

HypoExplore: Agentic Discovery with Active Hypothesis Exploration

Jaywon Koo¹, Jefferson Hernandez¹, Ruozhen He¹, Hanjie Chen¹,
Chen Wei¹, Vicente Ordonez¹

Rice University
{jk125, vicenteor}@rice.edu

Abstract

We introduce HypoExplore, an agentic framework that formulates model discovery as a hypothesis-driven scientific inquiry. Given a human-specified high-level research direction, HypoExplore ideates, implements, evaluates, and improves neural architectures through evolutionary branching. Hypotheses are generated using a large language model by selecting a parent hypothesis to build upon, guided by a dual strategy that balances exploiting validated principles with resolving uncertain ones. Our proposed framework maintains a *Trajectory Tree* that records the lineage of all proposed architectures, and a *Hypothesis Memory Bank* that actively tracks associated confidence scores acquired through experimental evidence. After each experiment, multiple feedback agents analyze the results from different perspectives and consolidate their findings into hypothesis confidence updates. Under minimal research directions, HypoExplore discovers a model that achieves top results on two medical imaging tasks on MedMNIST, an improved encoder for named entity recognition on FiNER-ORD, and a lightweight 0.7M parameter model on CIFAR-10 that reaches 94.1% accuracy and transfers well to other benchmarks. Our analysis of hypothesis confidence scores across evolutionary lineages suggest that HypoExplore not only discovers stronger architectures, but can help build a genuine understanding of the design space.

1 Introduction

Scientific discovery in model design is often driven not by exhaustively searching a fixed space, but by starting from a broad research direction, formulating hypotheses about useful mechanisms, and testing those hypotheses through iterative experiments. This process is especially important for specialized domains, where discovering strong models still requires substantial human effort, repeated experimentation, and careful interpretation of failures. Recent advances in multi-agent systems (Si

et al., 2026; Chen et al., 2026; Cheng et al., 2025) have made it feasible to automate or support parts of this process, including brainstorming (Huang et al., 2024), code generation (team et al., 2025; Weng et al., 2026; Zhang et al., 2025), experiment execution (Huang et al., 2024; Si et al., 2026), debugging (Epperson et al., 2025), and result analysis (Koo et al., 2024; Panagopoulou et al., 2025). These developments suggest the possibility of scientific agents that can assist with, and potentially accelerate, model design (Wen et al., 2020; Ren et al., 2021; Cheng et al., 2025).

Unlike several prior works, we start from a human-specified research direction without any fixed search space (Yang et al., 2025b; Chang et al., 2025) or predefined seed architecture (Chen et al., 2026; Turcan et al., 2025; Yuksekgonul et al., 2026). A direction such as *novel attention mechanisms* or *diffusion-inspired denoising* is not itself a model. It is a scientific prior over possible mechanisms. The central challenge is therefore not only to generate stronger candidates, but to decide which architectural hypothesis should be tested next and what evidence supports that decision. We posit that this framing better matches how a researcher works, by turning a broad direction into competing mechanistic hypotheses, selecting informative experiments, and updating beliefs from accumulated results.

Meta-research on scientific practice provides a foundation for this view. It characterizes discovery as a coupled search over a *hypothesis space* and an *experiment space*, where progress depends on managing the interaction between proposing explanations and selecting informative tests (Klahr and Dunbar, 1988). This view further emphasizes maintaining *multiple competing hypotheses* to avoid fixation and redundancy, and prioritizing tests that can eliminate alternatives rather than only accumulate confirmations (Chamberlin, 1890; Platt, 1964; Wason, 1960). Similarly, work in organizational learning and the sociology of science highlights the

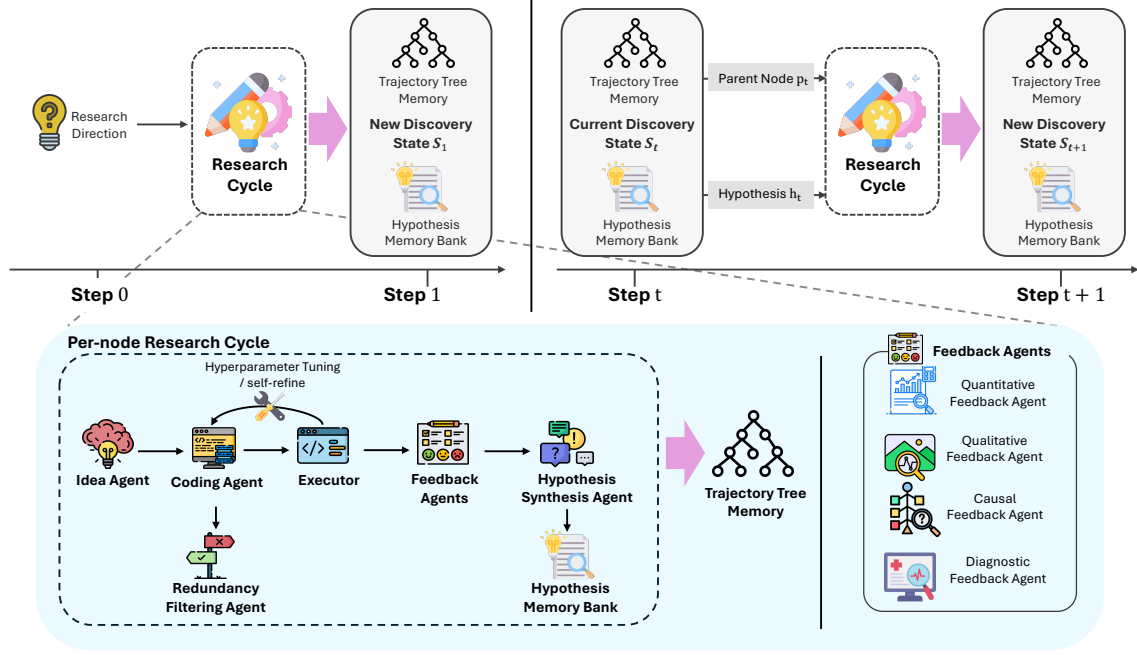


Figure 1: **Overview of HypoExplore.** **Top:** From a research direction, HypoExplore builds a discovery state with Trajectory Tree Memory and a Hypothesis Memory Bank, then iteratively selects a parent node and hypothesis for the next research cycle. **Bottom:** specialized agents instantiate, implement, evaluate, and analyze the selected hypothesis, then update memory with the resulting evidence.

exploration–exploitation tension and the benefits of division of cognitive labor, suggesting mechanisms for allocating effort across promising directions while still probing uncertain ones (March, 1991; Kitcher, 1990). These insights motivate the development of an agent grounded in generation and execution that can also prioritize evidence-driven hypothesis management.

Following this view, HypoExplore treats each candidate model variant as a test of an explicit architectural hypothesis. Starting from a human-specified research direction, HypoExplore instantiates hypotheses into runnable model code, evaluates them empirically, and writes the resulting evidence into structured memory (Figure 1). This memory has two components: A *Trajectory Tree* that records the lineage of research branches, including hypotheses, implementations, and outcomes; and a *Hypothesis Memory Bank* that tracks usage, evidence logs, and confidence estimates for each hypothesis. HypoExplore also uses specialized agents for idea generation, redundancy filtering, code implementation, experiment execution, and feedback analysis, enabling the system to reason about what has been tried, why it was tried, and which hypothesis should be tested next.

Using this memory, HypoExplore guides exploration at two levels. A parent-node selector chooses which research branch to expand by balancing em-

pirical promise with remaining unexplored potential. A hypothesis selector chooses which candidate hypothesis to evaluate by balancing supported mechanisms with uncertain ones. This separates where to continue the research from what mechanistic claim to test, making the discovery loop more deliberate and interpretable than repeated generation and execution.

Our contributions are summarized as follows:

- HypoExplore, a memory-grounded multi-agent framework for automated neural architecture design from scratch.
- A hypothesis-centered memory system, with a trajectory tree and memory bank, for non-redundant, interpretable exploration.
- A selection strategy over research branches and hypotheses that balances empirical promise with unexplored potential.

2 Related Work

Autonomous Scientific Discovery (ASD). These systems use LLMs to close the loop between ideation, implementation, execution, and reflection, but differ in what drives exploration and what the *discovered object* is. AutoDiscovery (Agarwal et al., 2025) prioritizes open-ended discovery using Bayesian surprise and MCTS over hypotheses, MARS (Chen et al., 2026) targets automated

AI research in codebases with budget-aware planning and reflective memory, and *Genesys* (Cheng et al., 2025) simulates the research lifecycle for discovering language modeling architectures, using genetic programming and staged verification. Other execution-grounded research agents and AI-scientist frameworks similarly emphasize planning, code execution, and memory across scientific and engineering tasks (Si et al., 2026; Yang et al., 2025a; Yu et al., 2025b; Mitchener et al., 2025; Yu et al., 2025a; Du et al., 2025). HypoExplore shares the goal of closing the research loop, but studies model discovery from a human-specified research direction. Its memory stores both branch histories and per-hypothesis evidence, so the agent can decide not only what artifact to keep, but what mechanism to test next.

Hypothesis Generation and Evaluation. A complementary line studies literature-grounded hypothesis generation and theory construction. ResearchAgent, BioDisco, MOOSE-Chem, and Hyper generate or refine hypotheses using literature, knowledge graphs, inspiration retrieval, or provenance-aware reasoning (Baek et al., 2025; Yang et al., 2025c; Vasu et al., 2025). Recent systems also validate, falsify, or refine free-form hypotheses using statistical evidence aggregation and uncertainty-aware updates (Jansen et al., 2026; Huang et al., 2025; Duan et al., 2025). In contrast, hypotheses in HypoExplore are not final textual outputs: each is an actionable model-design claim instantiated as runnable code, evaluated on a target task, and written back into structured memory.

Architecture Design and Agent Memory. Closest to our setting, NADER and RevoNAD use LLM agents for neural architecture design through collaborative reflection or evolutionary exploration (Yang et al., 2025b; Chang et al., 2025). HypoExplore differs by making the research direction the search prior and by selecting both branches and hypotheses from accumulated evidence. Related self-evolving agents such as Darwin Gödel Machine, Group-Evolving Agents, and AlphaEvolve improve programs or agent behavior through open-ended evolution (Zhang et al., 2025; Weng et al., 2026; Novikov et al., 2025), while memory-centric methods distill reusable reasoning traces or context for future tasks (Ouyang et al., 2025; Suzgun et al., 2025; Zhang et al., 2026). Our memory instead records evidence about discovered model lineages and the hypotheses that guide future experiments.

3 Method

HypoExplore is a hypothesis-grounded multi-agent framework for autonomous scientific discovery of neural architectures. As shown in Figure 1, HypoExplore operates in a predefined task domain. Given a human-specified research agenda, the system aims to discover effective neural architectures *from scratch*, without a seed backbone or a fixed search space. HypoExplore maintains a discovery state $\mathcal{S}_t = \{\mathcal{T}_t, \mathcal{M}_t\}$, where $\mathcal{T}_t$ records the experimental lineage and $\mathcal{M}_t$ stores hypothesis-level statistics. At iteration t , it selects a parent node p_t , selects a small *set* of hypotheses $\mathcal{Q}_t$ to test under that parent, and runs a research cycle to instantiate, execute, and analyze architectures conditioned on $(p_t, \mathcal{Q}_t)$. The resulting outcomes update both $\mathcal{T}_t$ and $\mathcal{M}_t$, yielding $\mathcal{S}_{t+1}$. Rather than mutating architectures within a predefined family, HypoExplore performs iterative hypothesis-driven discovery, using structured memory to decide *where* to explore next and *what* mechanisms to test. We describe the memory (Sec. 3.1), research cycle (Sec. 3.2), and dual selection mechanism (Sec. 3.3).

3.1 Structured Memory

The discovery state $\mathcal{S}_t = \{\mathcal{T}_t, \mathcal{M}_t\}$ is a structured memory with two complementary stores. This separation lets the system reason both about *branch trajectories* (what lines of exploration are promising) and *hypotheses* (what mechanistic claims are supported or contradicted across experiments).

Trajectory tree. The trajectory tree $\mathcal{T}_t$ records the branching structure of discovery. Each node $v \in \mathcal{T}_t$ corresponds to one executed research step and stores $v = \{h_v, a_v, r_v, p_v\}$, where h_v is the architectural hypothesis (or hypothesis set reference) used to guide the design, a_v is the instantiated architecture, r_v is the experimental outcome, and p_v is the parent node. By preserving parent-child relations, $\mathcal{T}_t$ exposes full exploration trajectories, enabling the system to identify promising, saturated, or repeatedly failing branches.

Hypothesis memory bank. Memory bank $\mathcal{M}_t$ aggregates statistics across related hypotheses. For each hypothesis h , the bank maintains $m_t(h) = \{N_t(h), c_t(h), \ell_t(h)\}$, where $N_t(h)$ is the number of times h has been tested, $c_t(h) \in [0, 1]$ is its current confidence score, and $\ell_t(h)$ stores logs such as supporting evidence, contradictions, failure modes, and implementation notes.

3.2 Per-Node Research Cycle

Each iteration executes a pipeline of specialized agents (Fig.1, bottom). Given a parent node p_t and a selected hypothesis set $\mathcal{Q}_t$, the system attempts to create up to $|\mathcal{Q}_t|$ child nodes by running the cycle once per selected hypothesis. If multiple hypotheses lead to duplicate proposals, redundancy filtering rejects and regenerates until novelty is satisfied or the retry budget is exhausted.

Idea Agent. The Idea Agent receives a research direction, parent context (architecture specification, performance, and multi-agent feedback), and the hypothesis memory $\mathcal{M}_t$. In *root node* (generation 0), it generates architectures from scratch guided by the research direction. In *evolution mode*, it conditions on a selected hypothesis $h \in \mathcal{Q}_t$ and parent node p_t to produce: an architecture specification, a reasoning trace, and up to K_{hyp} newly proposed hypotheses motivated by the design.

Coding Agent. This agent receives the architecture specification along with implementation notes from $\ell_t(h)$ (failure modes and recommended practices from prior experiments). It generates `model.py` and `config.py`. An error recovery loop reruns the agent with the error trace for up to R_{max} attempts. After successful compilation, a hyperparameter refinement loop adjusts `config.py` for up to F_{max} steps, with early stopping on accuracy plateau.

Redundancy filtering. Before execution, an LLM judge compares the proposed architecture against the top- k most similar archived concepts in $\mathcal{T}_t$. If the architecture is judged to be a duplicate, the node is rejected and the pipeline backtracks to regenerate another architecture. This enforces novelty as a hard constraint on node creation.

Executor. The Executor trains each proposed architecture a under a wall-clock timeout τ_{max} . A sanity-check phase (5 epochs) detects catastrophic failures early. The recorded outcome is $r = \{J(a), d\}$, where $J(a)$ is a validation task performance and d stores diagnostics (e.g., instability, timeout). Each successful execution appends a new node to the trajectory tree, storing (h, a, r) and its parent pointer.

Multi-perspective feedback. On success, four parallel agents analyze the outcome from complementary perspectives: (i) *Quantitative*: analyzes accuracy, loss curves, convergence speed, and computational efficiency, extracting hypothesis evidence from performance patterns. (ii) *Qualitative*: a VLM examines misclassified images and

attention maps. (iii) *Causal*: compares the parent and child architectures, attributing observed performance changes to specific structural modifications. (iv) *Diagnostic* (failure/timeout only): performs root-cause analysis and records implementation failure modes into $\ell_t(h)$.

Hypothesis synthesis. A hypothesis synthesis agent consolidates feedback in one LLM call, deduplicating overlapping updates, resolving disagreements in evidence interpretation, and capping new hypotheses at K_{synth} per node. Each hypothesis must pass a quality gate assessing mechanistic specificity, falsifiability, novelty w.r.t. $\mathcal{M}_t$, and actionability before admission to the memory bank.

Memory update. Confidence scores of all referenced hypotheses are updated using evidence type and strength $w \in [0, 1]$ produced by hypothesis synthesis:

$$c_{t+1}(h) = \begin{cases} c_t(h) + \eta w (1 - c_t(h)), & \text{if } e_t = 1, \\ c_t(h) - \eta w c_t(h), & \text{if } e_t = 0, \end{cases} \quad (1)$$

where $e_t = 1$ if the current evidence supports h , or $e_t = 0$ if the current evidence is in contradiction with h , and $\eta \in (0, 1]$ is a learning rate. The factors $(1 - c_t(h))$ and $c_t(h)$ keep confidence bounded: supporting evidence pushes confidence toward 1 with diminishing increments, while contradicting evidence pushes it toward 0. Hypotheses are initialized at $c_0(h) = 0.5$, representing maximum uncertainty. Finally, the hypothesis logs $\ell_t(h)$ and counts $N_t(h)$ are updated with newly synthesized evidence, failure modes, and references.

3.3 Dual Selection for Guided Discovery

To avoid undirected trial-and-error, a two-stage selection strategy separates *which branch* to expand from *which hypotheses* to test within that branch. Parent selection is deterministic at the branch level for stability, while hypothesis selection concentrates the exploration–exploitation trade-off.

3.3.1 Parent-Node Selection.

Let $\mathcal{V}_t^{\text{exp}}$ be the set of expandable nodes in the trajectory tree at iteration t . For each node $n \in \mathcal{V}_t^{\text{exp}}$, we compute a branch quality score by combining task performance and execution efficiency:

$$\text{quality}(n) = \lambda_{\text{acc}} \text{Acc}(n) + (1 - \lambda_{\text{acc}}) \left(1 - \frac{\min(\tau(n), \tau_{\text{max}})}{\tau_{\text{max}}} \right), \quad (2)$$

Method	Dermal MNIST	Tissue MNIST	Breast MNIST
ResNet-18 (Manzari et al., 2025)	75.4	68.1	83.3
ResNet-50 (Manzari et al., 2025)	73.1	68.3	82.8
ViT (Chowdary and Yin, 2024)	73.9	–	–
Swin (Chowdary and Yin, 2024)	75.3	–	–
MedViTV1-L (Manzari et al., 2023)	77.3	68.3	88.5
MedMamba-B (Yue and Li, 2024)	75.7	–	89.1
MedFormer (Chowdary and Yin, 2024)	78.3	–	–
NQNN (Rahman and Zhuang, 2025)	80.4	–	–
Med-LEGO (Zhu et al., 2025)	73.9	–	–
PRADA (Jang et al., 2025)	81.3	–	–
MedViTV2-L (Manzari et al., 2025)	<u>81.7</u>	<u>71.6</u>	91.0
MedNNS (Mecharbat et al., 2025)	79.7	69.2	92.3
NADER (Yang et al., 2025b)	80.6	68.5	85.9
Ours	82.1	73.9	<u>91.7</u>

Table 1: **Domain-specific architecture discovery on MedMNIST classification.** We report top-1 accuracy (%) on DermalMNIST, TissueMNIST, and BreastMNIST. “–” indicates the result is not reported. Best in **bold** and second-best underlined.

where $\text{Acc}(n) \in [0, 1]$ is normalized validation accuracy at node n , $\tau(n)$ is the training time, $\tau_{\max}$ is the maximum allowed training time, and $\lambda_{\text{acc}} \in [0, 1]$ controls the trade-off. On top of quality, we measure whether the node still contains useful unexplored directions. Let $\mathcal{H}_{\text{active}}(n)$ denote the set of active hypotheses associated with node n (excluding confirmed and refuted hypotheses), and let $\mathcal{H}_{\text{tested}}(n) \subseteq \mathcal{H}_{\text{active}}(n)$ denote the subset already tested. We define an availability score as:

$$\text{avail}(n) = 1 - \frac{|\mathcal{H}_{\text{tested}}(n)|}{|\mathcal{H}_{\text{active}}(n)|}, \quad (3)$$

with the convention $\text{avail}(n) = 0$ if $|\mathcal{H}_{\text{active}}(n)| = 0$. The final parent score is a weighted combination:

$$\begin{aligned} \text{score}(n) &= \lambda_{\text{parent}} \text{quality}(n) \\ &\quad + (1 - \lambda_{\text{parent}}) \text{avail}(n), \end{aligned} \quad (4)$$

where $\lambda_{\text{parent}} \in [0, 1]$ trades-off branch quality and remaining search potential. We then select as parent the expandable node with the highest score.

3.3.2 Hypothesis Selection.

For node p_t , let $\mathcal{H}_{\text{cand}}(p_t)$ denote the hypotheses not yet tested on its ancestors. We select an exploitation subset and an exploration subset from $\mathcal{H}_{\text{cand}}(p_t)$; their union forms set $\mathcal{Q}_t$ passed to the research cycle.

Setting	Method	Entity F1
<i>Published references</i>	GPT-4o-mini, 5-shot	0.774
	GPT-4o, 5-shot	0.820
	RoBERTa full fine-tune	0.879
<i>Frozen RoBERTa setting</i>	GPT-5-mini, 5-shot	0.755
	Linear head	0.765
	CRF head	0.648
	MLP head	0.780
	BiLSTM head	0.799
<i>LLM-guided frozen-head discovery</i>	GPT-5-mini iterative, n steps	0.788
	GPT-5-mini best-of- n	0.799
	HypoExplore, n iterations	0.836

Table 2: **FiNER-ORD frozen-encoder head discovery.** Except for the published RoBERTa full fine-tuning reference, all trained task-head methods freeze RoBERTa-base. Here $n = 20$. GPT-5-mini best-of- n samples n one-shot heads and selects the best. GPT-5-mini iterative runs sequential n -step proposal loop with execution feedback but without memory or dual selection.

Exploitation via Thompson sampling. To avoid committing to noisy early winners, we use Thompson sampling over weighted supporting and contradicting evidence (Chapelle and Li, 2011; Daniel et al., 2018). For each $h \in \mathcal{H}_{\text{cand}}(p_t)$, we define:

$$\alpha_h = \alpha_0 + \sum_{e \in \mathcal{E}_{\text{sup}}(h)} w_e, \quad \beta_h = \beta_0 + \sum_{e \in \mathcal{E}_{\text{con}}(h)} w_e, \quad (5)$$

with prior pseudo-counts $\alpha_0, \beta_0 > 0$, and sample $\theta_h \sim \text{Beta}(\alpha_h, \beta_h)$. Let π order $\mathcal{H}_{\text{cand}}(p_t)$ such that $\theta_{\pi(1)} \geq \theta_{\pi(2)} \geq \dots$. The exploitation subset is $\mathcal{H}_{\text{exploit}}^*(p_t) = \{\pi(1), \dots, \pi(K_{\text{hypo}})\}$.

Exploration via epistemic value. In parallel, we prioritize hypotheses whose current evidence is ambiguous. Using the confidence $c_t(h) \in [0, 1]$ stored in $\mathcal{M}_t$, we define

$$\text{epistemic}(h) = 1 - |2c_t(h) - 1|, \quad (6)$$

which is maximal at $c_t(h) = 0.5$ and decreases toward 0 or 1. We then define $\mathcal{H}_{\text{explore}}^*(p_t)$ as the K_{hypo} hypotheses in $\mathcal{H}_{\text{cand}}(p_t)$ with the largest $\text{epistemic}(h)$, analogous to uncertainty-based acquisition in active learning (Settles, 2009; Houlsby et al., 2011).

Final hypothesis set. We use deduplicated set:

$$\mathcal{Q}_t = \mathcal{H}_{\text{exploit}}^*(p_t) \cup \mathcal{H}_{\text{explore}}^*(p_t),$$

so that $|\mathcal{Q}_t| \leq 2K_{\text{hypo}}$. Together with parent selection, this yields branch-level continuation plus hypothesis-level exploitation and uncertainty-driven exploration.

Research Direction	Best discov. model	Acc.	Params	FLOPs
Generic efficient architectures	H-Edge Routed Hybrid Mixer	93.31	422K	35M
Pure basic tensor operations	DirBank MLP	91.42	749K	24M
Novel attention mechanisms	GSTN	94.11	668K	207M
Diffusion-based denoising	StabilizerNet	95.68	1.52M	210M

Table 3: **Direction-controlled CIFAR-10 architecture discovery.** All directions used the same search budget. The resulting models differ in accuracy, parameter count, and FLOPs, showing that research direction acts as a prior over the discovered architecture.

4 Experiments

4.1 Experimental Setup

The experiments answer three questions: whether HypoExplore discovers useful domain-specific models across medical imaging and financial NLP, how research directions influence the discovered architecture family, and whether discovered architectures generalize beyond their search dataset. We then ablate the discovery loop to identify which components drive continued improvement.

HypoExplore starts from a natural-language research direction and task-specific training recipe. All agents use GPT-5-mini (OpenAI, 2025). Unless otherwise stated, each run uses five root proposals, expands one parent node per iteration, and gives each candidate a $t_{\max} = 30$ minute wall-clock timeout. Parent ranking uses $\lambda_{\text{acc}} = 0.85$ in branch quality and $\lambda_{\text{parent}} = 0.60$ in the final parent score. Hypothesis exploitation uses a uniform Beta prior $(\alpha_0, \beta_0) = (1, 1)$ with uniform evidence weights $w_e = 1$. Memory updates use confidence rate $\eta_c = 0.20$. We set $K_{\text{hypo}} = 2$, so the dual selector can return up to four hypotheses after combining exploitation and exploration. Vision experiments discover architectures from scratch. FiNER-ORD (Lu and Huo, 2025) freezes RoBERTa-base and trains only the discovered task head. Baseline details, implementations, prompts, training recipes, and evaluation protocols are in Appendix.

4.2 Domain-Specific Discovery

Table 1 evaluates HypoExplore on DermalMNIST, a task where architecture iteration is costly and expert priors matter. We run for 30 iterations, then reuse the discovered architecture on TissueMNIST and BreastMNIST. This differs

Model	M	CIFAR-10	CIFAR-100	Tiny-IN
ResNet-18	11.7	95.4 / 99.8	78.5 / 94.1	69.3 / 84.8
MobileNetV3	2.5	95.5 / 99.9	73.0 / 92.5	58.5 / 83.4
ShuffleNetV2	1.4	90.1 / 99.4	67.5 / 85.2	50.9 / 73.4
SqueezeNet	1.2	91.1 / 99.7	67.3 / 83.8	54.7 / 77.1
GSTN (ours)	0.7	94.1 / 99.6	72.6 / 91.7	58.1 / 81.7

Table 4: **Cross-dataset generalization of the discovered GSTN architecture.** Each dataset column reports Acc@1 / Acc@5. The **M** column shows number of parameters in millions. GSTN was discovered on CIFAR-10 and evaluated on CIFAR-100 and Tiny-ImageNet. Manually engineered models of similar and larger capacity are included for reference.

from Neural Architecture Search (NAS) methods such as MedNNS (Mecharbat et al., 2025) and NADER (Yang et al., 2025b), which report task-specific search runs. Even under this transfer setting, we achieve the strongest performance on DermalMNIST (82.1%) and TissueMNIST (73.9%), improving over the best reported references by +0.4 and +2.3 points, while remaining competitive on BreastMNIST (91.7%). Direct GPT-5-mini prompting reaches 72.5% on DermalMNIST with one generated architecture and 79.0% with best-of-five sampling under the same conditions, suggesting that HypoExplore’s value goes beyond one-shot or repeated architecture generation.

Table 2 tests whether the same agentic loop transfers to an NLP module on top of a pretrained encoder. Although not a full-fine-tuning SOTA method, HypoExplore improves over direct, best-of-20, and iterative GPT-5-mini generation while reaching 95% of the full-fine-tuned RoBERTa reference. The discovered SAR-AELRR head builds numeric-masked and lexical-masked views, routes spans to numeric or lexicon anchors, and adds low-rank residuals over prototype logits. This matches financial NER, where entity type often depends on document-level numeric and lexical cues rather than local token identity alone.

4.3 Direction-Controlled Discovery

Table 3 uses CIFAR-10 to test whether changing only the starting direction changes the architecture HypoExplore discovers. The four outputs are qualitatively distinct. StabilizerNet uses reliability maps, gated denoising residuals, and stagewise consistency tokens. GSTN keeps most token interactions local and routes a few global shape tokens through learned prototypes. Hyperedge-Routed Hybrid

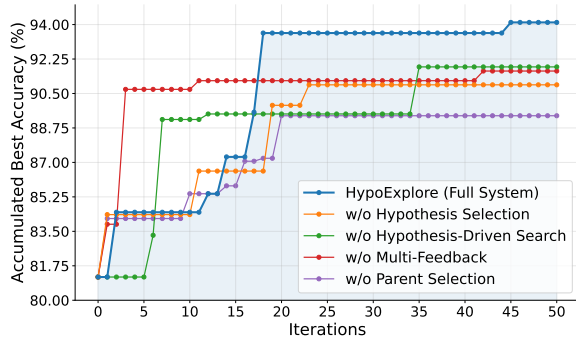


Figure 2: **CIFAR-10 discovery ablation over 50 iterations.** All methods start from the same five root architectures (iteration 0, best 81.2%). Each ablation removes one component, showing that the full hypothesis-guided loop is needed for continued improvement.

Mixer splits content from texture and sends content through sparse hyperedge tokens. DirBank-MLP replaces attention and post-patch convolutions with directional banks and median/MAD reweighting. Thus the direction changes both mechanisms and accuracy-efficiency tradeoffs, not only names. This shows that the human-provided research direction meaningfully guides what HypoExplore discovers, highlighting that human guidance remains an important part of the process. Detailed architectural descriptions and implementation details for these discovered models are provided in the Appendix.

Finally, we compare our best discovered model with LLM-based Neural Architecture Discovery (NAD) methods on CIFAR-10. NADER (Yang et al., 2025b) and RevoNAD (Chang et al., 2025) start LLM-guided architecture discovery from seed designs such as ResNet-style blocks and NAS benchmark settings. They report 94.62% and 95.22% accuracy, respectively. In contrast, HypoExplore starts from zero without a predefined search space. Despite this more open ended setup, StabilizerNet reaches 95.68% accuracy.

4.4 Cross-Dataset Generalization

Tab. 4 shows that the architecture discovered on CIFAR-10 transfers well to harder datasets. With only **0.7M** parameters, **GSTN** achieves **72.6/91.7** on CIFAR-100 and **58.1/81.7** on Tiny-ImageNet, matching MobileNetV3 within **0.4** Top-1 on both while using $\sim 2.8\times$ fewer parameters. It also outperforms ShuffleNetV2 and SqueezeNet by 5–7 Top-1 points, suggesting that the discovered mechanisms transfer beyond CIFAR-10. Although ResNet-18 remains stronger in absolute accuracy, it requires roughly $13\times$ more parameters, leaving GSTN on a favorable accuracy–efficiency frontier.

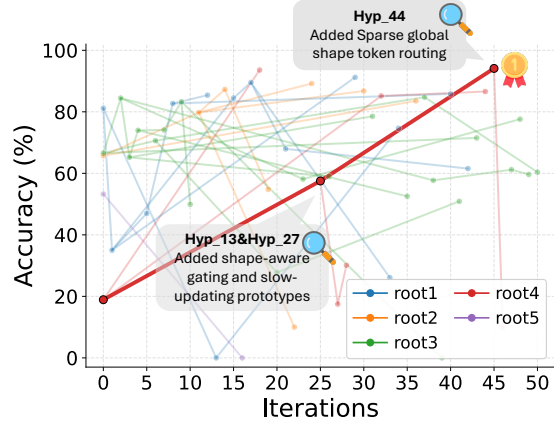


Figure 3: **Hypothesis-guided evolutionary branching.** HypoExplore explores multiple architecture lineages, with some improving gradually and one recovering from a weak root to discover the best-performing design.

4.5 Ablation of the Discovery Loop

Fig. 2 shows the accumulated best accuracy over a 50-iteration CIFAR-10 search for the novel-attention direction. All runs start from the same five root architectures. The full system reaches 94.11%, while one-shot GPT-5-mini architecture generation reaches only 79.53%, indicating that improvement comes from iterative discovery rather than the model prior alone.

The component ablation in Fig. 2 shows that later progress depends on the full loop. Removing hypothesis-driven search makes the system choose parents using only accuracy and time, which gives early gains but quickly saturates. Removing multi-agent feedback weakens the diagnostic signal about why changes help or fail. Removing hypothesis selection lets memory accumulate evidence, but prevents it from focusing the next experiment. Without parent selection (replaced with greedy, accuracy-based selection), leads to a similar plateau. These variants show that memory must guide both decisions: which branch to expand and which hypothesis to test. Comparisons with alternative parent selection strategies are in the Appendix.

To test whether the selection findings on CIFAR-10 extend beyond vision, we separately ablate the two selection decisions in the FiNER-ORD setting. Under the same setup, removing hypothesis selection while retaining the memory bank and memory-guided parent selection yields 0.797 entity F1. Replacing memory-guided parent selection with greedy accuracy selection while retaining the memory bank and hypothesis selection yields 0.798. Both variants perform below full HypoExplore at 0.836, while the iterative baseline, which

Memory design	Best Acc.
Execution-log memory	91.27
Reflection memory	91.63
Experience replay memory	92.36
Raw trajectory, token-matched	92.86
HypoExplore	94.11

Table 5: Comparison of memory designs on CIFAR-10. All systems use the same roots, search budget, executor, feedback agents, training protocol, and base LLM. The raw-trajectory baseline is matched to the average HypoExplore evolution-prompt length.

uses neither memory nor dual selection, reaches 0.788. This consistent pattern supports the contribution of both hypothesis selection and memory-guided parent selection across vision and NLP.

4.6 Alternative Memory Designs

We compare HypoExplore with four alternative ways of retaining past search experience under the same initial roots in Tab. 5. HypoExplore achieves 94.11% accuracy, outperforming token-matched raw trajectory memory (92.86%), experience replay memory (92.36%), reflection memory (91.63%), and execution-log memory (91.27%). These results support our design choice to encode search experience as structured, confidence-tracked hypotheses. In particular, the token-matched raw-trajectory control shows that improvement is not explained by additional context alone. In our setting, the Hypothesis Memory Bank provides a more effective basis for guiding architecture discovery than alternative ways of storing past search experience.

5 Analysis

5.1 Search Trajectory

HypoExplore keeps a trajectory tree over the CIFAR-10 discovery run rather than collapsing the search to the current best architecture. Fig. 3 shows why this matters. The five root branches do not follow a single smooth climb. Some improve steadily, some decline before recovering, and one branch eventually separates from the rest after reusing key hypotheses such as Hyp_44. This structure lets the system preserve partially successful directions long enough for later hypotheses to make them useful.

5.2 Hypothesis Memory

HypoExplore accumulated 117 hypotheses in the CIFAR-10 discovery run. Of these, 19 reached confirmed status (confidence > 0.7), 95 remained

Task	Hyp.	With	Removed	Δ
DermalMNIST	hyp_40	82.10	79.15	-2.95
FiNER-ORD	hyp_5	83.60	82.40	-1.20
CIFAR-10	hyp_44	94.10	93.10	-1.00

Table 6: Functional validation of confirmed hypotheses. We remove only the architectural component associated with each hypothesis. DermalMNIST and CIFAR-10 report accuracy and FiNER-ORD reports entity F1. Values are percentages or percentage points.

uncertain, and 3 were actively refuted. Hyp_44 suggests that a few learnable global tokens can collect spatial summaries and broadcast shape-aware updates, which helps reduce texture dominance at low cost. Another high-confidence hypothesis, Hyp_17, suggests separating shape-oriented and texture-oriented channels into dedicated representation banks so the network can preserve global object outlines while retaining local detail. These examples show that the memory stores reusable design rationales, not only winning architectures.

Hypothesis Prediction Accuracy. To test whether confidence scores carry signal, we compare each hypothesis’s predicted effect with the child outcome at the time it is selected. A prediction is correct when a positive-effect hypothesis improves over its parent, or a negative-effect hypothesis degrades. We then bin hypothesis-experiment pairs by confidence scores before testing. As shown in Fig. 4 (left), prediction accuracy rises with confidence: 58% for $[0.25, 0.5)$ ($N=24$), 65% for $[0.5, 0.75)$ ($N=109$), and 80% for $[0.75, 1.0]$ ($N=30$). All nonempty bins exceed the 50% baseline, suggesting that confidence provides a useful signal for selecting hypotheses to revisit.

Frozen-memory audit. The hypothesis prediction analysis above uses hypothesis applications chosen during the original search, where confidence updates and hypothesis selection operate jointly. The observed relationship may therefore partly reflect selection effects. To examine whether confidence is informative under randomized application selection, trajectory tree, and memory at four historical parent states, disable Thompson sampling and epistemic ranking, and randomly audit eight eligible hypothesis applications using matched parent and child seeds. The audit outcomes are excluded from memory updates and do not influence subsequent search decisions. Mean paired accuracy changes are -12.5 , $+5.9$, and $+10.9$ percentage points in the $[0.25, 0.50)$, $[0.50, 0.75)$, and

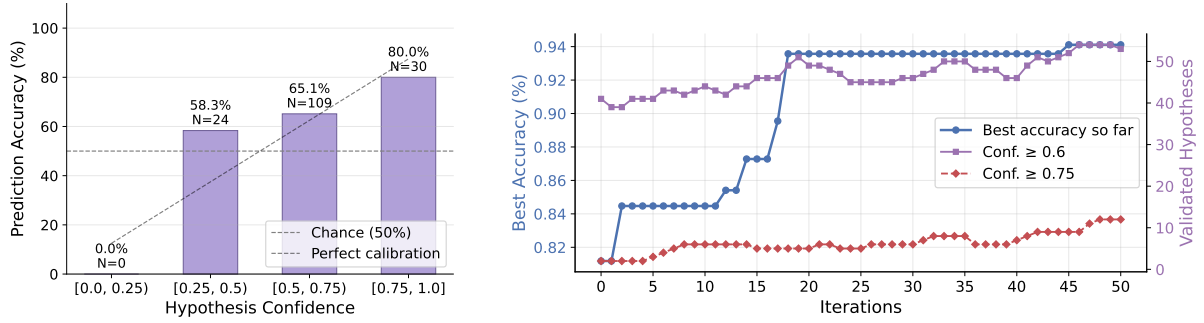


Figure 4: **Hypothesis memory analysis on CIFAR-10.** Left: Hypothesis prediction accuracy. Higher-confidence hypotheses predict experimental outcomes more accurately. Right: Knowledge accumulation curve. Best accuracy achieved and validated hypothesis count grows together.

[0.75, 1.00] confidence bands, respectively. The two higher-confidence bands show positive mean changes, whereas the lower observed band shows a negative change. This pattern suggests that confidence carries predictive information at frozen decision points even when applications are sampled rather than selected by the system. Because the audit includes only eight applications, we view the result as directional evidence rather than a statistically conclusive finding.

Knowledge Accumulation. We ask whether validated hypotheses accumulate alongside improvements in the best architecture discovered. Fig. 4 (right) plots best accuracy and validated hypothesis counts at confidence thresholds 0.6 and 0.75. The curves move together at the main transition points. The largest accuracy jump, from roughly 85% to 93.5% between nodes 15 and 25, coincides with rapid growth in validated hypotheses at the 0.6 threshold. After node 30, both curves largely plateau before a smaller late increase around node 45. This pattern is consistent with an explore-then-exploit dynamic in which early experiments build reusable evidence and later experiments consolidate it into stronger architectures.

Cross-Lineage Knowledge Transfer. HypoExplore maintains five independent root lineages and we classify each hypothesis application as within-lineage when the hypothesis and target node share the same root, and cross-lineage otherwise. An application is successful if the child improves over its parent. Cross-lineage applications succeed 65% of the time ($N=171$), comparable to within-lineage applications at 57% ($N=93$), while occurring nearly twice as often. This suggests that the selector actively reuses hypotheses across branches and that the memory captures transferable design principles rather than lineage-specific patches.

5.3 Functional Validation of Hypotheses

The preceding analyses examine whether hypothesis confidence and reuse carry a predictive signal. We test whether confirmed hypotheses identify functionally important architectural mechanisms. For one confirmed hypothesis from each task, we remove only the corresponding architectural component while holding all remaining architectural choices and task-specific training and evaluation protocol fixed. Tab. 6 shows that removing a hypothesis-specific component reduces performance on three datasets. These interventions show that, for the tested hypotheses, the memory captures mechanisms that materially contribute to the discovered models rather than merely storing plausible post-hoc rationales.

6 Conclusion

We introduced HypoExplore, a multi-agent framework for hypothesis-driven model discovery from a research direction. HypoExplore maintains a trajectory tree and hypothesis memory to decide where to search and what to test using accumulated evidence. Our findings suggest that scientific agents should not only generate candidate solutions, but also maintain testable beliefs that guide future experiments. HypoExplore’s underlying methodology could be further applied to other aspects of model development such as the discovery of better data augmentation strategies e.g. [Cascante-Bonilla et al. \(2021\)](#) or better optimizers e.g. [Chen et al. \(2023\)](#).

Acknowledgments. This project was partially supported by NSF Award No 2201710, and NSF Award No 2601419, the Ken Kennedy Institute at Rice University, NSF Campus Cyberinfrastructure Award OAC-2019007, and by Rice University’s Center for Research Computing (CRC)

Limitations

Our vision experiments discover architectures from scratch without a seed backbone or fixed search space, while the FiNER-ORD experiment studies a complementary frozen-encoder setting in which the system discovers the task head. Outcomes from using our proposed framework can still depend on the research direction, task recipe, search budget, timeout, evaluation protocol, and base LLM used by the agents. Additionally, while the discovered architectures presented in our work transfer across several benchmarks and the ablations support the value of memory-guided selection, broader validation on larger-scale tasks, additional modalities, and different compute regimes would further clarify when hypothesis-driven discovery provides the strongest gains.

Ethical considerations

A potential risk of automated discovery systems is that they could be misused as autonomous paper-generation tools, producing large numbers of plausible but weakly validated ideas, implementations, or narratives that burden reviewers and obscure scientific accountability. We do not advocate such use. HypoExplore is designed as a human-guided research assistant for execution-grounded model discovery, not as a replacement for human scientific judgment or peer review.

In our intended workflow, human researchers provide the research direction, define the task and evaluation protocol, inspect generated code and experimental results, assess novelty against prior work, and decide whether any discovered architecture or hypothesis warrants scientific communication. The agent’s role is to make hypothesis exploration more systematic, auditable, and evidence-grounded. Practical safeguards include disclosure of agent use, reproducibility checks, human verification of generated hypotheses and code, and treating discovered artifacts as candidates requiring external validation rather than final scientific conclusions.

References

Dhruv Agarwal, Bodhisattwa Prasad Majumder, Reece Adamson, Megha Chakravorty, Satvika Reddy Gavireddy, Aditya Parashar, Harshit Surana, Bhavana Dalvi Mishra, Andrew McCallum, Ashish Sabharwal, and 1 others. 2025. Autodiscovery: Open-

ended scientific discovery via bayesian surprise. *arXiv preprint arXiv:2507.00310*.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziem, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. 2026. *Gepa: Reflective prompt evolution can outperform reinforcement learning*. Preprint, arXiv:2507.19457.

Jinheon Baek, Sujay Kumar Jauhar, Silviu Cucerzan, and Sung Ju Hwang. 2025. *ResearchAgent: Iterative Research Idea Generation over Scientific Literature with Large Language Models*. Preprint, arXiv:2404.07738.

Paola Cascante-Bonilla, Arshdeep Sekhon, Yanjun Qi, and Vicente Ordonez. 2021. Evolving image compositions for feature representation learning. In *Proceedings of the 32nd British Machine Vision Conference (BMVC)*. BMVA Press.

Thomas C Chamberlin. 1890. The method of multiple working hypotheses. *Science*, (366):92–96.

Gyusam Chang, Jeongyoon Yoon, Shin han yi, JaeHyeok Lee, Sujin Jang, and Sangpil Kim. 2025. *RevoNAD: Reflective Evolutionary Exploration for Neural Architecture Design*. Preprint, arXiv:2512.05403.

Olivier Chapelle and Lihong Li. 2011. An empirical evaluation of thompson sampling. *Advances in neural information processing systems*, 24.

Jiefeng Chen, Bhavana Dalvi Mishra, Jaehyun Nam, Rui Meng, Tomas Pfister, and Jinsung Yoon. 2026. MARS: Modular Agent with Reflective Search for Automated AI Research. *arXiv preprint arXiv:2602.02660*.

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V Le. 2023. Symbolic discovery of optimization algorithms. In *Advances in Neural Information Processing Systems*, volume 36, pages 49205–49233. Curran Associates, Inc.

Junyan Cheng, Peter Clark, and Kyle Richardson. 2025. Language modeling by language models. *arXiv preprint arXiv:2506.20249*.

G Jignesh Chowdary and Zhaozheng Yin. 2024. Medformer: A transformer based architecture for medical image classification. In *International conference on medical image computing and computer-assisted intervention*, pages 448–457. Springer.

J Russo Daniel, Van Roy Benjamin, Kazerouni Abbas, Osband Ian, and Wen Zheng. 2018. A tutorial on thompson sampling. *Foundations and Trends® in Machine Learning*, 11(1):1–99.

- Shangheng Du, Xiangchao Yan, Dengyang Jiang, Jiakang Yuan, Yusong Hu, Xin Li, Liang He, Bo Zhang, and Lei Bai. 2025. Automlgen: Navigating fine-grained optimization for coding agents. *arXiv preprint arXiv:2510.08511*.
- Shiyang Duan, Yuan Tian, Qi Bing, and Xiaowei Shao. 2025. [Bayes-Entropy Collaborative Driven Agents for Research Hypotheses Generation and Optimization](#). *Preprint*, arXiv:2508.01746.
- Will Epperson, Gagan Bansal, Victor C Dibia, Adam Fourney, Jack Gerrits, Erkang Zhu, and Saleema Amershi. 2025. Interactive debugging and steering of multi-agent ai systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–15.
- Neil Houlsby, Ferenc Huszár, Zoubin Ghahramani, and Máté Lengyel. 2011. Bayesian active learning for classification and preference learning. *arXiv preprint arXiv:1112.5745*.
- Kexin Huang, Ying Jin, Ryan Li, Michael Y. Li, Emmanuel Candès, and Jure Leskovec. 2025. [Automated Hypothesis Validation with Agentic Sequential Falsifications](#). *Preprint*, arXiv:2502.09858.
- Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2024. MLAGentBench: evaluating language agents on machine learning experimentation. In *Proceedings of the 41st International Conference on Machine Learning*, ICML’24. JMLR.org.
- Jinhyeok Jang, Hong Joo Lee, Nassir Navab, and Seong Tae Kim. 2025. Prada: Protecting and detecting dataset abuse for open-source medical dataset. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 463–473. Springer.
- Peter Jansen, Peter Clark, Doug Downey, and Daniel S. Weld. 2026. [Generating Literature-Driven Scientific Theories at Scale](#). *Preprint*, arXiv:2601.16282.
- Philip Kitcher. 1990. The division of cognitive labor. *The journal of philosophy*, 87(1):5–22.
- David Klahr and Kevin Dunbar. 1988. Dual space search during scientific reasoning. *Cognitive science*, 12(1):1–48.
- Jaywon Koo, Ziyang Yang, Paola Cascante-Bonilla, Baishakhi Ray, and Vicente Ordonez. 2024. Proptest: Automatic property testing for improved visual programming. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 8241–8256.
- Yi-Te Lu and Yintong Huo. 2025. Financial named entity recognition: How far can LLM go? In *Proceedings of the Joint Workshop of the 9th Financial Technology and Natural Language Processing (FinNLP), the 6th Financial Narrative Processing (FNP), and the 1st Workshop on Large Language Models for Finance and Legal (LLMFinLegal)*, pages 164–168, Abu Dhabi, UAE. Association for Computational Linguistics.
- Omid Nejati Manzari, Hamid Ahmadabadi, Hossein Kashiani, Shahriar B Shokouhi, and Ahmad Ayatollahi. 2023. Medvit: a robust vision transformer for generalized medical image classification. *Computers in biology and medicine*, 157:106791.
- Omid Nejati Manzari, Hojat Asgariandehkordi, Taha Koleilat, Yiming Xiao, and Hassan Rivaz. 2025. Medical image classification with kan-integrated transformers and dilated neighborhood attention. *Applied Soft Computing*, page 114045.
- James G March. 1991. Exploration and exploitation in organizational learning. *Organization science*, 2(1):71–87.
- Lotfi Abdelkrim Mecharbat, Ibrahim Almakky, Martin Takac, and Mohammad Yaqub. 2025. Mednns: supernet-based medical task-adaptive neural network search. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 448–458. Springer.
- Ludovico Mitchener, Angela Yiu, Benjamin Chang, Mathieu Bourdenx, Tyler Nadolski, Arvis Sulovari, Eric C. Landsness, Daniel L. Barabasi, Siddharth Narayanan, Nicky Evans, Shriya Reddy, Martha Foiani, Aizad Kamal, Leah P. Shriver, Fang Cao, Asmamaw T. Wassie, Jon M. Laurent, Edwin Melville-Green, Mayk Caldas, and 18 others. 2025. [Kosmos: An AI Scientist for Autonomous Discovery](#). *Preprint*, arXiv:2511.02824.
- Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. [AlphaEvolve: A coding agent for scientific and algorithmic discovery](#). *Preprint*, arXiv:2506.13131.
- OpenAI. 2025. Introducing gpt-5 for developers. OpenAI product announcement, accessed March 5, 2026.
- Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. 2025. [ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory](#). *Preprint*, arXiv:2509.25140.
- Artemis Panagopoulou, Honglu Zhou, Silvio Savarese, Caiming Xiong, Chris Callison-Burch, Mark Yatskar, and Juan Carlos Niebles. 2025. Viunit: Visual unit tests for more robust visual programming. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 24646–24656.
- John R Platt. 1964. Strong inference: Certain systematic methods of scientific thinking may produce

- much more rapid progress than others. *science*, 146(3642):347–353.
- Maqsudur Rahman and Jun Zhuang. 2025. Nqnn: Noise-aware quantum neural networks for medical image classification. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 433–442. Springer.
- Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. 2021. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)*, 54(4):1–34.
- Burr Settles. 2009. Active learning literature survey.
- Chenglei Si, Zitong Yang, Yejin Choi, Emmanuel Candès, Diyi Yang, and Tatsunori Hashimoto. 2026. Towards execution-grounded automated ai research. *arXiv preprint arXiv:2601.14525*.
- Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. 2025. [Dynamic Cheat-sheet: Test-Time Learning with Adaptive Memory](#). *Preprint*, arXiv:2504.07952.
- FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jan-nik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, and 32 others. 2025. [CWM: An Open-Weights LLM for Research on Code Generation with World Models](#). *Preprint*, arXiv:2510.02387.
- Alistair Turcan, Kexin Huang, Lei Li, and Martin JinYE Zhang. 2025. Tusoi: Agentic optimization for scientific methods. *arXiv preprint arXiv:2509.23986*.
- Rosni Vasu, Chandrayee Basu, Bhavana Dalvi Mishra, Cristina Sarasua, Peter Clark, and Abraham Bernstein. 2025. HypER: Literature-grounded hypothesis generation and distillation with provenance. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 25413–25438, Suzhou, China. Association for Computational Linguistics.
- Peter C Wason. 1960. On the failure to eliminate hypotheses in a conceptual task. *Quarterly journal of experimental psychology*, 12(3):129–140.
- Wei Wen, Hanxiao Liu, Yiran Chen, Hai Li, Gabriel Bender, and Pieter-Jan Kindermans. 2020. Neural Predictor for Neural Architecture Search. In *Computer Vision – ECCV 2020*, pages 660–676, Cham. Springer International Publishing.
- Zhaotian Weng, Antonis Antoniadis, Deepak Nathani, Zhen Zhang, Xiao Pu, and Xin Eric Wang. 2026. [Group-Evolving Agents: Open-Ended Self-Improvement via Experience Sharing](#). *Preprint*, arXiv:2602.04837.
- Xu Yang, Xiao Yang, Shikai Fang, Yifei Zhang, Jian Wang, Bowen Xian, Qizheng Li, Jingyuan Li, Minrui Xu, Yuante Li, Haoran Pan, Yuge Zhang, Weiqing Liu, Yelong Shen, Weizhu Chen, and Jiang Bian. 2025a. [R&D-Agent: An LLM-Agent Framework Towards Autonomous Data Science](#). *Preprint*, arXiv:2505.14738.
- Zekang Yang, Wang Zeng, Sheng Jin, Chen Qian, Ping Luo, and Wentao Liu. 2025b. Nader: Neural architecture design via multi-agent collaboration. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 4452–4461.
- Zonglin Yang, Wanhao Liu, Ben Gao, Tong Xie, Yuqiang Li, Wanli Ouyang, Soujanya Poria, Erik Cambria, and Dongzhan Zhou. 2025c. [MOOSE-Chem: Large Language Models for Rediscovering Unseen Chemistry Scientific Hypotheses](#). *Preprint*, arXiv:2410.07076.
- Haofei Yu, Keyang Xuan, Fenghai Li, Kunlun Zhu, Zijie Lei, Jiaxun Zhang, Ziheng Qi, Kyle Richardson, and Jiaxuan You. 2025a. [TinyScientist: An Interactive, Extensible, and Controllable Framework for Building Research Agents](#). *Preprint*, arXiv:2510.06579.
- Zhaojian Yu, Kaiyue Feng, Yilun Zhao, Shilin He, Xiaoping Zhang, and Arman Cohan. 2025b. [AlphaResearch: Accelerating New Algorithm Discovery with Language Models](#). *Preprint*, arXiv:2511.08522.
- Yubiao Yue and Zhenzhang Li. 2024. Medmamba: Vision mamba for medical image classification. *arXiv preprint arXiv:2403.03849*.
- Mert Yuksekgonul, Daniel Kocaja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and 1 others. 2026. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*.
- Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. 2025. [Darwin Godel Machine: Open-Ended Evolution of Self-Improving Agents](#). *Preprint*, arXiv:2505.22954.
- Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. 2026. [Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models](#). *Preprint*, arXiv:2510.04618.
- Yitao Zhu, Yuan Yin, Jiaming Li, Mengjie Xu, Zihao Zhao, Honglin Xiong, Sheng Wang, and Qian Wang. 2025. Med-lego: Editing and adapting toward generalist medical image diagnosis. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 438–447. Springer.

Appendix

A Experiment Details

In this section, we provide implementation details for reproducing our experiments. All discovery runs use the same multi-agent pipeline described in Section 3 of the main paper and only the dataset-specific training recipe differ between CIFAR-10 and MedMNIST.

A.1 Experiment Setups

All agents in the pipeline use GPT-5-mini with a maximum output length of 32 768 tokens. Generation agents (Idea, Architect, Coding, and Feedback) use temperature 0.7, synthesis and memory agents use 0.3, and the redundancy-filter LLM Judge uses 0.1. For the redundancy filtering agent, we use Gemini Embedding API (gemini-embedding-001, 256-dim) and the top-3 most similar archived concepts are retrieved by cosine similarity. All experiments are executed on a single NVIDIA A40 GPU. Each architecture is trained with a hard wall-clock timeout of 30 minutes.

A.2 CIFAR-10 Training Protocol

Table 7 shows the training protocol of CIFAR-10.

Table 7: Default CIFAR-10 training configuration. All values may be overridden by the Coding Agent via the generated `config.py`. Additionally, new hyperparameters can be added by the Coding Agent.

Setting	Value
Resolution	32×32 (native)
Normalisation	$\mu = (0.4914, 0.4822, 0.4465)$, $\sigma = (0.2470, 0.2435, 0.2616)$
Augmentation	Random horizontal flip (50%), random translation (± 4 px, reflect pad), Cutout (12×12)
Data loader	GPU-resident (entire dataset in VRAM); all augmentation on-GPU; FP16, channels-last
Optimiser	SGD ($\text{lr}_{\text{default}} = 0.1$, momentum = 0.9, weight decay = 5×10^{-4} , Nesterov)
LR schedule	Cosine annealing with linear warmup (5 epochs)
Label smoothing	$\epsilon = 0.1$ (default)
Gradient clipping	$\ g\ \leq 1.0$
Precision	FP16 mixed-precision
Batch size	1 024 (default)
Wall-clock budget	1 800 s (30 min) per experiment
Error recovery	Up to 10 retries on code errors
HP refinement	Up to 5 steps after first successful run

A.3 MedMNIST Training Protocol

Table 8 shows the training protocol of MedMNIST.

Table 8: Default MedMNIST training configuration. All values may be overridden by the Coding Agent via the generated `config.py`. Additionally, new hyperparameters can be added by the Coding Agent.

Setting	Value
Datasets	DermaMNIST (7 cls)
Resolution	224×224 (aligned with the MedViTV2 SOTA protocol)
Channel conversion	Grayscale $\rightarrow$ RGB (as_rgb=True)
Normalisation	$\mu = (0.5, 0.5, 0.5)$, $\sigma = (0.5, 0.5, 0.5)$
Augmentation	RandomResizedCrop(224), AugMix (sev=3, width=3, $\alpha = 0.4$), RandomHorizontalFlip ($p = 0.4$)
Test/val transform	Resize(224) + normalisation only
Optimiser	Same SGD defaults as CIFAR-10 (Table 7)
Precision	FP16 mixed-precision
Batch size	128 (4 data-loader workers, pinned memory)
Wall-clock budget	1 800 s (30 min) per experiment
Error recovery	Same as CIFAR-10 (up to 10 retries, 5 refinement steps)

A.4 Search Budget

Table 9 summarizes the search budget for each discovery setting. We report the number of root candidates, parent-conditioned expansions, and trained candidates that reached the executor. Within each task, all compared HypoExplore variants use the same trained-candidate budget, timeout, data split, and validation-based selection metric. All parent selection, hypothesis updates, feedback generation, and final model selection during search use only validation metrics.

B Additional Results and Ablations

B.1 Comparison with alternative parent selection strategies

We also compare alternative parent selection strategies while holding all other components fixed (Fig. 5) on CIFAR-10. Greedy and DGM-style selection plateau early (Zhang et al., 2025; Agrawal et al., 2026). A fixed exploration-exploitation annealing schedule, similar to schedules used in prior coding agents (Si et al., 2026; Novikov et al., 2025), finds strong models early but stops improving after shifting toward exploitation. Random selection is

Task	Roots	Exp.	Executor evals	Failed	Dup. rej.	Max GPU	Search metric
CIFAR-10	5	45	50	3	1	≤ 25 A40 GPU-h	Val. accuracy
FiNER-ORD	5	15	20	8	0	≤ 10 A40 GPU-h	Val. entity F1
MedMNIST	5	25	30	0	2	≤ 15 A40 GPU-h	Val. accuracy

Table 9: Search budgets for HypoExplore. *Roots* are initial candidates generated without a parent, and *Exp.* denotes parent-conditioned expansions. *Executor evals* counts candidates that reached the training executor, including candidates that later failed or timed out. *Failed* counts candidates that compiled but did not return a valid validation score. *Dup. rej.* counts redundant candidates filtered before training. Max GPU budget is computed from executor evaluations and the per-candidate 30-minute wall-clock timeout on a single NVIDIA A40 GPU. All parent selection, hypothesis updates, feedback generation, and model selection use validation metrics only; test-set information is never used during discovery.

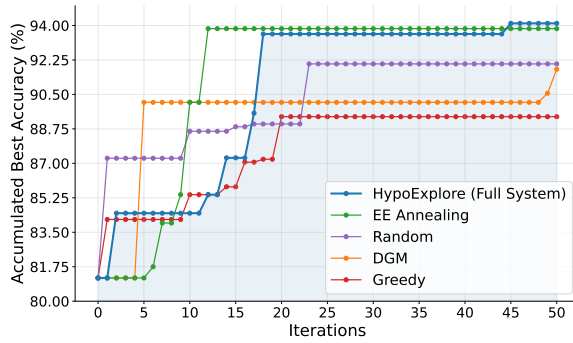


Figure 5: **CIFAR-10 parent selection strategy comparison.** All methods share the same five root architectures (iteration 0, best 81.2%). All methods have hypothesis memory, hypothesis selection, and multi-agent feedback, and only the parent selection mechanism varies.

surprisingly competitive, suggesting that hypothesis memory and feedback make broad exploration useful. HypoExplore continues improving through the full run because it scores branches using accumulated evidence rather than fitness or novelty alone.

B.2 Trajectory Exploration Dynamics

Figure 6 visualizes an example trajectory tree generated during the architecture discovery process for CIFAR-10. Each node corresponds to a candidate architecture evaluated during exploration, while node color indicates the achieved accuracy. The figure highlights that high-performing architectures do not emerge from a purely greedy search trajectory; instead, the agent explores multiple branches with varying intermediate performance before converging to stronger candidates. Interestingly, some branches with initially modest results later produce competitive descendants, suggesting that the hypothesis-guided exploration mechanism helps preserve potentially useful research direc-

tions rather than prematurely pruning them. This behavior supports our claim that maintaining a trajectory tree and hypothesis memory enables more structured and reusable scientific exploration compared to simple score-based candidate selection.

B.3 Results with Gemini-3.1-pro

Figure 7 visualizes HypoExplore’s run using Gemini 3.1 Pro across 45 iterations excluding the five root nodes. The x-axis represents iteration order (0 for the 5 root ideas, then 1 through 45 for subsequent architectures), while the y-axis shows CIFAR-10 accuracy (%). Each color corresponds to one of the 5 root lineages: root1 (node_0, "PolyMixer," 31.0%), root2 (node_1, "LatentMixer," 44.4%), root3 (node_2, "ScanMLP," 0.0%), root4 (node_3, "DilatedPatchMLP," 92.1%), and root5 (node_4, "HyperCubeMLP," 29.2%). The thick green line highlights the best-performing lineage path, where it originates from root3’s node_2 ("ScanMLP"), which completely failed at 0.0% accuracy due to repeated CUDA out-of-memory errors. Despite this unpromising start, HypoExplore selected root3 in Generation 31, due to its high exploration bonus from being unvisited and found a neural network with 28.3% accuracy ("DualFreqScanMLP"). Then in Generation 35, it was selected and produced "PyramidGateMLP: Multi-Scale Pooled Gating with Full-Capacity Global Context", which achieved the highest accuracy of 94.9%, a dramatic jump of +66.6 percentage points in a single generation and the overall best across all 50 evaluated architectures.

C Implementation Details

This section specifies the exact input/output contracts and behavioral modes of each component in the HypoExplore pipeline, complementing the high-level description in Section 3.

2. {parent_architecture}: parent node’s full brainstorming output (title, description, intuition, novelty, architecture spec) + performance metrics (accuracy, training time, novelty score)
3. {feedback_summary}: concatenated outputs from all feedback agents, including per-agent reasoning, actionable recommendations, hypothesis updates with current confidence from $\mathcal{M}_t$, and newly proposed hypotheses
4. {hypothesis_memory}: compiled hypothesis context from $\mathcal{M}_t$, grouped into confirmed ($c > 0.75$), refuted ($c < 0.25$), and uncertain patterns, each with full evidence log and agent attribution
5. Selected hypothesis h^* to test

Output

A single evolved architecture containing:

- Structured reasoning trace: parent_analysis (what worked), failure_analysis (what failed), hypothesis_usage (which hypotheses guide design), proposed_changes (targeted modifications)
- Architecture fields: title, description, architecture_spec, etc.
- existing_hypotheses: referenced hypothesis IDs (e.g., ["hyp_3", "hyp_7"])
- Up to K_{hyp} new hypotheses motivated by the proposed design

Behavior

Hypothesis-guided design conditioned on parent context. Makes targeted 1–2 aspect changes (not full redesigns). Builds on confirmed patterns, investigates uncertain hypotheses, avoids refuted patterns.

C.2 Coding Agent

The Coding Agent operates in three modes, each with distinct input/output.

First, here is the specific details for the initial generation (iteration 1):

Input

Architecture specification from the Idea Agent, problem specification (dataset, input shape, number of classes), and implementation notes from $\ell_t(h^*)$ recording failure modes and recommended practices from prior experiments testing

the same hypothesis.

Output

Two files: model.py (PyTorch nn.Module) and config.py (hyperparameter dictionary).

Constraints: <10M parameters, mixed-precision (AMP) support, channels-last memory format, training must complete within t_{max} .

Here is the specific details for Error recovery (fix mode):

Input

Previous model.py, error traceback, and the original architecture specification.

Output

Corrected model.py; the existing config.py is reused.

Behavior

Retries up to R_{max} times. Each retry receives the latest traceback, enabling iterative debugging.

We specify the details for the Hyperparameter refinement (refine mode):

Input

Previous config.py, training logs (loss curves, accuracy progression, convergence diagnostics), and the current model.py.

Output

Updated config.py; the existing model.py is reused.

Behavior

Runs for up to F_{max} steps with early stopping if accuracy plateaus between consecutive iterations.

C.3 Redundancy Filtering Agent

Before execution, a two-stage filter prevents re-exploration of previously visited concepts.

Stage 1: Embedding-based retrieval. The candidate architecture’s concept description is embedded (256-dimensional, Gemini Embedding API) and the top- $k=3$ most similar archived concepts in $\mathcal{T}_t$ are retrieved by cosine similarity.

Stage 2: LLM judge.

Input

Candidate architecture (title, description, generated code) together with the top-3 retrieved concepts from $\mathcal{T}_t$.

Output

Structured judgment:

- **novel**: boolean decision
- **reasoning**: explanation of the decision
- **most_similar_to**: ID of the most similar existing concept
- **shared_principles**: what the candidate shares with existing work
- **new_contribution**: what is genuinely new

Behavior

Compares structural design principles rather than surface-level differences (e.g., different hyperparameters or activation choices do not constitute novelty). Same innovation claim + same structural pattern = duplicate. If rejected, the Idea Agent may regenerate with explicit instructions to avoid the flagged overlap, for up to 2 retries.

C.4 Feedback Agents

After each experiment, four specialized agents analyze the outcome from complementary perspectives. All agents share a common output schema containing reasoning, actionable_feedback, and hypothesis_updates. Each hypothesis update contains evidence_type $\in \{\text{supports, contradicts, neutral}\}$, strength $\in [0, 1]$, reasoning, and new_hypotheses.

We specify the details of the Quantitative Feedback Agent:

Input

Training curves (loss and accuracy per epoch), final top-1/top-5 accuracy, per-class accuracy breakdown, parameter count, FLOPs, and training time.

Output

Statistical analysis of convergence patterns, efficiency characterization, and hypothesis evidence. Identifies overfitting signatures, class-specific weaknesses, and computational bottle-

necks.

We specify the details of the Qualitative Feedback Agent (VLM):

Input

N_{err} highest-confidence misclassified images with corresponding heatmap overlays. Heatmap method is auto-detected: GradCAM (CNN-dominant), Attention Rollout (transformer-dominant), Input Gradient Saliency (universal fallback, e.g. MLP-Mixers).

Output

Per-image analysis of attention patterns, texture vs. shape bias identification, confusion patterns between visually similar classes, and hypothesis evidence about failure modes.

We specify the details of the Causal Feedback Agent:

Input

Parent and child architecture specifications, structural diff between them, and performance delta ($\Delta\text{accuracy}$, $\Delta\text{training time}$).

Output

Causal attribution of observed performance changes to specific structural modifications (e.g., “replacing batch normalization with layer normalization improved accuracy by 1.2%”). Produces directed hypothesis updates linking architectural changes to outcomes.

We specify the details of the Diagnostic Feedback Agent (failure/timeout only):

Input

Error traceback or timeout logs, the generated model.py, and the architecture specification.

Output

Root cause analysis (e.g., OOM, numerical instability, excessive FLOPs) and structured implementation notes:

- **common_failures**: failure modes with descriptions, recorded into $\ell_t(h)$
- **recommended_practices**: concrete guidance to avoid recurrence

These notes persist in the hypothesis memory and are provided to the Coding Agent in future iterations testing the same hypothesis, enabling the system to learn from implementation failures.

C.5 Hypothesis Synthesis Agent

Input

Concatenated outputs of all feedback agents (hypothesis updates and new hypothesis proposals) and the current hypothesis memory $\mathcal{M}_t$.

Output

Consolidated update containing:

- `hypothesis_updates`: deduplicated evidence updates for existing hypotheses, each with `hyp_id`, `evidence_type`, `strength`, and cross-agent reasoning
- At most $K_{\text{synth}}=2$ new_hypotheses, each in the structured IF/THEN/BECAUSE/DISPROVED IF format with scope, prediction, falsification_criteria, and tags
- `implementation_notes`: failure modes and recommended practices keyed to hypothesis IDs

Behavior

All feedback agent outputs are processed in a single LLM call. Resolves three types of conflicts:

- overlapping hypothesis updates from different agents are merged, with synthesized reasoning citing all contributing perspectives;
- contradictory evidence assessments are resolved by weighing the specificity of each agent's analysis;
- redundant new hypothesis proposals are collapsed.

Quality gate. Every new hypothesis must pass a 7-dimension quality gate before admission to $\mathcal{M}_t$:

7-Dimension Hypothesis Quality Gate

1. *Mechanistic*—explains *why* via a causal mechanism, not just correlation
 2. *Scoped*—states where the hypothesis applies and where it does not
 3. *Predictive*—specifies the expected direction and magnitude of effect
 4. *Falsifiable*—defines what experimental result would disprove it
 5. *Novel*—is not a restatement of an existing hypothesis in $\mathcal{M}_t$
 6. *Transferable*—applies across architectures, not just the current model
 7. *Actionable*—can be tested with a single targeted architectural change
- Hypotheses failing any dimension are either revised by the synthesis agent or discarded.

C.6 Hypothesis Memory Bank

The hypothesis memory bank $\mathcal{M}_t$ stores per-hypothesis records and provides structured retrieval for downstream agents.

Per-hypothesis storage.

Each hypothesis $h \in \mathcal{M}_t$ maintains:

- Confidence $c_t(h) \in [0.01, 0.99]$, initialized at $c_0 = 0.5$;
- An evidence log: list of (node_id, type, strength, reasoning, agent) tuples
- Implementation notes $\ell_t(h)$: accumulated failure modes with frequency and recommended practices
- Bidirectional connections to related hypotheses
- Structured fields: scope, prediction, falsification_criteria
- Provenance: created_by agent identifier and source node.

Confidence update. Evidence updates follow Equation 1 from the main paper with learning rate $\eta = 0.20$. Hypotheses are classified as *confirmed* when $c > 0.75$, *refuted* when $c < 0.25$, and *uncertain* otherwise.

C.7 Trajectory Tree Memory

The trajectory tree $\mathcal{T}_t$ stores the complete research history as a forest of parent-child node relationships.

Per-node storage

Each IdeaNode $v \in \mathcal{T}_t$ stores:

- **idea**: the full brainstorming output dictionary;
- **architecture_graph**: DAG representation (nodes and edges) for novelty scoring;
- **wl_embedding**: 1024-dimensional Weisfeiler-Lehman embedding vector;
- **code_attempts**: list of all (model.py, config.py) pairs attempted;
- **experiments**: training run results (accuracy, loss curves, timing, FLOPs);
- **feedback**: per-agent feedback records;
- **Hypothesis tracking**:
tested_hypothesis_id,
hypothesis_prediction,
hypothesis_evidence_type, and
hypothesis_evidence_strength;
- **Scores**: best_accuracy,
novelty_score, selection_score,
times_selected.

D Discovered Architectures

This section presents the complete Idea Agent output and final implementation code for the three highest-performing architectures discovered.

D.1 GST-Guarded NPIN (94.11%)

Idea Agent Output: GST-Guarded NPIN: Sparse Global Shape Tokens + Per-Band Normalized Super-Particles

Description. Build on NPIN-Guard (particle dynamics + EMA-stabilized super-particles + FiLM gating + band-split) and introduce two minimal, targeted changes: (1) add a very small set of stage-local Global Shape Tokens (GSTs, $G \leq 4$) that aggregate low-frequency token summaries via sparse top- k routing and provide a compact, shape-focused global routing path; (2) apply per-band magnitude normalization to super-particle (slot) anchor messages before FiLM blending, plus make FiLM gate compute a GST-conditioned coherence score (GST-conditioned gate). These changes are introduced to (a) provide an explicit, tiny shape-oriented global aggregator that supplements semi-parametric super-particles

and reduces reliance on many texture-driven slots, and (b) prevent any anchor message (esp. high-frequency) from overwhelming token identity during broadcast. All remaining NPIN-Guard components ($S=2$ local integrator, locality grid neighbor lookup, EMA+usage-normalized slot updates, band-split, per-band bookkeeping) are preserved so differences are causal and small.

Architecture pipeline: PatchEmbed $\rightarrow$ BandSplit (DWConv low-pass, residual high-pass) $\rightarrow$ ParticleDynamicsBlock (per-band local neighbor gather; $S=2$ integrator; depthwise MLP PotentialNet) $\rightarrow$ SuperParticleCoarsen (per-band K_b slots, soft assignment, EMA + per-slot usage normalization) $\rightarrow$ GST Module (G small learned tokens per stage that collect low-frequency token summaries via sparse top- k routing and compute compact GST states) $\rightarrow$ GST-conditioned FiLM Gate & Per-band Anchor Normalization $\rightarrow$ FiLM-Gated Broadcast (anchor messages modulate tokens only when gated) $\rightarrow$ Token MLP & Norm.

Intuition. NPIN-Guard already demonstrated that stabilizing super-particle prototypes and FiLM-gating multiscale messages dramatically reduces texture-driven failures. Two residual failure modes remain plausible: (a) some global shape groupings require a compact cross-token aggregator not easily captured by semi-static slots because slots are optimized for multi-band prototype coverage rather than explicitly coding global shape templates, and (b) large anchor messages (especially from high-frequency slots) can still overwhelm token identity despite gating if magnitudes vary across bands. Adding a tiny set of learned GSTs supplies an explicit, low-capacity global routing channel that focuses on low-frequency/shape cues (G small keeps compute trivial). Per-band L2-normalization of anchor messages prevents magnitude-driven takeovers and makes blending with residual token features numerically stable, further reducing prototype monopolies.

Novelty. The architecture fuses three mechanisms in a unique, minimal way: (1) particle-ODE local dynamics for ef-

efficient local mixing, (2) semi-parametric EMA-stabilized super-particles for multi-scale aggregation, and (3) a tiny stage-local GST pathway that performs sparse top- k low-frequency routing to provide explicit, compact shape aggregation. The GSTs do not implement full cross-attention and are constrained ($G \leq 4$, sparse routing) to remain structurally distinct from standard global attention methods. Per-band L2-normalization of anchor messages (applied before FiLM blending) combined with GST-conditioned gating is a novel practical stabilizer that directly targets prototype takeover while preserving local dynamics. **Target Improvement:** both (accuracy and efficiency).

Architecture Specification:

Core Ideas: 1) Preserve NPIN particle dynamics (local neighbor ODE steps) for efficient local mixing. 2) Keep EMA-stabilized, per-band super-particle slots with per-slot usage normalization to avoid drift. 3) Add stage-local Global Shape Tokens (GSTs, $G \leq 4$) that aggregate low-frequency token summaries via sparse top- k routing ($O(G \cdot N)$ cheap ops) and broadcast compact shape updates back to tokens. 4) Apply per-band L2-normalization to anchor messages before FiLM blending and compute FiLM gate as a GST-conditioned coherence score, so broadcasts occur only when GSTs and low-frequency tokens agree.

Core Blocks:

- **PatchEmbed:** 3×3 conv stride 2 (or stride 1 + small pooling) $\rightarrow$ produce token features and initial learned-position offsets. Use small base channels ($C=64$) to control params.
- **BandSplit:** DWConv low-pass ($k=3$) $\rightarrow$ low-frequency stream; high-frequency = residual ($\text{feat} - \text{lowpass}$). Reduce per-band channels with grouped 1×1 conv to keep budget low.
- **ParticleDynamicsBlock:** For each band share neighbor grid; for each token gather neighbors within radius r (small). PotentialNet: depthwise MLP producing neighbor messages aggregated via sum/mean. Integrator: $S=2$ explicit steps updat-

ing per-token feature residual and small learned position offsets. Residual connections preserved.

- **SuperParticleCoarsen** (per-band): Soft assignment (temperature τ) of tokens to K_b slots; aggregate normalized per-slot summaries; update prototypes via EMA: $\text{proto} \leftarrow m \cdot \text{proto} + (1-m) \cdot (\text{agg}/(\text{usage} + \epsilon))$. Keep usage as EMA of assignment mass. Small learned gradient step permitted but heavily scaled.
- **GST Module** (stage-local): G learnable tokens per stage ($G=1 \dots 4$). Each token computes a tiny low-frequency key ($\text{MLP} \rightarrow k_{\text{dim}}$). Compute top-1 or top-2 scores for G (cheap when G small). Aggregate token low-frequency summaries into chosen GSTs (sum or weighted sum). Update GST vectors with small gradient steps; optionally warmup EMA to stabilize early training.
- **GST-conditioned FiLM Gate & Anchor Norm:** Per-band anchor vectors (super-particle $\rightarrow$ token messages) L2-normalized independently. Compute gate $g_{\text{token}} = \sigma(w \cdot \cos_{\text{sim}}(\text{token_lf}, \text{GST_state}) + b)$. Gates multiplicatively scale FiLM parameters and broadcast mass.
- **FiLM-Gated Broadcast:** For each token, compute slot-weighted (soft assignment) anchor message but multiply by g_{token} and apply FiLM: $\text{feat}' = \text{feat} \cdot (1 + g_{\text{token}} \cdot \gamma_{\text{slot}}) + g_{\text{token}} \cdot \beta_{\text{slot}}$. Because anchors are normalized, the FiLM scale remains bounded.
- **Token MLP & Norm:** Depthwise MLP + residual normalization to finalize block output.

Network Structure: 3-stage backbone: Early stage (no coarsen): 3 ParticleDynamicsBlocks, base channels $C=48 \rightarrow 64$. Mid stage (first coarsen): 3 ParticleDynamicsBlocks with per-band slots ($K_{\text{low_mid}}=24$, $K_{\text{high_mid}}=12$) and GSTs $G_{\text{mid}}=2$. Final stage: 2 ParticleDynamicsBlocks with coarsen ($K_{\text{low_final}}=8$, $K_{\text{high_final}}=4$) and GSTs $G_{\text{final}}=1$. Global avg pooling $\rightarrow$ classifier head. Residuals across blocks; per-band bookkeeping passed for-

ward. Use grouped 1×1 convs and depth-wise ops to keep params low. Estimated param/FLOP budget: $\sim 1.0\text{--}3.5\text{M}$ params (config-dependent) and $< 200\text{M}$ FLOPs for CIFAR-size inputs.

Tunable Aspects:

- GST count G per stage (1–4)
- Super-particle counts K_b per band/stage
- EMA momentum for prototypes (0.98–0.999)
- Assignment temperature τ
- Neighbor radius r and integrator steps S (S default = 2)
- Anchor L2-norm scaling factor (post-normalization scale)
- Gate computation (cos-sim vs. MLP), thresholding

Invariants:

- All local propagation inside ParticleDynamicsBlock uses only local neighbors (no all-to-all N^2 ops).
- Super-particles are semi-parametric EMA-updated slots with per-slot usage normalization retained.
- Bandwise bookkeeping (low/high) is preserved end-to-end and normalization is applied per-band, not globally.
- GSTs remain small ($G \leq 4$) and perform sparse $O(G \cdot N)$ routing, not full $N \times N$ attention.

New Hypotheses:

1. **IF** a small number ($G \leq 4$) of stage-local Global Shape Tokens (GSTs) are added that aggregate low-frequency token summaries via sparse top- k routing and broadcast compact shape-aware updates back to tokens **IN** NPIN-Guard-style particle+slot backbones on CIFAR-scale tasks, **THEN** overall shape-sensitive accuracy will increase and texture-driven confusions will decrease, **BECAUSE** GSTs provide a low-cost, explicit global routing channel specialized for low-frequency/shape cues that assembles spatially distributed evidence without relying on many texture-prone super-particle slots. **DISPROVED IF** adding GSTs (with controlled extra params and matched FLOPs) fails to reduce texture-driven

misclassifications or GST activations remain near-zero across validation images.

Tags: global-routing, sparse-routing, shape-aggregation. *Initial confidence:* 0.55. *Connected:* hyp_44, hyp_27.

Reasoning: GSTs are a compact, low-capacity mechanism targeted to the same failure mode (shape starvation) identified previously. hyp_44 already suggested GSTs with moderate evidence; here we test a GST design constrained to sparse top- k routing and low-frequency conditioning to keep overhead tiny and maximize shape focus.

2. **IF** super-particle anchor messages are L2-normalized per-band before FiLM blending **IN** multi-band NPIN-style backbones, **THEN** the incidence of anchor-driven prototype-takeover misclassifications will decrease and shape-sensitive per-class recall will improve, **BECAUSE** bounding anchor magnitudes prevents high-frequency anchors from numerically overwhelming token identity vectors and allows gating/FiLM modulation to effectively preserve local shape evidence. **DISPROVED IF** per-band L2-normalization yields no measurable reduction in concentrated confusion mass onto dominant texture classes and no improvement in shape-sensitive per-class recall beyond random run variance. *Tags:* normalization, stability, anchor-magnitude. *Initial confidence:* 0.6. *Connected:* hyp_58, hyp_18.

Reasoning: hyp_58 articulated the same mechanism; NPIN-Guard improvements hinted that message magnitude could be a remaining knob. Per-band normalization is simple, inexpensive, and falsifiable by measuring confusion mass and per-prototype assignment distributions.

Reasoning:

Parent Analysis: NPIN-Guard preserved NPIN’s core strength—local particle dynamics (learned potential + short ODE integration) with an efficient locality grid—for

cheap and effective local mixing. The introduction of EMA-stabilized super-particles, per-slot usage-normalization, FiLM gating conditioned on low-frequency features, and band-splitting all correlated with a large, robust improvement in Top-1 accuracy (from $\sim 19\%$ to $\sim 57\%$), stable training dynamics, and reduced texture-dominated collapse.

Failure Analysis: Although NPIN-Guard made big gains, residual risks remain: (1) some global shape groupings may not be captured by slot prototypes alone, causing remaining confusions; (2) anchor messages, especially from high-frequency slots, can still have magnitude imbalances that numerically overwhelm token identity even when gated; (3) causal attribution was confounded because multiple interventions were introduced simultaneously in NPIN-Guard.

Hypothesis Usage: We prioritize hyp_13 (prototype drift mitigated by EMA + usage-normalization) and hyp_27 (FiLM-gated retrieval conditioned on low-frequency shape cues) as confirmed drivers—so we keep EMA + per-slot usage-normalization and low-frequency FiLM gating unchanged. hyp_44 (GSTs) performed moderately in memory and directly motivates adding a tiny explicit global routing channel; we adopt this but constrain GSTs to be sparse and low-capacity to avoid large compute/overfitting. hyp_18 and hyp_58 (per-band starvation and anchor magnitude bounds) are uncertain but actionable; we incorporate per-band L2-normalization of anchor messages to test hyp_58 and maintain per-band bookkeeping to test hyp_18.

Proposed Changes: (1) Insert stage-local GSTs ($G \leq 4$) that aggregate low-frequency token summaries via sparse top- k routing (cheap $O(G \cdot N)$) and broadcast compact shape summaries back to tokens; this supplements semi-parametric super-particles with an explicit global shape channel (tests hyp_44). (2) L2-normalize anchor messages per-band prior to FiLM blending, and compute FiLM gate as a GST-conditioned coherence score (σ of cosine similarity) so that broadcasts occur

only when GSTs and token low-frequency cues align (tests hyp_58 and strengthens hyp_27).

Existing Hypotheses Referenced: hyp_13, hyp_27, hyp_18, hyp_44, hyp_58.

config.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class BasicBlock(nn.Module):
    """A small ResNet-like basic block (kept
    lightweight)"""
    def __init__(self, in_planes, planes, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(in_planes, planes,
                                kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(planes, planes,
                                kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        if stride != 1 or in_planes != planes:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_planes, planes,
                           kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes),
            )
        else:
            self.downsample = None

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)
        return out

class GSTModule(nn.Module):
    """
    Lightweight GST module:
    - Maintains a small set of GST vectors per stage
    - Computes similarity between L2-normalized
      tokens and normalized GSTs
    - Uses temperature + softmax to compute
      assignments
    - Optionally enforces top-k sparsity per token (
      masking other assignments), renormalizes
    - Aggregates GST vectors weighted by assignments
      and merges back via 1x1 conv residual
    """
    def __init__(self, channels, G=0, topk=1, temp
                 =0.1, eps=1e-6):
        super().__init__()
        self.channels = channels
        self.G = max(0, int(G))
        self.topk = max(1, int(topk)) if self.G > 0
        else 0
        self.temp = float(temp)
        self.eps = float(eps)

        if self.G > 0:
            # GST vectors: [G, C]
```

```

        self.gsts = nn.Parameter(torch.randn(self.
G, channels))
        # small projection to mix GST effect into
features
        self.proj = nn.Conv2d(channels, channels,
kernel_size=1, bias=True)
        # initialize proj to near-zero so
residual starts near identity
        nn.init.constant_(self.proj.bias, 0.0)
        nn.init.constant_(self.proj.weight, 0.0)
    else:
        # placeholders to simplify forward logic
        self.register_parameter('gsts', None)
        self.proj = nn.Identity()

def forward(self, x):
    # x: [B, C, H, W]
    if self.G == 0:
        return x

    B, C, H, W = x.shape
    N = H * W

    # tokens: [B, N, C]
    tokens = x.permute(0, 2, 3, 1).reshape(B, N,
C)

    # L2-normalize tokens and GSTs along feature
dim
    t_norm = tokens / (tokens.norm(dim=-1,
keepdim=True) + self.eps)
    gst = self.gsts # [G, C]
    gst_norm = gst / (gst.norm(dim=-1, keepdim=
True) + self.eps) # [G, C]

    # similarity: [B, N, G]
    sim = torch.matmul(t_norm, gst_norm.t()) / (
self.temp + 1e-12)

    # soft assignment
    assn = F.softmax(sim, dim=-1) # [B, N, G]

    # top-k masking (sparse selection). If topk
== G then nothing changes.
    if 0 < self.topk < self.G:
        # get indices of topk per token
        topk_vals, topk_idx = assn.topk(self.topk,
dim=-1) # [B, N, topk]
        # create mask
        mask = torch.zeros_like(assn)
        # scatter ones into mask
        mask.scatter_(-1, topk_idx, 1.0)
        assn = assn * mask
        # renormalize selected assignments so
each token sums to 1 (or to small eps)
        denom = assn.sum(dim=-1, keepdim=True)
        assn = assn / (denom + self.eps)

    # weighted aggregation of GSTs: [B, N, C]
    gst_effect = torch.matmul(assn, gst_norm) #
[B, N, C]
    # reshape to [B, C, H, W]
    gst_effect = gst_effect.view(B, H, W, C).
permute(0, 3, 1, 2).contiguous()

    # merge via 1x1 conv and residual add
    out = x + self.proj(gst_effect)
    return out

class Model(nn.Module):
    def __init__(self, config=None):
        super().__init__()
        # Extract hyperparameters from config with
sensible defaults
        if config is None:
            config = {}
        self.base_channels = int(config.get('
base_channels', 48))
        self.stage_channels = list(config.get('
stage_channels', [48, 64, 96]))
        self.blocks_per_stage = list(config.get('
blocks_per_stage', [3, 3, 2]))
        self.gsts_per_stage = list(config.get('
gsts_per_stage', [0, 2, 1]))

        self.gst_topk = int(config.get('gst_topk', 1)
)
        self.assignment_temp = float(config.get('
assignment_temp', 0.1))
        self.integrator_steps = int(config.get('
integrator_steps', 2))
        self.eps = float(config.get('eps', 1e-6))
        self.num_classes = int(config.get('
num_classes', 10))
        # ensure lengths match expected 3 stages
        assert len(self.stage_channels) == 3, "
stage_channels must be length 3"
        assert len(self.blocks_per_stage) == 3, "
blocks_per_stage must be length 3"
        assert len(self.gsts_per_stage) == 3, "
gsts_per_stage must be length 3"

        # Stem
        self.stem = nn.Sequential(
            nn.Conv2d(3, self.base_channels,
kernel_size=3, stride=1, padding=1, bias=False),

            nn.BatchNorm2d(self.base_channels),
            nn.ReLU(inplace=True),
        )

        # Stages
        in_ch = self.base_channels
        self.stages = nn.ModuleList()
        self.gst_modules = nn.ModuleList()
        for idx, out_ch in enumerate(self.
stage_channels):
            blocks = []
            n_blocks = self.blocks_per_stage[idx]
            # downsample at stage transitions except
first stage (keep spatial 32x32->16x16->8x8)
            stride = 1 if idx == 0 else 2
            # first block may downsample
            blocks.append(BasicBlock(in_ch, out_ch,
stride=stride))
            for _ in range(1, n_blocks):
                blocks.append(BasicBlock(out_ch,
out_ch, stride=1))
            self.stages.append(nn.Sequential(*blocks)
)

            # GST module for this stage
            G = int(self.gsts_per_stage[idx])
            gst_mod = GSTModule(out_ch, G=G, topk=
self.gst_topk, temp=self.assignment_temp, eps=
float(config.get('gst_eps', 1e-6)))
            self.gst_modules.append(gst_mod)
            in_ch = out_ch

        # classifier head
        self.global_pool = nn.AdaptiveAvgPool2d((1,
1))
        self.classifier = nn.Linear(self.
stage_channels[-1], self.num_classes)

        self._initialize_weights()

def forward(self, x):
    # Input: [B, 3, 32, 32]
    # Output: [B, 10] (num_classes)
    # No explicit memory_format or dtype
conversions here (AMP / channels_last
compatible)
    B = x.shape[0]
    out = self.stem(x) # [B, C0, 32, 32]

    # pass through stages, applying GST modules
and simple integrator steps
    for stage, gst_mod in zip(self.stages, self.
gst_modules):
        out = stage(out) # residual blocks
        # apply GST module possibly multiple
integrator steps (simple recurring refinement)
        # We keep this efficient: repeated small
conv/1x1 operations, avoiding large allocations
        for _ in range(self.integrator_steps):
            out = gst_mod(out)

    # classification head
    out = self.global_pool(out) # [B, C, 1, 1]
    out = out.view(B, -1) # [B, C]

```

```

        out = self.classifier(out) # [B, num_classes]
    ]
    return out

def _initialize_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            # standard Kaiming init for conv
            nn.init.kaiming_normal_(m.weight,
mode='fan_out', nonlinearity='relu')
            if m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.Linear):
            # small std normal for linear layers
            nn.init.normal_(m.weight, 0, 0.01)
            if m.bias is not None:
                nn.init.constant_(m.bias, 0)

```

```

'anchor_scale_init': 1.0, # post-L2
    normalization scalar for anchors
'integrator_steps': 2, # S=2 integrator
    steps in ParticleDynamicsBlock
'anchor_eps': 1e-6,
'gst_eps': 1e-6,
# classifier
'num_classes': 10,
# small numeric eps
'eps': 1e-6,
}

```

D.2 Hierarchical Hub Routing Network (93.57%)

Idea Agent Output: Hierarchical Hub Routing Network (HHRN)

Description. HHRN replaces the NPIN particle-dynamics connectivity with a lightweight, soft hub-mediated routing backbone that produces sparse, content-adaptive, multi-hop token interactions. Tokens are produced by a conv-based PatchEmbed (depthwise separable convs + small per-scale phase-offsets). Each token computes a low-dimensional “fingerprint” key which is used to softly route (via temperatureed softmax/residual blend) to a small set of global hub vectors ($H \approx 16-48$). Hubs aggregate token messages (weighted averages), run a short sparse GNN among hubs (1–2 hops) to exchange higher-order/global context, and then broadcast refined hub messages back to tokens via the same soft routing weights (symmetry via shared keys). The token update is a residual blend between local conv features and hub-broadcast correction; local convs capture fine spatial detail while hubs carry long-range, structured relational information. Per-scale small learnable phase-offsets are applied in the local conv analysis transforms to break perfect shift-symmetry.

All routing is deliberately soft (temperatureed / small initial blending weight) to preserve per-token uncertainty and avoid hard prototype collapse. The hub set is small ($H \ll N$), updated online (learnable + lightweight EMA carrying state between blocks) but not computed via heavy differentiable clustering; hub adjacency (hub graph) is learned as a sparse trainable connectivity (top- s per hub) computed from hub features, enabling multi-hop hub chaining without

config.py

```

config = {
    # Experiment info
    'experiment_name': 'GST-Guarded NPIN',
    'notes': 'CIFAR-scale NPIN refinement: tiny GSTs
        (G<=4) + per-band L2-normalized super-particle
        anchors + particle dynamics',

    # Data settings
    'data_dir': 'cifar10',

    # Training hyperparameters
    'epochs': 200,
    'batch_size': 1024, # large batch to utilize GPU
        memory; reduce if OOM
    'learning_rate': 0.1,
    'momentum': 0.9,
    'weight_decay': 5e-4,
    'nesterov': True,

    # Learning rate schedule
    'lr_schedule': 'cosine',
    'warmup_epochs': 5,

    # Regularization
    'label_smoothing': 0.1,

    # Data augmentation
    'flip': True,
    'translate': 4,
    'cutout': 12,

    # Evaluation
    'tta_level': 0,
    'eval_every': 1,

    # Model settings
    'use_half_precision': True,
    'channels_last': True,
    'verbose': True,

    # Model-specific hyperparameters
    'base_channels': 48, # base channel
        width (kept small to meet param budget)
    'stage_channels': [48, 64, 96], # stage channels
        for 3 stages
    'blocks_per_stage': [3, 3, 2], # blocks per
        stage
    # Super-particle slot counts per band per stage:
        list of tuples (K_low, K_high)
    'slots_per_stage': [(0, 0), (24, 12), (8, 4)],
    # GST counts per stage (G small)
    'gsts_per_stage': [0, 2, 1],
    'gst_topk': 1, # top-k selected
        GSTs per token (1..G)
    'assignment_temp': 0.1, # temperature for
        soft assignment (before any topk masking)
    'ema_momentum': 0.995, # for prototypes
        and GSTs

```

$N \times N$ token pairwise operations. Computation stays efficient (no $O(N^2)$ expansions): token→hub and hub→token operations cost $O(N \cdot H)$ with small H , and hub→hub GNN costs $O(H \cdot s)$ with small s .

Intuition. The research direction asks for fundamentally different token connectivity. Instead of simulating local particle dynamics or computing dense attention, HHRN routes token information via a small set of shared intermediary hubs arranged into a sparse graph. This yields an expressive family of structured sparse interaction patterns: tokens influence one another indirectly by co-attending to the same hubs and by hub-graph propagation which chains multi-hop interactions. Soft routing and residual blending preserve per-token representational variance (addressing hyp_14), while per-scale phase offsets improve asymmetry and shape sensitivity (addressing hyp_26). Using hubs avoids quadratic costs and offers explicit control of global mixing capacity via H and hub-graph sparsity. Because tokens still include efficient local convs for fine-grained detail, the model preserves low-level spatial inductive bias required for CIFAR-10 while enabling long-range relational reasoning through a compact hub layer. The design prioritizes parameter/FLOP efficiency (target <10M params and <2B FLOPs) and low memory (no pairwise $N \times N$), enabling very fast training on a single GPU.

Novelty. HHRN is novel because it replaces continuous latent-position ODEs and multiscale coarsening with a learned, soft hub-routing + sparse hub-graph propagation mechanism that: (1) uses a compact set of reusable global hub vectors (learned + light online state) rather than dynamically clustering tokens each block; (2) runs explicit sparse GNN propagation among hubs to achieve higher-order chaining; (3) enforces soft temperatureed routing plus learned residual blending to avoid hard prototype collapse. Unlike NPIN’s geometric particle interactions or hyperedge attention/hypergraph methods, HHRN uses a two-stage token↔hub exchange with explicit hub-hub

GNN chaining as the primary global mixing primitive, and augmenting local convs with tiny learnable phase offsets to break shift-symmetry.

Target Improvement: both (accuracy and efficiency).

Architecture Specification:

Core Ideas: 1) Replace dense or particle-based global mixing with a compact hub layer: tokens route softly to H learned hubs; hubs aggregate, run sparse GNN steps among themselves, and broadcast back. 2) Preserve strong local inductive bias via depthwise separable convs and small per-scale phase-offsets. 3) Keep routing soft with residual blending (α) to avoid hard-prototype collapse; learn α but initialize small. 4) Keep computations $O(N \cdot H) + O(H \cdot s)$, avoid $N \times N$ expansions.

Core Blocks:

- **PatchEmbed:** 3-layer conv stem using depthwise separable convs (kernel sizes $3 \rightarrow 3$), producing token features ($C=96-128$) and initial low-dim fingerprints via a 1×1 projection ($d_k=48-64$). Small learnable per-scale phase-offsets are added to depthwise sampling indices before applying kernels (parameter budget tiny).
- **LocalConvBlock:** Two lightweight residual depthwise separable conv layers + pointwise FFN to capture fine spatial detail. LayerNorm/BatchNorm as appropriate for fast CIFAR training.
- **HubRoutingBlock:** (a) Token→Hub routing: compute similarity between token fingerprints and H hub keys (cosine or dot), apply temperatureed softmax across hubs to get soft routing weights $w_{t \rightarrow h}$. (b) Hub aggregation: hubs receive weighted average messages $m_h = \sum_t w_{t \rightarrow h} \cdot \text{token_value}_t$ (value is a small projection of token features). Hubs also maintain a small learnable state vector updated via a small MLP and optional EMA to carry information between blocks. (c) Hub-GNN propagation: construct a sparse adjacency by computing top- s neighbors per hub from hub features ($s=2-4$) and run 1–2 GNN message-passing steps (edge MLPs

with shared parameters) to allow hub-hub chaining. (d) Hub→Token broadcast: compute corrections for tokens as $\sum_h w_{t \rightarrow h} \cdot \text{hub_message}_h$ (reuse w for symmetry). (e) Token update: $\text{token}' = \text{LayerNorm}(\text{token} + \alpha \cdot \text{hub_correction})$ followed by local FFN; α is learnable and initialized small (e.g., 0.15).

- **TokenFFN & Norm:** Standard 2-layer MLP with GELU and residual + LayerNorm after each HubRoutingBlock.

Network Structure: Stack: PatchEmbed → [(LocalConvBlock × 2) → HubRoutingBlock] repeated across 3 stages with channel widths (e.g., 64 → 96 → 128) and spatial downsampling by strided convs between stages. Hubs are per-stage with separate hub banks (H small per stage: 16/24/32) to match stage capacity; hub states may be carried across consecutive blocks in the same stage via a lightweight per-stage EMA update; hub graph is recomputed each block from current hub features but kept sparse. Final global pooling + classification head (linear). Total depth is modest (10–14 blocks) to keep training fast.

Tunable Aspects:

- Number of hubs per stage H (e.g., 16–48)
- Hub-graph sparsity s (neighbors per hub, e.g., 2–4)
- Fingerprint dimension d_k (e.g., 32–64)
- Routing temperature τ and initial residual blend α_{init}
- Number of hub-GNN hops (1–2) and whether hub EMA state is used across blocks

Invariants:

- No pairwise $N \times N$ token attention: all global mixing uses token↔hub $O(N \cdot H)$ ops and hub-hub sparse GNN $O(H \cdot s)$.
- Routing remains soft/probabilistic (no hard top- k during forward pass) and hub corrections are residual-blended.
- Local conv path is preserved to supply high-frequency spatial detail; hubs supply global relational context.

New Hypotheses:

1. **IF** token-to-hub routing is implemented as a temperatureed soft-assignment combined with a small learnable residual

blend weight (α) in the token update **IN** mid/high network stages that perform global mixing, **THEN** the model will reduce high-confidence wrong predictions and improve top-1 accuracy without harming top-5, **BECAUSE** soft routing preserves per-token uncertainty and residual blending prevents abrupt prototype collapse while still allowing hubs to inject global corrective context. **DISPROVED IF** models with identical capacity/hubs but with hard top- k routing or zero residual blending produce equal or better top-1 and calibration metrics across repeated controlled runs.

Tags: soft-routing, calibration, prototype. *Initial confidence:* 0.65. *Connected:* hyp_14.

Reasoning: hyp_14 evidence shows hard one-step retrieval yields overconfident, lower top-1 predictions. By keeping routing soft (temperature) and blending hub-corrections via a learned α initialized small, the architecture preserves fine-grained token variance while enabling hubs to provide global information.

2. **IF** small per-scale, per-channel-group additive phase-offset parameters are added to the local depthwise separable conv sampling locations **IN** early/mid stages of the hierarchical backbone, **THEN** shift-equivariant failure modes will be reduced and shape-sensitive class accuracy will improve, **BECAUSE** tiny phase offsets break exact translation symmetry in filter responses allowing filters to specialize to asymmetric relative positions and enabling FiLM-like modulators to gate based on relative positional cues without large parameter cost. **DISPROVED IF** adding these offsets yields no measurable improvement in validation accuracy or shape-focused metrics, and analysis visualizations show no detectable asymmetric responses compared to the phase-less baseline.

Tags: phase-offset, shift-equivariance, positional-bias. *Initial confidence:* 0.55. *Connected:* hyp_26.

Reasoning: This builds on hyp_26 which found small phase-offsets can help capture asymmetric token relationships. The offsets are cheap, add only a few hundred parameters, and break perfect shift symmetry that can hinder distinguishing object-centric vs. background features.

Reasoning:

Parent Analysis: NPIN demonstrated the value of indirect, sparse local interactions plus multiscale coupling to achieve expressive global behavior without dense attention. Its emphasis on preserving local computations (via neighborhood cutoffs) combined with some mechanism for long-range coupling (coarsening) is an effective recipe for efficiency and accuracy on vision tasks.

Failure Analysis: The parent tested hyp_9 (particles + few ODE steps + coarsening) and the hypothesis was contradicted: NPIN in that configuration did not reach desired accuracy (accuracy reported 0.1891 indicating severe underperformance). Possible failure modes include (a) insufficient capacity of local dynamics with very shallow integration steps to propagate discriminative information, (b) clustering/coarsening instability or monopolization (hard clusters collapsing prototypes, related to hyp_14), and (c) latent-geometry nonidentifiability making learned positions brittle and hard to optimize under short training budgets.

Hypothesis Usage: hyp_14 (soften retrieval) is central—NPIN-like clustering or prototype hops risk hard projection and overconfidence; HHRN uses soft routing + residual blending (α) to deliberately avoid hard retrieval effects. hyp_26 (small learnable phase-offsets) is adopted for local conv sampling to break shift-symmetry cheaply and improve shape sensitivity. The uncertain hypotheses hyp_18 and hyp_24 (prototype normalization and one-way cross-band consistency) guide caution: avoid global, small-batch one-way pulls; instead use per-stage small hub banks and soft normalization so no band-starvation-like effects occur. hyp_9 (parent tested) is recorded as contradicted and motivated a departure from

particle ODEs toward hub-mediated sparse chaining while preserving the principle of local + compact global mixing.

Proposed Changes: (1) Replace particle-latent-position + ODE integration + differentiable clustering coarsening with a hub-based routing layer that is cheap ($O(N \cdot H)$), avoids heavy per-step neighborhood searches and integration loops, and reduces training overhead for very fast runs. (2) Make routing soft and residual-blended (α) to counter hard-prototype collapse (hyp_14). (3) Add small per-scale phase-offsets in local convs to break shift symmetry (hyp_26). (4) Use sparse hub-graph GNN chaining to obtain multi-hop global mixing without pairwise token interactions, enabling expressive higher-order interactions while keeping FLOPs low.

Existing Hypotheses Referenced: hyp_9, hyp_14, hyp_26, hyp_18, hyp_24.

model.py

```
import math
import torch
import torch.nn as nn
import torch.nn.functional as F

class SeparableConv2d(nn.Module):
    """Depthwise separable conv: depthwise conv
    followed by pointwise conv."""
    def __init__(self, in_ch, out_ch, kernel_size=3,
                 stride=1, padding=1, bias=False):
        super().__init__()
        # depthwise
        self.depth = nn.Conv2d(in_ch, in_ch,
                                kernel_size, stride, padding, groups=in_ch,
                                bias=bias)
        # pointwise
        self.point = nn.Conv2d(in_ch, out_ch,
                                kernel_size=1, bias=bias)

    def forward(self, x):
        x = self.depth(x)
        x = self.point(x)
        return x

class PatchEmbed(nn.Module):
    """3-layer conv stem using separable convs +
    small per-stage phase offsets injection.
    Kept lightweight for CIFAR-10."""
    def __init__(self, in_ch=3, out_ch=32,
                 phase_groups=4, spatial_size=32):
        super().__init__()
        mid = max(out_ch // 2, 12)
        self.conv1 = SeparableConv2d(in_ch, mid,
                                       kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(mid)
        self.conv2 = SeparableConv2d(mid, mid,
                                       kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(mid)
        self.conv3 = SeparableConv2d(mid, out_ch,
                                       kernel_size=3, stride=1, padding=1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_ch)
        self.act = nn.ReLU(inplace=True)
```

```

        # phase groups for small positional bias
        self.phase_groups = max(1, int(phase_groups))
        gx = torch.linspace(0, 1, steps=spatial_size).
        unsqueeze(1).repeat(1, spatial_size)
        gy = torch.linspace(0, 1, steps=spatial_size).
        unsqueeze(0).repeat(spatial_size, 1)
        # buffers shaped [1,1,H,W]
        self.register_buffer('grid_x', gx.unsqueeze
        (0).unsqueeze(0), persistent=False)
        self.register_buffer('grid_y', gy.unsqueeze
        (0).unsqueeze(0), persistent=False)

        self.phase_x = nn.Parameter(torch.zeros(self.
        phase_groups))
        self.phase_y = nn.Parameter(torch.zeros(self.
        phase_groups))
        self.spatial_size = spatial_size

    def _phase_bias(self, channels, device, dtype):
        """Produce [1,channels,H,W] small bias based
        on sin/cos of grid + per-group phases."""
        G = self.phase_groups
        # grid [1,1,H,W]
        gxs = self.grid_x.to(device=device, dtype=
        dtype)
        gys = self.grid_y.to(device=device, dtype=
        dtype)
        # expand groups
        gxs_g = gxs.expand(G, -1, -1, -1) # [G,1,H,W]
        ]
        gys_g = gys.expand(G, -1, -1, -1)
        phx = self.phase_x.view(G, 1, 1, 1).to(device
        =device, dtype=dtype)
        phy = self.phase_y.view(G, 1, 1, 1).to(device
        =device, dtype=dtype)
        basis = torch.sin(2 * math.pi * (gxg_g + phx)
        ) + torch.cos(2 * math.pi * (gys_g + phy)) # [
        G,1,H,W]

        # tile groups across channels and slice
        chunk = math.ceil(channels / G)
        basis_tiled = basis.repeat_interleave(chunk,
        dim=0)[:channels, :, :, :] # [channels,1,H,W]
        bias = basis_tiled.transpose(0, 1) # [1,
        channels, H, W]
        return 0.04 * bias # small amplitude

    def forward(self, x):
        # x: [B,3,H,W]
        b, c, h, w = x.shape
        device = x.device
        dtype = x.dtype

        bias_in = self._phase_bias(c, device, dtype)
        x = x + bias_in

        x = self.conv1(x)
        x = self.bn1(x)
        x = self.act(x)

        mid_c = x.shape[1]
        bias_mid = self._phase_bias(mid_c, device,
        dtype)
        x = x + bias_mid
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.act(x)

        bias_out = self._phase_bias(x.shape[1],
        device, dtype)
        x = x + bias_out
        x = self.conv3(x)
        x = self.bn3(x)
        x = self.act(x)
        return x

class LocalConvBlock(nn.Module):
    """Two lightweight residual depthwise separable
    conv layers + small pointwise FFN."""
    def __init__(self, channels, mlp_ratio=1.5):
        super().__init__()
        self.channels = channels
        hidden = max(int(channels * 0.5), 8)

        # residual depthwise separable convs
        self.conv1 = SeparableConv2d(channels,
        channels, kernel_size=3, stride=1, padding=1,
        bias=False)
        self.bn1 = nn.BatchNorm2d(channels)
        self.conv2 = SeparableConv2d(channels,
        channels, kernel_size=3, stride=1, padding=1,
        bias=False)
        self.bn2 = nn.BatchNorm2d(channels)
        self.act = nn.ReLU(inplace=True)
        # lightweight pointwise FFN (keeps
        representation in conv domain - efficient)
        inner = max(int(channels * mlp_ratio), hidden
        )
        self.ffn = nn.Sequential(
            nn.Conv2d(channels, inner, kernel_size=1,
            bias=True),
            nn.GELU(),
            nn.Conv2d(inner, channels, kernel_size=1,
            bias=True),
        )
        # a small layernorm-like scaling (channel-
        wise) to stabilize training
        self.gamma = nn.Parameter(torch.ones(channels
        ) * 0.1)

    def forward(self, x):
        # x: [B, C, H, W]
        residual = x
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.act(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.act(x)
        x = x + residual

        # FFN
        f = self.ffn(x)
        x = x + self.gamma.view(1, -1, 1, 1) * f
        return x

class HubRoutingBlock(nn.Module):
    """
    Token->hub routing, hub aggregation, hub
    propagation, and hub->token broadcast.
    Designed to be efficient: small hub banks,
    compact projections.
    """
    def __init__(self, channels, H=4, fingerprint_dim
    =16, value_dim=24, hub_dim=32,
        hub_s=2, routing_tau=0.07,
        alpha_init=0.12, hub_hops=1, use_hub_ema=True,
        hub_ema_beta=0.08):
        super().__init__()
        self.C = channels
        self.H = max(1, int(H))
        self.d_k = max(8, int(fingerprint_dim))
        self.v_dim = max(8, int(value_dim))
        self.hub_dim = max(8, int(hub_dim))
        self.hub_s = max(1, int(hub_s))
        self.tau = float(routing_tau)
        self.hub_hops = max(1, int(hub_hops))
        self.use_hub_ema = bool(use_hub_ema)
        self.hub_ema_beta = float(hub_ema_beta)

        # lightweight linear projections for tokens
        self.key_proj = nn.Linear(channels, self.d_k,
        bias=False)
        self.value_proj = nn.Linear(channels, self.
        v_dim, bias=False)
        # hub -> value, correction back to channels
        self.hub_to_val = nn.Linear(self.hub_dim,
        self.v_dim, bias=False)
        self.correction_proj = nn.Linear(self.v_dim,
        channels, bias=False)

        # small learned hub bank and hub routing keys
        self.hub_bank = torch.randn(self.H, self.hub_dim)
        * 0.02
        self.hub_keys = torch.randn(self.H, self.d_k) *
        0.02
        self.register_parameter('hub_bank', nn.
        Parameter(hub_bank))

```

```

        self.register_parameter('hub_keys', nn.
Parameter(hub_keys))

    # MLPs for hub message processing (kept small
)
    self.hub_msg_in = nn.Sequential(
        nn.Linear(self.v_dim, self.hub_dim),
        nn.GELU(),
        nn.Linear(self.hub_dim, self.hub_dim)
    )
    self.hub_mlp = nn.Sequential(
        nn.Linear(self.hub_dim, self.hub_dim),
        nn.GELU(),
        nn.Linear(self.hub_dim, self.hub_dim)
    )

    # blending scalar for residual correction
    self.alpha = nn.Parameter(torch.tensor(
alpha_init, dtype=torch.float32))

    # normalization for token features after
update
    self.token_norm = nn.LayerNorm(channels)

def forward(self, x):
    # x: [B, C, H, W]
    b, c, h, w = x.shape
    n = h * w
    # flatten tokens
    tokens = x.flatten(2).transpose(1, 2) # [B,
N, C]

    # project to key and value (no dtype casts)
    keys = self.key_proj(tokens) # [B,
N, d_k]
    values = self.value_proj(tokens) # [B,
N, v_dim]

    # normalize keys and hub_keys for cosine-like
similarities
    keys_norm = F.normalize(keys, dim=-1) # [B,
N, d_k]
    hub_keys_norm = F.normalize(self.hub_keys,
dim=-1) # [H, d_k]

    # compute logits: [B, N, H]
    logits = torch.matmul(keys_norm,
hub_keys_norm.t()) / (self.tau + 1e-8)
    weights = F.softmax(logits, dim=-1) #
routing weights token->hub

    # aggregate token values into hubs: [B,H,
v_dim]
    hub_agg = torch.bmm(weights.transpose(1, 2),
values)

    # initialize hub state from bank per batch
hub_state = self.hub_bank.unsqueeze(0).expand
(b, -1, -1).contiguous() # [B,H,hub_dim]
    # incorporate token messages
    hub_in = self.hub_msg_in(hub_agg) # [B,H,
hub_dim]
    new_hubs = hub_state + hub_in

    if self.use_hub_ema:
        hub_state = (1.0 - self.hub_ema_beta) *
hub_state + self.hub_ema_beta * new_hubs
    else:
        hub_state = new_hubs

    # Sparse hub->hub propagation (small H keeps
this cheap)
    hub_feats = hub_state
    s = min(self.hub_s, self.H)
    for _ in range(self.hub_hops):
        # similarity [B,H,H]
        hnorm = F.normalize(hub_feats, dim=-1)
        sim = torch.matmul(hnorm, hnorm.transpose
(1, 2))
        if s < self.H:
            # topk neighbors per hub
            topk_vals, _ = sim.topk(s, dim=-1)
            kth = topk_vals[... , -1].unsqueeze
(-1)
            mask = sim >= (kth - 1e-6)

```

```

        # safe negative fill value for fp16/
AMP
        neg_val = float(torch.finfo(sim.dtype
).min) * 0.9
        sim_masked = sim.masked_fill(~mask,
neg_val)
    else:
        sim_masked = sim
        adj = F.softmax(sim_masked, dim=-1)
        neigh = torch.bmm(adj, hub_feats)
        hub_feats = hub_feats + self.hub_mlp(
neigh)

    # map hub_feats to values and broadcast back
to tokens
    hub_msgs = self.hub_to_val(hub_feats) # [B,H,
v_dim]
    correction = torch.bmm(weights, hub_msgs) #
[B,N,v_dim]
    correction_c = self.correction_proj(
correction) # [B,N,C]

    # blend and normalize
    tokens_updated = tokens + (self.alpha.to(
correction_c.dtype) * correction_c)
    tokens_updated = self.token_norm(
tokens_updated)

    x_out = tokens_updated.transpose(1, 2).view(b,
c, h, w)
    return x_out

class ClassificationHead(nn.Module):
    def __init__(self, in_channels, num_classes=10):
        super().__init__()
        self.pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Linear(in_channels, num_classes)

    def forward(self, x):
        x = self.pool(x).view(x.size(0), -1)
        x = self.fc(x)
        return x

class Model(nn.Module):
    """Main model. Input [B,3,32,32] -> Output [B,10].
    Lightweight defaults to train quickly."""
    def __init__(self, config=None):
        super().__init__()
        cfg = config or {}

        # sensible lightweight defaults for CIFAR-10
/ single GPU training
        stages = int(cfg.get('stages', 3))
        default_channels = cfg.get('
channels_per_stage', [32, 48, 64])
        channels_per_stage = (default_channels + [
default_channels[-1]] * stages)[:stages]
        blocks_per_stage = cfg.get('blocks_per_stage',
[1] * stages)[:stages]
        hubs_per_stage = cfg.get('hubs_per_stage', [4,
6, 8])[:stages]

        # hub routing and projection dims (kept small
)
        d_k = cfg.get('fingerprint_dim', 16)
        value_dim = cfg.get('value_dim', 24)
        hub_dim = cfg.get('hub_dim', 32)
        hub_s = cfg.get('hub_sparsity', 2)
        tau = cfg.get('routing_tau', 0.07)
        alpha_init = cfg.get('alpha_init', 0.12)
        hub_hops = cfg.get('hub_hops', 1)
        use_hub_ema = cfg.get('use_hub_ema', True)
        hub_ema_beta = cfg.get('hub_ema_beta', 0.08)
        mlp_ratio = cfg.get('mlp_ratio', 1.5)
        phase_groups = cfg.get('phase_groups', 4)

        # stem
        self.stem = PatchEmbed(in_ch=3, out_ch=
channels_per_stage[0], phase_groups=
phase_groups, spatial_size=32)

        # build stages: each stage has optional
downsample then N blocks

```

```

self.stages = nn.ModuleList()
in_ch = channels_per_stage[0]
for i in range(stages):
    out_ch = channels_per_stage[i]
    blocks = []
    # downsample between stages (except first)
    if i > 0:
        down = SeparableConv2d(in_ch, out_ch,
                                kernel_size=3, stride=2, padding=1, bias=False)
        bn_down = nn.BatchNorm2d(out_ch)
        blocks.append(nn.Sequential(down,
                                     bn_down, nn.ReLU(inplace=True)))
    # Add blocks: LocalConvBlock +
    # HubRoutingBlock per block
    nblock = max(1, int(blocks_per_stage[i]))
    for _ in range(nblock):
        blocks.append(LocalConvBlock(out_ch,
                                      mlp_ratio=mlp_ratio))
        blocks.append(HubRoutingBlock(
            channels=out_ch,
            H=hubs_per_stage[i],
            fingerprint_dim=d_k,
            value_dim=value_dim,
            hub_dim=hub_dim,
            hub_s=hub_s,
            routing_tau=tau,
            alpha_init=alpha_init,
            hub_hops=hub_hops,
            use_hub_ema=use_hub_ema,
            hub_ema_beta=hub_ema_beta,
        ))
    self.stages.append(nn.Sequential(*blocks))
in_ch = out_ch

# head
self.head = ClassificationHead(in_channels=
                                channels_per_stage[-1], num_classes=10)

self.config = cfg
self._initialize_weights()

def forward(self, x):
    # Input: [B,3,32,32], Output: [B,10]
    # Avoid any dtype or memory_format casts here
    # to remain AMP / channels_last friendly
    x = self.stem(x)
    for stage in self.stages:
        x = stage(x)
    out = self.head(x)
    return out

def _initialize_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight,
                                     mode='fan_out', nonlinearity='relu')
            if getattr(m, 'bias', None) is not
            None and m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, 0, 0.01)
            if getattr(m, 'bias', None) is not
            None and m.bias is not None:
                nn.init.constant_(m.bias, 0)

```

```

'data_dir': 'cifar10',

# Training hyperparameters
'epochs': 200,
'batch_size': 1024, # large batch preferred on
                    # 48GB GPU
'learning_rate': 0.1,
'momentum': 0.9,
'weight_decay': 5e-4,
'nesterov': True,

# Learning rate schedule
'lr_schedule': 'cosine',
'warmup_epochs': 5,

# Regularization
'label_smoothing': 0.1,

# Data augmentation
'flip': True,
'translate': 4,
'cutout': 12,

# Evaluation
'tta_level': 0, # 0=none, 1=mirror, 2=mirror+
               # translate
'eval_every': 1,

# Model settings
'use_half_precision': True,
'channels_last': True,
'verbose': True,

# Model-specific hyperparameters below
# Stage/channel configuration (3 stages)
'stages': 3,
'channels_per_stage': [64, 96, 128],
'blocks_per_stage': [2, 2, 2], # number of (
                                # LocalConvBlock x 2 -> HubRoutingBlock) repeats
                                # per stage

# Hub routing specifics (per-stage lists)
# Hubs per stage: small banks (H << N)
'hubs_per_stage': [16, 24, 32],
# hub-graph sparsity (neighbors per hub)
'hub_sparsity': 3, # top-s neighbors per hub
# fingerprint dim used for routing (token keys)
'fingerprint_dim': 48,
# token value dim used when aggregating (small)
'value_dim': 64,
# hub internal dimension
'hub_dim': 64,
# routing temperature tau
'routing_tau': 0.07,
# initial residual blending alpha
'alpha_init': 0.15,
# number of hub-GNN hops (1 or 2)
'hub_hops': 1,
# whether to use per-stage EMA within forward
# pass (kept local to forward)
'use_hub_ema': True,
'hub_ema_beta': 0.1,

# Other architectural niceties
'drop_path_rate': 0.0,
'mlp_ratio': 2.0,

# For reproducibility / safety
'seed': 42,
}

```

config.py

```

config = {
    # Experiment info
    'experiment_name': 'HHRN_cifar10',
    'notes': 'Hierarchical Hub Routing Network: soft
             token->hub routing + sparse hub GNN, depthwise
             separable conv stem, per-stage phase offsets.',

    # Data settings

```

D.3 Band-Aware Wavelet Token Mixer (91.22%)

Idea Agent Output: Band-Aware Wavelet Token Mixer with Gated Cross-Band Residuals (BA-WTM+)

Description. An evolution of MSL-WTM that retains the learned multi-scale lifting-based token analysis/synthesis but adds two targeted mechanisms: (1) per-band usage-normalized FiLM controllers (Band-Aware FiLM Norm) that produce multiplicative/additive per-channel gains per subband while normalizing their statistics and controller gradients per-band to avoid low-pass starvation; and (2) lightweight gated cross-band residual connectors that let low-pass summaries multiplicatively gate band-pass processors per token (low→high gating), preventing isolated high-frequency textures from dominating decisions. The rest of the backbone (patch embedding, hierarchical patch-merging, shallow per-band depthwise convs / token MLPs, inverse lifting synthesis, and global low-pass broadcast context) is preserved to enable causal attribution to these two additions. The design enforces linear-time cross-token ops (depthwise 1D convs, downsampling, pooling, small MLPs) and keeps peak memory low (no N^2 expansions).

Intuition. The parent MSL-WTM demonstrated that explicit multi-scale decomposition plus per-band modulation reduces texture bias relative to single-scale circulant mixers. Remaining failure modes appear to come from band-imbalance (high-frequency tokens overwhelming low-pass signals) and from isolated high-frequency activations that spuriously trigger downstream responses. Per-band usage-normalization (in the FiLM controllers) counters statistical imbalance at the controller/update level so low-pass channels are neither starved during forward routing nor drowned in controller gradient updates. Gated cross-band residuals give band-pass processors per-token access to coherent low-pass priors that can suppress false positives from isolated high-frequency cues without adding pairwise attention. Together these changes aim to fur-

ther reduce texture-driven errors and improve fine-grained class accuracy while preserving efficiency.

Novelty. Combines learned lifting-style multi-scale token transforms with band-aware controller normalization and explicit low→high gated residuals. This is structurally distinct from (a) global fixed transforms (FNet/FFT), (b) dense token MLP mixing (MLP-Mixer/gMLP), (c) windowed attention/hub routing (Swin/BigBird), and (d) axial or linear attention factorizations. The key novelty is low-cost, per-band controller normalization plus multiplicative cross-band gating applied in the wavelet subband domain—i.e., manipulating the transform-domain controllers and residuals rather than adding pairwise attention or prototypes.

Target Improvement: both (accuracy and efficiency).

Architecture Specification:

Core Ideas: 1) Keep MSL-WTM’s learned lifting-based multi-scale token analysis/synthesis for explicit scale separation. 2) Replace shared FiLM controllers with per-band usage-normalized FiLM controllers whose normalization state and gradient scaling are tracked per-band to prevent band starvation. 3) Insert gated cross-band residual connectors: compute a lightweight low-pass summary per token and apply multiplicative gates to band-pass processors (residual connection) before their local nonlinear blocks. 4) Preserve linear-time operations (depthwise 1D convs, small MLPs, pooling/downsampling) and hierarchical patch-merging to fit the efficiency constraints.

Core Blocks:

- **PatchEmbed:** 3×3 conv (stride=1) followed by 2×2 strided conv to form patch tokens, output token dims (N, C).
- **WaveletAnalysis** (Lifting-style): Depthwise 1D conv across token index ($k=3$) + linear predictor to split tokens into Low and multiple Band subbands; optionally downsample for subsequent scales (2 scales recommended for CIFAR).
- **Band-Aware FiLM Controller:** Per-

band small MLP (input = token embedding concat global low-pass summary) $\rightarrow$ outputs γ, β per channel-group. Each controller maintains per-band normalization stats (running mean/var) and per-band controller gradient scaling/EMA for stable updates.

- **Gated Cross-Band Residual:** Compute per-token low-pass summary (per-channel or grouped) $\rightarrow$ tiny gating MLP (hidden dim = $C/16$) $\rightarrow$ sigmoid gate $g(\text{token}, \text{band})$ applied multiplicatively to band-pass residual path: $\text{out} = \text{band_proc}(\text{in}) \cdot g + \text{residual}(\text{in})$. Low-cost and applied per-token per-band.
- **Subband Processor:** Per-band depthwise separable 1D conv ($k=3$) + channel MLP (1×1 conv) + residual, kept shallow (1–2 layers) for FLOP budget.
- **WaveletSynthesis** (Inverse lifting): Depthwise 1D conv + learned interpolation to reconstruct full-resolution token sequence.
- **Stage Head:** LayerNorm $\rightarrow$ tokenwise MLP (channel mixing) $\rightarrow$ residual. Patch Merging: adjacent token concatenation $\rightarrow$ linear proj (reduces N , increases C).
- **Classifier:** Global average pool over tokens $\rightarrow$ small MLP $\rightarrow$ softmax.

Network Structure: 3-stage hierarchical backbone tuned for CIFAR-10. Stage 1 (tokens $\sim 32 \times 32$ patches $\rightarrow N \sim 1024$): channels $C=64$, apply 2 WTM-blocks (each block = Analysis(2-scale) $\rightarrow$ Band-Aware FiLM $\rightarrow$ GatedCrossBand $\rightarrow$ SubbandProc $\rightarrow$ Synthesis $\rightarrow$ StageHead). Patch merge $\rightarrow$ Stage 2: $C=128$, N halved, 2 WTM-blocks. Patch merge $\rightarrow$ Stage 3: $C=192$, N halved, 2 WTM-blocks. Final global avg pool $\rightarrow$ classifier. Per-block parameter controls: per-band MLP hidden dims = $C/8$; gating MLP hidden dim = $C/16$. With these choices the model fits under 10M params and $<2\text{B}$ FLOPs for CIFAR input (32×32), and trains efficiently (no N^2 ops).

Tunable Aspects:

- Number of scales (1–3)
- Per-band controller hidden dim and normalization momentum

(controller_ema)

- Use gated residuals (on/off) and gating hidden dim
- Number of WTM repeats per stage
- Channel widths per stage

Invariants:

- All token-mixing across tokens is linear-time (depthwise convs, downsampling, pooling); no N^2 attention or pairwise expansions.
- Global context is injected only via compact low-pass summaries (broadcast), not via hub-routing/prototypes with aggressive online updates.
- Per-band computation state (normalization and controller EMA) is maintained per-band to avoid cross-band statistical interference.

New Hypotheses:

1. **IF** per-band usage-normalized FiLM controllers replace naive/shared FiLM controllers **IN** multi-scale wavelet-style token-mixing stages for CIFAR-scale hierarchical backbones, **THEN** low-pass (shape) channels will retain sufficient forward activation mass and controller responsiveness so that shape-sensitive per-class accuracy improves while avoiding high-frequency prototype monopolization, **BECAUSE** normalizing controller statistics and gradient scales per band prevents high-frequency token abundance from dominating controller outputs and updates, maintaining balanced per-band modulation and thereby preserving low-pass evidence for downstream decisions. **DISPROVED IF** per-band FiLM normalization yields no measurable improvement in shape-reliant per-class accuracy ($>1\%$ abs) nor measurable rebalancing of per-band activation/assignment statistics compared to shared FiLM controllers.

Tags: band-normalization, FiLM, multi-scale. *Initial confidence:* 0.6.

Connected: hyp_18, hyp_13.

Reasoning: High-frequency bands produce many more active tokens; if controllers/updates are pooled globally they will be dominated by those signals

(hyp_18). By computing separate normalization statistics, gradient scaling, and optional per-band controller EMA momentum, the controller parameters and outputs will be proportionally responsive to signals present in each band, preventing starvation of low-pass modulators. This reuses principles from hyp_13 (stabilize online updates) and hyp_10 (avoid large intermediate tensors).

2. **IF** gated cross-band residual connectors (low-pass $\rightarrow$ band-pass multiplicative gates with tiny gating MLPs) are added to band-pass processors **IN** WTM stages, **THEN** the model will reduce texture-triggered false positives and improve discrimination for visually-similar classes, **BECAUSE** low-pass summaries provide spatially-coherent priors that multiplicatively suppress or amplify band-pass responses per-token so that isolated high-frequency textures cannot unduly dominate final features. **DISPROVED IF** adding these gated connectors produces no measurable reduction in texture-driven confusion mass nor improvement in shape-sensitive per-class recall (no $>1\%$ abs change) and Grad-CAMs remain equally texture-focused.
Tags: cross-band-gating, residuals, texture-robustness. *Initial confidence:* 0.55. *Connected:* hyp_28.

Reasoning: Low-pass channels encode global shape/region context; gating band-pass processors with those low-pass-derived signals biases band-pass nonlinearities to respect coherent shape boundaries, preventing small high-frequency islands from triggering strong downstream activations. This is a low-cost, local alternative to adding pairwise attention for context-aware suppression, inspired by hyp_28.

Reasoning:

Parent Analysis: The parent MSL-WTM successfully replaced single-scale circulant mixers with a learned lifting-style multi-scale transform. This gave explicit low/high-frequency decomposition,

enabling separation of shape vs. texture cues, and allowed efficient global receptive fields via downsampling/upsampling while keeping operations linear-time (depthwise convs, small MLPs). Hierarchical patch-merging and local channel mixing preserved strong local feature extraction and training stability.

Failure Analysis: Residual failure modes remain consistent with band imbalance and texture dominance: high-frequency tokens produce abundant activations and can dominate controller signals and downstream processing, starving low-pass channels (hyp_18). Isolated high-frequency textures still trigger false positives in some classes. There is a risk that any associative/prototype mechanisms (if added) could suffer attractor drift from overly-aggressive online updates (hyp_13), so we avoid such aggressive online updates and instead prefer per-band normalization and conservative controller EMA.

Hypothesis Usage: (1) hyp_13 (worked): warns that aggressive online updates cause prototypes to drift toward dominant textures; we therefore avoid online prototype banks and, if any persistent controller state is kept, use conservative EMA/usage-weighting per-band. (2) hyp_10 (worked): encourages low-rank or low-cost projections to avoid memory blowups; we preserve linear-time depthwise convs and small MLPs and avoid any dense token $\leftrightarrow$ hub expansions. (3) hyp_18 (uncertain): suggests global usage-normalization can starve low-pass channels; we implement per-band usage-normalization in FiLM controllers to directly test/mitigate this. (4) hyp_28 (uncertain): proposes gated cross-band connectors; we add explicit gated low $\rightarrow$ high residuals to test whether providing low-pass priors to band-pass processors reduces texture-driven errors.

Proposed Changes: (A) Replace shared FiLM controllers with per-band usage-normalized FiLM controllers: implement running statistics and gradient scaling per band (and optionally per-stage) so controllers are not overwhelmed by high-

frequency token abundance—addresses hyp_18 and reduces risk highlighted by hyp_13. (B) Insert gated cross-band residual connectors for band-pass processors that multiplicatively gate band responses with a low-pass-derived gate—this reduces isolated texture false positives (tests hyp_28). Keep all other aspects of MSL-WTM (analysis/synthesis lifting, subband processing, hierarchical stages) unchanged so that any improvements can be attributed to these additions.

Existing Hypotheses Referenced: hyp_13, hyp_10, hyp_18, hyp_28.

model.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class ConvBNGELU(nn.Module):
    def __init__(self, in_ch, out_ch, kernel=3,
                 stride=1, padding=1, groups=1):
        super().__init__()
        self.conv = nn.Conv2d(in_ch, out_ch, kernel,
                               stride=stride, padding=padding, groups=groups,
                               bias=False)
        # Use GroupNorm(1) for small-batch stability
        # and GPU efficiency on small images
        self.norm = nn.GroupNorm(1, out_ch, eps=1e-5)
        self.act = nn.GELU()

    def forward(self, x):
        return self.act(self.norm(self.conv(x)))

class PatchEmbed(nn.Module):
    """Patch embedding: a light conv stem preserving
    spatial resolution."""
    def __init__(self, in_ch=3, out_ch=32):
        super().__init__()
        # single conv stem with small width
        self.conv1 = ConvBNGELU(in_ch, out_ch, kernel
                                =3, stride=1, padding=1)

    def forward(self, x):
        # x: [B,3,32,32] -> [B,C,32,32]
        return self.conv1(x)

class BandFiLMController(nn.Module):
    """
    Efficient per-band controller implemented via
    pointwise convs (1x1):
    - Accepts band features and low-pass features (
      both spatial),
    - Projects low to band channels via 1x1 conv,
    - Concatenates along channel dim and runs a small
      pointwise MLP (1x1 convs),
    - Normalizes controller outputs with GroupNorm(1)
      and produces gamma/beta.
    This avoids heavy per-token reshapes and large
    linear layers.
    """
    def __init__(self, band_channels, low_channels,
                 hidden_ratio=0.25):
        super().__init__()
        self.band_ch = int(band_channels)
        self.low_ch = int(low_channels)
        # Project low to band channels with 1x1 conv
```

```
        self.low_proj = nn.Conv2d(self.low_ch, self.
band_ch, kernel_size=1, bias=False)
        # Small MLP implemented as 1x1 convs
        hid = max(4, int(self.band_ch * hidden_ratio)
        )
        self.mlp = nn.Sequential(
            nn.Conv2d(self.band_ch * 2, hid,
kernel_size=1, bias=False),
            nn.GELU(),
            nn.Conv2d(hid, self.band_ch * 2,
kernel_size=1, bias=True) # outputs gamma+beta
        )
        # Normalize controller outputs per-token
        across channels
        self.norm = nn.GroupNorm(1, self.band_ch * 2,
eps=1e-5)
        # small learnable gating scale
        self.out_scale = nn.Parameter(torch.ones(self.
band_ch * 2))
```

```
def forward(self, band_x, low_x):
    # band_x: [B, Cb, H, W], low_x: [B, Cl, H, W]
    # Project low to band channels (spatial
    preserved)
    low_p = self.low_proj(low_x) # [B, Cb, H, W]
    # concat along channel
    x = torch.cat([band_x, low_p], dim=1) # [B,
2*Cb, H, W]
    out = self.mlp(x) # [B, 2*Cb, H, W]
    out = self.norm(out) * self.out_scale.view(1,
-1, 1, 1)
    # split
    gamma, beta = out.split(self.band_ch, dim=1)
    return gamma, beta
```

```
class GatedCrossBandResidual(nn.Module):
    """
    Efficient low->band gating implemented with 1x1
    conv:
    - Produces per-channel gates for the band from
    low features (spatial preserved).
    """
    def __init__(self, low_channels, band_channels,
                 gating_hidden_ratio=0.25):
        super().__init__()
        hid = max(4, int(low_channels *
gating_hidden_ratio))
        # small MLP as 1x1 convs to produce gates per
        band channel
        self.net = nn.Sequential(
            nn.Conv2d(low_channels, hid, kernel_size
=1, bias=False),
            nn.GELU(),
            nn.Conv2d(hid, band_channels, kernel_size
=1, bias=True),
            nn.Sigmoid()
        )

    def forward(self, low_x, band_x):
        # low_x: [B, Cl, H, W], band_x: [B, Cb, H, W]
        gates = self.net(low_x) # [B, Cb, H, W]
        return band_x * gates
```

```
class SubbandProcessor(nn.Module):
    """
    Per-band shallow processor: depthwise separable
    convs + small channel MLP + residual.
    GroupNorm used for efficiency and stability.
    """
    def __init__(self, channels, hidden_ratio=0.5):
        super().__init__()
        self.channels = channels
        # depthwise conv
        self.dw = nn.Conv2d(channels, channels,
kernel_size=3, padding=1, groups=channels, bias
=False)
        self.dw_norm = nn.GroupNorm(1, channels, eps
=1e-5)
        # pointwise expansion and projection (channel
        MLP)
        hidden = max(4, int(channels * hidden_ratio))
        self.pw1 = nn.Conv2d(channels, hidden,
kernel_size=1, bias=False)
```

```

        self.pw1_norm = nn.GroupNorm(1, hidden, eps=1e-5)
        self.act = nn.GELU()
        self.pw2 = nn.Conv2d(hidden, channels, kernel_size=1, bias=True)
        self.pw2_norm = nn.GroupNorm(1, channels, eps=1e-5)

    def forward(self, x):
        identity = x
        x = self.act(self.dw_norm(self.dw(x)))
        x = self.act(self.pw1_norm(self.pw1(x)))
        x = self.pw2_norm(self.pw2(x))
        return x + identity

class WaveletBlock(nn.Module):
    """
    Efficient WTM block:
    - Analysis (depthwise conv + pointwise)
    - Band-FiLM via pointwise conv controller producing spatial gamma/beta
    - Gated cross-band residual (from low to band) using 1x1 convs
    - Subband processors
    - Synthesis (concat + 1x1 conv + depthwise smooth)
    - Lightweight stage head with GroupNorm and 1x1 token MLP residual
    """
    def __init__(self, channels, num_bands=2, controller_hidden_ratio=0.25, use_gated_residuais=True, gating_hidden_ratio=0.25):
        super().__init__()
        assert num_bands == 2, "Efficient implementation only implements low+band"
        self.channels = channels
        # Analysis
        self.analysis_dw = nn.Conv2d(channels, channels, kernel_size=3, padding=1, groups=channels, bias=False)
        self.analysis_pw = nn.Conv2d(channels, channels, kernel_size=1, bias=False)
        self.analysis_norm = nn.GroupNorm(1, channels, eps=1e-5)

        # split sizes
        self.low_ch = channels // 2
        self.band_ch = channels - self.low_ch

        # controllers and gating
        self.band_controller = BandFiLMController(self.band_ch, self.low_ch, hidden_ratio=controller_hidden_ratio)
        self.use_gated = use_gated_residuais
        if self.use_gated:
            self.gated = GatedCrossBandResidual(low_channels=self.low_ch, band_channels=self.band_ch, gating_hidden_ratio=gating_hidden_ratio)

        # Subband processors
        self.band_proc = SubbandProcessor(self.band_ch, hidden_ratio=0.5)
        self.low_proc = SubbandProcessor(self.low_ch, hidden_ratio=0.5)

        # Synthesis
        self.synth_pw = nn.Conv2d(channels, channels, kernel_size=1, bias=False)
        self.synth_dw = nn.Conv2d(channels, channels, kernel_size=3, padding=1, groups=channels, bias=False)
        self.synth_norm = nn.GroupNorm(1, channels, eps=1e-5)

        # Stage head (light)
        self.head_norm = nn.GroupNorm(1, channels, eps=1e-5)
        self.head_fc1 = nn.Conv2d(channels, channels * 2, kernel_size=1, bias=False)

```

```

        self.head_fc2 = nn.Conv2d(channels * 2, channels, kernel_size=1, bias=True)
        self.head_act = nn.GELU()

    def forward(self, x):
        # x: [B, C, H, W]
        a = self.analysis_dw(x)
        a = self.analysis_pw(a)
        a = self.analysis_norm(a)
        # split into low and band
        low, band = a.split([self.low_ch, self.band_ch], dim=1) # [B,C1,H,W], [B,Cb,H,W]

        # Band-FiLM modulation
        gamma, beta = self.band_controller(band, low)
        # both [B, Cb, H, W]
        band = band * (1.0 + gamma) + beta

        # Gated cross-band residual (multiplicative gating)
        if self.use_gated:
            band = band + self.gated(low, band)

        # Subband processing
        band = self.band_proc(band)
        low = self.low_proc(low)

        # Synthesis
        merged = torch.cat([low, band], dim=1) # [B, C, H, W]
        merged = self.synth_pw(merged)
        merged = self.synth_norm(self.synth_dw(merged))

        # Stage head
        out = self.head_norm(merged)
        out = self.head_act(self.head_fc1(out))
        out = self.head_fc2(out)
        out = out + merged # residual
        return out

class PatchMerging(nn.Module):
    """Reduce spatial by 2x (H/2,W/2) and increase channels using stride-2 conv."""
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.conv = nn.Conv2d(in_ch, out_ch, kernel_size=2, stride=2, bias=False)
        self.norm = nn.GroupNorm(1, out_ch, eps=1e-5)
        self.act = nn.GELU()

    def forward(self, x):
        return self.act(self.norm(self.conv(x)))

class ClassifierHead(nn.Module):
    def __init__(self, in_ch, num_classes):
        super().__init__()
        self.pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Linear(in_ch, num_classes)

    def forward(self, x):
        # x: [B, C, H, W]
        x = self.pool(x).flatten(1)
        x = self.fc(x)
        return x

class Model(nn.Module):
    def __init__(self, config=None):
        super().__init__()
        cfg = config or {}
        # Defaults chosen to be lightweight and efficient for CIFAR-10 while preserving the BA-WTM idea.
        self.num_classes = cfg.get('num_classes', 10)
        # widths and depths define a small, efficient encoder. Users can override via config.
        widths = cfg.get('widths', [32, 64, 128])
        depths = cfg.get('depths', [1, 1, 1])
        assert len(widths) == len(depths), "widths and depths must match"

        self.num_bands = cfg.get('num_bands', 2)

```

```

        self.controller_hidden_ratio = cfg.get('
controller_hidden_ratio', 0.25)
        self.use_gated_residuals = cfg.get('
use_gated_residuals', True)
        self.gating_hidden_ratio = cfg.get('
gating_hidden_ratio', 0.25)

        # Patch embed
        self.patch_embed = PatchEmbed(in_ch=3, out_ch
=widths[0])

        # Build stages
        self.stages = nn.ModuleList()
        self.patch_merges = nn.ModuleList()
        in_ch = widths[0]
        for stage_idx, (w, d) in enumerate(zip(widths,
depths)):
            blocks = []
            # if needed, a 1x1 projection at start of
stage
            if stage_idx == 0:
                if in_ch != w:
                    blocks.append(nn.Conv2d(in_ch, w,
kernel_size=1, bias=False))
                    in_ch = w
            else:
                # in_ch already equals current width
(from previous patch merging)
                pass
            for _ in range(d):
                blocks.append(WaveletBlock(w,
num_bands=
self.num_bands,

controller_hidden_ratio=self.
controller_hidden_ratio,

use_gated_residuals=self.use_gated_residuals,

gating_hidden_ratio=self.gating_hidden_ratio))
            self.stages.append(nn.Sequential(*blocks)
)
            if stage_idx < len(widths) - 1:
                out_ch = widths[stage_idx + 1]
                self.patch_merges.append(PatchMerging
(in_ch=w, out_ch=out_ch))
                in_ch = out_ch

        # classifier head
        self.head = ClassifierHead(in_ch, self.
num_classes)

        self._initialize_weights()

    def forward(self, x):
        # Input: [B, 3, 32, 32]
        # No memory-format or dtype conversions here
(AMP / channels_last preserved by caller).
        x = self.patch_embed(x) # [B, C, 32, 32]
        for idx, stage in enumerate(self.stages):
            x = stage(x)
            if idx < len(self.patch_merges):
                x = self.patch_merges[idx](x) #
downsample spatial dims by 2
            x = self.head(x) # [B, num_classes]
        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight,
mode='fan_out', nonlinearity='relu')
                if getattr(m, 'bias', None) is not
None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.GroupNorm):
                # initialize scale and bias
                if getattr(m, 'weight', None) is not
None:
                    nn.init.constant_(m.weight, 1)
                if getattr(m, 'bias', None) is not
None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)

```

```

        if getattr(m, 'bias', None) is not
None:
            nn.init.constant_(m.bias, 0)

```

config.py

```

config = {
    # Experiment info
    'experiment_name': 'BA-WTM+',
    'notes': 'Band-aware wavelet token mixer with per
-band FiLM normalization and low->high gated
residuals for CIFAR-10',

    # Data settings
    'data_dir': 'cifar10',

    # Training hyperparameters
    'epochs': 200,
    'batch_size': 1024,
    'learning_rate': 0.03,
    'momentum': 0.9,
    'weight_decay': 5e-4,
    'nesterov': True,

    # Learning rate schedule
    'lr_schedule': 'cosine',
    'warmup_epochs': 5,

    # Regularization
    'label_smoothing': 0.1,

    # Data augmentation
    'flip': True,
    'translate': 4,
    'cutout': 12,

    # Evaluation
    'tta_level': 0,
    'eval_every': 1,

    # Model settings
    'use_half_precision': True,
    'channels_last': True,
    'verbose': True,

    # Model-specific hyperparameters
    'stages': 3,
    'depths': [2, 2, 2], # number of WTM
        blocks per stage
    'widths': [64, 128, 192], # channels per
stage
    'num_classes': 10,
    'num_bands': 2, # low + band
    'controller_hidden_ratio': 1/8, # per-band
controller MLP hidden dim = C_band * ratio
    'controller_ema': 0.05, # running stats
momentum for per-band controllers
    'use_gated_residuals': True,
    'gating_hidden_ratio': 1/16, # gating MLP
hidden dim = C * ratio
    'drop_path_rate': 0.0,
    'eps': 1e-5,
    # Operational constraints
    'input_size': (3, 32, 32),
}

```

E Prompts

E.1 Idea Agent Prompts

Idea Agent (Root Node) — System Prompt

You are an expert Deep Learning Research Scientist specialized in designing novel neural architectures. You will be given a research direction and must propose EXACTLY {count} DISTINCT architectures that could each realistically be published at NeurIPS/ICML/ICLR.

Guidelines. Design {count} fundamentally novel neural network architectures that each introduce at least one new structural principle. The architectures will be evaluated on CIFAR-10 but should be generalizable to other computer vision tasks. Each architecture must be hypothesis-driven—propose testable hypotheses about WHY it should work.

Key Rules:

- Analyze curated reference papers to extract architectural principles; all proposals must be structurally distinct from these works.
- Follow the research direction exactly. Innovate within constraints.
- Each architecture must introduce at least one fundamentally new structural or computational principle, grounded in inductive bias, geometry, optimization, or representation theory.
- Prioritize both accuracy and efficiency (fast training, fewer parameters/FLOPs).
- Each idea must propose DIFFERENT hypotheses. No two architectures may share the same core mechanism.
- Do not propose minor variations of known models or hyperparameter tuning suggestions.

Hypothesis Format: IF [architectural choice] IN [scope], THEN [predicted effect], BECAUSE [mechanism]. DISPROVED IF [falsification criterion].

At most 2 new hypotheses per architecture. Each must be mechanistic, scoped, and falsifiable.

Curated Reference Papers:

{curated_related_papers} [injected at runtime]

Output Format (JSON):

- "ideas": array of exactly {count} objects, each containing:
- "title", "description" (2–3 paragraphs), "intuition" (1–2 paragraphs), "novelty", "target_improvement": accuracy | efficiency | both
- "related_papers":
[{"title", "year", "url", "key_idea", "why_relevant", "difference_from_proposal"}]
- "new_hypotheses": [{"text", "scope", "prediction", "falsification_criteria", "tags", "initial_confidence", "reasoning"}]
- "architecture_spec":
{"core_ideas", "core_blocks", "network_structure", "tunable_aspects", "invariants"}

Idea Agent (Root Node) — User Prompt

Design exactly {count} fundamentally DISTINCT neural network architectures following the research direction below. Each architecture must be novel, achieve high accuracy, train fast, and propose DIFFERENT hypotheses. They should generalize to other computer vision datasets.

Performance Targets (for ALL ideas):

- 95% accuracy on CIFAR-10, better efficiency than ResNet-50
- Training around 1 minute on single GPU

RESEARCH DIRECTION:
{research_direction}

Idea Agent (Evolution Mode) — System Prompt

You are a Brainstorming Agent specialized in evolving neural architectures. You analyze parent architectures and hypothesis evidence to propose novel, efficient, high-performing architectures that advance the research direction.

Key Rules:

- Analyze the parent architecture's

strengths and weaknesses based on feedback, then propose a child that addresses identified issues and tests promising hypotheses.

- *Hypotheses that worked*: build on and extend these patterns.
- *Hypotheses that failed*: avoid these patterns or try alternatives.
- *Uncertain hypotheses*: design targeted changes to gather clearer evidence.
- Make targeted changes (1–2 aspects), not a full redesign, to enable causal attribution while preserving successful components.
- New hypotheses must be general, mechanistic, novel, and testable. Reference existing hypothesis IDs in `connected_hypotheses`.

Output Format (JSON):

- "reasoning": { "parent_analysis", "failure_analysis", "hypothesis_usage", "proposed_changes" }
- "title", "description", "intuition", "novelty", "target_improvement"
- "existing_hypotheses": list of referenced hypothesis IDs
- "new_hypotheses": [{ "text", "tags", "initial_confidence", "reasoning", "connected_hypotheses" }]
- "architecture_spec": { "core_ideas", "core_blocks", "network_structure", "tunable_aspects", "invariants" }

Idea Agent (Evolution Mode) — User Prompt

Research Direction:
{research_direction}
Performance Targets:
• 95% accuracy on CIFAR-10, better efficiency than ResNet-50
• Training around 1 minute on single GPU
Parent architecture:
{parent_architecture}
Parent Performance:
{parent_performance}

Feedback from Parent architecture:
{feedback_summary}
Hypothesis Memory:
{hypothesis_memory}

E.2 Coding Agent Prompts

Coding Agent (Initial Generation) — System Prompt

You are a Coding Agent specialized in implementing novel neural network architectures in PyTorch. You translate architecture proposals into clean, efficient, runnable code.

Key Rules:

- Implement the research idea as two files: `config.py` (hyperparameter dictionary) and `model.py` (PyTorch `nn.Module`).
- The Model class must accept an optional config parameter, take input `[B, 3, 32, 32]`, and output `[B, 10]`.
- Ensure AMP and `channels_last` compatibility; no per-forward dtype casts or memory-format conversions.
- Include Kaiming initialization for Conv2d, constant initialization for BatchNorm2d.
- Total parameters MUST be <10M (target <5M). Avoid materializing large intermediate tensors (e.g., pairwise `[B, N, M, D]` expansions).
- Training must complete within 15 minutes on a single GPU.

Performance Rules:

- `forward()`: no loops over tokens/groups/neighbors; no `where+scatter`; no `split+cat` for grouping; no batch-expanded weights.
- Prefer Conv2d (1×1 /grouped/depthwise), BN2d/GroupNorm, pooling, einsum/matmul; minimize permute/contiguous.

Output: Two complete Python files: `config.py` with full hyperparameter dictionary (epochs, batch_size, learning_rate, lr_schedule, regularization, augmentation, model-specific parameters) and `model.py` with complete `Model(nn.Module)` implementation.

Coding Agent (Initial Generation) — User Prompt

Research Idea: {research_idea}
Architecture Summary: {architecture_summary}

Coding Agent (Error Recovery) — System Prompt

You are a Coding Agent specialized in fixing neural network implementation errors in PyTorch. You diagnose failures from error logs and produce corrected, runnable code.

Key Rules:

- Identify the failing code from the error log or stack trace, then rewrite the buggy functions with corrected logic.
- Make real, substantive code changes—do not copy the previous implementation unchanged.
- Preserve the core research innovation, overall architecture structure, and the Model class interface.
- If the error is CUDA OOM, MUST reduce model size (cut hidden dimensions, remove layers, replace memory-heavy operations). Do NOT just reduce batch size.
- Output only model.py unless the error is caused by a config issue.

Output: Complete, corrected model.py that runs without modification.

Coding Agent (Error Recovery) — User Prompt

Previous Model Implementation (Iteration {iteration}): {previous_code}
Error Feedback: {feedback}

Coding Agent (Hyperparameter Refinement) — System Prompt

You are a Coding Agent specialized in hyperparameter optimization for neural network training. You refine configuration files to improve accuracy while maintaining training efficiency.

Key Rules:

- Analyze training logs to diagnose underfitting, overfitting, or convergence issues,

then make targeted adjustments.

- Only modify config.py. The model architecture must not be changed.
- Review previous refinement attempts to avoid repeating failed approaches.
- Make incremental changes—avoid drastic hyperparameter shifts.
- GPU has 48GB; use batch size 1024 to maximize throughput.

Output: A brief change summary followed by the complete config.py file.

Coding Agent (Hyperparameter Refinement) — User Prompt

Current Configuration: {config_code}
Best Accuracy So Far: {best_accuracy}
Previous Refinement Attempts: {refinement_history}
Training Logs: {stdout_log}

E.3 Redundancy Filtering Prompt

Redundancy Filter (LLM as a Judge) — System Prompt

You are an expert neural architecture reviewer. Your ONLY task is to decide whether a candidate architecture represents a NEW concept or is a DUPLICATE (regeneration) of a concept already in the archive.

Context: These architectures come from an evolutionary search where children naturally “build upon” parents. A child that introduces a genuinely new mechanism IS novel—we are detecting REDUNDANT regenerations of the SAME concept.

Mark as DUPLICATE (novel=false) when:

- The candidate’s core innovation claim is the same as an archive entry.
- Same structural pattern AND same design principle, differing only in hyperparameters or minor implementation details.
- Different terminology masks conceptual identity (e.g., “adaptive residual” vs “enhanced residual” may describe the same mechanism).
- Different implementations of the SAME concept are still duplicates. Focus on

WHAT the mechanism achieves, not HOW it is implemented.

Mark as NOVEL (novel=true) when:

- The candidate introduces a mechanism or structural principle NOT in any archive entry.
- The candidate combines ideas in a way not seen in the archive.
- The candidate represents a meaningfully different design philosophy.

Structured Reasoning: Before deciding, identify: (1) shared_principles with the closest archive entry, and (2) new_contribution—a genuinely new HIGH-LEVEL design principle (or “none”).

Output Format (JSON):

- "shared_principles": what candidate shares with closest entry
- "new_contribution": genuinely new principle, or "none"
- "novel": true or false
- "reasoning": 1–3 sentence explanation
- "most_similar_to": node_id or null

ficiency patterns).

- New hypotheses must satisfy all 7 quality dimensions [see Section C.5].

Output Format (JSON):

- "reasoning": why the experiment succeeded or failed based on metrics
- "hypothesis_updates": [{"hyp_id", "evidence_type", "strength", "reasoning"}]
- "new_hypotheses": [{"text", "scope", "prediction", "falsification_criteria", "tags", "initial_confidence", "reasoning"}]

Quantitative Agent — User Prompt

Research	Direction:
{research_direction}	
Proposed	Architecture:
{architecture_summary}	
Experiment Metrics and Logs:	
{experiment_metrics_and_logs}	
Hypotheses Being Tested:	
{tested_hypothesis_section}	

E.4 Feedback Agent Prompts

Quantitative Feedback Agent — System Prompt

You are a Quantitative Analysis Agent specialized in discovering causal relationships between neural architecture design and performance. You will be given a research direction, a proposed architecture, experiment metrics and logs, and optionally a hypothesis being tested.

Key Rules:

- Analyze quantitative metrics (accuracy, loss curves, convergence, training dynamics) to assess whether the experiment succeeded or failed relative to the research goal.
- Evaluate evidence as supports, contradicts, or neutral; strength ranges from 0.0 (negligible) to 1.0 (large, clear effect).
- PREFER updating existing hypotheses over creating new ones. Focus on observations unique to your perspective (quantitative metrics, convergence behavior, ef-

Qualitative Feedback Agent (VLM) — System Prompt

You are an Error Analysis Agent specialized in discovering causal relationships between neural architecture design and classification errors. You analyze misclassified images with attention heatmaps via VLM to identify behavioral patterns and evaluate hypotheses.

Heatmap Methods:

- **GradCAM:** class-specific spatial attention; separate heatmaps for predicted (wrong) and true (correct) class.
- **Attention Rollout:** class-agnostic spatial attention aggregated across transformer layers.
- **Input Gradient Saliency:** class-specific input sensitivity; warm regions indicate pixels most influencing the class score.

Analysis Framework: (1) Compare attention between predicted and true class heatmaps; (2) identify texture vs. shape bias; (3) assess whether attention focuses on dis-

criminative features or background; (4) connect attention failures to specific architectural choices.

Key Rules:

- Evaluate evidence as supports, contradicts, or neutral; strength $\in [0, 1]$.
- PREFER updating existing hypotheses. Propose at most 3 new hypotheses.
- Use heatmap evidence to ground reasoning (e.g., “heatmaps show diffuse attention on background textures”).

Output Format (JSON):

- "reasoning",
"actionable_feedback",
"overall_summary",
"attention_analysis"
- "hypothesis_updates": [{"hyp_id",
"evidence_type", "strength",
"reasoning"}]
- "new_hypotheses": [{"text",
"tags", "initial_confidence",
"reasoning"}]

Qualitative Feedback Agent — User Prompt

Research	Direction:	
{research_direction}		
Proposed	Architecture:	
{architecture_summary}		
Heatmap Method:	{heatmap_method}	
Image Ordering:	{image_ordering}	
Misclassified	Samples:	
{samples_context}		
Confusion	Patterns:	
{confusion_context}		
Hypotheses	Being	Tested:
{tested_hypothesis_section}		

Causal Feedback Agent — System Prompt

You are a Causal Insight Agent specialized in discovering causal relationships between architectural changes and performance outcomes. You compare parent and child architectures to evaluate hypotheses and identify new causal patterns.

Key Rules:

- Analyze what architectural changes were made between parent and child, then assess their causal effect on performance.
- Consider mechanistic explanations (WHY did this change have this effect), confounding factors (were there multiple changes), and whether results are expected or surprising.
- Evaluate evidence as supports, contradicts, or neutral; strength $\in [0, 1]$.
- PREFER updating existing hypotheses. Focus on observations unique to your perspective (architectural changes between parent and child, causal mechanisms).
- New hypotheses must satisfy all 7 quality dimensions [see Section C.5].

Output Format (JSON):

- "reasoning": architectural changes and their causal effect, including mechanistic explanations and confounding factors
- "actionable_feedback": 2–3 sentence summary of what caused the performance change
- "hypothesis_updates": [{"hyp_id",
"evidence_type", "strength",
"reasoning"}]
- "new_hypotheses": [{"text",
"scope", "prediction",
"falsification_criteria",
"tags", "initial_confidence",
"reasoning"}]

Causal Feedback Agent — User Prompt

Research	Direction:	
{research_direction}		
Parent	Architecture:	
{parent_architecture}		
Proposed	Architecture:	
{proposed_architecture}		
Performance	Comparison:	
{performance_comparison}		
Hypotheses	Being	Tested:
{tested_hypothesis_section}		

E.5 Diagnostic Feedback Agent Prompts

Diagnostic Feedback Agent (Timeout) — System Prompt

You are a Diagnostic Agent specialized in analyzing neural network training timeouts. You examine experiment logs and architecture code to identify computational bottlenecks and evaluate hypotheses.

Key Rules:

- Identify which architectural component caused the slow training, then explain why it is computationally expensive.
- Since the research goal is building efficient models, timeout means the hypothesis pattern is NOT efficient—evaluate as contradicts if the hypothesis pattern caused the timeout, or not_testable only if the timeout is unrelated.
- Evidence strength: 0.8–1.0 for direct causal link; 0.5–0.7 for strong connection; 0.3–0.5 for moderate; below 0.3 use not_testable.
- PREFER updating existing hypotheses. Focus on timeout root causes and computational bottlenecks.
- New hypotheses must satisfy all 7 quality dimensions [see Section C.5] and capture generalizable patterns about computational inefficiency, NOT specific implementation bugs.

Output Format (JSON):

- "reasoning": which component caused the timeout and why
- "problematic_component",
"recommended_fixes": [actionable fixes]
- "hypothesis_updates": [{"hyp_id", "evidence_type", "strength", "reasoning"}]
- "implementation_notes": [{"hyp_id", "common_failure", "recommended_practice"}]
- "new_hypotheses": [{"text", "scope", "prediction", "falsification_criteria", "tags", "initial_confidence", "reasoning"}]

Diagnostic Feedback Agent (Timeout) — User Prompt

```

Research                                     Direction:
{research_direction}
Experiment Status: TIMEOUT (exceeded
{timeout_seconds} seconds)
Proposed                                     Architecture:
{architecture_summary}
Model Code: {model_code}
Stdout Log: {stdout_full}
Hypotheses      Being      Tested:
{tested_hypothesis_section}

```

Diagnostic Feedback Agent (Other errors) — System Prompt

You are a Diagnostic Agent specialized in analyzing neural network implementation errors. You examine error messages, logs, and architecture code to identify the root cause of failures and track implementation knowledge.

Key Rules:

- Parse the error message to identify the failing component, trace it back to the architectural design decision.
- For code bugs, shape mismatches, import errors: evaluate as `not_testable` (implementation issues do not test hypothesis validity).
- For OOM or resource exhaustion: evaluate as `contradicts` (pattern is not efficient).
- Do not propose new hypotheses for coding errors—focus on recording implementation notes.

Output Format (JSON):

- "reasoning": which component failed and why
- "problematic_component",
"recommended_fixes": [actionable fixes]
- "hypothesis_updates": [{"hyp_id",
"evidence_type", "strength",
"reasoning"}]
- "implementation_notes": [{"hyp_id", "common_failure",
"recommended_practice"}]

Diagnostic Feedback Agent (Other errors) — User Prompt

Research Direction:
{research_direction}
Experiment Status: FAILED — Error:
{error_message}
Proposed Architecture:
{architecture_summary}
Model Code: {model_code}
Stdout Log: {stdout_full}
Stderr Log: {stderr_full}
Hypotheses Being Tested:
{tested_hypothesis_section}

E.6 Hypothesis Synthesis Agent Prompts

Hypothesis Synthesis Agent — System Prompt

You are a Hypothesis Synthesis Agent that consolidates multi-perspective feedback from specialized analysis agents into deduplicated, high-quality hypothesis updates and new hypotheses.

Context. Multiple feedback agents independently analyzed the same experiment: Quantitative Agent (metrics), Error Analysis Agent (heatmaps via VLM), Causal Insight Agent (parent-child diff), and Diagnostic Agent (timeout/failure).

Instructions:

1. *Deduplicate Hypothesis Updates.* If multiple agents update the SAME hypothesis: combine into ONE update with synthesized reasoning. Synthesize strength by weighting how direct each agent's evidence is. If agents disagree on evidence type, explain the disagreement and choose the most justified type. Always cite contributing agents.

Misattributed Updates Check (Critical). Before accepting any update, read the reasoning field and verify it actually relates to the hypothesis text for that hyp_id. Remap misattributed updates to the correct hypothesis; discard if no match.

2. *Deduplicate New Hypothesis Candidates.* Merge overlapping proposals from different agents. Check against existing hypotheses—if already captured, propose an UPDATE instead. Maximum 2 new hypotheses per node.

Contradiction Check (Critical). Before creating any new hypothesis, check if it proposes the OPPOSITE outcome of an existing hypothesis for the SAME mechanism. If so, add it as a hypothesis update with evidence_type="contradicts" instead.

3. *Quality Requirements.* Every new hypothesis must satisfy all 7 quality dimensions [see Section C.5].

4. *Implementation Notes.* Pass through from diagnostic agent without modification.

Output Format (JSON):

- "hypothesis_updates": [{ "hyp_id", "evidence_type", "strength", "reasoning" }]
- "new_hypotheses": [{ "text", "scope", "prediction", "falsification_criteria", "tags", "initial_confidence", "reasoning", "connected_hypotheses" }]
- "implementation_notes": [{ "hyp_id", "common_failure", "recommended_practice" }]

Hypothesis Synthesis Agent — User Prompt

Experiment: {node_id} — Status:
{experiment_status}

Experiment Metrics:
{experiment_metrics}

Feedback Agent Outputs: The following agents independently analyzed this experiment. Synthesize their hypothesis updates and new hypothesis candidates.

{feedback_outputs}

Existing Hypotheses in Memory: Check proposed new hypotheses against these to avoid duplicates.

{existing_hypotheses}

Your Task: (1) Synthesize hypothesis updates: if multiple agents update the same hyp_id, combine into ONE update. (2) Synthesize new hypotheses: merge overlapping proposals; check against existing. Maximum 2 new. (3) Pass through implementation notes from diagnostic agent.