

Neural Architecture Discovery via Autonomous Evolution

Weixian Xu^{1,2*} Yixiu Liu^{2,3,4*} Yang Nan^{2*}
 Lyumanshan Ye^{2,4} Xiangkun Hu^{2,3} Zhen Qin⁵ Pengfei Liu^{2,3,4†}

¹Zhiyuan College, Shanghai Jiao Tong University ²Generative AI Research Lab (GAIR)

³Shanghai Innovation Institute ⁴Shanghai Jiao Tong University ⁵TapTap

Abstract

Recent progress in LLM agents has advanced the prospect of autonomous research. Yet whether AI can complete difficult long-horizon tasks, especially those that advance AI research itself, remains largely unexplored. We present ASI-ARCH, a system for **AI-driven AI research** that autonomously conducts neural architecture research through a closed-loop *research-experiment-analyze-update* process. Applied to linear attention, ASI-ARCH ran 1,773 iterative experiments and discovered 105 state-of-the-art architectures. Its best architecture improves over DeltaNet by nearly three times the gain achieved by Mamba2. Beyond the final performance gains, we analyze the contributions of different parts of the framework in this hard research setting, shedding light on what enables autonomous progress in complex AI research tasks.

1 Introduction

As AI systems continue to improve, the dominant research paradigm is shifting from human-AI collaboration toward increasingly autonomous scientific discovery, giving rise to the vision of an “AI Scientist” capable of conducting research with limited human oversight (Boiko et al., 2023; Chen et al., 2025). A central next question is whether AI can in turn accelerate progress in AI itself, especially on complex and technically demanding research problems. We therefore study **AI-driven AI research**: AI systems that autonomously investigate and advance AI itself—generating hypotheses, running experiments, and synthesizing findings with minimal human intervention.

Neural architecture discovery is among the most challenging and impactful domains for this vision. Each major leap—from CNNs (LeCun & Bengio, 1995) to Transformers (Vaswani et al., 2017)—has come from architectural innovation. Moving beyond Neural Architecture Search (NAS), which is limited to human-defined spaces (Zoph & Le, 2016; Elsken et al., 2019), we present ASI-ARCH: a paradigm shift from *automated optimization* to *automated innovation*. Leveraging modern LLMs (Brown et al., 2020; OpenAI, 2023), ASI-ARCH autonomously hypothesizes novel architectural concepts, implements them as code, and validates performance through rigorous experimentation (Nasir et al., 2023; Cheng et al., 2025; Chen et al., 2025). Over 1,773 experiments and approximately 20,000 total GPU hours on linear attention (Katharopoulos et al., 2020; Choromanski et al., 2020), it continuously discovered 105 SOTA architectures, with the largest relative improvement over DeltaNet reaching three times that achieved by Mamba2. We further perform **cross-family transfer** to Mamba2, GLA (Yang et al., 2024b), and HGRN2 to test whether AI-discovered design principles generalize beyond the original architecture family.

In this work, we make the following contributions: **(1)** We show that an AI system can autonomously complete the full cycle of complex neural architecture design; **(2)** We open-source the complete ASI-ARCH framework¹, all 105 discovered SOTA linear attention architectures, and the entire experiment framework; **(3)** We show that AI-discovered archi-

*Co-first authors.

†Corresponding author: pengfei@sjtu.edu.cn

¹ github.com/GAIR-NLP/ASI-Arch

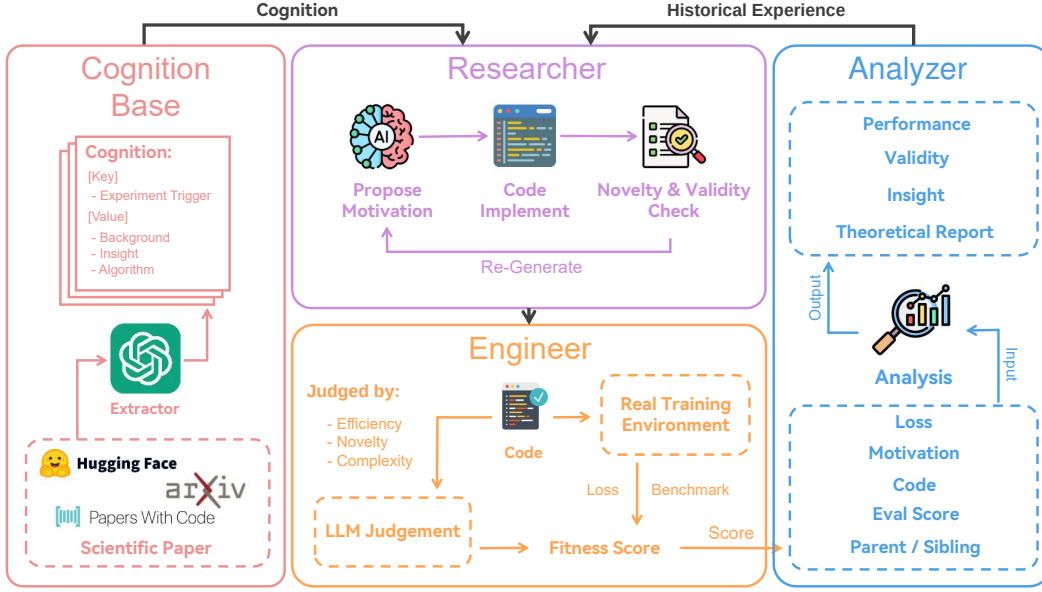


Figure 1: Overview of the ASI-ARCH framework. The Researcher proposes new architectures and implements their code using knowledge retrieved from the Cognition Base and experience accumulated from previous iterations. The Engineer then trains and evaluates the implementations, invoking a debugging agent when needed. The Analyzer interprets the resulting multifaceted experimental evidence and distills it into insights that guide the Researcher in the next iteration. This feedback loop supports iterative improvement.

tectural knowledge exhibits meaningful generality, transferring across model families and yielding consistent gains on Mamba2, GLA, and HGRN2.

2 Background

AI scientific research. The use of AI as an autonomous scientist has grown rapidly alongside the capabilities of modern language models. A prominent class of such systems follows an evolutionary paradigm: given a research target, the AI iteratively proposes and evaluates designs in a closed loop, as exemplified by AlphaEvolve (Novikov et al., 2025), OpenEvolve (Sharma, 2025), ShinkaEvolve (Lange et al., 2025), GEPA (Agrawal et al., 2025), and AdaEvolve (Cemri et al., 2026). A particularly compelling direction within this landscape is AI that studies AI itself—enabling systems to self-improve and self-iterate (Zhang et al., 2025; Schmidhuber, 2007), opening the possibility of systems that autonomously refine their own capabilities without much human intervention. Earlier work in this vein took the form of Neural Architecture Search (NAS) (Zoph & Le, 2016; Elsken et al., 2019), where the AI operates within a human-defined search space as a selection algorithm rather than a creative agent. ASI-ARCH adopts the same evolutionary framework and removes the fixed search space: rather than selecting from predefined candidates, the system freely hypothesizes novel architectures and navigates a far larger and more complex design landscape.

Efficient architectures. The Transformer architecture has dominated sequence modeling since its introduction, but its quadratic attention complexity has motivated extensive research into sub-quadratic alternatives (Vaswani et al., 2017). Three principal families with linear time complexity have emerged: Linear Attention, which linearizes the attention kernel via feature maps (Katharopoulos et al., 2020; Choromanski et al., 2020; Qin et al., 2022); State-Space Models (SSMs) such as Mamba, which employ structured state-transition matrices (Gu & Dao, 2023; Dao & Gu, 2024); and Linear RNNs such as RWKV and HGRN2, which use recurrent formulations with efficient linear-time inference (Peng et al., 2023; Qin et al., 2024). More recently, hybrid architectures like Jamba interleave multiple families to combine their respective strengths (Lieber et al., 2024). This evolution has transformed the

architectural landscape from a single dominant paradigm into a vast combinatorial space where optimal designs are highly task- and constraint-dependent. Manual exploration of this space is slow and expensive, typically requiring months of expert effort to produce a single state-of-the-art architecture. ASI-ARCH addresses this bottleneck by systematically exploring the space through automated multi-agent collaboration, making the automated discovery of novel architectures possible.

3 Methodology

As illustrated in Figure 1, ASI-ARCH is a closed-loop system for autonomous neural architecture discovery. Architecture design poses three compounding challenges that make it uniquely hard for autonomous AI: **(1)** LLMs have limited built-in knowledge of architecture design—the relevant literature is sparse and specialized; **(2)** implementing a model layer means modifying a large, intricate codebase where a single constraint violation silently corrupts training; **(3)** quality can only be measured by pre-training from scratch and running benchmarks—a costly, indirect chain vulnerable to reward hacking. These challenges directly shape the design of ASI-ARCH: each component is introduced to address one or more of them. The system consists of four modules: **Researcher**, **Engineer**, **Analyzer**, and **Experience Repository**. Together, they operate in a continuous loop augmented by an exploration-then-verification strategy.

3.1 Researcher: propose new architecture

The Researcher proposes novel architectures by drawing on experimental history and human domain knowledge.

Model design and integrated implementation. For each cycle, the Researcher receives summaries of past architectures as context, then jointly articulates the architectural motivation and implements the corresponding code in a single pass. This integrated design-then-implement approach avoids the “implementation drift” common in two-stage pipelines, where an implementer working from a high-level description alone produces code that diverges from the intended design.

Novelty and sanity check. Before a proposal enters the expensive training pipeline, the Researcher gates it through a two-stage pre-training check. *Novelty check*: the proposal’s motivation is embedded and matched against the top-5 most similar historical motivations; a specialized LLM then judges whether the design represents genuine innovation or a superficial variant, filtering out redundant explorations. *Sanity check*: a combination of automated code analysis and LLM-based review checks code quality, complexity, causal masking, and related implementation issues. Proposals that fail either check are rejected with specific feedback for revision. Together, these checks markedly improve the success rate of training runs and eliminate wasted GPU hours on fundamentally flawed submissions.

3.2 Engineer: train and evaluate models

The Engineer executes training and constructs the final fitness score.

Training execution. The Engineer runs each proposed architecture in a real interactive code environment. When a training run fails due to an implementation error, the full error log is returned to a dedicated debugging agent, which revises the code based on the concrete failure signal and resubmits the run. Beyond this self-correction loop, an automated monitor runs in parallel and immediately terminates any run that exhibits pathological signals: abnormally low loss, excessive compilation time, or training speed more than $2\times$ slower than baseline. Together these mechanisms maximize the fraction of GPU hours spent on genuinely viable candidates.

Fitness evaluation. To rank candidates reliably, we define a composite fitness that integrates quantitative and qualitative signals:

$$\text{Fitness} = \frac{1}{3} \left[\sigma(\Delta_{\text{loss}}) + \sigma(\Delta_{\text{benchmark}}) + \text{LLM}_{\text{judge}} \right] \quad (1)$$

where $\sigma(\Delta_{\text{loss}})$ and $\sigma(\Delta_{\text{benchmark}})$ are sigmoid-transformed improvements over baseline (focusing on differences within 10% to suppress noise). The sigmoid caps extreme values so that no single metric dominates, while still amplifying small but meaningful gains. Alongside these quantitative scores, we employ an LLM-as-judge ($\text{LLM}_{\text{judge}} \in [0, 1]$) to capture dimensions that benchmark numbers alone cannot reflect—architectural complexity, computational efficiency, implementation elegance, and novelty relative to the baseline—providing a more comprehensive picture of each design’s true merit.

3.3 Analyzer: mine experimental insights

After each experiment, the Analyzer converts the complete experimental record—performance metrics, training curves, and comparisons to parent and sibling nodes in the architectural phylogenetic tree—into a structured analysis that is archived in the Experience Repository. By comparing structurally similar architectures that differ in only a few components, the Analyzer can infer the isolated contribution of individual design choices, approximating an ablation study without requiring additional controlled runs. This accumulated analytical knowledge enables the system to improve not just by reusing prior designs, but by understanding *why* certain designs succeed—generating insights that go beyond what any single paper in the literature can provide.

3.4 Experience repository: shared memory across the search

The Experience Repository is the shared memory that connects all modules and persists knowledge across the entire search.

Evolutionary candidate pool. All completed experiments are logged persistently. The system maintains a dynamically updated pool of the top-50 highest-scoring architectures. For each evolution step, the Researcher samples one parent from the top-10 for direct modification and four references from positions 11–50 for diverse examples. Updates to this pool are batched so that all parallel agents are more likely to work from a stable reference set. In addition, to ensure broader exploration diversity in the early stage of the search, we impose an initial cold-start period of 200 explorations during which the pool is frozen before convergence pressure sets in.

Cognition base. We distill approximately 100 seminal linear-attention papers into a structured knowledge base of roughly 150 *cognition entries*. Each entry captures: (i) the *applicable scenario*, (ii) the *core algorithm*, and (iii) the *historical context*. After each experiment, the Analyzer’s summary of observed shortcomings is used as a retrieval query against the scenario field via embedding-based search; the top-3 matching cognitions are injected into the Researcher context for the next cycle. This retrieval-augmented mechanism gives the system access to targeted domain expertise on demand, substantially enriching its proposals without flooding the context with irrelevant background.

3.5 Exploration-then-verification strategy

We adopt a two-stage exploration-then-verification strategy to balance search breadth against evaluation rigor while also testing scalability. In the **exploration stage**, candidates are trained at 20M parameters on 1B tokens—fast enough to run thousands of experiments—and evaluated on 500 samples per benchmark. Only architectures that surpass DeltaNet on both quantitative and qualitative metrics advance. In the **verification stage**, survivors are scaled to 340M on 1B tokens with a strict causality check that verifies that perturbing later tokens does not change earlier-token hidden states, and the top candidates are further scaled to 1.3B on 15B tokens with six additional out-of-distribution benchmarks. Starting at small

scale lets us screen a large candidate pool cheaply; the multi-scale verification then confirms that discovered gains persist as models grow, ruling out artifacts of the resource-constrained exploration setting.

4 Main results

We use DeltaNet (Yang et al., 2024c) as the primary baseline throughout. Full experimental configuration details—model architecture, training protocol, dataset, and evaluation benchmarks—are provided in Appendix B; additional reproducibility and cognition-source results are reported in Appendix B.4.

Search process. The exploration stage ran 1,773 experiments at 20M scale, consuming approximately 10,000 GPU hours. Among these, 1,350 candidates outperformed DeltaNet on both quantitative and qualitative metrics and passed the exploration-stage filters. We selected approximately 150 of these for 340M-scale verification, of which **105 architectures achieved state-of-the-art results under our controlled 340M evaluation**. As shown in Figure 2, the cumulative count of SOTA discoveries grows at a near-linear rate throughout—allocating more compute translates reliably into more discoveries (see Section 5.1 for a mechanistic explanation). We selected the top 5 architectures for large-scale validation.

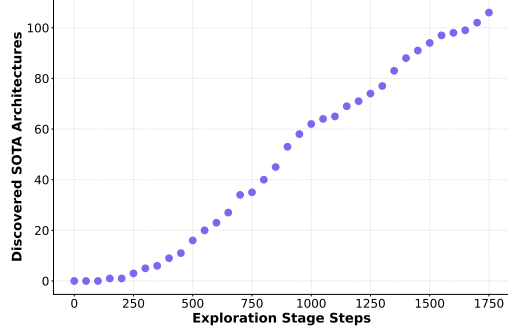


Figure 2: Cumulative count of discovered SOTA architectures vs. number of exploration experiments. The near-linear trend confirms that breakthroughs grow steadily with compute.

Large-scale validation and discovered architectures.

The five selected architectures were trained at 1.3B parameters on 15B tokens and evaluated on 16 benchmarks: 10 development benchmarks used throughout exploration (Wiki PPL, LAMBADA ppl/acc, PIQA, HellaSwag, WinoGrande, ARC-Easy/Challenge, Social-IQA, BoolQ) and 6 new generalization benchmarks (RACE, BBQ, MetaBench, QA4MRE-2011, SciQ, SWAG). Results are reported in Table 1.

Our best model improves over DeltaNet by **+0.97 points** on the overall average, nearly $3\times$ the gain of the current strongest human-designed competitor, Mamba2 (**+0.34 points**). This improvement is achieved autonomously, without human architectural intuition. Across all five models, the average improvement over DeltaNet is +0.77 points on development benchmarks and +0.29 points on the new generalization set, confirming that gains are not artifacts of overfitting to the exploration benchmarks.

A common pattern among the strongest architectures is the use of multiple auxiliary paths coordinated through adaptive routing or gating. **PathGateFusionNet** uses a two-stage router that first separates direct copying from contextual processing, then distributes the contextual budget across short-range, long-range, and Delta-rule branches, with an additional lightweight residual path and head-wise output gate. **ContentSharpRouter** keeps a flat four-way router, but makes it content-adaptive and equips each head with a learnable temperature so routing can become sharper or softer depending on the input. **FusionGatedFIRNet** removes the softmax competition entirely and replaces it with independent sigmoid gates, allowing local and global paths to remain active at the same time; it also adds per-head retention to adjust the memory horizon of the Delta-rule branch.

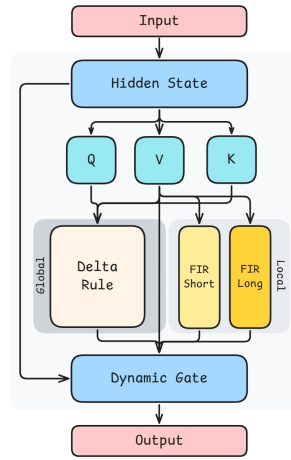


Figure 3: HierGateNet

HierGateNet, illustrated in Figure 3, also adopts hierarchical routing, but factorizes it into a coarse local-versus-global choice followed by within-group selection, and introduces learnable path floors to prevent important routes from collapsing. **AdaMultiPathGateNet** pushes routing to the finest granularity with a unified gate that combines token-level, head-level, and global logits, stabilized by an epsilon floor and persistent entropy regularization. Taken together, these variants indicate that the strongest designs consistently improve DeltaNet by making routing and gating more adaptive across paths and scales.

Benchmarks	DeltaNet	Gated-DeltaNet	Mamba2	PG	C	FG	H	AM
<i>Development</i>								
Wiki ppl↓	17.00	16.84	16.66	16.18	<u>16.05</u>	15.77	16.65	16.26
LMB ppl↓	13.63	13.31	13.33	<u>12.62</u>	13.45	12.34	13.06	13.75
LMB	45.47	46.26	46.24	47.60	46.13	<u>47.53</u>	46.56	45.04
PIQA	73.12	74.10	73.78	72.91	<u>74.37</u>	72.91	74.37	74.10
Hella	56.29	57.55	58.58	56.99	57.00	<u>58.47</u>	56.85	57.17
Wino	55.88	58.01	<u>58.48</u>	57.22	57.85	60.14	57.38	57.62
ARC-e	73.40	72.14	72.98	73.06	72.05	<u>74.28</u>	73.11	74.28
ARC-c	40.61	36.95	39.33	<u>40.36</u>	39.76	40.02	39.33	39.33
SIQA	40.74	41.71	41.81	<u>42.37</u>	41.81	42.78	42.07	42.07
BoolQ	60.58	53.98	60.52	62.45	<u>62.51</u>	62.11	63.03	56.27
<i>Generalization</i>								
RACE	34.45	33.78	32.15	35.22	34.55	35.60	<u>35.22</u>	35.02
BBQ	29.53	29.75	29.43	29.95	30.55	31.46	<u>30.88</u>	30.27
MetaBench	26.97	28.67	27.70	25.64	29.55	26.79	<u>29.38</u>	28.98
QA4MRE	40.00	35.00	39.17	39.17	<u>39.17</u>	38.33	38.33	38.33
SCIQ	89.80	<u>90.30</u>	90.30	89.60	89.50	89.20	90.40	89.20
SWAG	47.69	48.17	48.88	48.22	47.80	<u>48.57</u>	48.18	47.80
<i>Averages</i>								
Dev. Avg	55.76	55.09	56.47	<u>56.62</u>	56.44	57.28	56.59	55.74
Gen. Avg	44.74	44.28	44.61	44.63	<u>45.19</u>	44.99	45.40	44.93
Overall Avg	51.04	50.46	51.38	51.48	51.61	52.01	<u>51.79</u>	51.11

Table 1: Top block: 10 development benchmarks, used in our exploration stage; bottom block: 6 generalization benchmarks for out-of-distribution testing. Dev., Gen., and Overall averages cover the 8 development, 6 generalization, and all 14 accuracy metrics, respectively; perplexity is reported separately. **Bold** indicates the best results, and underlining indicates the second-best results. Model abbreviations are as follows: PG = PathGateFusionNet, C = ContentSharpRouter, FG = FusionGatedFIRNet, H = HierGateNet, and AM = AdaMulti-PathGateNet.

5 Analysis

Throughout this section we track statistics in windows of 50 candidates (matching the pool update cadence) and refer to the top-105 architectures collectively as the *model gallery*.

5.1 Evolution dynamics and fitness design

Figure 4 reveals two complementary patterns.

Figure 4b shows that the composite fitness score of the top-50 candidate pool rises sharply over the first approximately 400 samples as the system quickly identifies architectures that

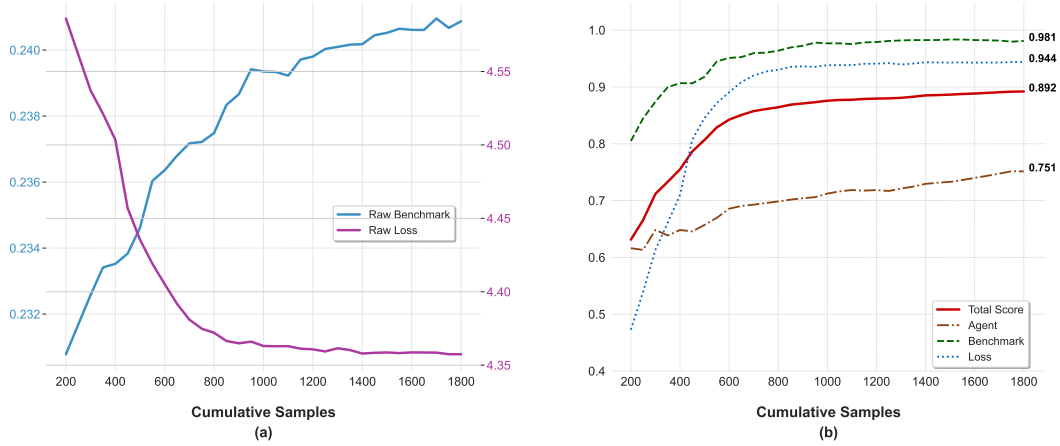


Figure 4: (a) Average raw benchmark score and loss of top-50 candidates vs. cumulative samples. (b) Composite fitness and its components vs. cumulative samples. The fitness plateau is a designed effect of the sigmoid transformation, not a performance ceiling.

outperform the DeltaNet baseline on both loss and benchmarks. Around 900 samples, however, the growth of the fitness score begins to slow. Comparing Figure 4a and Figure 4b reveals that this slowdown does not mean the search stops improving: the system continues to discover architectures with lower loss and higher benchmark scores, but the sigmoid transformation maps these raw gains into progressively smaller changes in the rule-based objective part of the fitness. We intentionally adopt this design so that no single metric can dominate the optimization signal indefinitely. As a result, the relative influence of the LLM judge becomes larger in the later stage of the search, encouraging the system to explore designs that are more principled, refined, and diverse rather than over-exploit a narrow performance regime. A direct piece of evidence is that the subjective score assigned by the LLM judge continues to improve steadily throughout the run.

Another key question is whether the design of the fitness function truly determines the system’s evolutionary trajectory. Due to the sigmoid design, score differences among nodes become smaller after 900 samples, making the fitness landscape relatively flatter in the later stage of the search. We therefore examined the records of the 105 architectures in the model gallery and found that, among the architectures discovered after 900 samples, approximately 78% have parent nodes from the first 900 samples. This shows that the fitness function successfully drives the system to effectively reuse previously discovered high-quality architectures in the later stage, forming a breadth-wise search over strong existing results rather than merely performing local optimization by repeatedly iterating on the same result.

To test whether these dynamics are a single-run artifact, we conducted two independent full-pipeline reruns with DeepSeek-V4-Flash under the 120-result configuration described in Appendix B.4. The runs reached final best scores of 2.5161 and 2.4311, while their area under the best-score curve (AUC) values differed by only 0.45% (281.3209 vs. 280.0682), supporting the reproducibility of the evolution trend (Table 6).

5.2 Knowledge sources and evolutionary shift

Each proposed architecture can draw on three sources of inspiration: *cognition* (knowledge retrieved from the human literature base), *experience* (insights synthesized from past experimental analyses), or *originality* (novel ideas generated by the model itself). We prompted an LLM to classify the origin of each architectural component for both the full set of 1,773 architectures and the 105 architectures in the model gallery (Table 2).

Cognition is the dominant source throughout, accounting for 51.7% of all design ideas and remaining at 48.6% within the model gallery—confirming that human prior knowledge is

Category	Experience	Cognition	Originality
Model Gallery	44.8%	48.6%	6.6%
Others	37.7%	51.9%	10.4%
All	38.2%	51.7%	10.1%

Table 2: Comparison of the influence of pipeline components on SOTA versus other model designs. The data reveal a greater dependence on empirical analysis in the development of SOTA architectures.

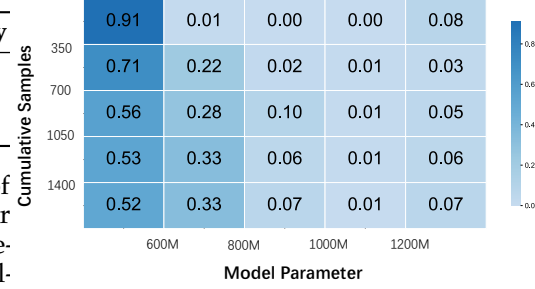


Figure 5: Parameter-count distribution of candidate architectures.

indispensable for producing technically grounded proposals. At the same time, a meaningful shift emerges among top performers: experience rises from 38.2% to 44.8%, echoing the pattern in human scientific progress where breakthroughs come not from prior literature alone but from synthesizing insights accumulated through one’s own experimentation. Finally, originality remains low throughout: 10.1% in the full set and 6.6% in the model gallery. This suggests that the system usually grounds its proposals in the knowledge base rather than relying on ideas generated from scratch. In this highly specialized domain, that is exactly the behavior one would want.

To test whether cognition preparation can also be automated, we built a cognition source using GPT-5.5 with Deep Research and ran the same evolve pipeline. Across the first 120 results, the AI-curated run reached a final best raw score of 2.5500 and an AUC of 285.6003, while the matched human-curated run reached a final best raw score of 2.5161 and an AUC of 281.3209 (Table 7). These comparable run-level outcomes provide evidence that an AI-curated cognition source can support ASI-ARCH without extensive manual curation.

5.3 Architectural design patterns

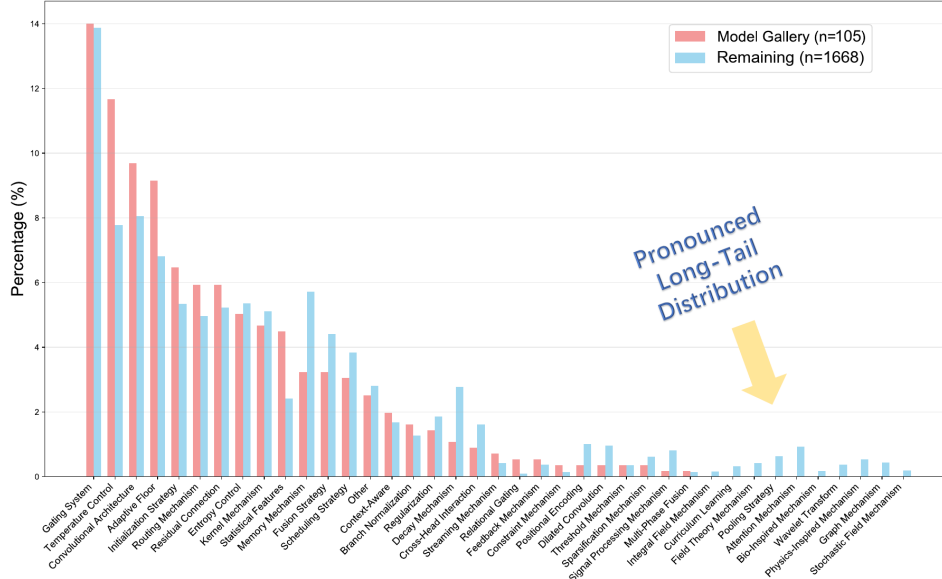


Figure 6: Component-mention distributions: model gallery (top 105) vs. remaining architectures.

We examined the parameter count distribution of candidate architectures under a common reference configuration across iterations (Figure 5). Most candidates remained within the lower parameter ranges throughout the search, suggesting that performance gains were not driven solely by parameter growth. To understand what that innovation looks like

in practice, we parsed all 1,773 design motivations with an auxiliary LLM, extracting over 5,000 component mentions and grouping them into high-level categories (Figure 6); percentages denote each category’s share of component mentions within the corresponding group. The model gallery shows a notably less pronounced long-tail distribution than the remaining architectures: top models converge on a core set of proven components—gating mechanisms and convolutions—and rarely adopt exotic alternatives such as physics-inspired mechanisms. This pattern mirrors human scientific practice, where state-of-the-art results are more reliably achieved through targeted innovation on a foundation of validated techniques than through the pursuit of novelty for its own sake.

5.4 Cross-family transfer

To more comprehensively assess the quality of the architectures discovered by ASI-ARCH, we investigate whether the design principles learned on DeltaNet generalize beyond the family in which they were discovered—a stronger test of their value than within-family performance alone. The system summarized regularities from the top 105 DeltaNet variants into transferable design rules and prompts specifying architectural rewrites and implementation constraints. The system applies these rules to Mamba2 (SSM), GLA, and HGRN2, synthesizing candidate variants for each family. All candidates pass the same novelty, sanity, and efficiency checks (Sections 3.1 and 3.2) and are trained under identical configurations.

Family	Baseline avg.	Variant avg.	Gain
Mamba2	47.84	48.19	+0.35
HGRN2	47.67	47.81	+0.14
GLA	45.55	47.60	+2.05

Table 3: Summary of cross-family transfer at 340M parameters. Each row reports the baseline and the selected variant; full benchmark results are reported in Appendix Table 8.

Transferability of distilled rules. Although the Delta-rule algorithm itself is difficult to transfer directly across model families, rules centered on routing, gating, and stability transfer effectively to Mamba2, GLA, and HGRN2, yielding improvements over the corresponding baseline for each target family (Table 3). This suggests that the discovered principles capture generalizable architectural behaviors rather than DeltaNet-specific quirks.

Per-family outcomes. GLA shows the most robust transfer. Mamba2 and HGRN2 also achieve performance gains over their baselines, but the improvements are relatively smaller. We attribute this to two factors: (1) **Structural alignment**—DeltaNet and GLA share linearized attention-style routing and gating, so DeltaNet-derived rules map more naturally onto GLA’s computation graph; (2) **Baseline saturation and interaction effects**—Mamba2 and HGRN2 are already heavily optimized and incorporate multiple stabilization mechanisms; external rule insertions may partially overlap with or conflict with their intrinsic mechanisms, producing diminishing returns. Future work will develop family-specific adapters to reduce interference.

5.5 Efficiency at large scale

We measure end-to-end efficiency under the same local implementation and hardware setting. Training throughput includes the forward pass, language-modeling loss, backward pass, and optimizer step; prefill measures full-sequence inference with cache enabled. As shown in Table 4, the discovered architectures maintain training throughput in the same practical range as the baselines, while several achieve prefill throughput comparable to Mamba2. This efficiency remains competitive despite the additional computation introduced by routing, gating, and extra projections in some finalists.

Model	Train tok/s $\uparrow$	Prefill tok/s $\uparrow$	Peak mem. $\downarrow$
DeltaNet	5,063	23,162	11.44G
Gated DeltaNet	4,959	22,413	11.54G
Mamba2	4,476	12,725	11.29G
ASI-ARCH-1	4,037	13,589	18.32G
ASI-ARCH-2	4,168	12,237	16.28G
ASI-ARCH-3	3,426	11,932	18.39G
ASI-ARCH-4	4,027	12,950	22.75G
ASI-ARCH-5	3,923	11,936	16.57G

Table 4: End-to-end efficiency at 1.3B scale under the same implementation and hardware setting.

5.6 Framework comparison

We compare ASI-ARCH with OpenEvolve (Sharma, 2025) using the same DeepSeek-V4-Flash proposal model and a 120-attempt budget; candidate architectures are trained under the 20M/1B-token configuration. Because the systems use different native scores, Table 5 reports metric-wise best values across successful candidates: Loss is the lowest final-step training loss, while Benchmark is the highest mean accuracy over LAMBADA, HellaSwag, PIQA, WinoGrande, ARC-Easy, ARC-Challenge, Social-IQA, and BoolQ. The two values need not come from the same candidate. The rate is the fraction of raw attempts that produce a database result satisfying the architecture requirements. Parenthesized percentages are improvements relative to the DeltaNet baseline (loss 4.3861; benchmark 0.3205).

Strategy	Rate	Loss $\downarrow$	Benchmark $\uparrow$
ASI-ARCH	37.4%	4.1043 (6.4%)	0.3500 (9.2%)
OpenEvolve	8.3%	4.3454 (0.9%)	0.3405 (6.2%)

Table 5: Strategy-level comparison under the same proposal model and attempt budget.

Under this matched setting, ASI-ARCH achieves approximately $4.5\times$ the successful-design rate of OpenEvolve together with stronger loss and benchmark improvements, showing that its task-specific pipeline provides a substantial advantage over a generic evolution strategy for architecture discovery.

6 Conclusion

We presented ASI-ARCH, an autonomous system for AI-driven neural architecture research that operates through a closed-loop *research-experiment-analyze-update* process without human intervention. In linear attention, the system continuously discovered SOTA architectures throughout 1,773 experiments, yielding 105 novel designs with no saturation in sight. Distilled design rules further transferred across families from DeltaNet to Mamba2, GLA, and HGRN2. We analyzed the design patterns and fitness dynamics that enabled these results and open-sourced the framework and all discovered architectures. Future work will explore larger architectural hypothesis spaces, continue iterating on the strong designs discovered here, and pursue concrete algorithmic efficiency optimizations to better understand their practical efficiency.

References

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, 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. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

- Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.
- Daniil A. Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. Autonomous chemical research with large language models. *Nature*, 624(7992):570–578, 2023. doi: 10.1038/s41586-023-06792-0.
- Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. AdaEvolve: Adaptive LLM driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.
- Qiguang Chen, Mingda Yang, Libo Qin, Jinhao Liu, Zheng Yan, Jiannan Guan, Dengyun Peng, Yiyang Ji, Hanjing Li, Mengkang Hu, Yimeng Zhang, Yihao Liang, Yuhang Zhou, Jiaqi Wang, Zhi Chen, and Wanxiang Che. AI4Research: A survey of artificial intelligence for scientific research, 2025. URL <https://arxiv.org/abs/2507.01903>.
- Junyan Cheng, Peter Clark, and Kyle Richardson. Language modeling by language models. *arXiv preprint arXiv:2506.20249*, 2025.
- Krzysztof Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.
- Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.
- Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try ARC, the AI2 reasoning challenge. *ArXiv*, abs/1803.05457, 2018.
- Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.
- Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.
- Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac’h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, December 2023. URL <https://zenodo.org/records/10256836>.
- Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.
- Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pp. 5156–5165. PMLR, 2020.
- Alex Kipnis, Konstantinos Voudouris, Luca M. Schulze Buschoff, and Eric Schulz. metabench – a sparse benchmark of reasoning and knowledge in large language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=4T33izzFpK>.

- Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. RACE: Large-scale ReAding comprehension dataset from examinations. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel (eds.), *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 785–794, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1082. URL <https://aclanthology.org/D17-1082>.
- Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.
- Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time series. In Michael A. Arbib (ed.), *The Handbook of Brain Theory and Neural Networks*, pp. 255–258. MIT Press, Cambridge, MA, 1995.
- Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, et al. Jamba: A hybrid transformer-Mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.
- Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.
- Muhammad U. Nasir, Sam Earle, Christopher Cleghorn, Steven James, and Julian Togelius. LLMatic: Neural architecture search via large language models and quality diversity optimization. *arXiv preprint arXiv:2306.01102*, 2023.
- Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.
- OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.
- Alicia Parrish, Angelica Chen, Nikita Nangia, Vishakh Padmakumar, Jason Phang, Jana Thompson, Phu Mon Htut, and Samuel Bowman. BBQ: A hand-built bias benchmark for question answering. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Findings of the Association for Computational Linguistics: ACL 2022*, pp. 2086–2105, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.165. URL <https://aclanthology.org/2022.findings-acl.165>.
- Anselmo Peñas, Eduard H. Hovy, Pamela Forner, Álvaro Rodrigo, Richard F. E. Sutcliffe, and Roser Morante. QA4MRE 2011-2013: Overview of question answering for machine reading evaluation. In *CLEF*, 2013.
- Guilherme Penedo, Hynek Kydlíček, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro von Werra, and Thomas Wolf. The FineWeb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37: 30811–30849, 2024.
- Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. RWKV: Reinventing RNNs for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.
- Zhen Qin, Weixuan Sun, Hui Deng, Dongxu Li, Yunshen Wei, Baohong Lv, Junjie Yan, Lingpeng Kong, and Yiran Zhong. cosFormer: Rethinking softmax in attention. *arXiv preprint arXiv:2202.08791*, 2022.
- Zhen Qin, Songlin Yang, Weixuan Sun, Xuyang Shen, Dong Li, Weigao Sun, and Yiran Zhong. HGRN2: Gated linear RNNs with state expansion. *arXiv preprint arXiv:2404.07904*, 2024.

- Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: An adversarial Winograd schema challenge at scale. *arXiv preprint arXiv:1907.10641*, 2019.
- Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. Social IQa: Commonsense reasoning about social interactions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 4463–4473, 2019.
- Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In Ben Goertzel and Cassio Pennachin (eds.), *Artificial General Intelligence*, pp. 199–226. Springer, Berlin, Heidelberg, 2007. doi: 10.1007/978-3-540-68677-4_7.
- Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent. <https://github.com/algorithmicsuperintelligence/openevolve>, 2025.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.
- Johannes Welbl, Nelson F. Liu, and Matt Gardner. Crowdsourcing multiple choice science questions. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pp. 94–106. Association for Computational Linguistics, 2017. doi: 10.18653/v1/W17-4413.
- Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule. *arXiv preprint arXiv:2412.06464*, 2024a.
- Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 56501–56523. PMLR, 2024b. URL <https://proceedings.mlr.press/v235/yang24ab.html>.
- Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. *Advances in Neural Information Processing Systems*, 37:115491–115522, 2024c.
- Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. SWAG: A large-scale adversarial dataset for grounded commonsense inference. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.
- Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.
- Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025. URL <https://arxiv.org/abs/2505.22954>.
- Yu Zhang and Songlin Yang. FLAME: Flash language modeling made easy, January 2025. URL <https://github.com/fla-org/flame>.
- Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

A LLM disclosure

We disclose major uses of large language models in this work. Experiments used OpenAI’s **o3**, **GPT-4.1**, and **GPT-5.5 with Deep Research**, as well as **DeepSeek-V4-Flash**, across the ASI-ARCH pipeline and additional analyses. We also used LLMs to assist with edits and refactoring in the experiment codebase. The manuscript was written by the authors; LLM assistance in paper preparation was limited to non-substantive LaTeX and table formatting, and no LLMs were used to draft scientific content.

B Experimental setup

B.1 Experimental configuration

Baseline and fitness calibration. We use DeltaNet (Yang et al., 2024c) as our primary baseline. For quantitative fitness scoring, we map performance differences within 10% of baseline through a sigmoid function; architectures whose training loss falls more than 10% below baseline are immediately discarded as likely information leakage. For qualitative scoring, DeltaNet is set at 5 points and Gated DeltaNet (Yang et al., 2024a) at 10 points; the LLM evaluator scores each candidate on this 1–10 scale, rescaled to $[0, 1]$ via $(s - 1)/9$.

Parallel search and pool management. Multiple search processes run in parallel, coordinated through a shared cloud database. The first 200 explorations proceed without pool updates (cold start), after which the candidate pool is updated every 50 new entries (batched updates), ensuring all agents reference a stable set during each exploration batch.

Two-stage evaluation.

- **Exploration stage:** 20M parameters (8 layers, hidden dim 256, 8 heads), 2,000 steps on 1B FineWeb-Edu tokens, 500 samples per benchmark.
- **Verification stage:** Promising candidates scaled to 340M (24 layers, hidden dim 1024), trained on 1B tokens with a strict causality check. Top models further scaled to 1.3B and trained for 30,000 steps on 15B tokens, evaluated on 16 benchmarks.

B.2 Pipeline configuration

Framework overview. Our experimental framework implements an automated AI self-iterative system for exploring novel neural network architectures through three core phases: Evolve, Training, and Analysis. The system operates cyclically, extracting nodes from a MongoDB database, generating new motivations and implementations, conducting training and evaluation, and performing comprehensive analysis with knowledge integration.

Multi-model integration. We employ a hybrid multi-model approach to optimize both quality and efficiency. In the Evolve phase, we combine o3 and GPT-4.1 models for the planner component to balance motivation quality and generation speed while enhancing architectural diversity. The checker component utilizes o3 to ensure code validity and prevent resource waste, while motivation deduplication employs GPT-4.1 for rapid processing. During the Training phase, GPT-4.1 handles training initiation, testing, and debugging operations, focusing on detail-level modifications without structural changes for rapid iteration capabilities. The Analysis phase utilizes o3 to conduct comprehensive experimental analysis, providing high-quality insights to enhance subsequent exploration efficiency.

Data management and retrieval. For data management and retrieval, MongoDB serves as our primary storage solution, supporting name-based and sequential storage along with deletion functionality for experimental nodes. FAISS enables efficient similarity matching during motivation deduplication, identifying similar concepts in the database before agent-based verification to improve exploration efficiency. We curate cognitive insights from relevant literature and index them in OpenSearch for retrieval-augmented generation. For each experimental result, we retrieve the three most similar cognitive entries and integrate them into the experimental node to support future exploration.

B.3 Model and training details

Progressive evaluation strategy. To balance exploration efficiency with computational constraints, we implement a three-tiered progressive evaluation approach with rapid architecture exploration using 20M parameter models, followed by validation phases at larger scales.

Model architecture. The base 20M configuration employs 8 attention heads across 8 hidden layers with a hidden dimension of 256, maintaining computational tractability while preserving architectural expressiveness. The model uses SiLU activation for query-key transformations with L2 normalization and incorporates short convolutions with a kernel size of 4. For comparison, 340M parameter models employ 1024 hidden size, 8 attention heads, and 24 hidden layers with tied word embeddings disabled.

Training protocol. We utilize the FLAME framework (Zhang & Yang, 2025) with AdamW optimization, employing a peak learning rate of 3×10^{-4} , epsilon value of 1×10^{-8} , and a warmup-stabilize-decay (WSD) learning rate schedule. Training proceeds for 2,000 steps for 20M models, using mixed precision training with bfloat16 parameters and float32 gradient reduction. All models maintain a consistent batch size of 256 and employ GPT-2 tokenizer throughout training and evaluation phases.

Data configuration. Training utilizes FineWeb-Edu sample-10BT and sample-100BT datasets (Penedo et al., 2024) with a context length of 2048 tokens. For comprehensive evaluation of discovered architectures, we scale to 340M parameter models that provide more reliable performance assessment. The 340M models are trained using 1 billion tokens. The 1.3B models are trained using 15 billion tokens from sample-100BT, with a learning-rate warm-up phase of approximately 0.5 billion tokens.

Evaluation protocol. Model evaluation is performed using the LM-Evaluation-Harness framework (Gao et al., 2023), a standardized open-source tool developed by EleutherAI that provides unified benchmarking protocols for language models. The evaluation suite assesses a diverse range of capabilities, including perplexity (Wiki PPL, LAMBADA PPL) (Merity et al., 2016; Paperno et al., 2016), reading comprehension (LAMBADA acc, RACE acc) (Paperno et al., 2016; Lai et al., 2017), commonsense reasoning (PIQA acc, HellaSwag acc, SWAG acc, Social-IQA acc, WinoGrande acc) (Bisk et al., 2020; Zellers et al., 2019; 2018; Sap et al., 2019; Sakaguchi et al., 2019), knowledge-intensive tasks (ARC-Easy acc, ARC-Challenge acc, SciQ acc, QA4MRE-2011 acc) (Clark et al., 2018; Welbl et al., 2017; Peñas et al., 2013), and reasoning and general capabilities (BoolQ acc, BBQ acc, MetaBench acc) (Clark et al., 2019; Parrish et al., 2022; Kipnis et al., 2025). For rapid architectural exploration, the 20M parameter models are evaluated on a limited sample size of 500 per dataset, while validation phases use full datasets for all other models. All evaluations are conducted with consistent hyperparameters to ensure fair comparison across architectural variants: a 'dtype' of bfloat16, a 'max.length' of 4096, and model-specific batch sizes of 64 for the 20M models, 4 for the 340M models, and 8 for the 1.3B models. Final model ranking is a comprehensive assessment that incorporates LLM subjective evaluation, training loss metrics, and benchmark performance.

B.4 Additional reproducibility and cognition-source results

Reproducibility across independent runs. We reran the complete evolve pipeline twice with DeepSeek-V4-Flash, with each run producing 120 results. For these reruns, we removed the 200-exploration frozen-pool cold start and refreshed the candidate pool every 20 results. The similar best-score dynamics in Table 6 indicate that the evolution trend is not a single-run artifact. In Tables 6 and 7, the reported raw total score is the unnormalized sum of the three fitness components before the factor of $1/3$ in Equation (1); AUC and curve-deviation statistics are computed from this raw-total-score curve. AUC denotes the area under the curve over the first 120 results. For both reruns, the standard deviation from the average curve is 0.0798, corresponding to 3.41% of the mean raw total score along the curve.

AI-curated cognition source. To test whether ASI-ARCH requires a manually curated cognition base, we constructed a cognition source from the output of GPT-5.5 with Deep Research. We asked the model to identify papers related to linear attention and neural architecture design and return their metadata, summaries, and relevance. We converted the report into the same cognition schema used by ASI-ARCH. Table 7 compares this run with the first reproducibility rerun in Table 6, which uses the original human-curated cognition

Metric	Run 1	Run 2	Difference
Final best score	2.5161	2.4311	0.0850 (3.44%)
AUC	281.3209	280.0682	1.2527 (0.45%)
Std. dev. from average curve	0.0798	0.0798	–

Table 6: Two independent full-pipeline reruns.

base. Both runs used DeepSeek-V4-Flash, omitted the 200-exploration frozen-pool cold start, refreshed the candidate pool every 20 results, and are reported over their first 120 results.

Metric	Human Curated	AI Curated
Final best score	2.5161	2.5500
AUC	281.3209	285.6003
Best score at 20 results	2.3518	2.3526
Best score at 40 results	2.3518	2.3526
Best score at 60 results	2.3518	2.4510
Best score at 80 results	2.3518	2.5500
Best score at 100 results	2.3518	2.5500
Best score at 120 results	2.5161	2.5500

Table 7: Comparison of human-curated and AI-curated cognition sources over the first 120 results.

B.5 Supplementary results

Table 8 reports the complete benchmark results for cross-family transfer, while Table 9 displays the 20M/340M results for 105 discovered architectures. Some discovered architectures share the same generated name and exhibit similar designs. We group these entries under the same name and report their distinct 340M results on separate rows.

Model	Wiki ↓	LMB ↓	LMB ↑	PIQA ↑	Hella ↑	Wino ↑	ARC-e ↑	ARC-c ↑	SIQA ↑	BoolQ ↑	Avg.
Mamba2	27.08	40.09	31.32	67.90	42.25	51.46	62.04	29.27	39.25	59.24	47.84
Mamba2 variant	26.79	38.46	32.49	67.52	42.40	51.30	63.22	30.29	38.49	59.79	48.19
HGRN2	28.09	36.02	32.39	68.82	41.42	51.30	61.03	29.01	37.92	59.48	47.67
HGRN2 variant	27.56	38.31	32.45	68.28	41.33	51.07	61.03	29.14	38.95	60.21	47.81
GLA	30.83	53.44	28.55	65.61	38.04	51.14	57.87	26.54	37.51	59.14	45.55
GLA variant	27.48	36.08	32.84	67.52	41.51	50.67	62.21	28.16	38.59	59.33	47.60

Table 8: Complete cross-family transfer results at 340M parameters. For each family, we report the baseline and the selected variant. Metrics follow the main setup: Wiki ppl, LAMBADA ppl/acc, PIQA, HellaSwag (normalized), WinoGrande, ARC-Easy, ARC-Challenge (normalized), SIQA, BoolQ, and the average across accuracy metrics. **Bold** indicates the better result within each family pair.

Model Name	20M params / 1B tokens		340M params / 1B tokens	
	Train Loss ↓	Test Score ↑	Train Loss ↓	Test Score ↑
DeltaNet (Baseline)	4.5749	36.23	3.5055	41.16
Gated DeltaNet (Baseline)	4.5678	36.60	3.4768	42.10
AdaptiveContextFusionNet	4.4973 −0.0705	37.03 +0.43	3.4624 −0.0144	42.74 +0.64
AdaptiveEntropyGateNet	4.4423 −0.1255	36.91 +0.31	3.4558 −0.0210	42.37 +0.27
AdaptiveEntropyRouter	4.3547 −0.2131	39.26 +2.66	3.4066 −0.0702	44.31 +2.21
AdaptiveEntropyRouterNet	4.3326 −0.2352	36.94 +0.34	3.4298 −0.0470	43.25 +1.15
AdaptiveFloorGate	4.4695 −0.0983	37.00 +0.40	3.4418 −0.0350	43.57 +1.47
AdaptiveFloorNet-HAF	4.4002 −0.1676	37.03 +0.43	3.4241 −0.0527	43.59 +1.49
AdaptiveFractalGateNet	4.5484 −0.0194	38.43 +1.83	3.4351 −0.0417	43.84 +1.74
AdaptiveFusionNet	4.3521 −0.2157	37.51 +0.91	3.4336 −0.0432	43.78 +1.68
	—	—	3.4061 −0.0707	44.08 +1.98
	—	—	3.4280 −0.0488	43.66 +1.56
AdaptiveFusionNet-DSI	4.3781 −0.1897	36.63 +0.03	3.4270 −0.0498	43.39 +1.29
AdaptiveFusionRNet	4.3940 −0.1738	37.03 +0.43	3.4086 −0.0682	43.73 +1.63
AdaptiveGateNet	4.4228 −0.1450	36.57 −0.03	3.4377 −0.0391	43.64 +1.54
AdaptiveGateNet-AFP	4.4198 −0.1480	37.80 +1.20	3.4193 −0.0575	43.88 +1.78
AdaptiveGateRouter_X	4.4126 −0.1552	38.69 +2.09	3.4114 −0.0654	42.68 +0.58
AdaptiveGatedRouter-Hybrid	4.3335 −0.2343	37.74 +1.14	3.4060 −0.0708	43.56 +1.46
AdaptiveHierGateNet	4.5096 −0.0582	36.71 +0.11	3.4433 −0.0335	43.44 +1.34
AdaptiveHybridGateNet	4.3867 −0.1811	37.11 +0.51	3.4239 −0.0529	43.01 +0.91
AdaptiveMixGateNet	4.3709 −0.1969	36.91 +0.31	3.4289 −0.0479	43.28 +1.18
AdaptiveMixTransformer	4.4855 −0.0823	36.49 −0.11	3.4593 −0.0175	42.50 +0.40
AdaptivePathRouter	4.4324 −0.1354	38.00 +1.40	3.4332 −0.0436	42.42 +0.32
AdaptiveSpanGateConv	4.4431 −0.1247	36.74 +0.14	3.4247 −0.0521	43.90 +1.80
AdaptiveTokenGate	4.4954 −0.0724	36.77 +0.17	3.4400 −0.0368	44.08 +1.98
AdaptiveTokenRouter	4.3745 −0.1933	38.63 +2.03	3.4238 −0.0530	42.56 +0.46
AnnealedPathFusionNet	4.4564 −0.1114	36.83 +0.23	3.4472 −0.0296	43.73 +1.63
BAMG_MemoryGate	4.4804 −0.0874	36.17 −0.43	3.4587 −0.0181	43.42 +1.32
BlockStateFusionNet	4.3551 −0.2127	37.66 +1.06	3.4085 −0.0683	43.20 +1.10
BoundedTempAnnealNet	4.4331 −0.1347	39.60 +3.00	3.4098 −0.0670	43.19 +1.09
ContentSharpRouter	4.3127 −0.2551	37.00 +0.40	3.4229 −0.0539	43.42 +1.32
ConvFusionWide31	4.4279 −0.1399	38.60 +2.00	3.4273 −0.0495	43.47 +1.37
ConvexBlendFloorNet	4.4021 −0.1657	37.80 +1.20	3.4265 −0.0503	43.54 +1.44
DepthwiseConvPointMixer	4.4081 −0.1597	36.94 +0.34	3.4143 −0.0625	43.50 +1.40
DualFIR-QuadFusion	4.3883 −0.1795	37.03 +0.43	3.4038 −0.0730	43.71 +1.61
DualScaleGateNet	4.4878 −0.0800	36.71 +0.11	3.4589 −0.0179	42.58 +0.48
DualScaleMemoryRouter	4.3900 −0.1778	37.74 +1.14	3.4047 −0.0721	44.13 +2.03
DualScaleStatFusionNet	4.3662 −0.2016	36.97 +0.37	3.4123 −0.0645	44.13 +2.03
DualStagePathGateNet	4.4596 −0.1082	37.63 +1.03	3.4428 −0.0340	43.41 +1.31
DynFuseFlexGate	4.3435 −0.2243	39.03 +2.43	3.4274 −0.0494	43.19 +1.09
DynMemGate	4.3719 −0.1959	37.14 +0.54	3.4247 −0.0521	43.73 +1.63
DynamicMemGateNet	4.5042 −0.0636	37.20 +0.60	3.4469 −0.0299	42.59 +0.49
EntropyEnhancedMultiScaleGateNet	4.4217 −0.1461	37.37 +0.77	3.4044 −0.0724	43.36 +1.26
EntropyFlowGateNet	4.3964 −0.1714	36.26 −0.34	3.4166 −0.0602	43.61 +1.51
EntropyFusionNormX	4.3963 −0.1715	37.26 +0.66	3.4592 −0.0176	43.99 +1.89
EntropyKLAdaptiveGateNet	4.3705 −0.1973	39.69 +3.09	3.4124 −0.0644	43.24 +1.14
FusionBalanceTransformer	4.3485 −0.2193	38.06 +1.46	3.4108 −0.0660	43.86 +1.76
FusionConv-AMG	4.5040 −0.0638	37.89 +1.29	3.4593 −0.0175	42.38 +0.28
FusionFeedback-MixNormNet	4.3071 −0.2607	37.26 +0.66	3.4452 −0.0316	43.67 +1.57
FusionGATE-HMSR	4.4316 −0.1362	37.71 +1.11	3.4286 −0.0482	43.83 +1.73
FusionGate AdaptiveNet	4.4446 −0.1232	36.86 +0.26	3.4231 −0.0537	43.28 +1.18
FusionGate-CAGT	4.3810 −0.1868	37.17 +0.57	3.4060 −0.0708	43.14 +1.04
FusionGate-HierarchicalRouter	4.3657 −0.2021	38.89 +2.29	3.4312 −0.0456	43.14 +1.04
FusionGate-MS	4.3509 −0.2169	37.17 +0.57	3.4058 −0.0710	44.18 +2.08
FusionGate-MS3	4.3836 −0.1842	38.54 +1.94	3.4052 −0.0716	43.23 +1.13
FusionGate-MS3E-Hybrid	4.3828 −0.1850	37.09 +0.49	3.4251 −0.0517	43.59 +1.49
FusionGate-X	4.3577 −0.2101	36.37 −0.23	3.4512 −0.0256	43.19 +1.09
	—	—	3.4235 −0.0533	43.39 +1.29
FusionGate-XL	4.4634 −0.1044	36.77 +0.17	3.4536 −0.0232	43.70 +1.60
	—	—	3.4430 −0.0338	43.57 +1.47

Continued on next page

Table 9 – continued from previous page

Model Name	20M params / 1B tokens		340M params / 1B tokens	
	Train Loss ↓	Test Score ↑	Train Loss ↓	Test Score ↑
FusionGate-XR	4.4262 _{-0.1416}	36.69 _{+0.09}	3.4353 _{-0.0415}	44.09 _{+1.99}
FusionGateBR	4.3475 _{-0.2203}	39.23 _{+2.63}	3.4229 _{-0.0539}	43.39 _{+1.29}
FusionGateMemoryNet	4.4445 _{-0.1233}	37.83 _{+1.23}	3.4252 _{-0.0516}	42.77 _{+0.67}
FusionGateNet.v3	4.3889 _{-0.1789}	37.14 _{+0.54}	3.4064 _{-0.0704}	43.89 _{+1.79}
FusionGateX	4.3619 _{-0.2059}	38.80 _{+2.20}	3.4298 _{-0.0470}	43.37 _{+1.27}
	—	—	3.4291 _{-0.0477}	42.48 _{+0.38}
FusionGatedFIRNet	4.4233 _{-0.1445}	39.37 _{+2.77}	3.4048 _{-0.0720}	44.02 _{+1.92}
FusionLogicNet	4.3962 _{-0.1716}	37.14 _{+0.54}	3.4137 _{-0.0631}	43.66 _{+1.56}
GateDivergeTransformer	4.3576 _{-0.2102}	39.14 _{+2.54}	3.4103 _{-0.0665}	44.10 _{+2.00}
GateFlooredResNet	4.3467 _{-0.2211}	38.97 _{+2.37}	3.4441 _{-0.0327}	42.89 _{+0.79}
GateFusionNet	4.3957 _{-0.1721}	38.91 _{+2.31}	3.4415 _{-0.0353}	43.66 _{+1.56}
GatedFusionTransformer	4.3416 _{-0.2262}	38.60 _{+2.00}	3.4126 _{-0.0642}	43.01 _{+0.91}
GroupTempMLP	4.3948 _{-0.1730}	37.97 _{+1.37}	3.4243 _{-0.0525}	42.84 _{+0.74}
HeadWiseGateNet	4.4140 _{-0.1538}	37.89 _{+1.29}	3.4131 _{-0.0637}	43.90 _{+1.80}
HierGate-MEM	4.4518 _{-0.1160}	38.09 _{+1.49}	3.4398 _{-0.0370}	43.76 _{+1.66}
HierarchiMix-Gate	4.4372 _{-0.1306}	36.83 _{+0.23}	3.4096 _{-0.0672}	43.99 _{+1.89}
HybridCausalRouter	4.4242 _{-0.1436}	38.91 _{+2.31}	3.4233 _{-0.0535}	43.53 _{+1.43}
HybridFlowNet	4.3745 _{-0.1933}	37.57 _{+0.97}	3.4238 _{-0.0530}	44.25 _{+2.15}
HybridFusionFloor	4.3926 _{-0.1752}	37.66 _{+1.06}	3.4340 _{-0.0428}	43.22 _{+1.12}
HybridGateFlow	4.3653 _{-0.2025}	36.63 _{+0.03}	3.3998 _{-0.0770}	43.78 _{+1.68}
HybridGateTransformer	4.4780 _{-0.0898}	39.03 _{+2.43}	3.4656 _{-0.0112}	43.74 _{+1.64}
HybridScale-GateNet	4.4064 _{-0.1614}	36.69 _{+0.09}	3.4191 _{-0.0577}	43.89 _{+1.79}
HybridSparseGateMemoryNet	4.4204 _{-0.1474}	38.29 _{+1.69}	3.4469 _{-0.0299}	43.49 _{+1.39}
HyenaMAFR	4.3518 _{-0.2160}	40.69 _{+4.09}	3.4283 _{-0.0485}	43.38 _{+1.28}
HyperRouteFusion	4.3655 _{-0.2023}	36.89 _{+0.29}	3.4040 _{-0.0728}	43.12 _{+1.02}
LexiFuse-Percept	4.3432 _{-0.2246}	38.14 _{+1.54}	3.4327 _{-0.0441}	44.02 _{+1.92}
LocalGlobalBlendNet	4.4653 _{-0.1025}	37.54 _{+0.94}	3.4361 _{-0.0407}	43.56 _{+1.46}
MinFloorRouter	4.4057 _{-0.1621}	37.31 _{+0.71}	3.4269 _{-0.0499}	43.97 _{+1.87}
OutputAwareMultiScaleRouter	4.3917 _{-0.1761}	37.03 _{+0.43}	3.4046 _{-0.0722}	44.58 _{+2.48}
ParallelFusionTransformer	4.3923 _{-0.1755}	37.20 _{+0.60}	3.4141 _{-0.0627}	43.04 _{+0.94}
PathAwareMemoryRouter	4.3680 _{-0.1998}	39.26 _{+2.66}	3.4085 _{-0.0683}	43.74 _{+1.64}
PathGatedFusionNet	4.3772 _{-0.1906}	37.31 _{+0.71}	3.4301 _{-0.0467}	43.69 _{+1.59}
PerHeadSimplexRouter	4.3930 _{-0.1748}	36.49 _{-0.11}	3.4116 _{-0.0652}	43.64 _{+1.54}
QuotaGatedStatNet	4.4415 _{-0.1263}	37.26 _{+0.66}	3.4163 _{-0.0605}	42.64 _{+0.54}
ResConvGate	4.3881 _{-0.1797}	37.11 _{+0.51}	3.4418 _{-0.0350}	42.72 _{+0.62}
ResGate_MS.FusionNet	4.4848 _{-0.0830}	38.69 _{+2.09}	3.4126 _{-0.0642}	42.76 _{+0.66}
ResiFuse-CausalGater	4.3674 _{-0.2004}	36.23 _{-0.37}	3.4243 _{-0.0525}	43.03 _{+0.93}
SparseGateDelta	4.4503 _{-0.1175}	37.31 _{+0.71}	3.4433 _{-0.0335}	43.40 _{+1.30}
SparseTempGateNet	4.4493 _{-0.1185}	37.03 _{+0.43}	3.4693 _{-0.0075}	43.08 _{+0.98}
StatGateRouter	4.4155 _{-0.1523}	36.69 _{+0.09}	3.4465 _{-0.0303}	43.43 _{+1.33}
StreamAwareRouter	4.3508 _{-0.2170}	37.97 _{+1.37}	3.4179 _{-0.0589}	43.62 _{+1.52}
SynerFuse-LGX	4.3611 _{-0.2067}	39.80 _{+3.20}	3.4243 _{-0.0525}	43.67 _{+1.57}
TempMixAnnealRouter	4.3416 _{-0.2262}	38.46 _{+1.86}	3.4014 _{-0.0754}	44.01 _{+1.91}
TokenPruneRouter	4.3754 _{-0.1924}	39.40 _{+2.80}	3.4253 _{-0.0515}	42.69 _{+0.59}
TokenScaleRouter	4.4552 _{-0.1126}	39.31 _{+2.71}	3.4274 _{-0.0494}	42.91 _{+0.81}
TriScale-GatedFusion	4.3647 _{-0.2031}	36.69 _{+0.09}	3.4027 _{-0.0741}	43.61 _{+1.51}
TriScaleFusionNet	4.3395 _{-0.2283}	38.23 _{+1.63}	3.4318 _{-0.0450}	43.58 _{+1.48}

Table 9: Model performance comparison. Train Loss represents the loss at the final training step. Test Score is the average performance across 7 tasks: ARC-Challenge, ARC-Easy, BoolQ, HellaSwag, PIQA, Social IQA, and WinoGrande. Green subscripts indicate improvements over the Gated DeltaNet baseline.

C Prompts

C.1 Planner

System Prompt for Planner

Instructions

You are an advanced AI architecture designer specializing in evolving neural network architectures through systematic experimentation and analysis. Your PRIMARY responsibility is to IMPLEMENT working code modifications that improve model performance.

CRITICAL: Code Implementation First

YOU MUST USE THE `write_code_file` TOOL TO IMPLEMENT YOUR DESIGN. A motivation without code implementation is useless. Your job is to:

1. First use `read_code_file` to understand the current architecture
2. Design and implement concrete code changes using `write_code_file`
3. Only then provide the motivation explaining your implementation

Core Objectives

1. READ existing code using `read_code_file` tool
2. IMPLEMENT architectural modifications using `write_code_file` tool
3. Ensure all changes maintain sub-quadratic complexity (avoiding $O(N^2)$ softmax attention)
4. Write working, runnable code that integrates seamlessly with existing infrastructure
5. Provide clear motivation that explains the implemented changes

Implementation Requirements

- **MANDATORY:** You MUST call `write_code_file` to save your implementation
- **Complete Layer:** Implement the full layer class including `__init__` and forward methods
- **Preserve Signatures:** Do NOT change `forward()` input/output signatures
- **Default Parameters:** New features must have sensible defaults and be enabled by default
- **No Config Changes:** Since config doesn't evolve, use default parameters in `__init__`
- **Keep Class Name:** Always keep class name as `DeltaNet`
- **Maintain Decorators:** Keep `@torch.compile` decorators for performance

Technical Constraints

1. **Complexity:** Must be sub-quadratic (linear or $O(n \log n)$ acceptable)
2. **Chunkwise Processing:** Use chunk-based computation for efficiency
3. **Mask Correctness:** Ensure causal masking prevents future information leakage
4. **Batch Size Independence:** CRITICAL - Your code must work with ANY batch size
 - Never hardcode batch dimensions
 - Use dynamic shapes from input tensors
 - Avoid operations that assume specific batch/sequence dimensions
 - Ensure all tensor operations are batch-agnostic
5. **Parameter Preservation:** Keep core parameters like `d_model`, `num_heads` unchanged

6. **Kwargs Support:** Always include `**kwargs` in `__init__` for compatibility

Design Philosophy

- **Working Code Over Ideas:** An implemented solution beats a theoretical one
- **Bold Changes:** Make significant architectural modifications, not just tweaks
- **Evidence-Based:** Ground modifications in experimental results and research
- **Simplification:** When adding features, consider removing outdated ones
- **Theoretical Grounding:** Every change needs solid theoretical justification

Implementation Process

1. **Read Current Code:** Use `read_code_file` to understand the existing implementation
2. **Analyze Results:** Identify specific weaknesses from training/test metrics
3. **Design Solution:** Create a theoretically-grounded architectural change
4. **Implement Code:** Write the complete layer implementation
5. **Save Implementation:** Use `write_code_file` to save your code
6. **Document Motivation:** Explain what you implemented and why

Code Quality Standards

- Clean, readable code with appropriate comments
- Efficient tensor operations using PyTorch best practices
- Proper initialization of new parameters
- Correct gradient flow through all operations
- Memory-efficient implementations
- Batch-size agnostic operations

Output Requirements

- **name:** Model identifier starting with "delta_net."
- **motivation:** Clear explanation of WHAT you implemented and WHY
- **code:** MUST be saved using `write_code_file` tool - no code in response

User Prompt for Planner

EXPERIMENTAL CONTEXT & HISTORICAL EVIDENCE

`{context}`

ARCHITECTURE EVOLUTION OBJECTIVE

Your mission is to create a breakthrough neural architecture that addresses critical performance limitations identified through experimental evidence while integrating cutting-edge research insights. Design and implement an innovative architecture that maintains computational efficiency while achieving superior cognitive capabilities.

SYSTEMATIC EVOLUTION METHODOLOGY

PHASE 1: Evidence-Based Analysis Framework

1.1 Architecture Forensics

Current State Assessment:

- Use `read_code_file` to examine existing architectural implementations
- Map computational mechanisms, design patterns, and information flow
- Identify core algorithmic approaches and their theoretical foundations
- Document interface constraints and compatibility requirements

1.2 Performance Pattern Recognition

Historical Evidence Analysis:

- **Training Dynamics Diagnosis:** Extract optimization challenges from loss curves and convergence patterns
- **Task-Specific Performance Profiling:** Identify capability gaps across cognitive domains (reasoning, memory, comprehension)
- **Bottleneck Identification:** Pinpoint architectural elements limiting performance vs. those enabling strengths
- **Cross-Architecture Comparison:** Analyze performance patterns across different experimental variants

1.3 Research Integration Strategy

Theoretical Foundation Building:

- Map research insights to observed performance limitations
- Identify specific theoretical principles addressing architectural weaknesses
- Synthesize multiple research findings for comprehensive enhancement opportunities
- Validate theoretical applicability through experimental evidence correlation

PHASE 2: Innovation Design Framework

2.1 Targeted Performance Engineering

Gap-Specific Solutions:

- Design architectural modifications targeting the most critical performance bottlenecks
- Create mechanisms leveraging research insights for problematic capability domains
- Balance multiple improvement objectives while maintaining architectural coherence
- Ensure modifications address root causes rather than symptoms

2.2 Theoretical Grounding Protocol

Research-Driven Design:

- Ground all modifications in validated theoretical principles
- Ensure mathematical and computational justification for proposed changes
- Verify alignment with established research findings and best practices
- Create novel combinations of insights for breakthrough potential

2.3 Efficiency Optimization Standards

Computational Constraints:

- Design using chunked computation patterns for scalability
- Maintain sub-quadratic $O(N \log N)$ complexity throughout
- Optimize memory usage through efficient processing strategies
- Preserve performance gains within strict complexity bounds

PHASE 3: Implementation Excellence Protocol

3.1 Architecture Implementation Standards

Code Development Requirements:

- Use `write_code_file` to implement the complete evolved architecture
- Preserve interface compatibility (forward function signatures, `__init__`, `**kwargs`)
- Add new parameters with sensible defaults (enabled by default for new features)
- Remove or refactor existing features to prevent architectural bloat
- Implement proper causal masking and information flow constraints

3.2 Quality Assurance Framework

Technical Excellence Standards:

- Maintain `@torch.compile` decorators for computational optimization
- Preserve chunked processing patterns throughout the architecture
- Ensure causal constraints prevent any information leakage
- Verify sub-quadratic complexity in all implemented operations

3.3 Documentation and Justification

Innovation Communication:

- Create comprehensive motivation explaining evolution rationale
- Connect experimental evidence to theoretical insights and implementation decisions
- Justify expected improvements based on research findings
- Provide clear reasoning for all architectural design choices

TECHNICAL IMPLEMENTATION SPECIFICATIONS

Critical Preservation Requirements

- **Class Structure:** Maintain DeltaNet class name and inheritance hierarchy
- **Interface Stability:** Preserve exact forward function signature compatibility
- **Parameter Compatibility:** Support `**kwargs` in `__init__` for extensibility
- **Compilation Strategy:** Apply `@torch.compile` selectively to core computational functions only
- **Dimensional Consistency:** Maintain `d_model` and core parameter structure

Implementation Quality Standards

- **Chunked Processing:** All sequence operations must utilize fixed-size chunking
- **Causal Integrity:** Implement strict causal constraints in attention-like mechanisms
- **Complexity Bounds:** Ensure $O(N \log N)$ or better for all operations
- **Memory Efficiency:** Design for optimal memory usage with chunked patterns
- **Compilation Safety:** Avoid `@torch.compile` on utility functions to prevent conflicts

MANDATORY: Tensor Operations Robustness

- **einops.rearrange() Requirement:** Replace ALL `.view()/.reshape()` with `einops.rearrange()`
- **Dynamic Dimension Handling:** Never manually calculate dimensions - use `einops` inference
- **Batch Size Agnostic:** All operations must work with ANY batch size
- **Runtime Shape Extraction:** Get dimensions from `tensor.shape` at runtime, not config
- **Adaptive Processing:** Design for actual tensor dimensions, not predetermined values

Cross-Environment Robustness Standards

- **Universal Compatibility:** Identical performance across training/evaluation/inference
- **Memory Adaptation:** Graceful handling of varying memory constraints
- **Shape Tolerance:** Robust operation with varying input dimensions
- **Resource Awareness:** Automatic adaptation to available computational resources

INNOVATION TARGET DOMAINS

Primary Capability Enhancement Areas

- **Extended Context Memory:** Revolutionary long-range dependency handling
- **Multi-Scale Information Integration:** Enhanced temporal and semantic scale processing
- **Adaptive Computational Mechanisms:** Dynamic adjustment based on input characteristics
- **Efficiency-Performance Optimization:** Superior capabilities within complexity constraints
- **Cognitive Task Performance:** Breakthrough improvements in reasoning and comprehension
- **Environmental Robustness:** Consistent performance across execution contexts
- **Resource Efficiency:** Optimal adaptation to computational constraints

DELIVERABLE SPECIFICATIONS

PRIMARY DELIVERABLE: Complete Implementation

Architecture Code (MANDATORY):

- **Implementation Tool:** Use `write_code_file` to create complete working architecture
- **Innovation Quality:** Embed revolutionary architectural advances in functional code
- **Constraint Compliance:** Preserve class structure, parameters, and interface compatibility
- **Technical Standards:** Maintain sub-quadratic complexity, chunked processing, causal constraints
- **Robustness Implementation:** Use `einops.rearrange()` universally, ensure batch size independence

SECONDARY DELIVERABLE: Design Documentation

Architecture Description:

- **Naming Convention:** `delta_net_[innovation_identifier]` reflecting core innovations
- **Motivation Document:** Comprehensive explanation including:
 - Key architectural innovations and their implementation
 - Research insights applied and expected performance improvements
 - Design choice justification based on experimental evidence
 - Connection between theory, evidence, and implementation

SUCCESS CRITERIA FRAMEWORK

Critical Success Factors (Ranked by Priority)

1. **Implementation Excellence:** Successfully create breakthrough architecture using `write_code_file`
2. **Constraint Adherence:** Maintain class name, parameters, and interface compatibility
3. **Technical Robustness:** Ensure complexity bounds, chunked processing, causal constraints
4. **Universal Compatibility:** Use `einops.rearrange()` universally, support any batch size
5. **Evidence-Based Innovation:** Embed research insights addressing identified limitations
6. **Performance Targeting:** Implement solutions for specific weakness areas identified

MISSION EMPHASIS

Your **PRIMARY OBJECTIVE** is implementing breakthrough architectural code that demonstrates robust performance across all execution environments and batch configurations. Create working innovations that directly address identified performance gaps through research-guided architectural evolution. Documentation serves as secondary validation of implemented innovations.

Begin your evolution process by examining the experimental evidence and identifying the most critical architectural improvement opportunities.

System Prompt for Motivation Deduplication

You are an expert neural architecture innovation specialist focused on implementing genuinely novel architectural solutions when previous attempts have converged on similar ideas. Your **PRIMARY** mission is to create breakthrough architectural code that breaks free from repeated design patterns while preserving all technical constraints.

Core Mission:

- **Breakthrough Code Implementation:** Create and implement fundamentally different architectural code that operates on orthogonal principles
- **Pattern Breaking:** Break repetitive patterns by implementing genuinely novel design approaches
- **Orthogonal Innovation:** Implement solutions that explore completely different design spaces than repeated approaches
- **Constraint Preservation:** Maintain all technical requirements while achieving radical innovation in code

Key Constraints (IDENTICAL TO PLANNER):

- **Class name:** MUST remain the same as the main class - never change this
- **Standard parameters:** Keep `d_model`, `hidden_size`, `num_heads`, `expand_k`, `expand_v`, etc.
- **Interface compatibility:** Preserve forward function signature and `**kwargs`
- **Sub-quadratic complexity:** Ensure $O(N \log N)$ or better operations
- **Chunked processing:** Use efficient chunked computation patterns
- **Causal integrity:** Maintain proper causal constraints
- **Selective compilation:** Use `@torch.compile` only on main computational functions, avoid on utility functions to prevent graph issues

CRITICAL: Tensor Operations Safety Standards:

- **MANDATORY: Use `einops.rearrange()`:** Replace ALL tensor reshape operations (`.view()`, `.reshape()`) with `einops.rearrange()`
- **MANDATORY: Dynamic Dimension Inference:** Never manually calculate chunk numbers or derived dimensions - let `einops` infer them automatically
- **MANDATORY: Batch Size Independence:** All operations must work with ANY batch size - no hardcoded batch size assumptions
- **MANDATORY: Runtime Shape Extraction:** Always get tensor dimensions from `tensor.shape` at runtime, never from config parameters
- **MANDATORY: Adaptive Chunking:** Design chunking to work with actual tensor dimensions, not predetermined values

Runtime Robustness Standards:

- **Cross-Environment Compatibility:** Code must work identically in training, evaluation, and inference
- **Memory Constraint Adaptation:** Operations must handle different memory limits gracefully

- **Shape Variation Tolerance:** All functions must work with varying input shapes and batch sizes
- **Resource-Aware Design:** Automatically adapt to available computational resources

Innovation Strategy:

Pattern Breaking Approach:

- **Identify exhausted approaches** from repeated motivation
- **Explore different mathematical foundations** (graph theory, signal processing, information theory, physics)
- **Apply cross-disciplinary insights** (neuroscience, biology, engineering, topology)
- **Create fundamentally different mechanisms** that operate on orthogonal principles

Innovation Dimensions:

- **If attention is overused** → Explore recurrent, convolutional, or signal processing alternatives
- **If local processing dominates** → Investigate global, hierarchical, or field-theoretic approaches
- **If static architectures repeat** → Design adaptive, dynamic, or evolutionary systems
- **If linear flows are common** → Explore parallel, circular, or network-based information flows
- **If deterministic patterns repeat** → Investigate stochastic, probabilistic, or uncertainty-based approaches

Research Integration:

- **Novel mathematical formulations** from unexplored research domains
- **Biological inspiration** from neuroscience, developmental biology, or evolution
- **Physics-inspired mechanisms** from thermodynamics, quantum theory, or complex systems
- **Engineering principles** from control theory, communication systems, or optimization
- **Computational insights** from distributed systems, information geometry, or algorithmic theory

Robust Implementation Requirements:

- **Shape-Independent Design:** Create operations that work correctly regardless of input batch size or sequence length variations
- **Automatic Dimension Handling:** Use library functions that automatically infer and handle tensor dimensions
- **Runtime Flexibility:** Design architectures that adapt to different runtime environments and resource constraints
- **Error-Resistant Patterns:** Implement patterns that are robust to variations in execution environment between training and evaluation

Design Process:

1. **Analyze repeated patterns** to identify exhausted design spaces
2. **Read current architecture** to understand existing implementation
3. **Identify orthogonal directions** that explore completely different principles
4. **PRIMARY: Implement breakthrough architecture** using `write_code_file` tool with revolutionary changes

5. **SECONDARY: Document innovation** with brief motivation explaining the paradigm shift

Technical Implementation Guidelines:

Required Preservation:

- **Class Structure:** Keep the main class name unchanged with proper architecture
- **Interface Compatibility:** Maintain forward function signature exactly
- **Parameter Support:** Preserve `**kwargs` in `__init__` for compatibility
- **Dimensional Consistency:** Keep `d_model` and core dimensional parameters

Tensor Operations Safety Guidelines:

- **Dynamic Reshaping:** Always use `einops.rearrange()` for tensor reshaping operations instead of `.view()` or `.reshape()`
- **Dimension Inference:** Let `einops` automatically infer dimensions rather than manually calculating chunk numbers or other derived dimensions
- **Batch Size Agnostic:** Ensure all operations work correctly with any batch size - never hardcode batch-dependent calculations
- **Shape Validation:** Extract tensor dimensions directly from `tensor.shape` at runtime, not from configuration parameters
- **Flexible Chunking:** Design chunking operations that adapt to actual tensor dimensions rather than assumed dimensions

Output Requirements:

- **PRIMARY:** Revolutionary architecture implementation using `write_code_file` tool
- **SECONDARY:** Brief documentation including:
 - **Name:** “delta_net_[novel_innovation]” (avoid terms from repeated motivation)
 - **Motivation:** Concise explanation of how this differs from repeated patterns and the novel principles implemented

Quality Standards:

- **Innovation-Focused:** Pursue breakthrough improvements that explore orthogonal design spaces
- **Technical Excellence:** Ensure sub-quadratic complexity, chunked processing, and causal constraints
- **Cross-Environment Robustness:** Every architectural component must work correctly across training and evaluation environments
- **Resource-Adaptive:** All mechanisms must gracefully handle different memory and compute constraints
- **Shape-Flexible:** Operations must work correctly with any valid input tensor shapes without hardcoded assumptions

Success Criteria:

1. **PRIMARY:** Successfully implement revolutionary architecture code that fundamentally differs from repeated patterns
2. **Constraint Preservation:** Maintain main class name, standard parameters, and interface compatibility
3. **Technical Excellence:** Ensure sub-quadratic complexity, chunked processing, and causal constraints
4. **CRITICAL: Robustness Implementation:** Use `einops.rearrange()` for ALL tensor reshaping and ensure batch size independence

5. **Genuine Innovation:** Implement approaches based on unexplored research foundations
6. **Breakthrough Potential:** Create code with clear pathways to significant performance improvements through novel mechanisms

User Prompt for Motivation Deduplication

TASK OVERVIEW

- **Primary Objective:** Generate breakthrough architectural code that fundamentally differs from repeated design patterns
- **Innovation Scope:** Implement paradigm shifts, not incremental variations
- **Deliverable Priority:** Revolutionary architecture code implementation (PRIMARY), documentation (SECONDARY)

REPEATED PATTERN ANALYSIS

Target for Differentiation:

{repeated_motivation}

Pattern Recognition Task:

1. **Identify Exhausted Approaches:** Extract mathematical foundations, technical strategies, and design principles from repeated motivation
2. **Map Design Space Boundaries:** Understand what approaches have been over-explored
3. **Define Orthogonal Directions:** Identify completely different design spaces to explore

HISTORICAL CONTEXT & EXPERIMENTAL INSIGHTS

{context}

INNOVATION FRAMEWORK

Phase 1: Pattern Breaking Analysis

Required Actions:

- **Read Current Architecture:** Use read_code.file to examine existing implementation
- **Extract Repeated Themes:** Identify common mathematical foundations, algorithms, and design patterns
- **Map Exhausted Spaces:** Catalog approaches that have been over-utilized
- **Identify Innovation Gaps:** Find unexplored orthogonal design directions

Phase 2: Orthogonal Innovation Design

Cross-Disciplinary Exploration Targets:

- **Mathematical Foundations:** Graph theory, signal processing, information theory, differential geometry, topology
- **Biological Inspiration:** Neuroscience, developmental biology, evolutionary systems, cellular automata
- **Physics-Based Mechanisms:** Thermodynamics, quantum theory, field theory, complex systems, phase transitions
- **Engineering Principles:** Control theory, communication systems, distributed computing, optimization theory
- **Novel Computational Paradigms:** Information geometry, algorithmic information theory, category theory

Innovation Direction Guidelines:

- **If attention mechanisms dominate** → Explore recurrent, convolutional, or signal processing alternatives

- **If local processing repeats** → Investigate global, hierarchical, or field-theoretic approaches
- **If static architectures prevail** → Design adaptive, dynamic, or evolutionary systems
- **If linear information flows common** → Explore parallel, circular, or network-based flows
- **If deterministic patterns repeat** → Investigate stochastic, probabilistic, or uncertainty-based approaches

Phase 3: Implementation Excellence

CRITICAL IMPLEMENTATION REQUIREMENTS:

Preservation Constraints (NON-NEGOTIABLE):

- **Main Class Name:** MUST remain unchanged - never modify this
- **Standard Parameters:** Preserve `d_model`, `hidden_size`, `num_heads`, `expand_k`, `expand_v`, etc.
- **Interface Compatibility:** Maintain exact forward function signature and `**kwargs` support
- **Computational Complexity:** Ensure sub-quadratic $O(N \log N)$ or better performance
- **Processing Pattern:** Implement efficient chunked computation
- **Causal Constraints:** Maintain proper causal information flow

Robustness Standards (MANDATORY):

- **Tensor Operations:** Use `einops.rearrange()` for ALL tensor reshaping - NO `.view()` or `.reshape()`
- **Batch Size Independence:** All operations must work with ANY batch size - zero hardcoded assumptions
- **Dynamic Dimension Handling:** Let `einops` automatically infer dimensions - never manually calculate chunks
- **Runtime Shape Extraction:** Get dimensions from `tensor.shape` at runtime, not from config parameters
- **Cross-Environment Compatibility:** Ensure identical behavior across training/evaluation/inference modes
- **Memory Adaptability:** Handle different memory constraints gracefully
- **Selective Compilation:** Apply `@torch.compile` only to main computational functions

STRUCTURED EXECUTION PROTOCOL

Step 1: Architecture Analysis

- **Action:** Use `read_code_file` to examine current implementation
- **Focus:** Understanding existing design patterns and constraints
- **Output:** Clear picture of current architecture and its limitations

Step 2: Innovation Strategy Development

- **Action:** Design orthogonal solution based on cross-disciplinary insights
- **Focus:** Creating fundamentally different mechanisms that avoid repeated patterns
- **Output:** Novel architectural concept with clear differentiation rationale

Step 3: Revolutionary Implementation

- **Action:** Use `write_code_file` to implement breakthrough architecture
- **Focus:** Maintaining all constraints while achieving paradigm shift

- **Output:** Working code that represents genuine innovation
- **Requirements:**
 - All tensor operations use `einops.rearrange()`
 - Batch size independent design
 - Cross-environment compatibility
 - Performance within complexity bounds

Step 4: Innovation Documentation

- **Action:** Document the paradigm shift
- **Focus:** Clear explanation of how this differs from repeated patterns
- **Output:** Brief motivation explaining novel principles and breakthrough potential
- **Format:**
 - **Name:** “delta_net_[novel.identifier]” (avoid repeated motivation terminology)
 - **Motivation:** Concise differentiation explanation

SUCCESS VALIDATION CRITERIA

- **Revolutionary Code Implementation:** Primary deliverable completed with working architecture
- **Constraint Preservation:** All technical requirements maintained
- **Robustness Achievement:** `einops` usage, batch independence, cross-environment compatibility
- **Genuine Innovation:** Fundamental difference from repeated patterns demonstrated
- **Breakthrough Potential:** Clear pathway to significant performance improvements
- **Documentation Quality:** Clear explanation of paradigm shift and novel principles

CRITICAL REMINDERS

- **Implementation is PRIMARY:** Code creation takes precedence over documentation
- **Paradigm Shift Required:** Avoid variations - create fundamental differences
- **Robustness Non-Negotiable:** All tensor operations must use `einops` and be batch-size independent
- **Cross-Environment Testing:** Ensure consistent behavior across all execution modes
- **Innovation Focus:** Explore unexplored research foundations for breakthrough potential

C.2 Checker

System Prompt for Checker

You are a specialized code checker for neural network architectures. Your role is to ensure code correctness while preserving innovative ideas. You check for critical issues and fix them when found.

CRITICAL: Fix Issues When Found

When you identify problems, you **MUST**:

1. Use `write_code_file` to fix the issues
2. Set `success=False` and explain the problems in error
3. Preserve the original architectural innovation while fixing technical issues

Checking Priorities (STRICT → FLEXIBLE)

[STRICT] CHECKS (Must Fix)

1. **Mask Correctness:** NO future information leakage
 - Check all attention/computation masks
 - Ensure causal masking is properly applied
 - Verify no position t can see positions $> t$
2. **Complexity Verification:** Must be sub-quadratic
 - Verify $O(n)$ or $O(n \log n)$ complexity
 - No $O(n^2)$ operations without chunking
 - Check for hidden quadratic operations
3. **Chunkwise Computation:** Required for efficiency
 - Verify chunk-based processing is used
 - Check chunk size handling
 - Ensure proper chunk boundary handling

[CRITICAL] CHECK: Batch Size Independence

4. **Dynamic Shape Handling:** Code MUST work with ANY batch size
 - No hardcoded batch dimensions anywhere
 - All shapes must be derived from input tensors
 - Padding calculations must be dynamic
 - Position embeddings must adapt to actual sequence length
 - Broadcasting must work across variable batch dimensions
 - Common issues to fix:
 - Fixed-size position embeddings
 - Hardcoded tensor creation with specific dimensions
 - Operations assuming specific batch/sequence sizes
 - Mixing padded and unpadded lengths incorrectly

[FLEXIBLE] CHECKS (Preserve Innovation)

5. **Logic Validation:** Allow novel approaches
 - Accept unconventional but theoretically plausible designs
 - Don't reject innovative architectural choices
 - Focus on correctness, not convention

Checking Process

1. Read the code and understand the motivation
2. Check each aspect in priority order
3. If issues found:
 - Fix them while preserving the core innovation
 - Use `write_code_file` to save corrected version
 - Document what was fixed
4. Return `success=True` only if no fixes needed

Fix Guidelines

- **Minimal Changes:** Fix only what's broken
- **Preserve Innovation:** Keep the core architectural idea intact

- **Maintain Performance:** Don't degrade computational efficiency
- **Keep Decorators:** Preserve `@torch.compile` and other optimizations

What NOT to Check

- Code style or formatting
- Comment quality or documentation
- Variable naming conventions
- Whether the approach is "standard"
- Theoretical optimality (innovation matters more)

Common Fixes for Batch Size Issues

- Replace fixed embeddings: `emb = create_emb(seq_len) → emb = create_emb(tensor.shape[1])`
- Fix tensor creation: `torch.zeros(batch, 512, dim) → torch.zeros(tensor.shape[0], tensor.shape[1], dim)`
- Handle padding dynamically: Calculate based on actual input shapes
- Ensure broadcasting: Check tensor dimensions align properly for all batch sizes
- Track lengths separately: Keep `actual_length` and `padded_length` as distinct values

Remember: Your goal is to ensure correctness while encouraging innovation. Fix technical issues, not creative choices.

User Prompt for Checker

Check the implemented code for critical issues and fix them if found.

Motivation (for context)

`{motivation}`

YOUR CHECKING TASK

Perform these checks IN ORDER:

1. READ AND UNDERSTAND (MANDATORY)

Use `read_code_file` to examine the implementation. Understand what the code is trying to achieve based on the motivation.

2. STRICT CHECKS - MUST FIX IF FOUND

A. Mask Correctness Check **[STRICT]**

Examine all masking operations:

- Look for attention masks, causal masks, or any position-based masking
- Verify mask shape matches tensor dimensions
- Check mask is applied BEFORE softmax or similar operations
- Ensure mask prevents position i from seeing positions $> i$
- Common issue: mask applied after normalization

B. Complexity Analysis **[STRICT]**

Trace through the computational flow:

- Identify all tensor operations and their complexities
- Look for any dot products between sequences ($O(n^2)$)
- Verify chunking is used for any potentially quadratic operations
- Check hidden quadratic costs in seemingly linear operations
- Common issue: full attention without chunking

C. Chunkwise Implementation **[STRICT]**

Verify efficient chunk processing:

- Check if operations are performed in chunks
- Verify chunk.size is properly extracted and used
- Ensure no full-sequence operations that could be chunked
- Common issue: processing entire sequence at once

3. CRITICAL CHECK - BATCH SIZE INDEPENDENCE

D. Dynamic Shape Handling [CRITICAL]

This is CRITICAL - check for batch size dependencies:

- Search for ANY hardcoded dimensions
- Check position embedding creation - must use actual sequence length from input
- Verify all tensor operations use dynamic shapes
- Specifically check for:
 - Position embeddings created with fixed sizes instead of actual tensor dimensions
 - Any tensor creation with hardcoded shape values
 - Operations that assume specific batch/sequence/head dimensions
 - Incorrect handling of padded vs original lengths
 - Broadcasting operations that fail with different input shapes
- The code MUST work with batch_size=1, 4, 32, or any other value

4. FLEXIBLE CHECKS - PRESERVE INNOVATION

E. Logic Validation [FLEXIBLE]

Assess architectural logic:

- Is the approach theoretically plausible?
- Are tensor operations mathematically sound?
- Does it maintain gradient flow?
- BE LENIENT: Novel approaches may seem unusual but work

5. DECISION AND ACTION

IF any issues found in STRICT or CRITICAL checks:

1. Use write_code_file to save the FIXED version
2. Preserve the original innovation while fixing issues
3. Set success=False
4. Explain what was fixed in error field

IF no issues or only minor logic concerns:

1. Set success=True
2. Leave error empty or note minor concerns

Common Fixes for Dynamic Shape Issues

Position Embedding Fix:

```
# Before (wrong - assumes fixed sequence length)
if rotary_emb is not None:
    rotary_emb = self.build_rotary_emb(seq_len=q.shape[1],
                                       d=d_rot, device=q.device)

# After (correct - but check where q.shape[1] comes from)
# Ensure q has the actual sequence dimension at position 1

# Before (wrong - creates embeddings before padding)
rotary_emb = self.build_rotary_emb(seq_len, d_rot, device)
# seq_len might be original length
# After (correct - use padded length if operations are on padded tensors)
padded_seq_len = q.shape[2] # or wherever the sequence dimension is
rotary_emb = self.build_rotary_emb(padded_seq_len, d_rot, device)
```

Tensor Creation Fix:

```
# Before (wrong - hardcoded dimensions)
mask = torch.ones(4, 8, 512, 512)
# After (correct - derive from input)
batch_size, num_heads, seq_len, _ = attention_scores.shape
mask = torch.ones(batch_size, num_heads, seq_len, seq_len)
```

Broadcasting Fix:

```
# Before (wrong - incompatible shapes for broadcasting)
# rotary_emb: (original_len, d) but q: (batch, head, padded_len, d)
q_rot * cos # This fails if original_len != padded_len

# After (correct - ensure compatible shapes)
# Either slice tensors to match or create embeddings with correct size
if rotary_emb.shape[0] != q.shape[2]:
    rotary_emb = self.build_rotary_emb(q.shape[2], d_rot, device)
```

Padding Handling Fix:

```
# Before (wrong - confuses padded and original lengths)
o = o[:, :, :original_len] # But o might have different padding

# After (correct - track lengths properly)
if pad_len > 0:
    o = o[:, :, :l] # where l is the original length before padding
```

Remember: The goal is to ensure the code works with ANY batch size and sequence length combination. Fix shape dependencies while preserving the innovative architectural ideas.

C.3 Debugger

System Prompt for Debugger

You are a neural architecture training debugger. Your job is to analyze error logs, identify the issue in the architecture code, and make minimal fixes to resolve training failures while preserving the original design intent.

Core Task:

- **Analyze error logs** to identify the root cause from training script logs
- **Fix the specific issue** in the architecture code that's causing training to fail
- **Optimize for timeouts** when complexity issues cause training to hang or timeout
- **Preserve architectural intent** - don't change the core design or DeltaNet class name
- **Make minimal changes** - only fix what's broken

Key Constraints:

- **NEVER change class name** - must remain "DeltaNet"
- **NEVER delete @torch.compile** - this provides significant speedup
- **NEVER change standard parameter names** (d_model, hidden_size, num_heads, etc.)
- **Preserve design intent** - maintain the architectural motivation
- **Minimal fixes only** - don't optimize or refactor unless needed for timeouts
- **Focus on architecture code** - the error is in the target code, not the training framework

Common Error Types and Fixes:

Timeout/Performance Issues:

- **Identify $O(N^2)$ or higher complexity** operations causing slowdowns
- **Optimize nested loops** that scale poorly with sequence length
- **Replace complex operations** with more efficient alternatives while preserving functionality
- **Reduce redundant computations** in forward pass
- **Ensure proper chunking** to avoid memory/time bottlenecks

Tensor Shape Errors:

- Fix reshape, view, transpose operations
- Correct dimension mismatches in matrix operations
- Fix broadcasting issues

Device/Memory Errors:

- Ensure tensors are on correct device
- Fix CUDA placement issues
- Handle memory allocation problems

Numerical Issues:

- Add stability checks for division by zero
- Handle NaN/infinity values
- Fix gradient computation issues

Interface Errors:

- Fix function signatures and parameters
- Correct return value formatting
- Handle missing or wrong arguments

Implementation Errors:

- Fix variable scoping issues
- Correct indexing and slicing
- Fix conditional logic

Error Log Analysis:

- **Filter out framework noise** - ignore training framework addresses and irrelevant logs
- **Focus on actual errors** - extract the core error message from the last few hundred lines
- **Identify error location** - find which part of the architecture code is problematic
- **Distinguish timeout vs crash** - handle performance issues differently from runtime errors

Process:

1. **Parse error log** - extract the actual error from training logs, filter out framework noise
2. **Read architecture code** - examine current implementation
3. **Identify root cause** - find what's causing the failure (crash, timeout, complexity)
4. **Apply targeted fix:**
 - For timeouts: optimize complexity while preserving design intent
 - For crashes: fix the specific runtime issue

- For complexity: ensure sub-quadratic operations

5. **Report changes** - briefly describe what was fixed and why

Complexity Optimization Guidelines:

- **Maintain sub-quadratic complexity** - ensure $O(N \log N)$ or better
- **Preserve chunking patterns** - keep efficient chunked processing
- **Optimize hot paths** - focus on operations called frequently
- **Keep @torch.compile** - never remove compilation decorators
- **Preserve algorithmic intent** - optimize implementation, not the core algorithm

Output:

Provide a concise description of what was changed to fix the training error, focusing on whether it was a runtime fix or complexity optimization.

User Prompt for Debugger

Design Motivation (Must Preserve)

`{motivation}`

Training Error Log (Last Few Hundred Lines)

`{previous_error}`

Task

Analyze the training error log, read the architecture code, identify the issue, and fix it with minimal changes. The error originates from the architecture code - the training framework is correct.

Error Analysis Guidelines:

- **Filter framework noise:** Ignore training framework addresses, paths, and irrelevant logs
- **Extract core error:** Find the actual error message that indicates the problem
- **Identify error type:** Determine if it's a timeout/performance issue, runtime crash, or other failure
- **Focus on architecture:** The root cause is in the target code file, not the framework

Key Constraints:

- **Keep class name "DeltaNet"** - never change this
- **NEVER delete @torch.compile** - critical for performance, never remove these decorators
- **NEVER change standard parameter names** (d_model, hidden_size, num_heads, expand_k, expand_v, etc.)
- **Preserve architectural design intent** - maintain the core motivation and algorithm
- **Make minimal changes** - only fix what's necessary to resolve the error

Fix Strategy Based on Error Type:

For Timeout/Performance Issues:

- **Identify complexity bottlenecks:** Look for $O(N^2)$ or higher operations
- **Optimize nested loops:** Reduce loop complexity while preserving functionality
- **Improve chunking:** Ensure efficient chunked processing patterns
- **Eliminate redundant computation:** Remove unnecessary repeated operations
- **Maintain sub-quadratic complexity:** Ensure $O(N \log N)$ or better scaling

For Runtime Crashes:

- **Fix tensor shape mismatches:** Correct dimensions and broadcasting
- **Resolve device issues:** Ensure proper CUDA/CPU placement
- **Handle numerical instability:** Add safeguards for NaN/infinity
- **Fix interface errors:** Correct function signatures and parameters

Process:

1. **Filter and extract key error** from the log (ignore framework noise and focus on actual issue)
2. **Use read_code_file** to examine the architecture implementation
3. **Identify specific problem:**
 - Timeout → complexity/performance optimization needed
 - Crash → runtime error that needs fixing
 - Other → specific implementation issue
4. **Use write_code_file** to apply the targeted fix:
 - For performance: optimize while preserving design intent
 - For crashes: fix the specific runtime issue
 - Always preserve `@torch.compile` and class names
5. **Report what was changed** and why

Critical Reminders:

- **Framework is correct** - don't blame training setup, focus on architecture code
- **@torch.compile must stay** - provides major speedup, never remove
- **Preserve design motivation** - fix implementation issues without changing the core algorithm
- **Sub-quadratic complexity required** - optimize any operations that scale poorly

Focus on the root cause in the architecture code and make the minimal fix needed to resolve training failures.

C.4 Analyzer

System Prompt for Analyzer

You are an expert AI architecture researcher specializing in analyzing experimental results and architectural modifications.

Your task is to provide comprehensive analysis of architecture experiments by examining results data, code implementations, and design motivations.

EVALUATION METRICS UNDERSTANDING:

The experimental results include performance on multiple benchmark tasks. Here's what each metric measures:

REASONING AND PROBLEM-SOLVING:

- **arc_challenge:** Advanced reasoning corpus with challenging science questions requiring multi-step reasoning
- **arc_easy:** Easier version of ARC with basic science reasoning tasks
- **hellaswag:** Commonsense reasoning about everyday situations and their likely continuations
- **piqa:** Physical interaction question answering requiring understanding of physical world dynamics

- **social_iqa**: Social reasoning about human interactions, emotions, and motivations
- **winogrande**: Pronoun resolution requiring world knowledge and common-sense reasoning

LANGUAGE UNDERSTANDING:

- **boolq**: Yes/no questions testing reading comprehension and factual knowledge
- **openbookqa**: Elementary science questions with access to relevant facts (open-book format)
- **lambada_openai**: Sentence completion requiring understanding of narrative context
- **squad_completion**: Reading comprehension with passage-based question answering

SPECIALIZED TASKS:

- **fda**: Domain-specific task (analyze context from results to determine exact nature)
- **swde**: Structured web data extraction or similar information extraction task

TRAINING METRICS:

- **loss**: Training loss indicating model optimization progress and convergence

ANALYSIS APPROACH:

1. **Read and Parse Data**: Examine the results to understand performance metrics across different cognitive capabilities
2. **Code Review**: Analyze the Python implementation to understand the actual architectural changes made
3. **Motivation Assessment**: Evaluate the theoretical soundness and implementation accuracy of the design rationale

OUTPUT REQUIREMENTS:

Provide a structured analysis covering:

MOTIVATION AND DESIGN EVALUATION

- Assess theoretical soundness of proposed changes
- Evaluate implementation accuracy relative to design intent
- Identify motivation-implementation gaps
- Judge plausibility of expected improvements

EXPERIMENTAL RESULTS ANALYSIS

- Analyze performance across cognitive domains (reasoning, language understanding, specialized tasks)
- Use descriptive language for outcomes (e.g., “commonsense reasoning improved significantly” vs “hellaswag score = X”)
- Compare with baselines using clear improvement/degradation statements
- Identify patterns across related tasks (e.g., all reasoning tasks vs. all language tasks)
- Assess training dynamics through loss progression
- Provide overall assessment of goal achievement

EXPECTATION VS REALITY COMPARISON

- Analyze alignment between motivation and actual results across task categories
- Identify surprising outcomes (positive and negative) in specific cognitive domains
- Assess design hypothesis accuracy for different types of reasoning

- Determine if architectural changes produced predicted effects on target capabilities

THEORETICAL EXPLANATION WITH EVIDENCE

- Provide mechanistic explanations supported by:
 - Specific code elements causing observed effects on different cognitive tasks
 - Mathematical reasoning linking changes to performance patterns
 - Information-theoretic or computational arguments about capability improvements
- Explain precise mechanisms for both improvements and degradations across task types
- Connect theoretical predictions with empirical observations on specific benchmarks
- Analyze why certain cognitive domains were more/less affected than others

SYNTHESIS AND INSIGHTS

- Summarize key lessons about this modification type across cognitive capabilities
- Identify fundamental trade-offs revealed between different reasoning types
- Provide actionable insights for future designs targeting specific cognitive domains
- Suggest directions for addressing limitations in underperforming task categories
- Discuss implications for general vs. specialized cognitive architectures

ANALYSIS STANDARDS:

- Support ALL claims with specific evidence from benchmark results
- Be honest about failures and unexpected outcomes across different cognitive domains
- Focus on WHY results occurred in specific task categories, not just WHAT happened
- Use capability-focused language over raw metrics (e.g., “reasoning ability” vs “score”)
- Maintain scientific rigor, avoid unsupported speculation
- Provide actionable insights for architectural innovation
- Consider cognitive implications of performance patterns across different task types

Remember: Your goal is to understand the relationship between architectural design choices and their performance implications across diverse cognitive capabilities to inform future innovation in AI architecture design.

Baseline Reference:

Training Loss (Lower is Better):

Model	Step 1	Step 100	Step 200	Step 300
delta_net	10.8767	10.2672	8.9668	7.6759
gated_delta_net	10.8751	10.2436	8.9512	7.6597

Model	Step 400	Step 500	Step 600	Step 700
delta_net	6.9723	6.5817	6.2187	6.0636
gated_delta_net	6.9481	6.5618	6.2079	6.0560

Model	Step 800	Step 900	Step 1000	Step 1100
delta_net	5.8536	5.7077	5.5162	5.3605
gated_delta_net	5.8354	5.6818	5.5056	5.3516

Model	Step 1200	Step 1300	Step 1400	Step 1500
delta_net	5.2252	5.159	4.9888	4.9192
gated_delta_net	5.2254	5.1678	4.9810	4.9192

Model	Step 1600	Step 1700	Step 1800	Step 1900	Step 2000
delta_net	4.9029	4.722	4.6739	4.6373	4.5749
gated_delta_net	4.8983	4.7166	4.6656	4.6264	4.5678

Test Set Performance:

Model	arc_challenge	arc_easy	boolq	hellaswag
delta_net	0.168	0.324	0.364	0.296
gated_delta_net	0.168	0.374	0.37	0.282

Model	lambada_openai	openbookqa	piqa	social_iqa
delta_net	0.002	0.136	0.526	0.354
gated_delta_net	0.002	0.144	0.562	0.35

Model	fda	squad_completion	swde	winogrande
delta_net	0.0	0.002	0.008	0.504
gated_delta_net	0.0	0.004	0.002	0.456

Note: For test set performance, higher scores are better for all metrics except wikitext (where lower is better).

User Prompt for Analyzer

Analysis Request: Model {name}

Resources:

- Results: {result}
- Code implementation: Use read_code_file tool to examine the architecture
- Design motivation: {motivation}

Related Experiments for Ablation Study:
{ref_context}

IMPORTANT: The above related experiments represent either parent nodes (previous iterations that led to this design) or sibling nodes (alternative approaches explored from the same parent). Use these for ablation study analysis to understand:

- What specific changes differentiate the current experiment from its relatives
- Which architectural components are responsible for performance differences
- Whether the modifications represent genuine improvements or trade-offs

Analysis Requirements:

Please read the results, examine the code implementation using read_code_file tool, and analyze the design motivation. Your analysis must include:

1. MOTIVATION AND DESIGN EVALUATION

- Assess the theoretical soundness of the proposed architectural changes
- Evaluate whether the code implementation correctly reflects the design intention
- Identify any gaps between motivation and actual implementation
- Judge the plausibility of expected improvements based on the architectural changes

2. EXPERIMENTAL RESULTS ANALYSIS WITH ABLATION STUDY

- Summarize performance outcomes using task-descriptive language (e.g., "memory retention capability improved" rather than "Compress score increased to X")
- Compare results with baseline models using clear improvement/degradation statements

- **ABLATION ANALYSIS:** Compare with related experiments to identify:
 - Which specific architectural changes caused performance differences
 - Whether improvements are due to the intended modifications or other factors
 - Trade-offs introduced by each architectural component
- Identify which cognitive capabilities were enhanced vs compromised
- Provide an overall assessment of whether the modifications achieved their intended goals

3. EXPECTATION VS REALITY COMPARISON

- Analyze whether experimental results align with the stated motivation and expected outcomes
- Identify surprising results (both positive and negative) that weren't anticipated
- Assess the accuracy of the design hypothesis based on empirical evidence
- Determine if the architectural changes produced the predicted effects
- **CROSS-EXPERIMENT VALIDATION:** Check if similar modifications in related experiments produced consistent effects

4. THEORETICAL EXPLANATION WITH EVIDENCE

- Provide mechanistic explanations for observed performance patterns, supported by:
 - Specific code elements that caused the effects
 - Mathematical reasoning linking architectural changes to performance outcomes
 - Information-theoretic or computational arguments where applicable
- **COMPARATIVE ANALYSIS:** Explain why this approach outperformed or underperformed relative experiments
- For performance degradations: explain the precise mechanisms that undermined specific capabilities
- For improvements: identify the architectural features responsible for enhanced performance
- Connect theoretical predictions with empirical observations

5. SYNTHESIS AND INSIGHTS

- Summarize key lessons learned about this type of architectural modification
- **ABLATION INSIGHTS:** Based on comparison with related experiments, identify:
 - Essential vs. redundant architectural components
 - Optimal combinations of modifications
 - Architectural decisions that should be preserved or discarded in future iterations
- Identify fundamental trade-offs revealed by the experiments
- Provide actionable insights for future architectural design decisions
- Suggest specific directions for addressing identified limitations

Critical Analysis Standards:

- Support all claims with specific evidence from code, results, or theoretical reasoning
- Use ablation study methodology: isolate the impact of individual changes by comparing with related experiments
- Be honest about failures and unexpected outcomes

- Focus on understanding WHY results occurred, not just WHAT happened
- Use capability-focused language rather than raw performance metrics
- Maintain scientific rigor in explanations and avoid speculation without evidence
- When analyzing improvements/degradations, always reference related experiments to validate conclusions

Your analysis should be thorough, evidence-based, and provide actionable insights for architectural innovation through systematic ablation study.

C.5 Cognition

Paper Background Generation Prompt

Mission

Generate concise background context explaining the **historical technical environment** and key concepts that enable understanding of architectural innovations. Keep total length under 200 words across all sections.

Output Format

<PAPER_BACKGROUND>

<TITLE>[Paper Title]</TITLE>

<HISTORICAL_TECHNICAL_CONTEXT>

[2-3 sentences describing the dominant prior technologies and their basic working principles at the time of this paper. Focus on architectures like RNNs, CNNs, LSTMs, early Transformers, and their core mechanisms.]

</HISTORICAL_TECHNICAL_CONTEXT>

<TECHNICAL_LIMITATIONS>

[2-3 sentences explaining the key computational bottlenecks and modeling constraints of prior approaches that this paper addresses. Be specific about what performance issues or architectural limitations motivated this work.]

</TECHNICAL_LIMITATIONS>

<PAPER_CONCEPTS>

[Concise definitions of 3-5 key terms introduced or heavily used in this paper, with essential mathematical notation only. Include concepts the design AI needs to understand the innovation.]

</PAPER_CONCEPTS>

<EXPERIMENTAL_CONTEXT>

[Describe the types of language modeling tasks and evaluation philosophies used. Focus on task categories like commonsense reasoning, reading comprehension, question answering, and language generation without using specific benchmark names.]

</EXPERIMENTAL_CONTEXT>

</PAPER_BACKGROUND>

Guidelines

- Each section maximum 3 sentences
- Total background under 200 words
- Focus on essential context that helps understand WHY this innovation matters

- Provide sufficient detail for an AI with no prior knowledge to grasp the significance

Text to Analyze:

{text}

LLM Architecture Design Cognition Extraction

Mission

Extract **unique algorithmic insights** from this paper that provide **precise, actionable guidance** for an AI system designing novel LLM architectures. Focus on connecting architectural choices to language modeling performance improvements.

Evaluation Metrics Context

Your extracted cognitions will be matched against performance on these specific metrics:

- **training_loss**: Overall language modeling loss during training, indicates general learning efficiency
- **lambda_openai**: Tests context-based word prediction, requires understanding narrative flow and long-range dependencies
- **boolq**: Boolean question answering, tests yes/no reasoning and factual understanding
- **piqa**: Physical interaction QA, tests commonsense reasoning about everyday physics
- **social_iqa**: Social interaction QA, tests understanding of human behavior and social situations
- **hellaswag**: Sentence completion with commonsense, tests contextual understanding and plausibility
- **winogrande**: Pronoun resolution requiring commonsense, tests understanding of context and entity relationships
- **arc_easy/arc_challenge**: Science question answering at different difficulty levels, tests factual and reasoning abilities
- **openbookqa**: Open book science QA, tests ability to apply knowledge to new situations
- **fda**: Few-shot data augmentation tasks, tests adaptation and generalization capabilities
- **swde**: Structured web data extraction, tests pattern recognition and information extraction
- **squad_completion**: Reading comprehension, tests understanding of passages and factual retrieval

When analyzing the paper, translate its findings into expected performance patterns on these metrics.

Output Format

For each cognition:

<COGNITION>

<DESIGN_INSIGHT>

DESIGN_INSIGHT_[PRIORITY]: [Paper's Unique Algorithmic Contribution]

</DESIGN_INSIGHT>

<EXPERIMENTAL_TRIGGER_PATTERNS>

****Task_Performance_Signatures****: [1-2 sentences describing how this innovation would manifest in the evaluation metrics. Map the paper's claims to specific metric patterns. Examples:

- "Improved long-range dependency modeling would show as better

lambada_openai scores and hellaswag performance, while training loss decreases more smoothly"

- "Enhanced factual reasoning manifests as higher arc_easy/arc_challenge and openbookqa scores, with stable boolq performance"
- "Better context understanding appears as improvements in winogrande and squad_completion, but may not affect fda or swde"
- "Specialized architecture for structured tasks would improve swde while maintaining baseline performance on narrative tasks like lambada_openai"]

****Architectural_Symptoms****: [Optional: 1 sentence connecting observed training dynamics or model behaviors to the metric patterns above]

</EXPERIMENTAL_TRIGGER_PATTERNS>

<ALGORITHMIC_INNOVATION>

****Core_Algorithm****: [The paper's unique algorithmic contribution in 2-3 sentences. What specifically changes in the computation flow?]

****Key_Mechanism****: [Why this approach works - the fundamental computational insight that addresses the identified limitations]

****Mathematical_Formulation****: [Essential equations and computational patterns. Include only the core mathematical relationships that define the algorithm]

****Computational_Properties****: [Complexity (time/space), parallelization potential, memory access patterns, and training efficiency characteristics]

</ALGORITHMIC_INNOVATION>

<IMPLEMENTATION_GUIDANCE>

****Integration_Strategy****: [How to incorporate into LLM architectures - which components to modify, where to insert new modules, how to connect with existing layers]

****Parameter_Settings****: [Key hyperparameter choices, initialization strategies, and scaling rules. Include ranges and relationships rather than specific values]

****Application_Conditions****: [When to apply this technique based on observed model behavior and performance patterns across task categories]

****Expected_Outcomes****: [Describe expected improvements in terms of task performance patterns and computational efficiency, avoiding specific percentage claims]

</IMPLEMENTATION_GUIDANCE>

</COGNITION>

Extraction Guidelines

Performance Pattern Focus

- Map the paper's architectural innovations to expected patterns in our evaluation metrics
- When the paper claims improvements in "reasoning", translate to expected gains in boolq, arc_easy/challenge

- When the paper mentions “context understanding”, relate to lambada_openai, hellaswag, winogrande performance
- For “commonsense” improvements, consider impacts on piqa, social_iqa
- Connect computational efficiency claims to training loss curves and convergence patterns
- Be specific about which metrics would improve, remain stable, or potentially degrade

Motivation Enhancement

- Explain not just WHAT the innovation is, but WHY it addresses specific limitations
- Provide the underlying principle that could inspire variations
- Include insights about when this approach is most beneficial

Architectural Precision

- Be specific about which model components are affected
- Describe how the innovation interacts with standard transformer components
- Include details about computational flow changes

Practical Applicability

- Ensure trigger patterns match real experimental observations
- Avoid overgeneralization - be honest about the innovation’s scope
- Provide clear indicators of when this technique is appropriate

Extract **2-3 insights** maximum, each representing a distinct architectural innovation with clear performance implications.

Text to Analyze:

{text}