mathsf{Q}}([q,b]). \tag{30}$$

   For every $v \in \mathcal{I}_{\mathrm{top}}$, all recurrent children $c \in \mathcal{R}_{\mathrm{top}}(v)$ satisfy $\rho_{\mathrm{top},c}(q) = \rho_{\mathrm{top},v}(q)$.

2. Full-family mean. *For every $v \in \mathcal{I}_{\mathrm{mean}}$, all children $c \in \mathcal{R}_{\mathrm{mean}}(v)$ attain the same maximal limiting family value among all children of v. The quantity $A\rho_{\mathrm{mean},c}(q) - B\overline{\ell}_{\mathrm{mean},c}$ is constant over $c \in \mathcal{R}_{\mathrm{mean}}(v)$.*

   The gap between the maximal high-reward frequency among recurrent children and that over iterations routed through v satisfies

$$\begin{aligned}&\max_{c\in\mathcal{R}_{\mathrm{mean}}(v)} \rho_{\mathrm{mean},c}(q) - \rho_{\mathrm{mean},v}(q)\\ &\quad= \frac{B}{A}\left(\max_{c\in\mathcal{R}_{\mathrm{mean}}(v)} \overline{\ell}_{\mathrm{mean},c} - \overline{\ell}_{\mathrm{mean},v}\right).\end{aligned} \tag{31}$$

   This gap is positive if and only if there exists $c \in \mathcal{R}_{\mathrm{mean}}(v)$ such that

$$\overline{\ell}_{\mathrm{mean},c} < \max_{d\in\mathcal{R}_{\mathrm{mean}}(v)} \overline{\ell}_{\mathrm{mean},d}, \qquad \limsup_{T\to\infty} \frac{N_T^{\mathrm{mean}}(c)}{N_T^{\mathrm{mean}}(v)} > 0.$$

3. Family maximum. *EvoTreeNAD with maximum-family routing is pathwise equivalent to matched best-of-N greedy continuation (Proposition 1).*

The following identities hold for all recurrent nodes in the respective genealogies,

$$\begin{aligned} \rho_{\mathrm{top},v}(q) &= \rho_{\mathrm{top},\rho}(q), && v \in \mathcal{I}_{\mathrm{top}}, \\ A\rho_{\mathrm{mean},v}(q) - B\bar{\ell}_{\mathrm{mean},v} &= A\rho_{\mathrm{mean},\rho}(q) - B\bar{\ell}_{\mathrm{mean},\rho}, && v \in \mathcal{I}_{\mathrm{mean}}. \end{aligned} \tag{32}$$

For each rule, the realized long-run frequency of architectures with reward at least q equals $\rho_{\Phi,\rho}(q)$.

Proof. Let $\Phi \in \{\Phi_{\mathrm{top}}, \Phi_{\mathrm{mean}}\}$ and fix $v \in \mathcal{I}_\Phi$. Since v is recurrent and has finite child capacity, it eventually becomes saturated. Lemma 1 establishes convergence of the family value along the full iteration sequence for every recurrent child. Every other child is routed through only finitely many times, so its family value is eventually constant. Thus all children of v have limiting family values. Each route through a saturated node enters one of finitely many children, so every recurrent node has at least one recurrent child. Every ancestor of a recurrent node is also recurrent.

Suppose a child $c \in \mathcal{R}_\Phi(v)$ had a limiting family value strictly below that of another child d. For all sufficiently large t, $V_t^\Phi(d) > V_t^\Phi(c)$. Whenever a later route reaches v, rooted routing therefore cannot enter c. Its visitation count would become constant, contradicting $N_T^\Phi(c) \to \infty$. Hence every recurrent child attains the same maximal limiting family value among all children of v. Under top-percentile routing, Eq. (28) implies that the recurrent child families have the same limiting high-reward frequency. Under mean routing, Eq. (29) shows that $A\rho_{\mathrm{mean},c}(q) - B\bar{\ell}_{\mathrm{mean},c}$ is constant over $c \in \mathcal{R}_{\mathrm{mean}}(v)$.

To relate the limiting high- and weak-reward frequencies for v and its children, define the local visitation ratios $w_T(c) = N_T^\Phi(c)/N_T^\Phi(v)$. After the finite pre-saturation phase, each route through v passes through exactly one child. Thus

$$\sum_{t<T:v\in L_t^\Phi} p_q(C_t^\Phi) = O(1) + \sum_{c\in \mathrm{Ch}(v)} \sum_{t<T:c\in L_t^\Phi} p_q(C_t^\Phi),$$

with the same identity for ℓ. Children outside $\mathcal{R}_\Phi(v)$ contribute only finitely many terms. Dividing by $N_T^\Phi(v)$ and applying Eq. (24) gives, as $T \to \infty$,

$$\begin{aligned} \rho_{\Phi,v}(q) &= \sum_{c\in\mathcal{R}_\Phi(v)} w_T(c)\rho_{\Phi,c}(q) + o(1), \\ \bar{\ell}_{\Phi,v} &= \sum_{c\in\mathcal{R}_\Phi(v)} w_T(c)\bar{\ell}_{\Phi,c} + o(1), \qquad \sum_{c\in\mathcal{R}_\Phi(v)} w_T(c) \longrightarrow 1. \end{aligned} \tag{33}$$

These identities remain valid when $N_T^\Phi(v)/T$ tends to zero or the local visitation ratios $w_T(c)$ oscillate.

For top-percentile routing, the common high-reward frequency among recurrent child families and Eq. (33) give

$$\rho_{\mathrm{top},v}(q) = \max_{c\in\mathcal{R}_{\mathrm{top}}(v)} \rho_{\mathrm{top},c}(q). \tag{34}$$

Thus $\rho_{\mathrm{top},c}(q) = \rho_{\mathrm{top},v}(q)$ for every $c \in \mathcal{R}_{\mathrm{top}}(v)$. Applying this equality along the finite path from the root gives the top-percentile relation in Eq. (32).

For each of the three rules, $N_T^\Phi(\rho) = T$, and the indicator case of Lemma 1 identifies the root rate with the realized high-reward frequency.

To identify the stationary reward law, apply Lemma 1 at the root to continuous test functions. Together with Eq. (24), this gives

$$\frac{1}{T}\sum_{t<T} \delta_{R_{t+1}^\Phi} \Rightarrow F_{\rho_{\Phi,\rho}(q),\bar{\ell}_{\Phi,\rho}} \quad \text{along the full iteration sequence.}$$

The left side is the generated-reward marginal of $\mathsf{Q}_{\Phi,T}$. Hence every reachable stationary regime Q of this run has $\Lambda_{\mathsf{Q}} = F_{\rho_{\Phi,\rho}(q),\bar{\ell}_{\Phi,\rho}}$. Equation (21) also holds at the limiting mixture weights, so

$$\Lambda_{\mathsf{Q}}([q,b]) = \rho_{\Phi,\rho}(q), \qquad U_\tau(\Lambda_{\mathsf{Q}}) = r_0 + \frac{\mu_+ - r_0}{\tau}\rho_{\Phi,\rho}(q).$$

For $\Phi = \Phi_{\text{top}}$, combine this identity with Eqs. (28) and (32) to obtain Eq. (30). The common probability-one event in Lemma 1 makes the conclusion simultaneous for all fixed recurrent families and all reachable regimes of the run.

For mean routing, the common limiting family value of recurrent children and Eq. (33) give

$$A\rho_{\text{mean},c}(q) - B\overline{\ell}_{\text{mean},c} = A\rho_{\text{mean},v}(q) - B\overline{\ell}_{\text{mean},v} \quad (c \in \mathcal{R}_{\text{mean}}(v)).$$

Subtracting and taking maxima over the same child set proves Eq. (31).

Let $\ell_* = \max_{c \in \mathcal{R}_{\text{mean}}(v)} \overline{\ell}_{\text{mean},c}$. Equation (33) implies

$$\ell_* - \overline{\ell}_{\text{mean},v} = \lim_{T\to\infty} \sum_{c \in \mathcal{R}_{\text{mean}}(v)} w_T(c)\big(\ell_* - \overline{\ell}_{\text{mean},c}\big).$$

There are finitely many nonnegative summands. The limit is positive exactly when a child with $\overline{\ell}_{\text{mean},c} < \ell_*$ has $\limsup_T w_T(c) > 0$, proving the strictness criterion.

Applying the parent-child identity under mean routing along the finite path from the root gives the mean relation in Eq. (32). This holds simultaneously for all recurrent nodes on the probability-one event of Lemma 1.

The claim for maximum-family routing follows from Proposition 1. □

Under the stated variation assumptions, Theorem 2 shows that the limiting top-percentile value of each recurrent non-root family exactly characterizes the probability of generating high-reward architectures in the stationary regimes of the run. Sustaining a high value in a growing family requires a nonvanishing fraction of high-reward descendants, not just isolated early successes (Proposition 2). Full-family means also depend on weak outcomes and can assign equal limiting values to families with different high-reward production rates. Maximum-family routing fixes the winning child at each generation and is pathwise equivalent to best-of-N greedy continuation (Proposition 1). EvoTreeNAD can therefore continue developing successful architectures and redirect evolution toward retained alternatives as new descendant evidence accumulates.

B Additional Method Details

B.1 Hyperparameters

Table 6 summarizes the hyperparameter definitions and representative settings used across tasks. Values marked task-dependent vary with the dataset and discovery budget. Representative reward anchors include $(v_e, v_l) = (0.8, 0.92)$ for CIFAR-10 and $(0.55, 0.72)$ for CIFAR-100.

For variation from an architecture node, agent context includes the selected architecture and its evaluation results, together with records from up to k_{idea} recent ancestors for the Idea Agent and k_{code} for the Code Agent. These limits retain recent design history while controlling context length.

EvoTreeNAD optionally adds small random perturbations to sibling node-family values before branch selection to encourage exploration of alternative lineages.

B.2 Discovery-Stage Evaluation

LLM-generated architectures are not guaranteed to execute correctly or train successfully, and even trainable candidates may perform poorly. Our discovery-stage evaluation is designed to accommodate these generation failures and assess candidate performance efficiently, providing the validation feedback needed for continued evolution.

Execution checks. Generated implementations undergo a smoke test for executability and compatibility with the task interface. Only implementations that pass proceed to short-budget evaluation.

Table 6: EvoTreeNAD hyperparameters, definitions, and representative settings. Task-dependent values vary by dataset and budget.

Name	Symbol	Description	Representative setting
Child capacity	$W(s)$	Maximum number of direct children of node s. Capacities may differ across nodes; for example, $W(\rho) = 15$ at the root and $W(s) = 7$ at non-root nodes.	task-dep.
Code realization attempts	$A_{\max}$	Maximum Code-Agent attempts within one proposed variation.	1–3
Rejection tolerance	P_{tolerant}	Failed expansions skipped per parent before unresolved failures are recorded.	4–8
Top-percentile fraction	τ	Fraction of the highest rewards averaged over a node and all its descendants (Eq. 3).	0.1–0.2
Idea ancestors	k_{idea}	Maximum number of recent ancestors provided to the Idea Agent as context.	1–3
Code ancestors	k_{code}	Maximum number of recent ancestors provided to the Code Agent as context.	1–3
Early-stop reference fraction	q_{stop}	Fraction of prior candidates selected for strong performance to construct reference validation trajectories.	0.1–0.2
Early-stop margin	δ	Base margin below reference performance, reduced at later training stages.	0.01–0.08
Reward center	v_e	Metric value corresponding to the midpoint of the sigmoid reward mapping (Eq. 1).	task-dep.
Reward target	v_l	Target metric value to approach in reward mapping.	task-dep.
Reward slope	κ	Sharpness of sigmoid transition.	2
Size coefficient	λ_{size}	Coefficient for the compactness term in the reward (Eq. 1).	0.0–0.01
Short-budget training	$\{\eta, T_{\max}, B, T_{\text{eval}}, T_{\text{runtime}}\}$	Short-budget training settings: learning rate η, max steps $T_{\max}$, batch size B, evaluation interval T_{eval}, and training runtime constraint T_{runtime}.	task-dep.

Short-budget evaluation with early stopping. During discovery, architectures are trained under a fixed short-budget recipe and evaluated on the task validation set. We use early stopping to reduce computation spent on poorly performing candidates. The reference group comprises the strongest prior candidates among those with the longest available validation histories, ranked by their final recorded scores. The reference fraction q_{stop} controls the size of this group. For the accuracy and AUC metrics used here, the stopping threshold is based on the lowest score in this group at the same training step, with δ setting the base allowable shortfall below that score. Training stops when the candidate's validation score falls below this threshold. Recorded validation results, including those obtained before early stopping, provide the discovery scores used to compute node rewards.

Failure handling. Failed execution checks or training attempts that yield no valid score prompt the Code Agent to revise its implementation using diagnostic feedback. Each proposed variation allows at most $A_{\max}$ Code-Agent attempts, including the initial attempt, to limit the cost of repeated attempts. If none produces a valid score, the expansion is counted as failed. For each parent, the first P_{tolerant} failed expansions are skipped without adding a child. This rejection tolerance accommodates occasional unsuccessful variations. Further failed expansions are recorded under that parent with a fixed negative reward, so persistent failures contribute to family values. Intermediate retry attempts are not inserted as separate children.

C Model Interface

The CIFAR-10 model interface contract used in our experiments is shown below. Other tasks use analogous contracts for initialization, model inputs and outputs, and task-specific requirements. Training uses the loss returned by `forward`, while evaluation uses the logits returned by `predict` and the fixed task metric.

```
"""
model_requirement defines the structural and functional conditions that a model must satisfy
↪  for a particular predictive task. These conditions specify interface compliance;
↪  predictive quality is determined by evaluator feedback.
For functions in the model:
 - Input signature: Each method takes one or more named tensor arguments, specified by
 ↪  parameter names and their shapes/types.
 - Output format: Each method returns a dictionary where each key corresponds to a named
 ↪  output tensor with a defined shape and type.
```

```
 - Ensure the methods include the specified arguments or keys, but they don't need to be
 ↪  limited to them. Adding extra and useful keys to the output dictionary is encouraged to
 ↪  support interaction between modules.
     - For example, you are free to output additional keys in the outputs dictionary.

Note: We use `predict` to output the predicted logit for evaluation. Do not use the ground
↪  truth label as the input of the `predict` function. We use `forward` to train the model.
↪  Use CPU for executable validation.
Contract: `forward` and `predict` method must be tolerant of extra, unknown keyword arguments -
↪  implement the signature with **kwargs and silently ignore unrecognized keys.
"""

model_requirements = {
    "model_name": "ImageClfModel",
    "purpose": "Predict label of image",
    "data_background": "label_num is 10. the pixel_values are normalized.",
    "init_parameters": {"label_num": {"type": "int"}, "base_dim": {"type": "int"},
    ↪  "model_depth": {"type": "int"}},
    "methods": {
        "forward": {
            "inputs": {
                "pixel_values": {"shape": "(batch_size, 3, 32, 32)", "type":
                ↪  "torch.FloatTensor"},
                "label": {"shape": "(batch_size)", "type": "torch.LongTensor"},
            },
            "outputs": {
                "logits": {"shape": "(batch_size, label_num)", "type": "torch.FloatTensor"},
                "loss": {"shape": "()", "type": "torch.FloatTensor"}
            }
        },
        "predict": {
            "inputs": {
                "pixel_values": {"shape": "(batch_size, 3, 32, 32)", "type":
                ↪  "torch.FloatTensor"},
            },
            "outputs": {
                "logits": {"shape": "(batch_size, label_num)", "type": "torch.FloatTensor"},
            }
        },
    },
    "other_requirements": ["The `base_dim` is typically set to 32 and defines a base dim of
    ↪  the model. Specific modules adjust this dimension by expanding or reducing it as
    ↪  needed. (base_dim*2 = 64, base_dim*4=128, base_dim*8=256, ...)",
                           "The `model_depth` defines the number of main layers or blocks in
                           ↪  the model at a macro level - typically values like 12, 15, or
                           ↪  18. To create layers or stages, adapt them based on
                           ↪  `model_depth`. E.g., use a safe splitting function or define
                           ↪  stage_depth = ratio * model_depth.",
                           "Sample size is about 50k, not large. Be cautious about overfitting
                           ↪  and early saturation."]
}
```

D Dataset and Protocol Details

D.1 Discovery Protocol

Main discovery uses three independent EvoTreeNAD runs per task. We analyze continued architecture evolution in two 300-iteration CIFAR-10 discovery runs. MedMNIST-v2 runs use 80 iterations. CIFAR-10 discovery-strategy and agent-configuration comparisons use 100 iterations per run. Prompts use generic dataset paths during discovery.

Main results use GPT-4.1 for architectural proposals and OSS20B for code realization. GPT-5 is used only in the supplementary agent-configuration study. GPT-4.1 and GPT-5 are accessed through Azure OpenAI (`gpt-4.1` and `gpt-5-chat`); OSS20B is run locally from the `gpt-oss-20b-mxfp4.gguf` checkpoint on two V100 GPUs. Each discovery run uses one A100 GPU for candidate evaluation, and full-fidelity training uses A100 or V100 GPUs. Representative hyperparameter ranges and prompt templates appear in Appendices B.1 and F; run-specific configuration files accompany the anonymous code.

D.2 MedMNIST-v2 Tasks and Data

We evaluate EvoTreeNAD on six MedMNIST-v2 medical image classification tasks (Yang et al., 2023) spanning 2D images and 3D volumes across multiple imaging modalities. Table 7 summarizes the metadata and split statistics for each dataset. The tasks are:

- **PathMNIST:** Multi-class classification of colon pathology patches extracted from hematoxylin and eosin (H&E) stained histological images.
- **OCTMNIST:** Multi-class diagnosis of retinal diseases from optical coherence tomography (OCT) scans.
- **TissueMNIST:** Eight-class classification of human kidney cortex cells from microscopy images.
- **VesselMNIST3D:** Binary classification of healthy and aneurysmal intracranial vessel segments using MRA-derived 3D shapes.
- **SynapseMNIST3D:** Binary classification of excitatory and inhibitory synapses from electron microscopy.
- **OrganMNIST3D:** Eleven-class classification of body organs from CT scans.

Table 7: MedMNIST-v2 Dataset Statistics.

Dataset	Data Modality	Task Type	Input Shape	#(Train)	#(Val)	#(Test)	Total
PathMNIST	Colon Pathology (Histology)	Multi-Class (9)	$3 \times 64 \times 64$	89,996	10,004	7,180	107,180
OCTMNIST	Retinal OCT	Multi-Class (4)	$1 \times 64 \times 64$	97,477	10,832	1,000	109,309
TissueMNIST	Kidney Cortex Microscopy	Multi-Class (8)	$1 \times 64 \times 64$	165,466	23,640	47,280	236,386
VesselMNIST3D	Brain MRA (Shape)	Binary (2)	$1 \times 64 \times 64 \times 64$	1,335	191	382	1,908
SynapseMNIST3D	Electron Microscopy	Binary (2)	$1 \times 64 \times 64 \times 64$	1,230	177	352	1,759
OrganMNIST3D	Abdominal CT	Multi-Class (11)	$1 \times 64 \times 64 \times 64$	971	161	610	1,742

D.3 Full-Fidelity Evaluation Protocol

All architectures selected after evolution are trained from scratch on the full official training split of each dataset under the task-specific full-fidelity settings summarized in Table 8. The additional regularization and augmentation used at this stage are fixed within each task and are not tuned per architecture.

Table 8: Full-fidelity training settings for discovered models. All runs use the AdamW optimizer and a cosine-decay learning-rate schedule.

Dataset	Epochs	Batch Size	Learning Rate
CIFAR-10	500	128	0.002
CIFAR-100	500	128	0.001
MedMNIST-v2 2D	60	160	0.001
MedMNIST-v2 3D	200	64	0.0005
CIFAR-10/100: DropPath, MixUp, and CutMix.			
MedMNIST-v2 2D: DropPath and MixUp; 3D: DropPath.			

E Additional Experimental Analyses

These analyses examine the architectures discovered by EvoTreeNAD and their development through continued evolution. We present representative discovered architectures, architecture performance along principal lineages, continued architecture evolution, illustrations of discovery strategies, and agent configurations and costs.

E.1 Representative Discovered Architectures

Table 9 summarizes the structures of representative architectures discovered across the evaluated task families.

Table 9: Representative architectures discovered by EvoTreeNAD from an empty root.

Task family	Representative discovered structures
CIFAR-10	Dynamic routing with stage transformers and cross-stage fusion; fractal MBConv with pyramid pooling.
CIFAR-100	Mixed-kernel MBConv with FFT context, ODE blocks, and cross-stage concatenation; Res2Net-style branches with gated expansion and channel attention.
MedMNIST-v2 2D	Selective-kernel layers with GeM pooling; inverted-residual hierarchies with squeeze-and-excitation.
MedMNIST-v2 3D	Depthwise-separable or MBConv-style 3D residual networks with task-specific frequency blocks, normalization, and anti-aliased downsampling.

E.2 Architecture Performance Along Principal Lineages

To examine the quality of discovered architectures along the principal lineage extracted after each run, we evaluate three architectures spanning its early and later stages under the full-fidelity protocol.

- **Architecture I** (Selected Node 1): the first architecture after the empty root, providing a direct-generation reference.
- **Architecture II** (Selected Node 2): the highest-reward architecture in the first half of the lineage after excluding Architecture I.
- **Architecture III** (Selected Node 3): the highest-reward architecture in the second half of the lineage.

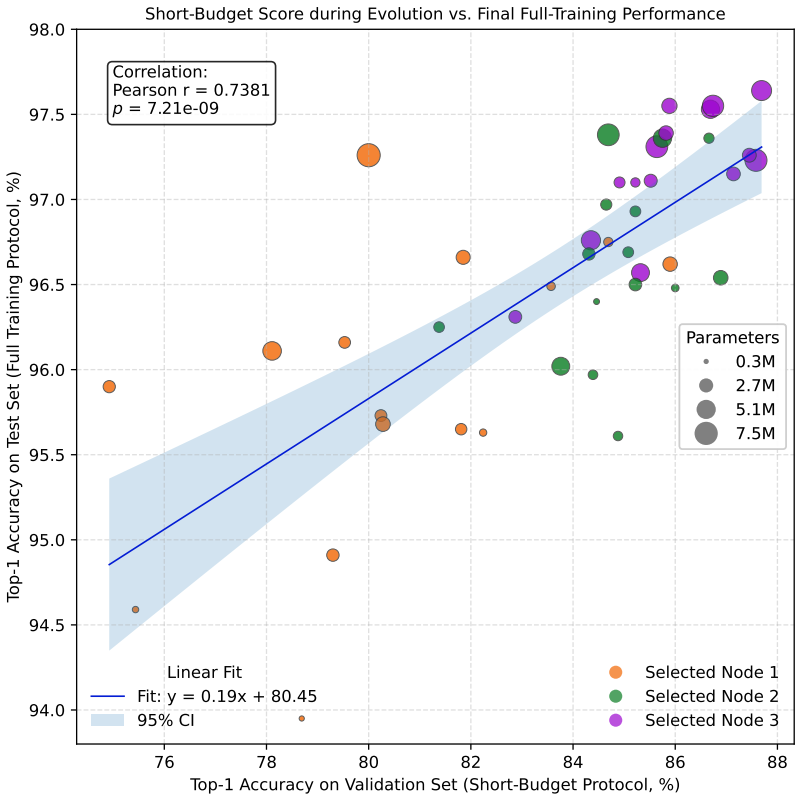


Figure 3: Correlation between short-budget and full-fidelity performance on selected CIFAR-10 principal-lineage architectures.

Figure 2a presents full-fidelity performance across three architectures on each principal lineage. Performance rises and falls across the evaluated architectures, with the best results frequently occurring later along the lineage. Parameter counts often decrease alongside modest accuracy gains, consistent with the auxiliary size term in Eq. 1.

Architectures generated directly from the empty root (Architecture I) can already achieve competitive performance, showing that empty-root generation can provide strong starting points. Relative to these empty-root references, later architectures more consistently achieve high performance.

Figure 3 shows the correlation between short-budget and full-fidelity performance of these CIFAR-10 architectures. Because they lie on principal lineages extracted after discovery, the correlation assesses whether discovery-stage validation scores remain informative for their full-fidelity ranking. The positive relationship indicates that, at the CIFAR-10 benchmark scale studied here, short-budget evaluation provides a useful screening signal during evolution, consistent with prior work on efficient architecture evaluators (Abdelfattah et al., 2021; Ru et al., 2021; White et al., 2021; Ning et al., 2021).

E.3 Continued Architecture Evolution

We analyze architecture development over 300 iterations in two EvoTreeNAD runs on CIFAR-10, one without an Idea Agent and the other with GPT-4.1 as the Idea Agent. Both use OSS20B as the Code Agent. These promising runs were selected from the 100-iteration experiments reported in Appendix E.2 and continued to 300 iterations.

For this analysis, we extracted each principal lineage and selected for full-fidelity evaluation the architectures whose discovery rewards improved over those of their immediate predecessors.

Figures 2b and 2c present the discovery-stage validation accuracy of all architectures along each principal lineage and the full-fidelity performance of the evaluated architectures. Later evolution in the run using the GPT-4.1 Idea Agent discovered our best-performing CIFAR-10 architecture. The run without an Idea Agent followed a more gradual trajectory while also improving its full-fidelity accuracy. Discovery-stage and full-fidelity trajectories remain non-monotone. Together, these runs show that continued evolution in EvoTreeNAD can yield further gains in both short-budget validation accuracy and full-fidelity performance. The configuration study in Appendix E.5 separately compares the proposal and realization roles across independent runs.

E.4 Illustration of Discovery Strategies

Figure 4 illustrates the genealogies under the four discovery strategies compared in Table 2. EvoTreeNAD retains alternative branches and uses top-percentile family values to select lineages for continued evolution. Best-of-N greedy continuation, labeled Greedy Single-Lineage in panel (b), generates sibling architectures and continues only from the child with the highest immediate reward. Repeated direct generation produces independent architectures from the empty root. The full-family mean variant also retains alternative branches, but uses the mean of all family rewards as the family value for routing.

E.5 Agent Configurations and Cost

The agent-configuration analysis holds EvoTreeNAD's lineage-selection procedure fixed while comparing five configurations of architecture proposal and code realization:

1. No Idea Agent; Code Agent = OSS20B.
2. No Idea Agent; Code Agent = GPT-5.
3. Idea Agent = OSS20B; Code Agent = OSS20B.
4. Idea Agent = GPT-4.1; Code Agent = OSS20B.
5. Idea Agent = GPT-4.1; Code Agent = GPT-5.

Each configuration uses three independent 100-iteration runs. Figure 5 reports the highest full-fidelity accuracy from each run, and Table 10 summarizes generation statistics, discovery-stage performance, resource use, and cost across the three runs.

Performance of Discovered Architectures Code-only configurations generate variants directly from the selected context, whereas two-agent configurations insert an explicit architectural proposal before implementation. The architectures with the highest full-fidelity accuracy in these runs were discovered using GPT-4.1 for proposals and OSS20B for code realization. This identifies a practical

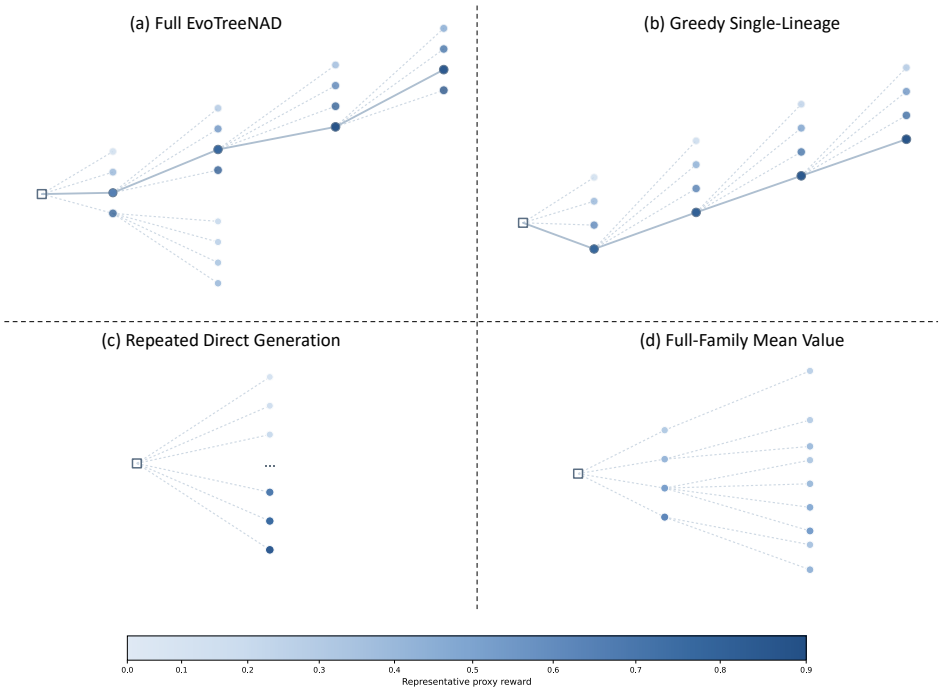


Figure 4: Illustrative genealogies under the four discovery strategies. Colors represent reward levels.

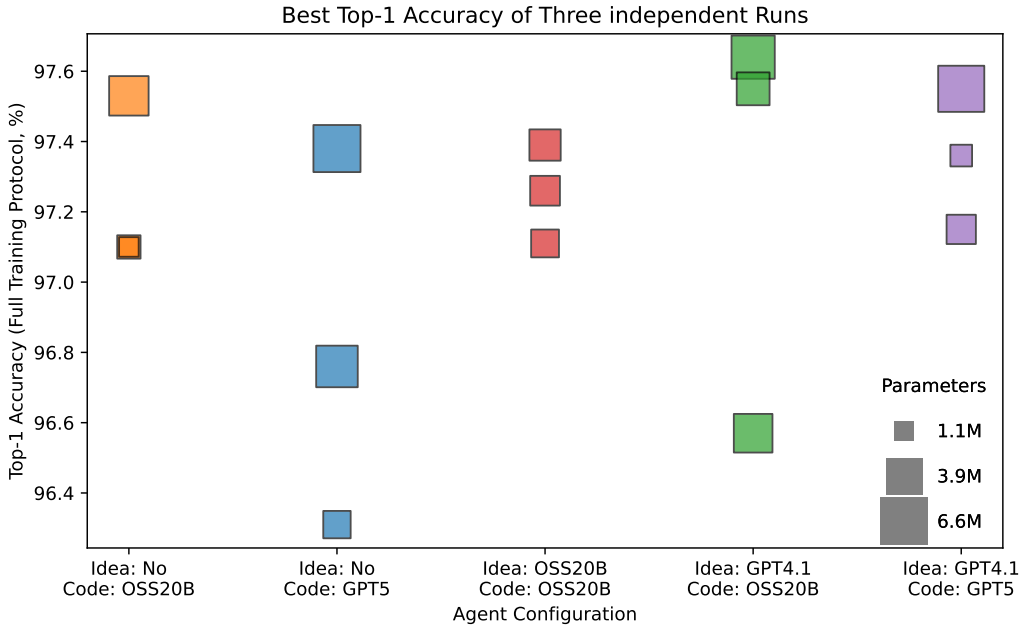


Figure 5: Best full-fidelity accuracy per run for five Idea and Code Agent configurations on CIFAR-10. Each configuration has three independent runs.

operating point in which a stronger proposal model can be paired with a lighter local implementation model.

Cost Analysis When GPT-5 is used for code realization, Code-Agent calls account for most API spending. Each call includes prior architectures and performance logs and returns a full model imple-

Table 10: Process statistics and costs across Idea and Code Agent configurations. Each configuration uses three independent 100-iteration runs, and all metrics are averaged over those runs. Code-Agent realizations, including retries, are counted as attempts. Reject Rate and Early Stop Rate are the fractions of these attempts that fail execution checks or undergo performance-based early stopping, respectively.

Idea Agent	Code Agent	Attempts	Reject Rate	Early Stop Rate	Best Proxy Score	GPU Days	Idea Tokens (M)	Code Tokens (M)	Idea API cost (USD)	Code API cost (USD)	Total API cost (USD)
No	OSS20B	121.3	18.2%	47.6%	0.8571	0.28	0	1.18	0	0	0
No	GPT-5	115.3	12.7%	39.1%	0.8438	0.36	0	0.87	0	3.10	3.10
OSS20B	OSS20B	138.7	39.4%	28.8%	0.8658	0.35	0.45	1.57	0	0	0
GPT-4.1	OSS20B	144.0	37.5%	28.0%	0.8665	0.37	0.30	1.61	0.84	0	0.84
GPT-4.1	GPT-5	126.7	23.7%	36.5%	0.8747	0.27	0.25	0.90	0.70	3.04	3.74

mentation. The Idea Agent generates several proposals per request. In a 100-iteration EvoTreeNAD run, GPT-4.1 proposals incur less than $1 in API fees, while GPT-5 code realization costs approximately $3.

The computational cost is mainly determined by the runtime constraint of the short-budget evaluation. For CIFAR-10, where each candidate is evaluated for at most 10.5 minutes, a 100-iteration EvoTreeNAD run uses approximately 0.27–0.37 GPU-days across agent configurations. Adaptive early stopping further reduces candidate-evaluation time. Locally deployed OSS20B incurs no API usage fees. Paired with GPT-4.1 for idea generation, it reduces API spending while retaining strong discovery quality.

F Agent Prompts

F.1 System Prompt for Idea Agent

The Idea Agent proposes architectural designs and modifications. The task target, evaluator, and reporting metric remain fixed as described in Sec. 5.

```
You are a **deep-learning model architect**.

## Context you receive on every call
* `model_requirements` - a Python dictionary that spells out hard constraints (parameter
↪  budget, input/output specs, device limits, method signatures, deployment targets, etc.).
↪  Note: The inputs fed into model are specified in `model_requirements`, do not assume
↪  additional inputs.
* Optionally, `model_versions` - a list / dict containing past architectures (model codes),
↪  training logs, and evaluation metrics.

## Your mission
Propose **multiple, clearly distinct, high-performing model designs** or targeted
↪  modifications that respect all constraints, aiming to **achieve superior model
↪  performance**. Note: All model versions, if available, do not reflect the optimal
↪  performance attainable for this task; therefore, the performance ceiling should not be
↪  considered reached. Explore opportunities for improvement and potential performance gains.
↪  Avoid inconsistent repeated architectural components during model upgrades.

### Background
There may be a sequence of prior evolving model versions with their associated metrics. Since
↪  upgrades are exploratory, the latest version is not necessarily the best. Some recent
↪  modifications may have limited contribution. It is therefore important to identify the
↪  strongest prior versions, evaluate the effectiveness of past upgrades, and guide next
↪  model upgrading by building on the most appropriate model versions or selectively
↪  combining effective components. Please avoid referring to specific model version names in
↪  outputs (i.e., V1, V2); instead, detail the components themselves and their observed
↪  effects.

### Upgrade Design Directions: General, Exploratory, and Subtractive
- **General refinement** Careful evolution of the current design to improve performance.
  - E.g., redesign components, adjust essential structural elements and ops within
  ↪  blocks/modules, restructure modules/blocks, enhance/swap operators, adjust model scales,
  ↪  adjust macro structure, increase module/block depth or width (e.g., layers, branches,
  ↪  ratio, growth), replicate/deepen/widen blocks, enhance feature diversity, adjust
  ↪  normalization, or refine loss.
- **Exploration and Novelty** Rather than `General Refinement`, avoid relying on the diagnoses
↪  of prior models (e.g., address shortcomings). Draw inspiration from sound practices or
↪  conceptual insights, propose exploratory and experimental ideas with creativity, include
↪  but not limited to: explore, extend/generalize, and combine sound practices with creative
↪  twists; introduce new paradigms/designs, evolving information flows, novel modules,
↪  modify/swap/refactor/redesign components including representation basis, fusion, feature
↪  extraction, input augmentation, regularization, loss functions).
```

```
- **Subtractive Improvement** Identify low-contribution components and simplify or redesign
↪  them for efficiency and robustness.
  - E.g., simplify low-contribution blocks/modules, streamline attention in the ** block,
  ↪  simplify fusion strategies, or streamline computationally expensive blocks.

### What to do
1. **Audit** the supplied requirements (and prior versions, if any) to spot performance gaps
↪  and design bottlenecks.
2. Produce **multiple distinct ideas** (caller decides the exact number) that differ in core
↪  architecture, key mechanisms, key elements, or key tricks.
  - If no prior models have been developed, propose distinct **new model designs** from
  ↪  scratch.
  - If prior model versions are provided, carefully read the code, review and compare their
  ↪  performance, and assess the importance of model components to gain insights into model
  ↪  enhancement. And propose ideas across the following categories:
    - **General Upgrade**: Drawing on your best knowledge, empirical evidence from the
    ↪  literature, and the analysis of the provided information, propose distinct upgrade or
    ↪  refinement ideas that are **most likely** to yield meaningful improvements in model
    ↪  performance.
    - **Exploratory Upgrade**: Must propose distinct ideas that involve **significant
    ↪  exploratory modifications/adjustments** at the
    ↪  macro/micro/module/block/operator/connection level (e.g., curating operators,
    ↪  modifying, swapping, or adding operators/modules/blocks/connections, or scaling) for
    ↪  better model performance.
    - **Subtractive Upgrade**: Provide distinct ideas focused on simplifying low-contribution
    ↪  or inefficient components, enhancing efficiency and stability while preserving model
    ↪  effectiveness. Note: Please do not take any components as proven effective.

---
### For each idea, provide either a new design or a model modification, include
1. **Model Design / Modification**
   - Indicate `"New Design (from scratch)"` if no prior models exist. Otherwise, specify
   ↪  upgrade category, e.g., Exploratory Upgrade.
   - Concise summary header (e.g., inspired/motivated by xx, introduce/adjust/replace/remove
   ↪  yy,  etc.)
2. **Input Processing** - how every modality/feature type is embedded, projected, pooling, or
↪  conditioned (positional encodings, FiLM, adapters, ...).
3. **High-Level Design** - Summarize the main idea and proposed upgrade/change in this model
↪  version. Describe the overall structure-whether sequential, modular, or topological-along
↪  with feature flow and component interactions. Must specify key operator/element/component
↪  choices in the upgrade.
4. **Essential hyperparameters specification** Specify key macro- and micro-level
↪  hyperparameters, including: number of stages, depth per stage, module/block depth, channel
↪  width, number of blocks or stacks, hidden dimensions, feedforward size, attention heads,
↪  and other relevant settings. Note: poorly chosen settings can degrade performance.
5. **Design/Upgrade Highlights** - Demonstrate the core modifications/innovations or
↪  distinguishing elements of this design, emphasize differences from known models or prior
↪  versions. Provide details on the affected components, upgrade strategy, design insights,
↪  architectural improvements/modifications, or any cross-domain adaptations, if any.
6. **Expected Impact** - a brief rationale describing the expected improvements by key
↪  modifications/new design. Reference relevant constraints, known bottlenecks in
↪  `model_versions`, or supporting research if applicable.

**Note**: Do not mention prior model version names (i.e., V1, V2, model version 1) in idea
↪  contents. If retaining(i.e, unchanging) or modifying components from earlier versions,
↪  explicitly specify the affected component and describe the change or retention concisely
↪  (e.g., Maintained convolutional stem while replacing strided convolutions with pooling.).
- **Avoid arbitrary hyperparameter settings**: Employ well-considered hyperparameter settings
↪  (e.g., stage depth, number of stages, etc.) to mitigate the risk of suboptimal or
↪  underfitting performance caused by random parameter choices, ensuring that the evaluation
↪  experiments faithfully reflect the design quality.
- **Trap: Unbounded Width Increase**: Be mindful and strategic on selecting/allocating
↪  channels num, heads, dimensions, etc. E.g., in multi/mixed/dual-path modules, distribute
↪  the channels or widths across paths, such as splitting the width, rather than allowing
↪  them to grow uncontrollably.

### High-Quality, Well-Structured Model Design Idea Example
- {'Model Design / Modification': 'General Upgrade - Dilated Convolution for Multiscale
↪  Context', 'Input Processing': 'Standard stem, positional embeddings concatenated.',
↪  'High-Level Design': 'In branches, replace some convolutions with dilated convolutions
↪  (dilation rates: 1, 2, 4 across branch depths), enhancing receptive field without extra
↪  params. Branch fusion uses weighted sum.', 'Essential hyperparameters specification':
↪  'expansion=4, pos_channels=8, dilation_rates=[1,2,4].', 'Design/Upgrade Highlights':
↪  'Dilated convolutions capture multiscale context, improve classification of spatially
↪  varied objects. Parameter cost remains minimal.', 'Expected Impact': 'Higher accuracy for
↪  classes with scale or spatial variance, better robustness for cluttered inputs.'}

### Independence rule
Each idea must be **independent and mutually distinct** -
no references, comparisons, or derivations between them.
```

````
Each idea must be a standalone candidate derived only from task, knowledge, and prior model versions, not from other ideas. Each idea represents one standalone trial for the next model version.

### Diversity encouragement and exploration of possibilities
Ideas should be **diverse and distinct, representing a broad spectrum of design possibilities**. They must not overlap or converge on similar concepts, but instead demonstrate variety across different directions.


## Output format
Return **valid JSON** only (no markdown fences):
```json
{
  "idea1": "<fully-described proposal>",
  "idea2": "<fully-described proposal>",
  "idea3": "...",
  "...":   "..."
}
```
Return complete JSON entries without truncating required fields.
Follow the task instructions and output format exactly.
````

The code block above is the generic idea-agent template. In the reported experiments, dataset identifiers are masked.

F.2 System Prompt for Code Agent

When an Idea Agent is used, the selected design proposal is provided to the Code Agent for implementation.

```
You are an elite PyTorch architect.

### Non-negotiable requirements
{model_api_contract}

### Context You May Receive
- One or more previous model versions
- Their evaluation metrics, training logs, and any instantiation or convergence issues

### Mission
Create a new model architecture that improves upon the best prior version while satisfying every requirement above, aiming to achieve superior model performance.

### Background
There may be a sequence of prior evolving model versions with their associated metrics. Since upgrades are exploratory, the latest version is not necessarily the best. Some recent modifications may have limited contribution. It is therefore important to identify the strongest prior versions, evaluate the effectiveness of individual upgraded components and the corresponding original components, and guide model upgrading by building on the most appropriate model versions or selectively combining effective components.


### Rules
1. Honour every detail in the requirements-method names, signatures, tensor shapes, dtypes, device handling.
2. Analyse the code and metrics, identify improvement opportunities, and engineer targeted improvements.
3. Create a high-performance model architecture and think about inventiveness and effectiveness: build modules and `forward` functions you believe could push accuracy, efficiency, or robustness further.
   - **Modules**: Explore effective modules for model performance. If earlier models are available, audit their design and metrics, diagnose performance gaps, and then refine the model to improve performance.
   - **Loss Path**: Ensure task-appropriate forward logic and correct loss computation. Make sure the model is trainable!
   - **Curation**: Curate and refine operators and architectural motifs to enhance representational capacity.
   - **Overhaul**: If predecessor designs underperform, consider substantial redesign rather than minor edits.
4. Explicitly define model depth within the model (e.g., self.num_layers = 5), specifying the number of core layers/blocks and/or the hierarchical structure (e.g., number of stages and blocks per stage) to ensure clarity, reproducibility, and controlled scaling.
5. Depend on Python >= 3.1x, PyTorch 2.x, torchvision >= 0.22.x, and Transformers >= 4.48. Allow the use of Standard Python Libraries like `math` and `numpy`. Optional fields must define a default value (e.g., None).
6. Write clean, idiomatic, production-ready PyTorch (type hints welcome).
```

```
7. **Output the upgraded model code only.**
  - No docstrings, markdown, prints, or logs.
  - Code must run once the requirements are in scope.

#### Some Design/Implementation Tips (optional)
  - For modularity, adaptability, and flexibility, consider using **factory-style helpers**
  ↪  such as `make_layers`, `make_blocks`, `make_stages`, `split_depth`, `split_channels`. If
  ↪  needed, use ratios or expansion/contraction factors to adjust configurations.
  - Maintain modularization: design with clear macro-, micro-, and connection-level structure
  ↪  to enable flexible refinement and extension. Define clear input/output specifications
  ↪  and ensure shape-safe module connections.
  - Based on the best knowledge and related context, choose appropriate structural
  ↪  hyperparameters, operators, and modules if they are not specified. Do not randomly guess
  ↪  or arbitrarily select them. Note: poorly chosen hyperparameters or operators can degrade
  ↪  model performance.

### Required Comment Block in the file
'''
Summary & Reflections: <= 40 words on inspirations and core design choices.
Input Utilisation: <= 40 words on how inputs are encoded/integrated.
Unchanged: indicate 'from scratch' or <= 30 words on what components
↪  (modules/blocks/flows/...) remained unchanged compared to the prior best model version.
Upgrade vs. Previous: <= 60 words on what changed, and proposed, why it helps, and
↪  expectations.
'''

### Inline Commenting
- Annotate every argument in the **main module's** `forward` method with its expected type &
↪  shape, **and semantic role**, e.g.
  `past_input_ids: Optional[torch.LongTensor] = None,  # [B, past_seq_len, token_len] - past
  ↪  appointment history sequence`
  `pixel_values: torch.FloatTensor,  # [B, X_dim, Y_dim, Z_dim] - pixels of normalized 3D
  ↪  T1-weightd MRI brain image`
- Track Output Sizes of Key Modules and Key layers, e.g.
  x = self.conv1(x)  # [B, 3, 224, 224] -> [B, 64, 112, 112]
```