AIRA-Compose: Agentic Discovery of Neural Architectures

Alberto Pepe* **Chien-Yu Lin*** **Despoina Magka*** **Bilge Acun** **Yannan Nellie Wu**

Anton Protopopov **Carole-Jean Wu** **Yoram Bachrach**

Abstract

We introduce **AIRA-Compose** – a framework that enables AI research agents to autonomously explore and discover novel neural architectures. AIRA-Compose leverages the previously-proposed Composer abstraction to define a combinatorial design space over fundamental model primitives (Attention, MLP, and Mamba), then uses a guided search policy to efficiently navigate this space for novel neural architecture design. Our approach operates in two stages: (1) agents iteratively designs and evaluates candidate architectures at the million-parameter scale, then (2) top-performing candidates are scaled to billion parameters for pre-training. Overall, AIRA-Compose discovers 14 novel architectures spanning two families – **AIRAformers** (Transformer-based) and **AIRAhybrids** (Transformer-Mamba Hybrid-based). When pre-trained at 1B scale under a fixed token budget, agent-discovered top-performing architectures consistently outperform both Llama 3.2 and Composer-found alternatives. On downstream tasks, AIRAformer-D and AIRAhybrid-D improve accuracy by 2.4% and 3.8% over Llama 3.2, respectively. AIRA-Compose also finds novel model architectures that achieve steeper, more efficient compute-optimal scaling frontiers. AIRAformer-C scales 54% and 71% faster than Llama 3.2 and the best Composer-found Transformer, while AIRAhybrid-C scales 23% and 37% faster than the modified Nemotron-2 and the best Composer-found hybrid, respectively. For the first time, AIRA-Compose demonstrates that AI research agents can discover hybrid architectures that surpass hand-designed baselines on the accuracy–compute frontier. It establishes a flexible paradigm for LLMs to discover next generation foundation models, marking a step towards recursive self-improvement.

1 Introduction

Agents powered by Large Language Models (LLMs) have evolved into autonomous systems capable of executing end-to-end research loops (Yamada et al., 2025). By formulating hypotheses, writing code, evaluating results, and iteratively refining their approaches, agents can now discover new mathematical constructions (Novikov et al., 2025), replicate existing AI literature (Starace et al., 2025), and drive open-ended machine learning discoveries (Lupidi et al., 2026). A compelling frontier in this space is **Recursive Self-Improvement (RSI)** (Good, 1966; Schmidhuber, 1987, 2003): agents autonomously designing and optimizing the very neural architectures that power them. While most LLMs today rely on Transformers (Vaswani et al., 2017; Radford et al., 2018), the community is increasingly shifting toward post-Transformer paradigms (Peng et al., 2023; Sun et al., 2023; Gu and Dao, 2024) to address inherent limitations like quadratic attention complexity and inference KV cache costs (Tay et al., 2022; Pope et al., 2023). Combining different computational primitives into *hybrid* LLMs has proven to yield superior, more efficient foundation models (Blakeman et al., 2025;

*Joint first author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

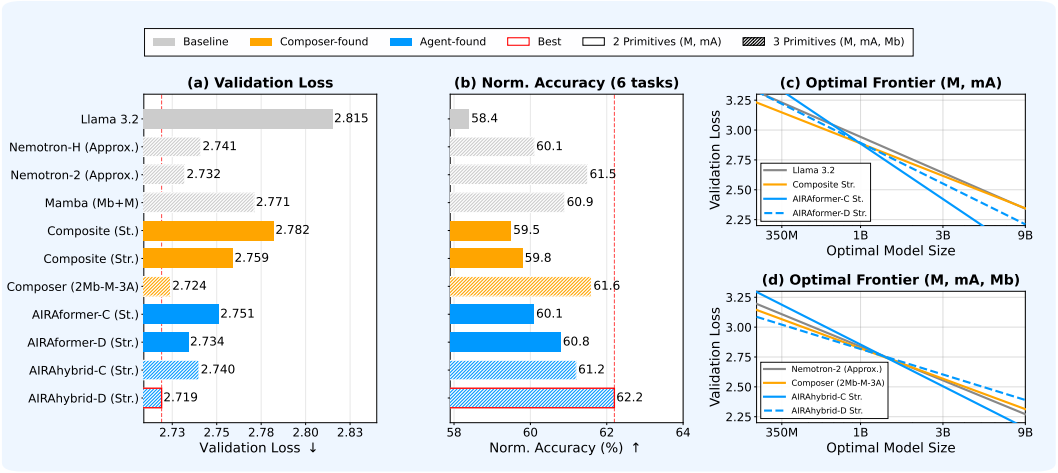

Figure 1: **AIRA-Compose.** Selected agent-discovered architectures compared to baselines and traditional NAS (Composer-found) models. (a–b) Downstream evaluations of architectures scaled-up to 1B scale with fixed token budget: (a) validation loss and (b) zero-shot average normalized accuracy across 6 tasks. (c–d) Compute-optimal scaling frontiers extrapolated from 350M/1B/3B isoFLOP experiments for (c) the 2-primitive and (d) the 3-primitive case, showing the best scaling architectures. Agent-powered NAS navigates the combinatorial search space effectively, discovering architectures that surpass both hand-designed and Composer-found baselines at scale.

Singhal et al., 2025; Alexiuk et al., 2026; Bae et al., 2025). Historically, model design has relied on human intuition. However, as hybrid design spaces become vast and combinatorial (Liu et al., 2021; Ren et al., 2021), manual exploration risks overlooking performant, non-obvious configurations. We therefore argue that **Neural Architecture Search (NAS)** is an ideal task for AI research agents: by pairing educated, context-aware hypotheses with iterative, automated search, agents can systematically explore this vast design space to propose fundamentally new architectures.

Motivated by this, we introduce **AIRA-Compose**, a framework for the agentic discovery of neural architectures for hybrid LLMs. We challenge agents to discover novel architectures by arranging computational primitives and evaluating them at small scale. We do so by recasting the exploration of the *Composer* framework (Acun et al., 2025) into AIRS-BENCH *tasks* (Lupidi et al., 2026) and execute them within the AIRA-DOJO harness (Toledo et al., 2025). Because AIRA-DOJO frames agentic exploration as a *search* algorithm, we argue that it provides the ideal infrastructure for agentic NAS. We summarize our contributions below:

- We introduce a flexible NAS framework which allows agents to navigate the architectural search space based on their reasoning capabilities, as opposed to rigid, incremental search policies.
- When pretrained from scratch at the 1B scale under fixed token budget, among the 14 agent-designed hybrid models, the top models achieve the lowest validation loss, highest 0-shot accuracy, and second best DCLM core score, outperforming established baselines and Composer-found alternatives.
- IsoFLOP analysis across 5 compute budgets and 3 model scales confirms that agent-found model exhibit superior scaling properties: all 6 AIRAformers scale faster than Llama 3.2 and Composer baselines, and top AIRAhybrids scale faster than the approximated Nemotron-2 and the best Composer-found model.

We summarize our findings in Figure 1. When scaled to 1B parameters and pretrained under a fixed budget of 37.5B tokens, agent-discovered models consistently surpass established baselines (Llama 3.2 and approximated Nemotron-2, Nemotron-H) as well as the strongest architectures found by Composer (Acun et al., 2025) in terms of (a) validation loss and (b) 0-shot normalized accuracy. The advantage is also noticeable at scale: IsoFLOP analysis (c–d) reveals that the best agent-found architectures follow steeper compute-optimal frontiers, with AIRAformer-C scaling 54% faster than

Llama 3.2 and AIRAhybrid-C scaling 23% faster than the approximated Nemotron-2, projecting 16% and 5% lower validation loss at 9B-equivalent scale, respectively.

2 Related Work

Hybrid Language Models. Transformers are the backbone of modern LLMs (Llama Team, 2024; Google Gemini Team et al., 2024; Google Gemini Team, 2025; OpenAI Team, 2025), but the quadratic cost of self-attention limits long-context processing. To address this, the field is moving towards hybrid architectures (Dao and Gu, 2024; Ren et al., 2024; Gu and Dao, 2024). These models interleave diverse computational primitives (e.g., Attention, SSMs) to balance efficiency with expressiveness (Bae et al., 2025). SSMs like Mamba-2 (Dao and Gu, 2024), Gated DeltaNet (GDN) (Yang et al., 2025b), Samba (Ren et al., 2024), and Hymba (Dong et al., 2024) represent a compelling alternative because of their linear complexity, striking a balance of performance and cost-efficiency while still successfully capturing complex data dynamics (Rahmani et al., 2025). The interleaving of SSM and Attention blocks does not act as a Transformer fallback, but it serves as the language modeling backbone of hybrid architectures (Borobia et al., 2026), yielding models that surpass the theoretical limits of either primitive alone (Merrill et al., 2026). This paradigm shift powers several SOTA models, including Qwen3-Next (Yang et al., 2025a), Nemotron-H (Blakeman et al., 2025), Nemotron Nano 2 (Basant et al., 2025) and Nemotron 3 Super (Alexiuk et al., 2026).

Neural Architecture Search. Hybrid models design implies a vast combinatorial search space, and manual tuning is not sufficient, hence researchers rely on Neural Architecture Search (NAS) (Elsken et al., 2019; Liu et al., 2021; White et al., 2023). Historically, NAS has been studied for smaller models using reinforcement learning (Zoph and Le, 2017) or evolutionary algorithms (Real et al., 2019) to combine small "cells" into full architectures (Zoph et al., 2018). Applying traditional NAS methods to LLMs is prohibitive, and most attempts rely on approximations. AutoMOE (Jawahar et al., 2023) utilizes a supernet approach, which trains an over-parameterized network to carve smaller sub-networks from it. LiteTransformerSearch (Javaheripi et al., 2022) employs zero-cost performance proxies, estimating a model's fully-trained accuracy based on untrained properties like initial gradient flow and parameter count. The Post Neural Architecture Search (PostNAS) paradigm prunes existing pretrained models or replaces standard attention blocks with linear variants (Bercovich et al., 2025; Gu et al., 2025). Mechanistic design principles are used to analytically derive the optimal allocation of diverse layers (Poli et al., 2024), while empirical search frameworks like Composer (Acun et al., 2025) bypass computational bottlenecks using small scale proxies to identify block arrangements that reliably perform well at scale.

Agents towards Recursive Self-Improvement. Advances in agentic reasoning have enabled LLMs to function as autonomous researchers, capable of scientific and algorithmic discovery (Novikov et al., 2025; Anthropic, 2025; Google DeepMind, 2026). More benchmarks are emerging to evaluate agentic capabilities: AIRS-Bench (Lupidi et al., 2026), assessing the full research cycle pipeline, MLGym (Nathani et al., 2025) and MLE-bench (Toledo et al., 2025), focusing on engineering tasks, and PaperBench (Starace et al., 2025), targeting scientific paper reproduction. An immediate next step in agentic research is represented by agents that iteratively discover and implement superior versions of themselves (Good, 1966; Schmidhuber, 1987; Yin et al., 2025). Recent efforts include the Automated LLM Speedrunning Benchmark (Zhao et al., 2025), which challenges agents to minimize LLM training time to a target loss, and Autoresearch (Karpathy, 2026), which automates model code and training loop optimization.

3 The AIRA-Compose Pipeline

AIRA-Compose (Figure 2) relies on the *Composer* framework (Acun et al., 2025). Composer discovers hybrid foundation models with a four-step process: (1) The *Search Engine* uses Bayesian Optimization combined with incremental layer search and width-scaling to discover primitive arrangements; (2) The *Evaluator* is a fast-proxy training and evaluation loop. Candidate architectures from the search are trained on small proxy datasets; (3) The *Aggregator* post-processes the top candidate architectures discovered during the search. It employs layer-wise clustering techniques to select the most frequent computational primitive to obtain a robust small scale architecture while smoothing out the noise and overfitting coming from proxy training; (4) The *Extrapolator* scales the aggregated small scale architecture to a desired target parameter count (e.g., 350M, 1B, or 3B). This

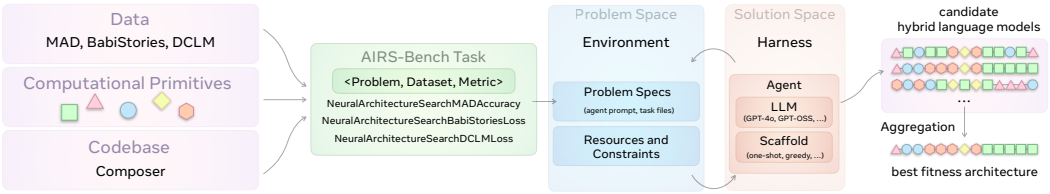

Figure 2: **The AIRA-Compose pipeline**. We recast *Composer* into equivalent AIRS-BENCH tasks. These tasks instruct agents to combine predefined computational primitives to assemble a 16-layer small scale architecture. Tasks are run employing the AIRA-DOJO harness with 6 LLMs as core reasoning models. Search results are then ranked, aggregated and scaled to 350M, 1B and 3B parameters size.

is achieved through *stretching* (proportionally expanding contiguous blocks) or *stacking* (repeating the entire discovered architecture sequentially).

AIRA-Compose recasts steps 1 and 2 as AIRS-BENCH tasks. Rather than relying on rigid Bayesian Optimization and deterministic incremental search, agents can freely formulate structural hypotheses and propose novel primitive arrangements (see Figure 3), evaluate them, and iteratively refine their designs based on their prior knowledge. We focus on 16 layer search, since they are small scale proxies that correlate well with equivalent large scale model. Once the agentic exploration is concluded, we leverage steps 3 and 4 to scale the agents' discoveries for final, large-scale evaluation.

Primitives. A *primitive* is a computational building block from which model architectures are assembled. We consider *two-primitive* and *three-primitive* search spaces. We employ MLPs (M), multi-head Attention (mA) and Mamba SSM (Mb). The two- and three- primitive search spaces span $2^{16} = 65{,}536$ and $3^{16} \approx 43$M possible 16-layer arrangements, respectively. Their configurations at small scale and 350M, 1B, and 3B scale are provided in Appendix C.

Datasets We evaluate architectures on three proxy datasets as in Acun et al. (2025), using metrics chosen to reliably predict at-scale performance: (i) **MAD** (Poli et al., 2024), a suite of six synthetic token-manipulation tasks (e.g., selective copying, compression, and in-context recall). Models are trained on 800 samples and tested on 1,280, using average accuracy as the metric; (ii) **BabiStories** (Zhang et al., 2025), a synthetic corpus of children's stories. Models train on 927,158 samples and are evaluated on 9,275 via cross-entropy loss; (iii) a fixed subset of **DCLM** (Li et al., 2024). Models train on 10,000 samples and test on 9,275 via cross-entropy loss. A larger portion of the DCLM corpus is reserved for the large-scale pretraining phase. We allow agents to validate their hypotheses by creating a 70:30 train/validation split on the original train set (Hambardzumyan et al., 2026). Submitted architectures are trained from scratch on the full training set and evaluated on the test set withheld from the agent.

4 Method

We adopt the definition of an *agent* as the combination of an LLM and a *scaffold* (Lupidi et al., 2026). The scaffold encompasses a search policy and operators that define how the agent explores the solution space. The scaffold is instantiated within a *harness*, that manages the agent's environment, execution, and tool access. To run the tasks we employ AIRA-DOJO (Toledo et al., 2025) – a harness designed to evolve code solutions through tree-based exploration. AIRA-DOJO guides the agent using structured search policies and interacts with candidate solutions through 4 operators: (i) *Draft:* Generates candidate solutions; (ii) *Debug:* Identifies and corrects execution or logical errors; (iii) *Improve:* Refines working solutions to optimize an evaluation metric; (iv) *Analyze:* inspects the agent solution. We employ one-shot and greedy scaffolds. The one-shot scaffold corresponds to calling the *draft* operator once, while the greedy scaffold explores several solutions through a best-first tree-based search policy. Each tree node represents a designed architecture, featuring validation fitness derived from the agent's code and test fitness from an independent evaluation script.

AIRA-Compose is instantiated in the form of AIRS-BENCH tasks (see Table 1). An AIRS-BENCH task is uniquely specified by a {problem, dataset, metric} triplet, e.g. `NeuralArchitectureSearchMAD3PrimitivesAccuracy`. AIRS-BENCH tasks have a standard-

Table 1: **The 4 AIRA-Compose tasks.** Each task requires agents to submit a `submission.csv` containing a sequence of 16 primitives describing a hybrid architecture. A 1∼20M proxy model is trained and evaluated at each step of the agentic search.

Task Name	Primitives	Data	Metric
`NeuralArchitectureSearchMAD2PrimitivesAccuracy`	M, mA	MAD (Poli et al., 2024)	Accuracy ↑
`NeuralArchitectureSearchBabiStories2PrimitivesLoss`	M, mA	BabiStories (Eldan and Li, 2023)	Val. Loss ↓
`NeuralArchitectureSearchDCLM2PrimitivesLoss`	M, mA	DCLM (Li et al., 2024)	Val. Loss ↓
`NeuralArchitectureSearchMAD3PrimitivesMbAccuracy`	M, mA, Mb	MAD (Poli et al., 2024)	Accuracy ↑

ized, modular file structure that makes open-ended ML problems into a format parsable by agentic harnesses (see Appendix E). It consists of: (i) `project_description.md`: The prompt provided to the agent. It includes the task objective, the dataset and the evaluation setup and how to structure its solution; (ii) `prepare.py` & `evaluate_prepare.py`: Responsible for data preparation; (iii) `evaluate.py`: The isolated scoring script used to evaluate the agent's submission against the test set. In our case `evaluate.py` trains the agent-generated architecture on the proxy dataset, effectively replacing step 2 (Evaluator) of Composer; (iv) `metadata.yaml`: A configuration file defining the task specifications and dependencies for the evaluation environment. AIRA-Compose differs from standard AIRS-BENCH tasks in several ways. Traditionally, `submission.csv` contains model predictions on test data (e.g., predicted classes or regression values), while in AIRA-Compose it specifies a neural architecture encoded as a string of 16 primitives. Similarly `evaluate.py` encapsulates the full training pipeline ported directly from the *Composer* codebase. This choice ensures consistent evaluation across all agent-generated architectures and it allows agents to focus exclusively on structural innovation rather than on the engineering overhead of writing training loops. Finally, AIRA-Compose expands the standard task folder with essential supplementary scripts, such as the PyTorch implementations of the computational primitives.

An example of agent-driven NAS is given in Figure 3. Compared to the Composer framework, which relies on fixed search methodologies, architectures designed at each step come from the agent own understanding of the problem as presented in `project_description.md`. At each node, the agent articulates its design choices, produces a candidate architecture as a `submission.csv` file, and writes an evaluation script to validate its reasoning. The submitted architecture is then trained and evaluated independently in `evaluate.py`. The node with the highest validation score is selected for further exploration via *improve* operations, which propose new architectures informed by the parent's reasoning and score. This process allows the agent to leverage domain knowledge to navigate the combinatorial space in a meaningful way, rather than relying on predefined structural templates or hand-crafted mutation operators. Scaled across several hundreds of nodes and multiple independent agents, this process yields a semantically diverse exploration, which we believe to be the ultimate advantage of AIRA-Compose. For the aggregation step, we collect the submitted architectures along with their test scores across all steps from all agents (see Appendix B).

5 Experiments

We test the 4 tasks on one-shot and greedy scaffolds within AIRA-DOJO. We run experiments on 20 seeds for one-shot agents and on 10 seeds for greedy agents. Each run lasts for at most 24 hours or at most 500 steps. Each agent has access to one H200 GPU to draft and validate its solutions. For the search on BabiStories and DCLM, which are larger, we allocate 60 hours per run. Within this time budget, each seed covers a search space of 100-200 small scale architectures. At large scale, 350M models are trained on 8 H200 GPUs, while 1B and 3B models on 16 H200 GPUs. Optimizers and hyperparameters at small and large scale are kept consistent with prior work (Acun et al., 2025). Dependencies available in the agent's container are given in Appendix A, Table 4.

Metrics. We rank small scale models based on their raw scores (accuracy or cross-entropy loss). We evaluate the scaled-up models across three metrics: **(i)** cross-entropy validation loss measured on 1000 samples of the DCLM validation set; **(ii)** 0-shot downstream accuracy and normalized accuracy on 6 tasks: HellaSwag, WinoGrande, ARC-Easy, ARC-Challenge, PIQA, and SciQ; and **(iii)** the **DCLM Core score** (Li et al., 2024), an unweighted average across 14 tasks evaluated under 0-shot (COPA, CoQA, LAMBADA, OpenBookQA, WinoGrande, XWinograd-EN), 3-shot

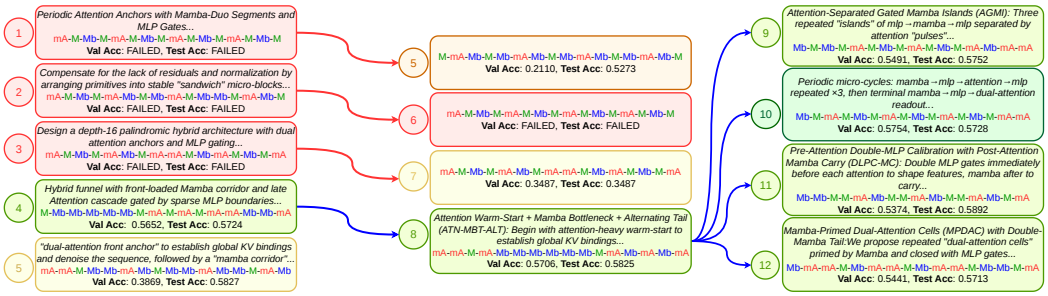

Figure 3: **Greedy GPT-5 search (partial) on the 3-primitive NAS task within AIRA-Compose**. The agent drafts five initial solutions and iteratively explores the architecture space via a greedy tree search. At each node, the agent reasons on different design choices, writes a candidate architecture and evaluates it. Red arrows indicate *debug* operations, while blue arrows indicate *improve* operations. At each improvement, the agent proposes a new architecture informed by the parent's reasoning and score. We report a snippet of the agent's design rationale, the submitted architecture, the agent's internal validation accuracy, and the true test accuracy from the isolated scoring script.

(AGIEval LSAT-AR), and 10-shot (ARC-Easy, ARC-Challenge, BoolQ, CommonsenseQA, HellaSwag, PIQA, SQuADv2). The primary metric is accuracy for most tasks, normalized accuracy for OpenBookQA, AGIEval LSAT-AR, ARC-Easy, ARC-Challenge, HellaSwag, and PIQA, and F1 score for CoQA and SQuADv2.

6 Results

6.1 2 primitives search (M, mA)

We evaluated 10 agents across three datasets by pairing five LLMs (Code World Model (Carbonneaux et al., 2025), o3-mini, gpt-oss-20b, gpt-oss-120b, and GPT-4o) with one-shot and greedy scaffolds. This yielded 150 runs per dataset (50 greedy and 100 one-shot). We further augmented the search with 20 additional greedy GPT-5 runs exclusively on the MAD dataset. Since agents only need to output a formatted string that maps to predefined PyTorch blocks, we deliberately employed non-SOTA models, which are fully capable of handling this structural simplicity. The search yielded 2,307 unique architectures explored, probing 3.17% of the search space. See Appendix A, Figures 9-12 for the search dynamic per agent. We rank all designed models across agents and seeds and rank them on their test accuracy (see Appendix B). We then aggregate them into robust base patterns and scale them. This resulted in 6 agent-discovered architectures (base 16-layer pattern in parentheses):

- **AIRAformer-A** (`((A + M) + (2A + 2M) + 4 × (A + M) + 2M)`): The aggregated 16-layer architecture is **stretched** to large scale.
- **AIRAformer-B** (`2A + 5 × (A + M) + 4M`): evaluated in its **stacked** variant.
- **AIRAformer-C** (`((2A + M) + 3 × (A + M) + (2A + M) + 4A)`: evaluated in its **stacked** and **stretched** variants.
- **AIRAformer-D** (`5 × (2A + M) + A`): evaluated in its **stacked** and **stretched** variants.

Aggregation techniques to obtain them and their composition at 350M, 1B, and 3B scale are shown in Appendix C. Notably, despite different aggregation techniques, these models converge on shared attention-to-MLP ratios: 7:9 for AIRAformers A and B, and 11:5 for C and D. The effect of aggregation on agentic search is discussed in Appendix D.

IsoToken Analysis. We pretrain 15 models: 6 AIRAformers, 2 Composer-found models (Acun et al., 2025) (Composite Stacked and Stretched, which are the best Composer-found models in 2-primitive space) and 7 baselines (Llama 3.2 (Llama Team, 2024), Sandwich Transformer (Press et al., 2020), Qwen3 (Yang et al., 2025a), OLMo3 (Ettinger et al., 2025) and 3 Striped Attention patterns (Poli et al., 2024)) at the 1B scale for a fixed, predefined token budget corresponding to 71,565 steps, derived from (Acun et al., 2025). With a batch size of 4, sequence length of 8,192,

and DP of 16, this corresponds to $\sim$37.5 billion tokens, or $\sim$ 38 TPP. Each architecture is trained with 3 random seeds, and we report the mean performance across them. We report the 3 metrics for the 15 pretrained models in Table 2. A comprehensive breakdown is given in Appendix A, Table 5, Figure 15.

Table 2: Pretraining results (2 primitives), 1B, fixed token budget (37.5B tokens). We report validation loss, 0-shot accuracy on 6 downstream tasks (in %), and few-shot DCLM Core score on 14 downstream tasks (in %), averaged over 3 seeds ($\pm$std). Values in parentheses show normalized accuracy. Best results are in **bold**; second-best results are underlined. Ties are highlighted equally.

Architecture	Val Loss ↓	ARC-C	ARC-E	HellaS.	PIQA	SciQ	WinoG.	Avg ↑	DCLM Core Score ↑
Llama 3.2	2.815$_{\pm.003}$	26.1 (30.0)	62.3 (56.0)	41.4 (53.1)	72.2 (72.5)	87.2 (80.2)	56.0	57.5 (58.4)	46.9$_{\pm0.3}$
Sandwich Transformer	2.761$_{\pm.0005}$	29.4 (31.9)	65.2 (**59.2**)	42.7 (55.4)	72.6 (**73.0**)	87.8 (83.1)	58.0	59.3 (60.1)	45.8$_{\pm0.3}$
Striped Attention 1:2	2.801$_{\pm.003}$	28.2 (30.3)	62.8 (56.8)	41.3 (52.7)	72.3 (72.0)	86.7 (80.1)	56.1	57.9 (58.0)	44.9$_{\pm0.2}$
Striped Attention 1:4	2.807$_{\pm.002}$	28.0 (29.2)	62.6 (55.7)	41.0 (52.4)	71.8 (71.7)	85.6 (78.4)	53.5	57.1 (56.8)	44.7$_{\pm0.1}$
Striped Attention 1:8	2.845$_{\pm.0003}$	26.0 (29.4)	61.0 (54.7)	39.8 (50.3)	71.5 (70.9)	82.9 (76.8)	52.4	55.6 (55.7)	43.4$_{\pm0.4}$
Qwen3	2.777$_{\pm.005}$	28.0 (30.5)	63.0 (57.5)	42.2 (54.2)	72.7 (72.9)	88.5 (82.9)	55.6	58.3 (59.6)	46.8$_{\pm0.2}$
OLMo3	2.826$_{\pm.003}$	26.9 (28.9)	61.7 (55.2)	40.3 (51.6)	72.0 (71.7)	86.3 (79.9)	55.3	57.1 (57.5)	45.8$_{\pm0.6}$
Composite (St.)	2.782$_{\pm.021}$	27.7 (30.2)	62.9 (57.1)	42.4 (55.1)	72.2 (72.3)	88.0 (82.7)	57.3	58.4 (59.5)	46.6$_{\pm1.4}$
Composite (Str.)	2.759$_{\pm.002}$	28.1 (31.0)	64.0 (58.5)	42.6 (54.8)	72.2 (72.6)	88.1 (81.8)	55.3	58.4 (59.8)	47.3$_{\pm0.1}$
AIRAf.-A (Str.)	2.752$_{\pm.002}$	**29.8** (**32.2**)	**65.4** (59.0)	43.2 (56.0)	72.6 (72.9)	88.0 (82.7)	58.4	59.6 (60.6)	48.5$_{\pm0.2}$
AIRAf.-B (St.)	2.752$_{\pm.001}$	29.4 (31.3)	65.3 (59.0)	43.2 (56.3)	**72.9** (72.9)	88.6 (83.7)	57.9	59.5 (60.6)	48.1$_{\pm0.1}$
AIRAf.-C (St.)	2.751$_{\pm.002}$	28.4 (31.4)	63.5 (57.5)	42.9 (55.3)	72.5 (72.7)	89.3 (83.6)	58.3	59.1 (60.1)	48.8$_{\pm0.2}$
AIRAf.-C (Str.)	2.750$_{\pm.001}$	29.1 (31.1)	64.0 (57.9)	42.9 (55.4)	71.8 (72.3)	**89.5** (83.5)	56.7	59.0 (60.1)	48.1$_{\pm0.2}$
AIRAf.-D (St.)	2.745$_{\pm.002}$	28.4 (31.6)	63.1 (58.9)	43.3 (56.2)	72.4 (72.5)	88.7 (83.1)	58.1	59.0 (60.5)	48.4$_{\pm0.2}$
AIRAf.-D (Str.)	**2.734**$_{\pm.001}$	29.4 (31.5)	63.7 (59.0)	**43.7** (**56.4**)	**72.9** (72.4)	89.4 (**84.5**)	**58.9**	**59.7** (**60.8**)	**48.9**$_{\pm0.4}$

All six AIRAformer variants outperform both Composite baselines in validation loss, average 0-shot accuracy, and DCLM Core score (Table 2). AIRAformer-D (Stretched) achieves the lowest validation loss (2.734), highest average 0-shot accuracy (59.7%; 60.8% normalized), and highest DCLM Core score (48.9%) among the evaluated architectures. Relative to Composite (Stretched), it reduces validation loss by 0.025 and improves average 0-shot accuracy and DCLM Core score by 1.3 and 1.6 percentage points, respectively. The corresponding gains over Llama 3.2 are 2.2 and 2.0 percentage points. Small standard deviations indicate consistent performance across the three training seeds.

IsoFLOP Analysis. We 8 models (keeping Llama 3.2 as baseline) across three parameter scales (350M, 1B, and 3B) under five FLOPs budgets: 2×10^{19}, 4×10^{19}, 8×10^{19}, 2×10^{20}, and 4×10^{20} FLOPs, for a total of 120 experiments (see Appendix C, Table 8 for FLOPs count). Attention-heavy AIRAformer-C and AIRAformer-D architectures achieve superior validation loss in the fixed-token-budget setting, as expected (Table 2), while balanced architectures are more compute-efficient in the isoFLOP regime. The additional attention layers in AIRAformer-C/D provide representational benefits when trained for longer, but they are less efficient when inference is the binding constraint.

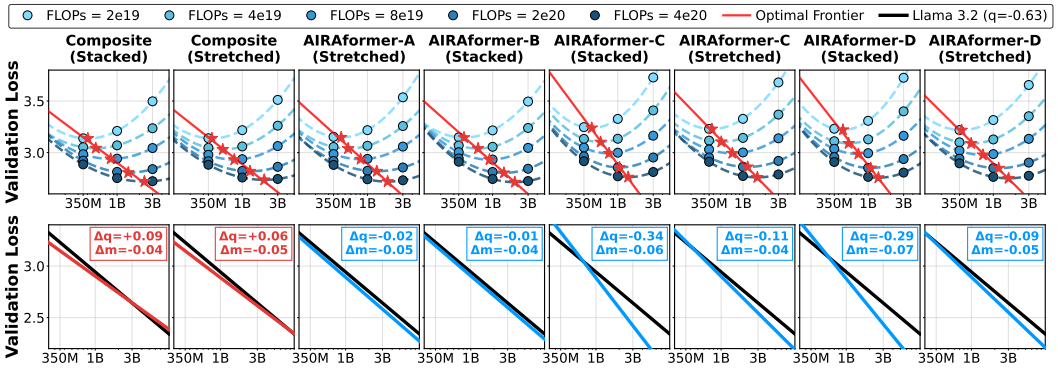

Figure 4: **IsoFLOP scaling curves and optimal frontier comparison (M, mA). Top:** Validation loss versus model size for each architecture across 5 FLOPs budgets. **Bottom:** Comparison of each architecture's optimal frontier against the Llama 3.2 scaling law derived from (Acun et al., 2025).

We fit isoFLOP parabolas to the validation loss as a function of model size for each FLOP budget (Figure 4, top) and infer the optimal frontier from their minima. Figure 4, bottom, compares each architecture's optimal frontier against Llama 3.2's reported scaling law, and reports Δq (slope difference, lower is better) and Δm (intercept offset, lower is better) for each architecture with respect

to it. AIRAformer-C Stacked and AIRAformer-D Stacked achieve the most favorable scaling coefficients, indicating steeper scaling than Llama 3.2. The Composite baselines exhibit slightly flatter slopes than Llama 3.2, suggesting diminishing advantage as model size increases.

6.2 3 primitives search (M, mA, Mb)

We use MAD for the 3 primitives search due to better agreement between the small scale ranking and large scale performance. A total of 170 agentic runs yielded 2,248 unique architectures, probing 0.0052% of the search space. Agent-powered AIRA-Compose discovers robust patterns by probing just a fraction of the design space, mitigating the combinatorial explosion of 3-primitive spaces in traditional NAS. We obtain 8 hybrid architectures:

- **AIRAhybrid-A** (`2Mb + M + 11Mb + 2M`): evaluated in its **stretched** variant.
- **AIRAhybrid-B** (`3 × (2Mb + M + A) + 2Mb + 2M`): evaluated in **stacked** and **stretched** variants.
- **AIRAhybrid-C** (`2Mb + A + 2 × (Mb + A) + M + 2 × (A + Mb + A + M)`): evaluated in its **stretched** variant.
- **AIRAhybrid-D** (`2Mb + M + 2 × (Mb + M) + A + M + Mb + M + A + M + Mb + M + A`): evaluated in **stacked** and **stretched** variants.
- **AIRAhybrid-E** (`5 × (Mb + M + A) + M`): evaluated in **stacked** and **stretched** variants.

IsoToken Analysis. We pretrain 19 models: 8 AIRAhybrids, 3 Composer-found architecture (Acun et al., 2025) and 8 hybrid baselines (Mamba (Gu and Dao, 2024), Nemotron-H (Blakeman et al., 2025), Nemotron-2 (Basant et al., 2025), Samba (Ren et al., 2024), Jamba (Lieber et al., 2024), Zamba2 (Glorioso et al., 2024), Qwen3.5 (Qwen Team, 2026) and OLMo-Hybrid (Merrill et al., 2026)) at the 1B scale at $\sim$ 38 TPP. Note that Nemotron-H and Nemotron-2 are only *approximated* versions to smaller scale, with MoEs replaced by MLP layers. The Composer architectures are obtained by performing the search over 6-layer small scale models, which is then stacked to obtain an equivalent model at large scale. Table 3 summarizes validation loss and downstream task performance. AIRAhybrids are competitive with the strongest hybrid baselines, although no architecture leads across all metrics. AIRAhybrid-D (Stretched) achieves the second-lowest validation loss (2.720), behind Zamba2 (2.716). Both AIRAhybrid-B (Stacked) and AIRAhybrid-D (Stretched) match Zamba2's highest average 0-shot accuracy of 60.4%, exceeding Mamba (59.3%) and approximated Nemotron-2 (60.2%). Individual AIRAhybrids also achieve leading task scores: AIRAhybrid-A (Stretched) obtains the highest PIQA accuracy (73.8%), while AIRAhybrid-B (Stacked) leads on ARC-C (31.3%) and ties for the highest SciQ accuracy (89.9%). Composer retains the highest DCLM Core score (49.4%), compared with the best AIRAhybrid score of 49.0%.

IsoFLOP Analysis. We pretrain the 11 most competitive models above at 3 scales, 350M, 1B and 3B parameters under 5 flop FLOPs budgets, for a total of 165 experiments. Results for selected models are shown in Figure 5, (see Appendix A for full results).

We compare each architecture's optimal frontier against the approximated Nemotron-H and Nemotron-2's scaling law, reporting slope difference and intercept offset relative to Nemotron-2. Architectures with balanced Mamba-to-attention ratios (Nemotron-2, AIRAhybrid-B, -D Stretched) consistently achieve the lowest validation loss across model sizes and FLOP budgets, while the attention-heavy AIRAhybrid-C Stretched ranks last. However, AIRAhybrid-C Stretched also achieves the steepest scaling slope, indicating faster improvement with scale than Nemotron-2. AIRAhybrid-A Stretched, the only pure-SSM design, also scales comparably. The Composer baseline, on the other hand, exhibits a flatter slope. These results mirror the two-primitive behavior: attention-heavy architectures are less compute-efficient but exhibit steeper scaling slopes that may yield superior performance at larger scale.

Finally, Figure 6 presents the 1B-scale latency – validation loss Pareto frontiers under isoToken and isoFLOP protocols. Model latency is computed by scaling the number of layers per block type by its corresponding GPU(H100)-profiled block latency. Under the isoToken analysis, AIRAhybrids-D (Stacked):17 and AIRAhybrids-D (Stretched):18 advance the previously-achieved frontier of Composer (Acun et al., 2025), leading to lower validation loss at strictly lower latencies than their Composer counterpart, which compensates strong downstream task performance with high latency.

Table 3: Pretraining results (3 primitives), 1B scale, fixed token budget (37.5B tokens). We report validation loss, 0-shot accuracy on 6 downstream tasks (in %), and few-shot DCLM Core score on 14 downstream tasks (in %), averaged over 3 seeds (±std). Values in parentheses show normalized accuracy. Best results are in **bold**; second-best results are underlined.

Architecture	Val Loss ↓	ARC-C	ARC-E	HellaS.	PIQA	SciQ	WinoG.	Avg ↑	DCLM Core Score ↑
Nemotron-H Approx.	2.743±.005	29.5 (31.8)	65.0 (59.2)	43.4 (56.3)	73.1 (73.8)	88.0 (82.9)	57.6	59.4 (60.3)	48.1±0.3
Nemotron-2 Approx.	2.733±.003	30.7 (33.0)	65.7 (61.0)	44.0 (57.3)	73.4 (73.9)	88.4 (82.8)	58.9	60.2 (61.1)	48.5±0.1
Mamba (Mb + M)	2.770±.001	29.7 (32.1)	64.7 (60.6)	43.1 (55.7)	72.9 (73.2)	87.8 (83.8)	57.4	59.3 (60.5)	46.7±0.6
Samba	2.736±.0003	29.3 (31.2)	65.4 (59.6)	44.3 (**57.9**)	73.2 (73.4)	88.3 (82.6)	58.6	59.8 (60.6)	48.4±0.4
Jamba	2.729±.001	30.6 (33.3)	65.7 (60.6)	44.1 (57.8)	73.0 (73.9)	88.8 (83.7)	59.1	60.2 (**61.4**)	48.3±0.2
Zamba2	**2.716**±.003	30.7 (33.0)	**66.4** (60.5)	44.6 (57.8)	73.2 (73.5)	89.0 (84.1)	58.6	60.4 (61.3)	48.8±0.2
Qwen3.5	2.764±.001	28.3 (30.3)	63.9 (57.8)	42.7 (55.3)	72.8 (72.8)	88.3 (82.6)	58.9	59.2 (59.8)	48.2±0.1
OLMo-Hybrid	2.766±.003	27.9 (30.3)	63.1 (57.2)	42.5 (54.7)	72.9 (73.3)	87.5 (81.3)	56.6	58.4 (59.4)	47.4±0.6
Composer (2Mb-M-3A)	2.724±.002	29.3 (32.5)	64.9 (60.9)	44.0 (57.2)	72.7 (73.7)	**89.9** (**85.2**)	58.8	60.0 (**61.4**)	**49.4**±0.6
Composer (M-2Mb-M-A-M)	2.735±.002	29.4 (32.1)	65.5 (60.4)	43.8 (56.6)	73.3 (73.6)	89.5 (84.3)	58.7	60.0 (61.0)	48.9±0.5
Composer (M-2Mb-2A-M)	2.728±.001	29.6 (32.0)	65.6 (60.9)	43.9 (57.1)	72.7 (72.9)	89.6 (84.4)	**59.2**	60.1 (61.1)	49.3±0.2
AIRAhybrid-A (Str.)	2.750±.000	30.4 (33.0)	66.0 (61.1)	43.8 (56.5)	**73.8** (73.8)	89.3 (84.3)	57.7	60.2 (61.1)	47.1±0.3
AIRAhybrid-B (St.)	2.731±.002	**31.3** (**33.6**)	65.5 (**61.2**)	44.0 (57.0)	73.3 (73.3)	**89.9** (84.6)	58.3	60.4 (61.3)	49.0±0.2
AIRAhybrid-B (Str.)	2.728±.001	29.5 (31.0)	65.0 (60.8)	44.2 (57.6)	73.1 (73.5)	89.6 (**85.2**)	58.3	59.9 (61.1)	48.7±0.5
AIRAhybrid-C (Str.)	2.742±.002	29.0 (31.0)	64.4 (59.4)	43.4 (56.0)	72.7 (73.0)	89.2 (84.2)	**59.2**	59.6 (60.5)	49.0±0.1
AIRAhybrid-D (St.)	2.721±.001	29.8 (32.1)	65.1 (60.8)	44.3 (57.6)	73.1 (73.5)	88.9 (84.3)	58.7	60.0 (61.2)	48.7±0.3
AIRAhybrid-D (Str.)	2.720±.001	30.4 (32.6)	66.1 (60.8)	44.3 (**57.9**)	73.4 (**74.4**)	89.3 (83.9)	58.6	60.4 (**61.4**)	48.7±0.5
AIRAhybrid-E (St.)	2.734±.002	28.9 (31.6)	65.0 (61.1)	43.8 (56.6)	72.8 (73.3)	88.9 (84.1)	58.8	59.7 (60.9)	49.0±0.3
AIRAhybrid-E (Str.)	2.733±.001	29.5 (31.6)	64.7 (60.4)	43.8 (56.7)	72.4 (73.2)	89.1 (**85.2**)	58.5	59.7 (60.9)	48.5±0.5

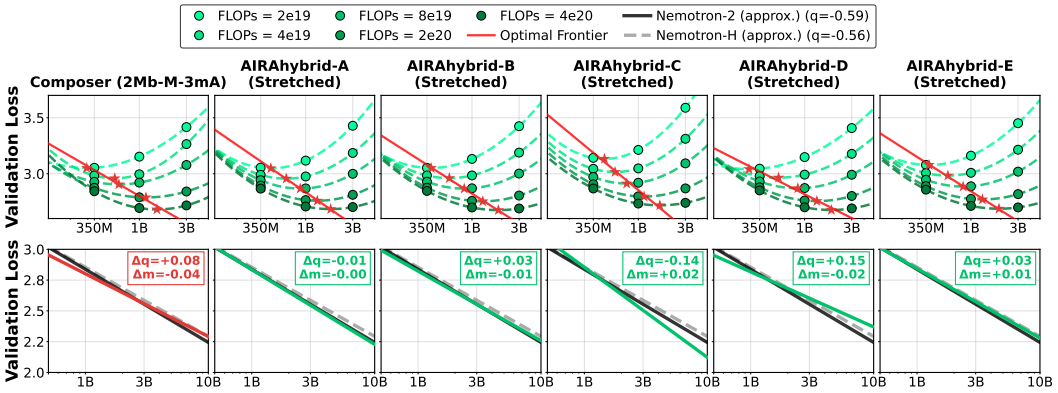

Figure 5: **IsoFLOP scaling curves and optimal frontier comparison (M, mA, Mb). Top:** Validation loss versus model size for each architecture across 5 FLOPs budgets. **Bottom:** Comparison of each architecture's optimal frontier against the approximated Nemotron-2 and Nemotron-H.

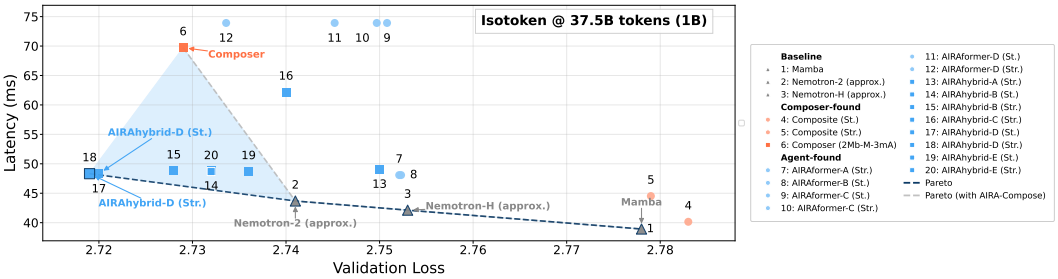

Figure 6: **Latency–validation loss Pareto frontiers at the 1B-parameter scale**. Isotoken comparison (37.5B). Total latency is the sum of per-block latencies across all layers. Dashed lines indicate Pareto frontiers with and without agent-discovered architectures; the shaded region highlights the gap between the two.

7 Conclusion

With AIRA-Compose, we deployed 11 agents across 340 24-hour and 300 60-hour runs, demonstrating their ability to autonomously formulate hypotheses on the optimal arrangement of computational primitives and construct original architectures. Unlike traditional methods constrained by rigid optimization objectives, these agents creatively and systematically navigate vast combinatorial search spaces. As a result, we introduce 14 novel architectures, AIRAformers and AIRAhybrids, which exhibit favorable downstream task performance, robust scaling properties, and promising loss–efficiency trade-offs.

Limitations and Future Work. AIRA-Compose relies on small-scale proxy evaluation, which does not always faithfully predict large-scale performance. This also limits the number of primitive combinations that can be reliably explored since they require careful tuning to scale. Testing AIRA-Compose on advanced harnesses (e.g., Hambardzumyan et al. (2026)) which can assign multiple GPUs per agent, will allow us to perform NAS on larger models and reduce the proxy–target gap. The greedy scaffold was chosen since it is considered the top-performing baseline (Lupidi et al., 2026), but incorporating Evolutionary or MCTS scaffolds may provide distinct advantages specific to our tasks. The aggregation and extrapolation steps also remain non-agentic: future work will introduce agentic tasks that do not just search and evaluate, but also aggregate and scale autonomously. Finally, optimizing only over primitive arrangements effectively underutilizes the agents' capabilities. Enriching the search space with tunable hyperparameters or normalization choices would significantly expand the design space and fully exploit the LLMs' potential. Because AIRA-Compose is built as a modular, LLM-agnostic template, we are confident it can be extended to incorporate these enhancements.

References

Bilge Acun, Prasoon Sinha, Newsha Ardalani, Sangmin Bae, Alicia Golden, Chien-Yu Lin, Meghana Madhyastha, Fei Sun, Neeraja J Yadwadkar, and Carole-Jean Wu. Composer: A search framework for hybrid neural architecture design. *arXiv preprint arXiv:2510.00379*, 2025.

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, 2023.

Chris Alexiuk, Chintan Patel, and NVIDIA. Nemotron 3 Super: Maximum Compute Efficiency for Complex Multi-Agent Applications, 2026. URL `https://developer.nvidia.com/blog`.

Anthropic. Claude Code, 2025. URL `https://www.anthropic.com/news/claude-code`.

Sangmin Bae, Bilge Acun, Haroun Habeeb, Seungyeon Kim, Chien-Yu Lin, Liang Luo, Junjie Wang, and Carole-Jean Wu. Hybrid architectures for language models: Systematic analysis and design insights. *arXiv preprint arXiv:2510.04800*, 2025.

Aarti Basant, Abhijit Khairnar, Abhijit Paithankar, Abhinav Khattar, Adithya Renduchintala, Aditya Malte, Akhiad Bercovich, Akshay Hazare, Alejandra Rico, Aleksander Ficek, et al. Nvidia nemotron nano 2: An accurate and efficient hybrid mamba-transformer reasoning model. *arXiv preprint arXiv:2508.14444*, 2025.

Akhiad Bercovich, Itay Levy, Izik Golan, Mohammad Dabbah, Ran El-Yaniv, Omri Puny, Ido Galil, Zach Moshe, Tomer Ronen, Najeeb Nabwani, et al. Llama-nemotron: Efficient reasoning models. *arXiv preprint arXiv:2505.00949*, 2025.

Aaron Blakeman, Aarti Basant, Abhinav Khattar, Adithya Renduchintala, Akhiad Bercovich, Aleksander Ficek, Alexis Bjorlin, Ali Taghibakhshi, Amala Sanjay Deshmukh, Ameya Sunil Mahabaleshwarkar, et al. Nemotron-h: A family of accurate and efficient hybrid mamba-transformer models. *arXiv preprint arXiv:2504.03624*, 2025.

Hector Borobia, Elies Seguí-Mas, and Guillermina Tormo-Carbó. Functional Component Ablation Reveals Specialization Patterns in Hybrid Language Model Architectures. *arXiv preprint arXiv:2603.22473*, 2026.

Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, et al. CWM: An open-weights LLM for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.

Xin Dong et al. Hymba: A Hybrid-head Architecture for Small Language Models. *arXiv preprint arXiv:2411.13676*, 2024.

Ronen Eldan and Yuanzhi Li. Tinystories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759*, 2023.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, et al. OLMo 3. *arXiv preprint arXiv:2512.13961*, 2025.

Paolo Glorioso, Quentin Anthony, Yury Tokpanov, Anna Golubeva, Vasudev Shyam, James Whittington, Jonathan Pilault, and Beren Millidge. The zamba2 suite: Technical report. *arXiv preprint arXiv:2411.15242*, 2024.

Irving John Good. Speculations concerning the first ultraintelligent machine. In *Advances in computers*, volume 6, pages 31–88. Elsevier, 1966.

Google DeepMind. Gemini 3.1 Pro: A Smarter Model for Your Most Complex Tasks. Google DeepMind Blog, 2026. URL `https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/`. Accessed: 2026-03-27.

Google Gemini Team. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Google Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First conference on language modeling*, 2024.

Yuxian Gu, Qinghao Hu, Shang Yang, Haocheng Xi, Junyu Chen, Song Han, and Han Cai. Jet-nemotron: Efficient language model with post neural architecture search. *arXiv preprint arXiv:2508.15884*, 2025.

Karen Hambardzumyan, Nicolas Baldwin, Edan Toledo, Rishi Hazra, Michael Kuchnik, Bassel Al Omari, Thomas Simon Foster, Anton Protopopov, Jean-Christophe Gagnon-Audet, Ishita Mediratta, et al. AIRA_2: Overcoming Bottlenecks in AI Research Agents. *arXiv preprint arXiv:2603.26499*, 2026.

Mojan Javaheripi, Gustavo de Rosa, Subhabrata Mukherjee, Shital Shah, Tomasz Religa, Caio Cesar Teodoro Mendes, Sebastien Bubeck, Farinaz Koushanfar, and Debadeepta Dey. Litetransformersearch: Training-free neural architecture search for efficient language models. *Advances in Neural Information Processing Systems*, 35:24254–24267, 2022.

Ganesh Jawahar, Subhabrata Mukherjee, Xiaodong Liu, Young Jin Kim, Muhammad Abdul-Mageed, Laks Lakshmanan, Ahmed Hassan, Sebastien Bubeck, Jianfeng Gao, et al. AutoMoE: Heterogeneous mixture-of-experts with adaptive computation for efficient neural machine translation. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 9116–9132, 2023.

Andrej Karpathy. Autoresearch. `https://github.com/karpathy/autoresearch`, 2026. GitHub repository.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Yitzhak Gadre, Hritik Bansal, Etash Guha, Sedrick Scott Keh, Kushal Arora, et al. Datacomp-lm: In search of the next generation of training sets for language models. *Advances in Neural Information Processing Systems*, 37:14200–14282, 2024.

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.

Yuqiao Liu, Yanan Sun, Bing Xue, Mengjie Zhang, Gary G Yen, and Kay Chen Tan. A survey on evolutionary neural architecture search. *IEEE transactions on neural networks and learning systems*, 34(2):550–570, 2021.

Llama Team. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783*, 2024.

Alisia Lupidi, Bhavul Gauri, Thomas Simon Foster, Bassel Al Omari, Despoina Magka, Alberto Pepe, Alexis Audran-Reiss, Muna Aghamelu, Nicolas Baldwin, Lucia Cipolina-Kun, et al. AIRS-Bench: a Suite of Tasks for Frontier AI Research Science Agents. *arXiv preprint arXiv:2602.06855*, 2026.

William Merrill, Yanhong Li, Tyler Romero, Anej Svete, Caia Costello, Pradeep Dasigi, Dirk Groeneveld, David Heineman, Bailey Kuehl, Nathan Lambert, et al. Olmo hybrid: From theory to practice and back. *arXiv preprint arXiv:2604.03444*, 2026.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, et al. MLGym: A New Framework and Benchmark for Advancing AI Research Agents. *arXiv preprint arXiv:2502.14499*, 2025.

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

OpenAI Team. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, et al. Rwkv: Reinventing rnns for the transformer era. In *Findings of the association for computational linguistics: EMNLP 2023*, pages 14048–14077, 2023.

Michael Poli, Armin W Thomas, Eric Nguyen, Pragaash Ponnusamy, Björn Deiseroth, Kristian Kersting, Taiji Suzuki, Brian Hie, Stefano Ermon, Christopher Re, et al. Mechanistic Design and Scaling of Hybrid Architectures. In *International Conference on Machine Learning*, pages 40908–40950. PMLR, 2024.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of machine learning and systems*, 5:606–624, 2023.

Ofir Press, Noah A Smith, and Omer Levy. Improving transformer models by reordering their sublayers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2996–3005, 2020.

Qwen Team. Qwen3.5: Towards Native Multimodal Agents. Alibaba Cloud Blog, 2026. URL `https://www.alibabacloud.com/blog/qwen3-5-towards-native-multimodal-agents_602894`.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Babak Rahmani, Mark Schöne, Heiner Kremer, Fabian Falck, Hitesh Ballani, and Jannes Gladrow. Implicit language models are RNNs: balancing parallelization and expressivity. *arXiv preprint arXiv:2502.07827*, 2025.

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

Liliang Ren, Yang Liu, Yadong Lu, Yelong Shen, Chen Liang, and Weizhu Chen. Samba: Simple hybrid state space models for efficient unlimited context language modeling. *arXiv preprint arXiv:2406.07522*, 2024.

Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)*, 54(4):1–34, 2021.

Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

Jürgen Schmidhuber. Gödel machines: Self-referential universal problem solvers making provably optimal self-improvements. *arXiv preprint cs.LO/0309048*, 2003.

Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Soumye Singhal, Jiaqi Zeng, Alexander Bukharin, Yian Zhang, Gerald Shen, Ameya Sunil Mahabaleshwarkar, Bilal Kartal, Yoshi Suhara, Akhiad Bercovich, Itay Levy, et al. Llama-nemotron: Efficient reasoning models. In *The Exploration in AI Today Workshop at ICML 2025*, 2025.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, et al. PaperBench: Evaluating AI's Ability to Replicate AI Research. *arXiv preprint arXiv:2504.01848*, 2025.

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6):1–28, 2022.

Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Maria Lupidi, Andrei Lupu, Roberta Raileanu, Kelvin Niu, Tatiana Shavrina, Jean-Christophe Gagnon-Audet, Michael Shvartsman, Shagun Sodhani, Alexander H. Miller, Abhishek Charnalia, Derek Dunfield, Carole-Jean Wu, Pontus Stenetorp, Nicola Cancedda, Jakob Nicolaus Foerster, and Yoram Bachrach. AI Research Agents for Machine Learning: Search, Exploration, and Generalization in MLE-bench, 2025. URL https://arxiv.org/abs/2507.02554.

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.

Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The AI Scientist-v2: Workshop-Level Automated Scientific Discovery via Agentic Tree Search, 2025. URL https://arxiv.org/abs/2504.08066.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *International Conference on Learning Representations*, volume 2025, pages 29687–29707, 2025b.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 27890–27913, 2025.

Jianyu Zhang, Niklas Nolte, Ranajoy Sadhukhan, Beidi Chen, and Leon Bottou. Memory mosaics. In *The Thirteenth International Conference on Learning Representations*, 2025.

Bingchen Zhao, Despoina Magka, Minqi Jiang, Xian Li, Roberta Raileanu, Tatiana Shavrina, Jean-Christophe Gagnon-Audet, Kelvin Niu, Shagun Sodhani, Michael Shvartsman, Andrei Lupu, Alisia Lupidi, Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Thomas Foster, Lucia Cipolina-Kun, Abhishek Charnalia, Derek Dunfield, Alexander H. Miller, Oisin Mac Aodha, Jakob Foerster, and Yoram Bachrach. The Automated LLM Speedrunning Benchmark: Reproducing NanoGPT Improvements, 2025. URL https://arxiv.org/abs/2506.22419.

Barret Zoph and Quoc Le. Neural Architecture Search with Reinforcement Learning. In *International Conference on Learning Representations*, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

Appendix

A Additional Details

Table 4: Environment requirements for AIRA-Compose tasks

Package	Version	Package	Version	Package	Version
torch	2.6.0	einops	–	transformers	–
torchvision	0.21.0	seaborn	–	tokenizers	–
torchaudio	2.6.0	torchmetrics	–	huggingface-hub	–
ninja	–	pytorch-lightning	–	safetensors	–
datasets	4.0.0	opt-einsum	–	filelock	–
scikit-learn	–	ray	–	fsspec	–
multiprocess	–	scipy	–	triton	3.1.0
ax-platform	–	absl-py	–	flash-attn	2.8.3
submitit	–	tiktoken	–	causal-conv1d	1.5.3.post1
blobfile	–	pyyaml	–	mamba-ssm	2.2.6.post3
numpy	–	tqdm	–		
pandas	–	packaging	–		

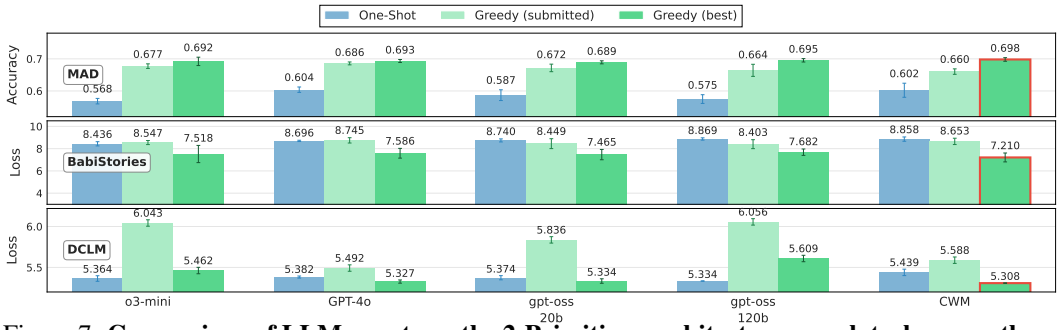

Figure 7: **Comparison of LLM agents on the 2-Primitives architecture search task across three datasets.** Performance of 11 agents, {o3-mini, GPT-4o, gpt-oss-20b, gpt-oss-120b, and CWM} $\times$ {one-shot, greedy} and greedy GPT-5. For greedy agents, we distinguish between *submitted* solutions (selected based on best validation fitness) and *best* solutions (achieving the highest test performance across all iterations). Top panel: MAD dataset (accuracy, higher is better); middle and bottom panels: BabiStories and DCLM datasets (loss, lower is better). The 2-Primitives search space is constrained to two primitives (M, mA). Error bars represent 95% confidence intervals. Best performing agent is highlighted in red.

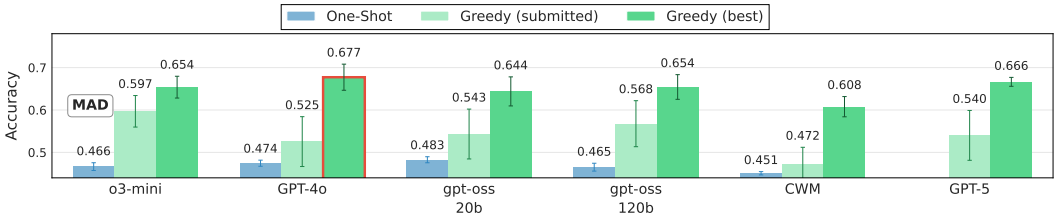

Figure 8: **Comparison of LLM agents on the 3-Primitives architecture search task on MAD datasets.** Performance of 10 agents, {o3-mini, GPT-4o, gpt-oss-20b, gpt-oss-120b, and CWM} $\times$ {one-shot, greedy}. For greedy agents, we distinguish between *submitted* solutions (selected based on best validation fitness) and *best* solutions (achieving the highest test performance across all iterations). The 3-Primitives search space is constrained to three primitives (M, mA, Mb). Error bars represent 95% confidence intervals. Best performing agent is highlighted in red.

Results per agent and per dataset are shown in Figure 7-Figure 8. CWM yields consistently the best overall results in the 2-primitive case, achieving the highest accuracy on MAD and the lowest

loss on both BabiStories and DCLM. Similarly, GPT-4 leads in the 3-primitive search. Greedy search, as expected, consistently outperforms one-shot inference across all agents, with the gap being most pronounced for CWM and gpt-oss-120b. Notably, the difference between submitted and best solutions for greedy agents reveals the challenge of validation-based selection: agents often identify superior architectures during search that are not selected due to noisy validation signals.

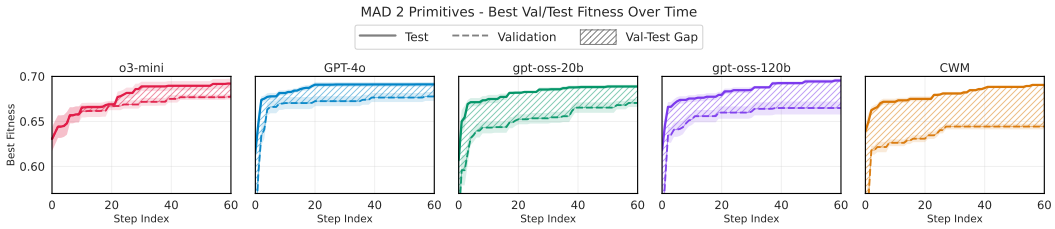

Figure 9: **Running Maximum Fitness Trajectories for MAD 2 Primitives (M, mA)**. Best validation (dashed line) and test (solid line) fitness achieved up to each step index, averaged across runs, for the 5 greedy agents: o3-mini, GPT-4o, gpt-oss-20b, gpt-oss-120b, and CWM. Shaded regions indicate 95% confidence intervals, and hatched areas highlight the generalization gap between validation and test fitness.

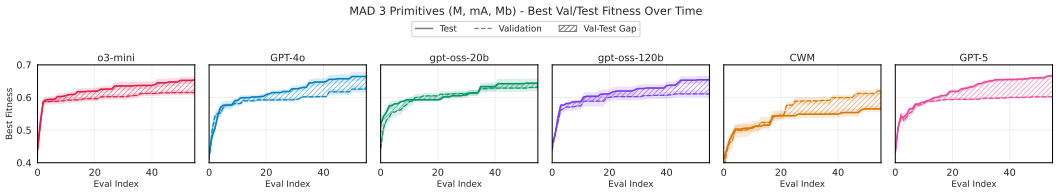

Figure 10: **Running Maximum Fitness Trajectories for MAD 3 Primitives (M, mA, Mb)**. Best validation (dashed line) and test (solid line) fitness achieved up to each step index, averaged across runs, for the 6 greedy agents: o3-mini, GPT-4o, gpt-oss-20b, gpt-oss-120b, CWM and GPT-5. Shaded regions indicate 95% confidence intervals, and hatched areas highlight the generalization gap between validation and test fitness.

Figures 9- 10 show the search dynamics of greedy agents on the MAD task. All agents exhibit rapid improvement during the early exploration phase (steps 0–20), with performance plateauing thereafter as the search converges. gpt-oss-120b achieves the highest average final test fitness ($\sim$0.70) on 2 primitives, demonstrating strong search efficiency. Similarly, GPT-5 shows the best average final test fitness on the 3-primitive case. CWM and gpt-oss-20b display the largest generalization gaps between validation and test curves, suggesting a tendency toward overfitting to the validation signal. This is even more evident in the 3-primitive case, with CWM consistently overestimating its own submission, which show higher validation accuracy than test accuracy. In contrast, o3-mini and gpt-oss-120b exhibit tighter alignment between validation and test fitness, indicating more robust architecture selection.

Figures 11 and 12 show the averaged exploration trajectories across all seeds for each greedy agent on the MAD benchmark, where the tree search has been flattened as a sequence of steps for visualization purposes. In both the 2-primitive and 3-primitive settings, the per-step accuracy fluctuates substantially as the agent explores diverse architectural hypotheses, reflecting the exploratory nature of the search rather than a monotonic improvement. A persistent gap between validation and test accuracy is visible across all models, highlighting the imperfect correlation between the agent-written validation proxy and the ground-truth evaluation. The high variance observed for some agents reflects the stochasticity of the search process, where different seeds can follow substantially different branches of the exploration tree.

Figures 13-14 show the distribution of architecture accuracy scores across agents in the 2-primitive and 3-primitive search space, respectively, on the MAD dataset. Distributions were employed to determine the best custoff and clustering technique for robust aggregation. The plots reveal distinct search behaviors: certain agents exhibit tighter distributions centered at higher accuracy values,

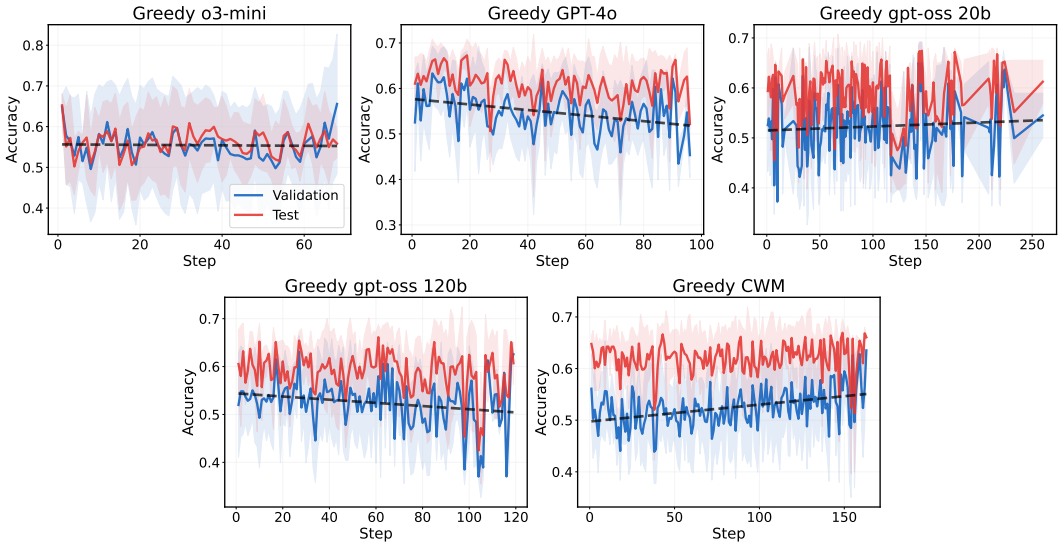

Figure 11: **Averaged exploration trajectories for the 2-primitive (M, mA) greedy search on the MAD benchmark**. Each panel shows the mean validation (blue) and test (red) accuracy per step across 10 seeds, with ± 1 standard deviation shading and validation accuracy linear trend lines. The gap between validation and test accuracy reflects the generalization challenge of agent-designed architectures under the proxy evaluation used for greedy selection.

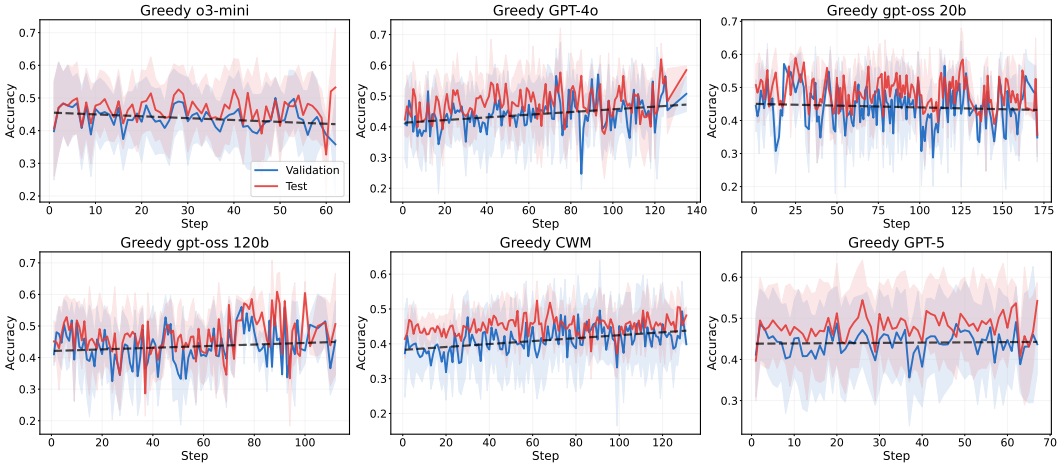

Figure 12: **Averaged exploration trajectories for the 3-primitive (M, mA, Mb) greedy search on the MAD benchmark**. Each panel shows the mean validation (blue) and test (red) accuracy per step across 10 seeds, with ± 1 standard deviation shading and validation accuracy linear trend lines. The gap between validation and test accuracy reflects the generalization challenge of agent-designed architectures under the proxy evaluation used for greedy selection.

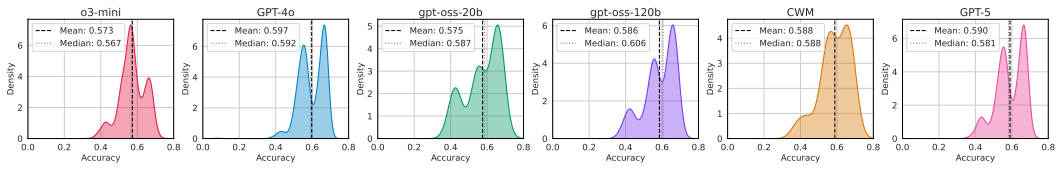

Figure 13: **Distribution of architecture accuracy scores across 6 greedy agents on MAD 2-Primitives (M, mA).** Each panel shows the distribution of accuracy values achieved by a single agent across all greedy runs. Vertical dashed black lines indicate the mean accuracy, while dotted gray lines indicate the median.

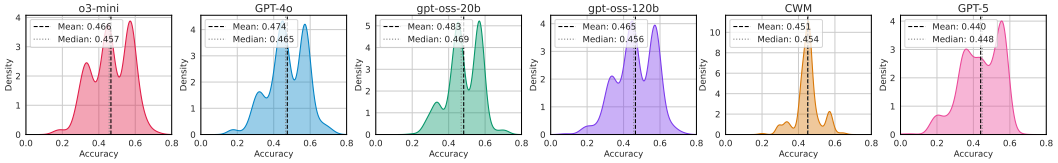

Figure 14: **Distribution of architecture accuracy scores across 6 greedy agents on MAD 3-Primitives (M, mA, Mb).** Each panel shows the distribution of accuracy values achieved by a single agent across all greedy runs. Vertical dashed black lines indicate the mean accuracy, while dotted gray lines indicate the median.

indicating consistent discovery of performant architectures, while others display broader, multi-peak profiles, suggesting greater exploration variance and a wider range of solution quality.

Table 5: Per-task DCLM Core Score breakdown at 1B scale (isotoken budget, 37.5B tokens). Metric per task in parentheses. **Bold** = best and underline = second best per column within each primitive group. The last column is the unweighted DCLM Core average (%). "–" indicates that the collaborator's per-task breakdown was unavailable.

Model	COPA (Acc.)	CoQA (F1 Score)	LAMBADA (Acc.)	OBQA (Norm. Acc.)	WinoGr. (Acc.)	XWino. (Acc.)	AGIEval (Norm. Acc.)	ARC-C (Norm. Acc.)	ARC-E (Norm. Acc.)	BoolQ (Acc.)	CSQA (Acc.)	HSwag (Norm. Acc.)	PIQA (Norm. Acc.)	SQuADv2 (F1 Score)	**Average**
2-Primitive (M, mA)															
Llama 3.2	78.0	45.4	46.8	33.4	56.4	76.7	20.9	33.1	64.3	37.9	20.1	53.3	73.3	17.1	46.9
Sandwich Transformer	75.3	12.0	45.1	34.7	58.0	78.8	17.1	36.1	67.0	37.9	20.1	56.6	73.2	28.6	45.8
Striped Attention 1:2	74.0	11.1	46.2	34.7	56.1	78.3	18.7	33.9	64.3	37.9	20.3	53.4	71.9	27.4	44.9
Striped Attention 1:4	**78.3**	11.1	41.7	34.3	53.5	74.7	20.6	33.8	64.9	37.9	19.6	53.0	72.3	**30.1**	44.7
Striped Attention 1:8	76.0	14.6	36.7	33.1	52.4	69.1	19.9	32.4	63.3	37.9	**21.2**	51.0	71.8	28.5	43.4
Qwen3	74.5	49.8	48.1	33.7	55.2	76.8	16.1	32.8	65.1	37.8	19.3	54.7	73.2	18.6	46.8
OLMo3	74.0	43.1	44.3	33.3	55.3	76.0	21.7	32.2	63.8	37.9	20.4	52.0	71.9	14.6	45.8
Composite (St.)	75.7	43.1	43.0	33.5	57.3	76.7	20.0	34.7	65.1	37.9	20.6	54.7	73.1	16.6	46.6
Composite (Str.)	**78.3**	44.6	44.5	34.5	56.2	78.5	20.3	**36.2**	67.8	37.9	20.3	55.7	72.5	14.3	47.3
AIRAformer-A (Str.)	75.7	50.4	47.2	34.6	57.7	**80.4**	**21.0**	35.9	67.9	37.9	19.8	57.3	**73.8**	19.1	48.5
AIRAformer-B (St.)	75.7	50.6	46.2	34.3	57.2	80.1	18.7	35.5	67.7	37.9	19.9	57.4	73.5	19.5	48.1
AIRAformer-C (St.)	77.7	53.2	49.0	**35.3**	58.3	80.2	20.3	35.3	67.2	37.9	18.9	56.6	73.4	20.2	48.8
AIRAformer-C (Str.)	75.0	52.1	**49.3**	33.5	56.6	78.8	18.6	34.7	67.2	37.9	20.6	57.0	73.1	19.2	48.1
AIRAformer-D (St.)	74.3	**53.9**	**49.3**	33.7	57.7	80.3	19.1	35.5	68.0	37.9	19.0	57.4	73.4	18.5	48.4
AIRAformer-D (Str.)	76.7	52.7	49.2	35.0	**59.0**	79.6	18.3	35.5	**68.3**	37.9	20.5	**58.1**	73.5	20.2	**48.9**
3-Primitive (M, mA, Mb/GDN)															
Nemotron-H (Approx.)	79.0	49.0	49.1	36.2	56.2	81.3	**23.0**	34.6	66.2	37.9	19.4	57.2	74.3	15.1	48.5
Nemotron-2 (Approx.)	76.0	47.6	48.6	35.8	58.3	81.1	20.9	35.7	66.8	**38.0**	21.1	57.9	73.0	16.6	48.4
Mamba (Mb+M)	79.0	34.4	45.2	35.6	57.7	80.1	16.1	35.6	64.8	37.9	19.8	55.9	73.6	9.6	46.1
Samba	76.3	45.6	50.6	35.7	58.6	81.4	18.8	36.1	68.0	37.9	20.5	**58.8**	74.1	15.6	48.4
Jamba	81.7	40.3	49.4	37.2	59.0	81.5	20.6	36.7	67.2	37.9	20.4	58.1	73.7	11.7	48.3
Zamba2	76.3	48.9	**52.1**	35.3	58.7	82.0	20.7	35.9	66.9	37.9	19.8	58.7	73.7	16.3	48.8
Qwen3.5	77.0	46.9	51.1	34.4	59.2	79.9	20.4	34.1	66.4	37.9	21.4	55.8	73.3	16.9	48.2
OLMo-Hybrid	77.0	44.9	48.1	34.7	56.6	78.6	20.7	34.6	66.2	37.9	20.0	55.1	72.9	15.9	47.4
Composer (2Mb-M-3A)	79.0	53.2	51.8	34.6	58.3	81.3	19.6	35.9	67.3	37.9	22.0	57.6	73.5	18.2	**49.3**
Composer (M-2Mb-M-A-M)	78.7	48.6	50.0	36.3	58.8	81.8	20.1	36.0	67.5	37.9	20.4	57.6	73.5	17.7	48.9
Composer (M-2Mb-2A-M)	77.7	51.5	51.9	35.8	59.2	81.9	22.0	34.9	**68.4**	37.9	20.2	58.0	74.0	17.5	**49.3**
AIRAhybrid-A (Str.)	**82.0**	40.6	47.6	35.2	57.2	79.3	19.6	34.7	67.5	37.9	**22.9**	57.2	72.7	9.9	47.5
AIRAhybrid-B (St.)	78.0	50.5	51.9	36.4	**59.9**	**82.6**	18.3	36.0	66.2	37.9	19.2	58.5	73.6	16.6	49.0
AIRAhybrid-B (Str.)	76.0	49.6	50.5	37.0	58.7	81.8	21.3	**36.8**	66.6	37.9	21.5	58.4	**75.0**	16.1	49.1
AIRAhybrid-C (Str.)	76.0	**53.5**	51.2	35.6	57.9	80.6	22.6	34.6	65.8	37.9	19.1	57.0	73.1	**19.6**	48.9
AIRAhybrid-D (St.)	79.0	49.1	51.4	35.8	58.3	81.8	16.1	35.1	66.8	37.9	19.2	57.2	73.2	16.0	48.4
AIRAhybrid-D (Str.)	74.0	50.8	50.6	37.2	57.9	81.5	17.4	**36.8**	67.7	37.9	20.0	57.8	73.8	16.0	48.5
AIRAhybrid-E (St.)	78.0	46.9	51.7	36.2	59.7	82.1	20.9	**36.8**	67.3	37.9	21.1	57.8	72.7	14.3	48.8
AIRAhybrid-E (Str.)	74.0	48.3	51.5	**37.4**	59.0	81.7	**23.0**	34.0	66.8	37.9	19.6	58.0	73.7	16.0	48.6

Table 5 expands the aggregate DCLM Core Scores reported in Tables 2 and 3 by providing the full per-task breakdown across all 14 evaluation tasks. The Composer-found (2Mb-M-3A) architecture achieves the highest overall average (49.3%), while the AIRAhybrid models show complementary strengths: AIRAhybrid-B variants achieve the best scores on WinoGrande (59.9), XWino-

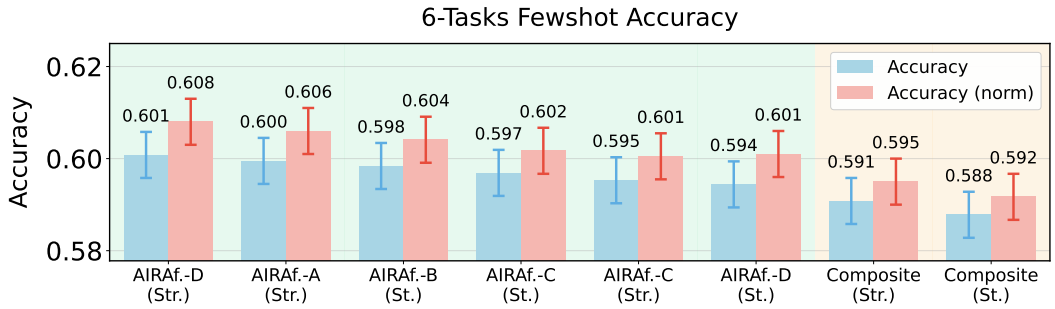

Figure 15: **Downstream evaluation of AIRAformer architectures at 1B scale**. 0-shot accuracy on 6 commonsense reasoning tasks (ARC-C, ARC-E, HellaSwag, PIQA, SciQ, WinoGrande), showing both raw and normalized accuracy across three seeds.

grad (82.6), HellaSwag (58.5), and PIQA (75.0), while AIRAhybrid-E (Stretched) leads on OpenBookQA (37.4) and ties for best on AGIEval (23.0). BoolQ remains essentially at chance level ($\sim$37.9%) across all architectures, indicating that 1B-scale models with our training setup uniformly fail under 10-shot evaluation regardless of architecture. Among the 2-primitive models, the AIRAformer variants consistently outperform the Composite and Llama baselines on CoQA, SQuADv2, and HellaSwag, with AIRAformer-D (Stretched) achieving the highest average (48.9%).

Figures 16–17 present the isoFLOP validation loss across three model sizes (350M, 1B, and 3B parameters) and five FLOP budgets for both the 2-primitive and 3-primitive settings. In the 2-primitive case, balanced architectures consistently outperform attention-heavy ones: AIRAformer-A Stretched, AIRAformer-B Stacked, and the Composite baselines achieve lower validation loss than AIRAformer-C and AIRAformer-D (which have A/M ratios exceeding 2) across all scales and budgets. For example, at the 1B scale with 4×10^{20} FLOPs, AIRAformer-A Stretched achieves a validation loss of 2.7415 compared to 2.7647–2.7732 for the attention-heavy variants.

In the 3-primitive hybrid setting, we observe a similar pattern: architectures that distribute compute more evenly across Mamba, MLP, and attention primitives—such as AIRAhybrid-C Stretched and the Composer (2Mb-M-3A) model—consistently reach the lowest validation losses, while Mamba-only configurations (AIRAhybrid-A) and attention-dominated hybrids underperform.

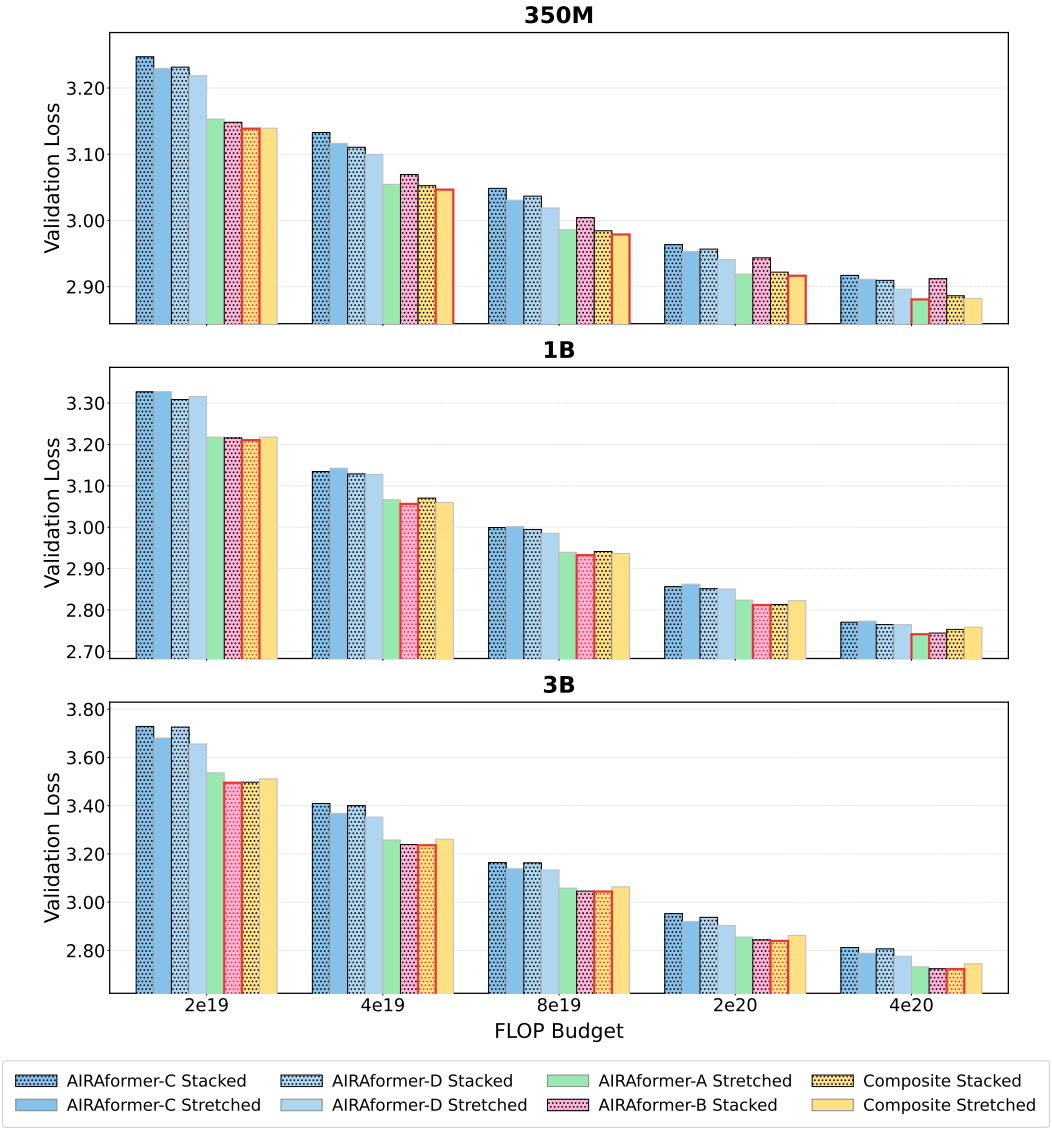

Figure 16: **IsoFLOP comparison of validation loss across model sizes.** Each subplot shows validation loss for 8 architectures (6 AIRAformer variants and 2 Composite baselines) at a given model size (350M, 1B, 3B), evaluated across 5 FLOP budgets ranging from 2×10^{19} to 4×10^{20}. Bars with red borders indicate the architecture achieving the lowest validation loss at each FLOP budget.

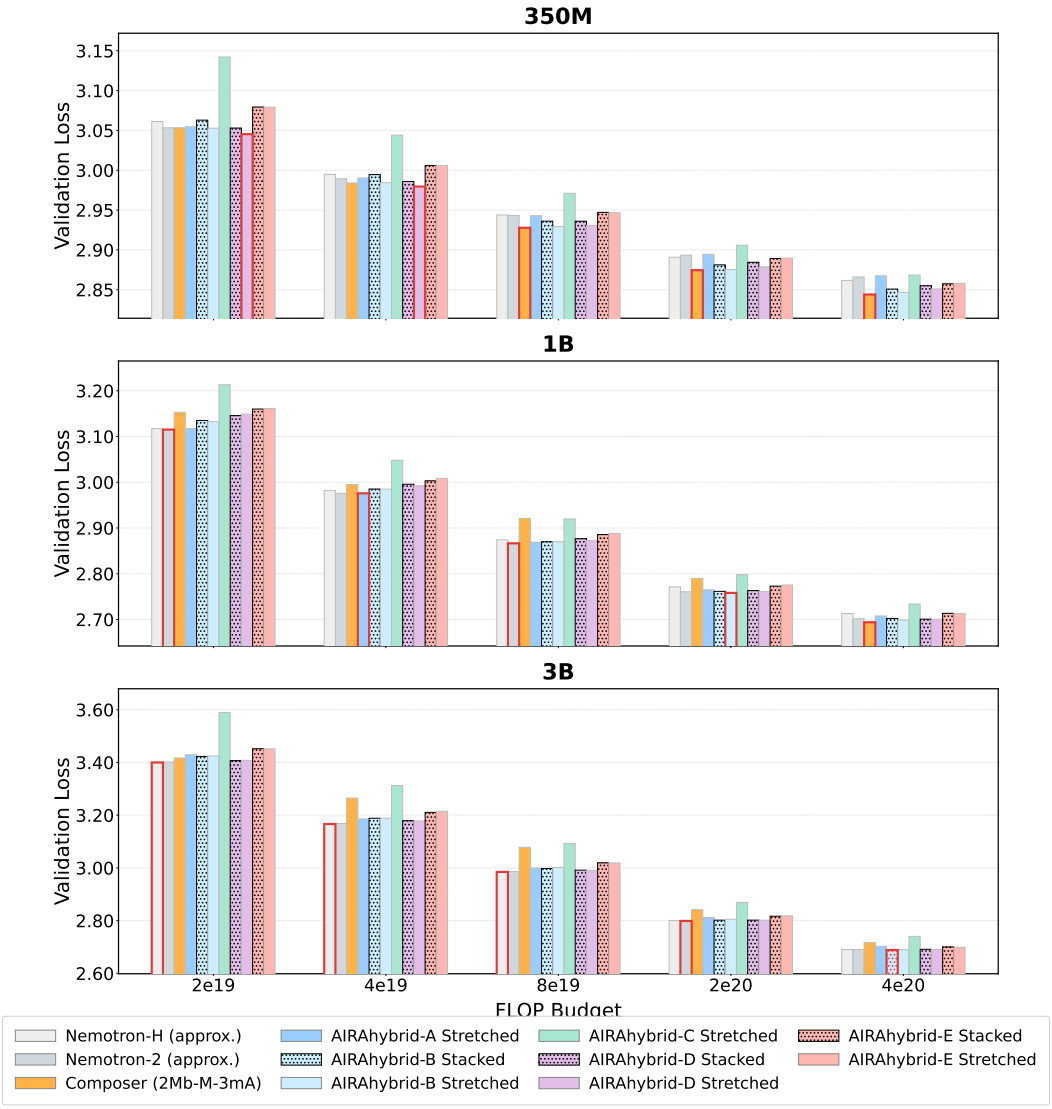

Figure 17: **IsoFLOP comparison of validation loss across model sizes.** Each subplot shows validation loss for 11 architectures (8 AIRAhybrids, 2 baselines and 1 Composer-found baselines) at a given model size (350M, 1B, 3B), evaluated across 5 FLOP budgets ranging from 2×10^{19} to 4×10^{20}. Bars with red borders indicate the architecture achieving the lowest validation loss at each FLOP budget.

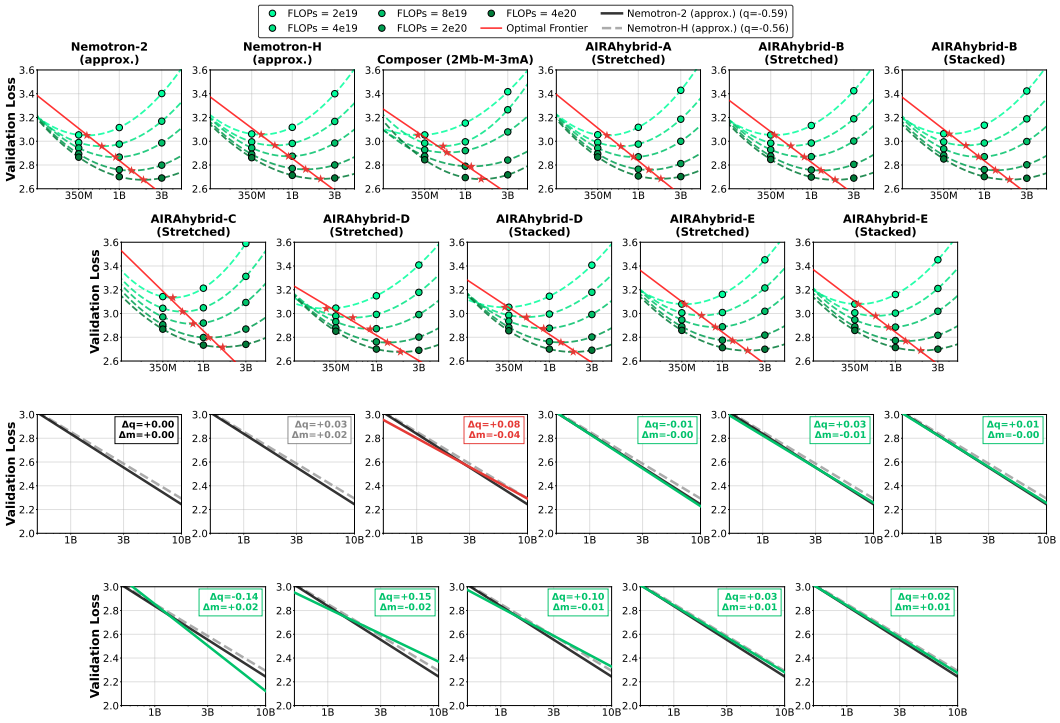

Figure 18: **IsoFLOP scaling curves and optimal frontier comparison (M, mA, Mb), full results. Top:** Validation loss versus model size for each architecture across 5 FLOPs budgets. **Bottom:** Comparison of each architecture's optimal frontier against the approximated Nemotron-2 and Nemotron-H.

B Small Scale Ranking

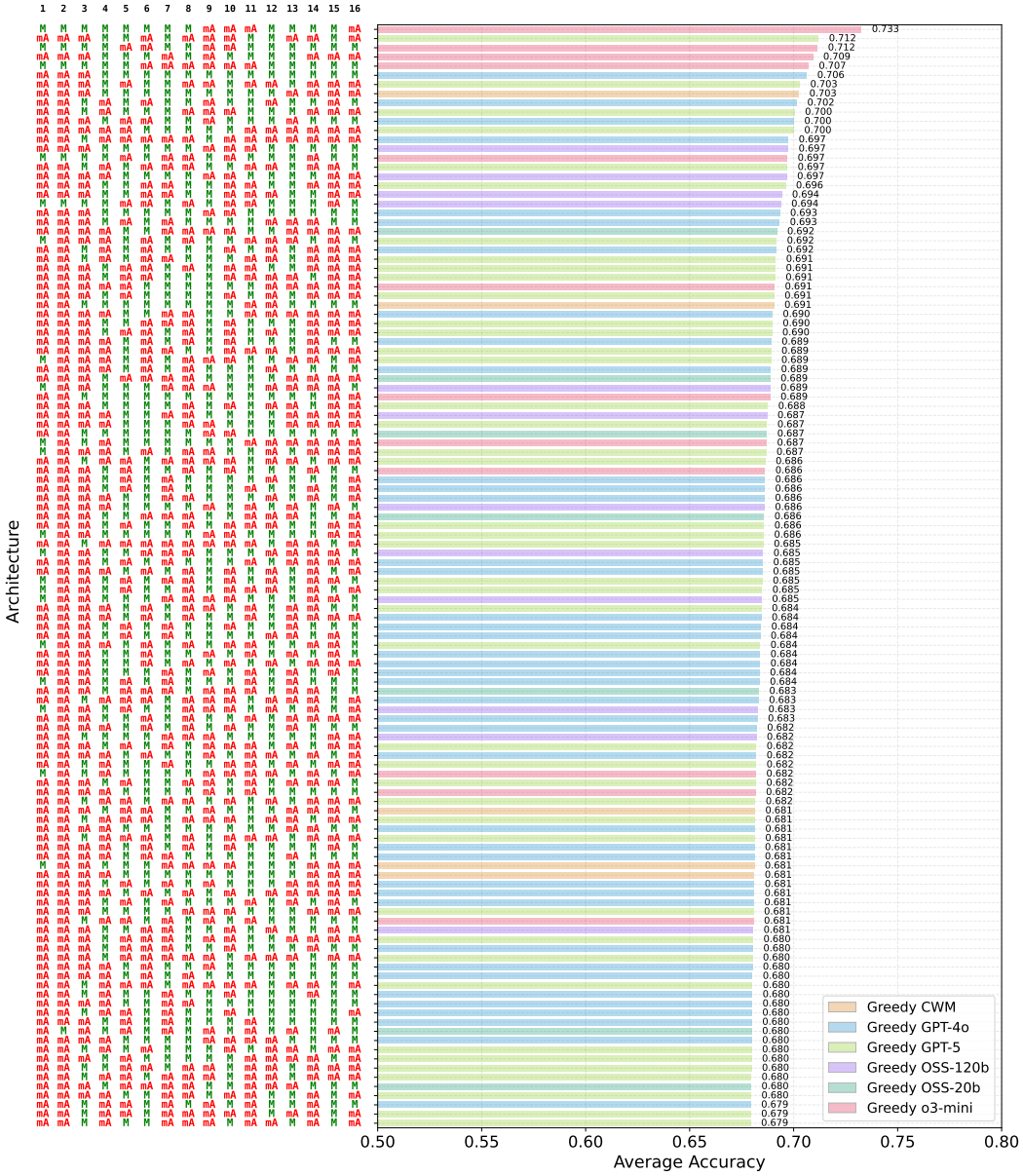

Figure 19: **Agent-discovered Architecture Ranking on MAD 2-Primitives (M, mA).** Top 120 small scale architectures discovered during greedy search, ranked by test performance. Each row displays the primitive sequence (M in green, mA in red) alongside its accuracy. Bar colors indicate the discovering agent.

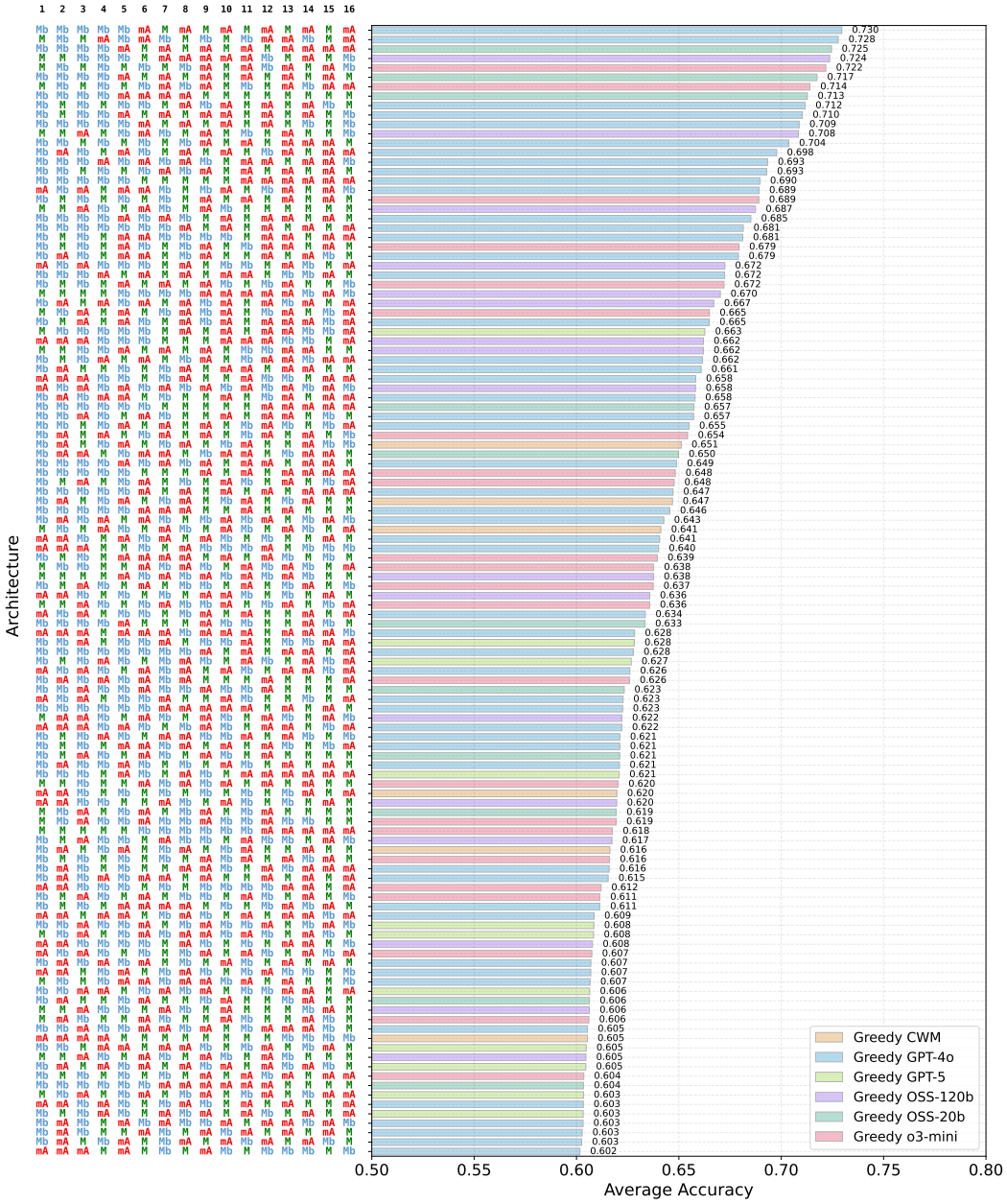

Figure 20: **Agent-discovered Architecture Ranking on MAD 3-Primitives (M, mA, Mb).** Top 120 small scale architectures discovered during greedy search, ranked by test performance. Each row displays the primitive sequence (M in green, mA in red, Mb in blue) alongside its accuracy. Bar colors indicate the discovering agent.

C Aggregation and Scaling Patterns

C.1 Aggregation methods

In the two-primitive space:

- **AIRAformer-A** (base pattern: `(A + M) + (2A + 2M) + 4 × (A + M) + 2M`): Obtained from the top-20 performing architectures on each of the 3 datasets. Within each dataset, the architectures are weighted using a strong exponential decay inversely proportional to their rank. Finally, the 3 datasets are given equal weight in the overall aggregation.
- **AIRAformer-B** (base pattern: `2A + 5 × (A + M) + 4M`): Obtained via N_1 aggregation on the top-400 architectures across all 5 agents on the MAD dataset.
- **AIRAformer-C** (base pattern: `(2A + M) + 3 × (A + M) + (2A + M) + 4A`): Obtained via N_1 aggregation on the top cluster after k-means clustering (using 3 clusters) on architectures derived from 20 seeds search of greedy GPT-5 agent on the MAD dataset.
- **AIRAformer-D** (base pattern: `5 × (2A + M) + A`): Obtained via N_2 aggregation on the top cluster after k-means clustering (using 3 clusters) on architectures derived from 20 seeds search of greedy GPT-5 agent on the MAD dataset.

In the three-primitive space:

- **AIRAhybrid-A** (base pattern: `2Mb + M + 11Mb + 2M`): Obtained via N_0 aggregation on the top cluster after k-means clustering (using 3 clusters) on the architectures found by greedy GPT-5.
- **AIRAhybrid-B** (base pattern: `3 × (2Mb + M + A) + 2Mb + 2M`): Obtained via N_1 aggregation on the top cluster after k-means clustering (using 3 clusters) on the architectures found by greedy GPT-5.
- **AIRAhybrid-C** (base pattern: `2Mb + A + 2 × (Mb + A) + M + 2 × (A + Mb + A + M)`): Obtained via N_2 aggregation on the top cluster after k-means clustering (using 3 clusters) on the architectures found by greedy GPT-5.
- **AIRAhybrid-D** (base pattern: `2Mb + M + 2 × (Mb + M) + A + M + Mb + M + A + M + Mb + M + A`): Obtained via N_1 aggregation on the top cluster after k-means clustering (using 3 clusters) on the architectures found by all 6 agents.
- **AIRAhybrid-E** (base pattern: `5 × (Mb + M + A) + M`): Obtained via N_2 aggregation on the top cluster after k-means clustering (using 3 clusters) on the architectures found by all 6 agents.

C.2 Scaling patterns

We provide a breakdown of the primitives and their implementation at small scale (16-layer proxy) and large scale (350M, 1B, 3B) below. At small scale, all primitives share a model dimension $d = 128$. At large scale, the model dimension d, number of attention heads n_h, head dimension d_h, and hidden dimensions are configured per scale as detailed in the IsoFLOP methodology section below.

- **MLP (M):** A feed-forward network. At small scale, a standard two-layer MLP with ReLU activation and hidden dimension $h_{\text{mlp}} = 258$. At large scale, a SwiGLU variant (Shazeer, 2020) with gated projections: gate and up projections $d \to h_{\text{mlp}}$ and a down projection $h_{\text{mlp}} \to d$, where h_{mlp} is computed from d and a scale-dependent expansion factor (see below).
- **Multi-head Attention (mA):** Standard multi-head causal self-attention (Vaswani et al., 2017). At small scale, we use $n_h = 16$ query heads with head dimension $d_h = 8$ and $n_{\text{kv}} = 16$ key-value heads (i.e., multi-head attention without grouping). At large scale, grouped-query attention (GQA) (Ainslie et al., 2023) is employed with $n_{\text{kv}} = 8$ key-value heads shared across a larger number of query heads, yielding a total KV dimension of $d_{\text{kv}} = n_{\text{kv}} \times d_h$.

- **Mamba (Mb):** A selective state-space model (SSM) based on Mamba-2 (Dao and Gu, 2024), which processes sequences with linear complexity through data-dependent gating and a hardware-efficient selective scan. At small scale, the SSM uses an expansion factor $e_{\text{ssm}} = 1.25$ (yielding SSM hidden dimension $d_{\text{ssm}} = e_{\text{ssm}} \cdot d = 160$), a state dimension $n_s = 4$, a convolution kernel size $k = 4$, and $n_g = 1$ group. At large scale, the SSM hidden dimension is instead computed as $d_{\text{ssm}} = \texttt{multiple_of}(\lfloor \frac{2}{3} \cdot 3d \rfloor, 256)$, with a larger state dimension n_s and SSM head dimension $d_h^{\text{ssm}} = 64$ (see below for per-scale values).

Table 6: 2-primitive architecture configurations across parameter scales. We report the full layer pattern at each scale, total depth (L), parameter counts (with / without embeddings, in billions), and the approximated ratio of Attention to MLP blocks (A:M).

Architecture	Scale	L	Params (B)	A:M	Pattern
Llama 3.2	**350M**	28	0.35 / 0.55	1.0:1	14×(A-M)
	1B	32	0.97 / 1.23	1.0:1	16×(A-M)
	3B	56	2.82 / 3.21	1.0:1	28×(A-M)
Composite (Stacked)	**350M**	24	0.35 / 0.55	0.5:1	4×(2A-4M)
	1B	27	1.00 / 1.26	0.5:1	4×(2A-4M)-A-2M
	3B	51	3.00 / 3.39	0.5:1	8×(2A-4M)-A-2M
Composite (Stretched)	**350M**	26	0.39 / 0.59	0.4:1	3A-8M-3A-5M-2A-5M
	1B	29	1.06 / 1.32	0.5:1	4A-9M-4A-5M-2A-5M
	3B	54	3.17 / 3.56	0.5:1	7A-16M-7A-10M-4A-10M
AIRAformer-A (Stretched)	**350M**	26	0.34 / 0.54	0.8:1	2×(A-M-2A-2M-3×(A-M)-A-3M)
	1B	32	1.05 / 1.31	0.8:1	2×(A-M-2A-2M)-4×(2A-2M)-4M
	3B	56	2.97 / 3.36	0.8:1	2×(A-M-4A-4M-6×(A-M)-2A-6M)
AIRAformer-B (Stacked)	**350M**	23	0.35 / 0.55	0.4:1	(3A-4×(M-A)-5M)-7M
	1B	32	1.05 / 1.31	0.8:1	2×(3A-4×(M-A)-5M)
	3B	53	2.94 / 3.33	0.7:1	3×(3A-4×(M-A)-5M)-5M
AIRAformer-C (Stacked)	**350M**	34	0.34 / 0.54	2.4:1	2×(2×(2A-M)-3×(A-M)-4A)-2A
	1B	48	1.10 / 1.36	2.2:1	3×(2×(2A-M)-3×(A-M)-4A)
	3B	76	2.92 / 3.31	2.4:1	5×(2×(2A-M)-3×(A-M)-4A)-6A
AIRAformer-C (Stretched)	**350M**	34	0.35 / 0.55	3.0:1	4A-2M-2A-2M-2A-2M-2A-8A
	1B	48	1.10 / 1.36	3.0:1	6A-3M-3A-3M-3A-3M-3A-3M-6A-3M-12A
	3B	70	2.92 / 3.31	2.4:1	10A-5M-5A-5M-5A-5M-5A-20A-3A
AIRAformer-D (Stacked)	**350M**	34	0.34 / 0.54	2.4:1	2×(5×(2A-M)-A)-2A
	1B	48	1.10 / 1.36	2.2:1	3×(5×(2A-M)-A)
	3B	76	2.92 / 3.31	2.4:1	5×(5×(2A-M)-A)-6A
AIRAformer-D (Stretched)	**350M**	34	0.35 / 0.55	2.0:1	4A-2M-4A-2M-4A-2M-4A-2M-4A-2M-2A-M
	1B	48	1.10 / 1.36	2.2:1	6A-3M-6A-3M-6A-3M-6A-3M-6A-3M-3A
	3B	70	2.92 / 3.31	2.0:1	10A-5M-10A-5M-10A-5M-10A-5M-10A-5M-5A-3M

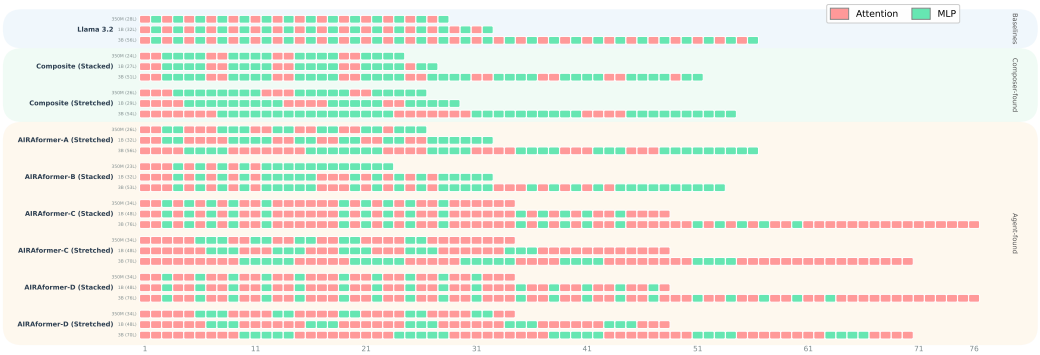

Figure 21: **Architecture layer patterns across three parameter scales (350M, 1B, 3B), 2-primitive case**. Each row within an architecture shows the sequence of Attention (red) and MLP (green) layers. We compare our 6 agent-discovered architectures (AIRAformer A-D) with 1 baseline, Llama 3.2, approximated to the desired scales, and 2 Composer-found architectures.

Table 7: 3-primitive architecture configurations across parameter scales. We report the full layer pattern at each scale, total depth (L), parameter counts (with / without embeddings, in billions), and the number of Attention (A), MLP (M), and Mamba (Mb) blocks.

Architecture	Scale	L	A	M	Mb	Params (B)	Pattern
Nemotron-H	**350M**	18	2	8	8	0.36 / 0.55	Mb-M-A-5×(M-Mb)-A-M-Mb-M-Mb
(Approx.)	**1B**	28	3	13	12	1.00 / 1.26	Mb-M-2×(A-4×(M-Mb)-M)-A-M-Mb-M
	3B	37	4	17	16	2.96 / 3.36	Mb-M-2×(A-4×(M-Mb)-M)-A-2×(M-Mb)-M-A-M-3×(Mb-M)
Nemotron-2	**350M**	17	2	7	8	0.33 / 0.52	3×(Mb-M)-Mb-A-4×(M-Mb)-A-M-Mb
(Approx.)	**1B**	29	3	13	13	1.02 / 1.28	2×(3×(Mb-M)-Mb-A-M)-Mb-M-Mb-A-5×(M-Mb)-M
	3B	37	4	16	17	2.91 / 3.30	2×(3×(Mb-M)-Mb-A-M)-Mb-M-Mb-A-7×(M-Mb)-A-M-Mb
Mamba	**350M**	17	0	8	9	0.36 / 0.56	8×(Mb-M)-Mb
(Mb+M)	**1B**	26	0	13	13	0.99 / 1.25	13×(Mb-M)
	3B	35	0	17	18	2.98 / 3.37	17×(Mb-M)-Mb
Composer	**350M**	26	12	4	10	0.34 / 0.53	4×(2Mb-M-3A)-2Mb
(2Mb-M-3A)	**1B**	45	21	8	16	1.04 / 1.30	7×(2Mb-M-3A)-2Mb-M
	3B	57	27	10	20	2.98 / 3.38	9×(2Mb-M-3A)-2Mb-M
AIRAhybrid-A	**350M**	20	0	3	17	0.33 / 0.53	2Mb-M-14Mb-2M-Mb
(Stretched)	**1B**	32	0	6	26	0.97 / 1.24	4Mb-2M-22Mb-4M
	3B	44	0	8	36	3.01 / 3.41	6Mb-3M-30Mb-5M
AIRAhybrid-B	**350M**	20	4	6	10	0.34 / 0.54	3×(2Mb-M-A)-3×(Mb-M)-A-Mb
(Stretched)	**1B**	32	6	10	16	0.98 / 1.24	3×(4Mb-2M-2A)-2×(2Mb-2M)
	3B	43	9	13	21	2.93 / 3.32	3×(5Mb-3M-3A)-3Mb-3M-3Mb-M
AIRAhybrid-B	**350M**	20	5	7	8	0.35 / 0.54	3×(Mb-M-A)-2×(3×(Mb-M)-A)-Mb
(Stacked)	**1B**	32	6	10	16	0.98 / 1.24	2×(3×(2Mb-M-A)-2×(Mb-M))
	3B	43	7	12	24	2.94 / 3.33	2×(3×(3Mb-M-A)-2×(Mb-M))-Mb-M-A-Mb-M
AIRAhybrid-C	**350M**	29	16	3	10	0.33 / 0.53	4Mb-2A-Mb-3×(2A-Mb-2A-M)-Mb-A-Mb-A
(Stretched)	**1B**	40	21	6	13	0.86 / 1.12	5Mb-3A-2Mb-3×(3A-2Mb-3A-2M)
	3B	61	35	6	20	2.73 / 3.12	8Mb-5A-3Mb-2×(5A-3Mb-5A-3M)-5A-3Mb-5A
AIRAhybrid-D	**350M**	18	3	8	7	0.35 / 0.54	2Mb-M-Mb-3×(M-Mb-M-A)-Mb-M
(Stretched)	**1B**	32	6	14	12	1.08 / 1.34	4Mb-2M-2Mb-3×(2M-2Mb-2M-2A)
	3B	39	6	17	16	3.01 / 3.41	5Mb-2M-2Mb-3×(2M-2Mb-2M-2A)-3×(Mb-M)
AIRAhybrid-D	**350M**	18	3	8	7	0.35 / 0.54	Mb-M-Mb-3×(M-Mb-M-A)-Mb-M-Mb
(Stacked)	**1B**	32	6	14	12	1.08 / 1.34	2×(2Mb-3×(M-Mb-M-A))
	3B	39	7	17	15	2.98 / 3.37	2×(2Mb-3×(M-Mb-M-A))-3×(Mb-M)-A
AIRAhybrid-E	**350M**	20	6	7	7	0.34 / 0.54	5×(Mb-M-A)-M-Mb-M-A-Mb
(Stretched)	**1B**	32	10	12	10	0.97 / 1.23	5×(2Mb-2M-2A)-2M
	3B	44	14	15	15	2.93 / 3.32	4×(3Mb-3M-3A)-3Mb-3M-2A
AIRAhybrid-E	**350M**	20	6	7	7	0.34 / 0.54	5×(Mb-M-A)-M-Mb-M-A-Mb
(Stacked)	**1B**	32	10	12	10	0.97 / 1.23	2×(5×(Mb-M-A)-M)
	3B	44	14	16	14	2.98 / 3.38	2×(5×(Mb-M-A)-M)-4×(Mb-M-A)

C.3 IsoFLOP Budget Methodology

We compute the training floating-point operations (FLOPs) per token per layer for each primitive using the standard convention. Each linear operation of shape $(M, K) \times (K, N)$ costs $2MKN$ FLOPs in the forward pass. This is multiplied by a factor of 3 for training (accounting for one forward and two backward passes), yielding $6 \cdot d_{\text{in}} \cdot d_{\text{out}}$ FLOPs per projection per token.

Model Configurations. All three model scales share a sequence length $s = 8{,}192$, $n_{\text{kv}} = 8$ key-value heads, and a constant batch size of $524{,}288$ tokens per step. The batch size is derived via DP × local_bs × acc_steps × s (e.g., $16 \times 4 \times 1 \times 8{,}192$ at the 1B scale). Embeddings are tied with vocabulary size $V = 128{,}256$.

For the **2-primitive experiments** (Attention + MLP):

- **350M:** $d = 1536$, $n_h = 24$, $d_h = 64$, $d_{\text{kv}} = n_{\text{kv}} \times d_h = 512$
- **1B:** $d = 2048$, $n_h = 32$, $d_h = 64$, $d_{\text{kv}} = 512$
- **3B:** $d = 3072$, $n_h = 24$, $d_h = 128$, $d_{\text{kv}} = 1024$

The SwiGLU MLP hidden dimension is computed as $h_{\text{mlp}} = \texttt{multiple_of}(\lfloor \frac{2}{3} \cdot 4d \cdot f_{\text{mlp}} \rfloor, 1024)$, where f_{mlp} is a scale-dependent expansion factor. For the 2-primitive case, $f_{\text{mlp}} = 1.0$ at 350M and 3B, and $f_{\text{mlp}} = 1.4$ at 1B. This yields $h_{\text{mlp}} = 4{,}096$ (350M), $h_{\text{mlp}} = 8{,}192$ (1B), and $h_{\text{mlp}} = 8{,}192$ (3B).



Figure 22: **Architecture layer patterns across three parameter scales (350M, 1B, 3B), 3-primitive case**. Each row within an architecture shows the sequence of Attention (red), MLP (green) and Mamba (blue) layers. We compare our 8 agent-discovered architectures (AIRAhybrid A-E) with 3 baselines (Nemotron-H, Nemotron-2, and Mamba) approximated to the desired scales, and 1 Composer-found architecture.

For the **3-primitive experiments** (Attention + MLP + Mamba):

- **350M:** $d = 1536$, $n_h = 24$, $d_h = 64$, $d_{\text{kv}} = 512$
- **1B:** $d = 2048$, $n_h = 32$, $d_h = 64$, $d_{\text{kv}} = 512$
- **3B:** $d = 3072$, $n_h = 32$, $d_h = 96$, $d_{\text{kv}} = 768$

The MLP expansion factor is $f_{\text{mlp}} = 1.4$ for 350M and 1B, and $f_{\text{mlp}} = 1.5$ for 3B, resulting in $h_{\text{mlp}} = 6{,}144$ (350M), $h_{\text{mlp}} = 8{,}192$ (1B), and $h_{\text{mlp}} = 12{,}288$ (3B). The Mamba-2 SSM uses: state dimension $n_s = 128$ (350M and 1B) or $n_s = 256$ (3B); convolution kernel size $k = 4$; $n_g = 1$ group; and SSM head dimension $d_h^{\text{ssm}} = 64$. The SSM hidden dimension is $d_{\text{ssm}} = \texttt{multiple_of}(\lfloor \frac{2}{3} \cdot 3d \rfloor, 256)$, yielding $d_{\text{ssm}} = 3{,}072$ (350M), $4{,}096$ (1B), and $6{,}144$ (3B). The number of SSM heads is $n_{\text{ssm}} = d_{\text{ssm}}/d_h^{\text{ssm}}$, giving 48, 64, and 96 heads, respectively.

Per-Primitive FLOPs per Token (Training). For **Grouped-Query Attention**, the computational cost comprises four linear projections—Q ($d \to d$), K ($d \to d_{\text{kv}}$), V ($d \to d_{\text{kv}}$), and O ($d \to d$)—alongside the quadratic attention computation ($QK^\top$ and softmax $\cdot\, V$):

$$F_{\text{Attn}} = 6(2d^2 + 2d \cdot d_{\text{kv}}) + 12sd \tag{1}$$

For the **SwiGLU MLP**, the three projections (gate: $d \to h_{\text{mlp}}$; up: $d \to h_{\text{mlp}}$; down: $h_{\text{mlp}} \to d$) result in:

$$F_{\text{MLP}} = 6 \cdot 3 \cdot d \cdot h_{\text{mlp}} = 18\, d\, h_{\text{mlp}} \tag{2}$$

For the **Mamba-2 SSM**, the cost decomposes into four components: (i) an input projection $d \to d_{\text{in_proj}}$ where $d_{\text{in_proj}} = 2d_{\text{ssm}} + 2n_g n_s + n_{\text{ssm}}$; (ii) a depthwise $\texttt{conv1d}$ over $d_{\text{conv}} = d_{\text{ssm}} + 2n_g n_s$ channels with kernel size k; (iii) the selective scan, costing $2 \cdot n_{\text{ssm}} \cdot d_h^{\text{ssm}} \cdot n_s$ per token; and (iv) an output projection $d_{\text{ssm}} \to d$. This yields the following total layer cost:

$$F_{\text{SSM}} = 6(d \cdot d_{\text{in_proj}} + d_{\text{conv}} \cdot k + 2\, n_{\text{ssm}}\, d_h^{\text{ssm}}\, n_s + d_{\text{ssm}} \cdot d) \tag{3}$$

Training Steps from FLOP Budget. Given an architecture with L_A attention layers, L_M MLP layers, and L_{Mb} Mamba layers, the total FLOPs per training step are:

$$C_{\text{step}} = (L_A \cdot F_{\text{Attn}} + L_M \cdot F_{\text{MLP}} + L_{\text{Mb}} \cdot F_{\text{SSM}}) \times B \tag{4}$$

where $B = 524{,}288$ is the constant batch size in tokens.

The number of training steps T for a target FLOP budget $\mathcal{F}$ is then $T = \lfloor \mathcal{F}/C_{\text{step}} \rfloor$.

For example, at the 1B scale, the Composite (Stacked) architecture ($L_A = 10$, $L_M = 19$) yields $C_{\text{step}} \approx 4.39 \times 10^{15}$ FLOPs/step, giving $T = \lfloor 2 \times 10^{19} / (4.39 \times 10^{15}) \rfloor = 4{,}552$ steps for a budget of 2×10^{19} FLOPs (matching Table 8).

The full set of training steps for all architectures and budgets is reported in Tables 8 and 9.

Table 8: Training steps required to reach target isoFLOP budgets. We include the count of Attention (A) and MLP (M) layers to contextualize the varying computational intensity per step. Calculations are based on a constant batch size of 524,288 tokens per step. Model base configurations: 350M ($d = 1536$, $h = 4096$), 1B ($d = 2048$, $h = 8192$), 3B ($d = 3072$, $h = 8192$).

				Target FLOP Budget				
Scale	**Architecture**	**A**	**M**	**2e19**	**4e19**	**8e19**	**2e20**	**4e20**
350M	Composite (Stacked)	8	16	11,484	22,967	45,934	114,835	229,670
	Composite (Stretched)	8	18	10,751	21,501	43,002	107,505	215,011
	AIRAformer-A (Stretched)	12	14	9,907	19,815	39,629	99,073	198,147
	AIRAformer-B (Stacked)	7	16	12,175	24,351	48,701	121,753	243,506
	AIRAformer-C (Stacked)	24	10	6,737	13,474	26,948	67,370	134,740
	AIRAformer-C (Stretched)	23	11	6,828	13,656	27,312	68,280	136,561
	AIRAformer-D (Stacked)	24	10	6,737	13,474	26,948	67,370	134,740
	AIRAformer-D (Stretched)	23	11	6,828	13,656	27,312	68,280	136,561
1B	Composite (Stacked)	10	19	4,552	9,104	18,208	45,520	91,041
	Composite (Stretched)	10	19	4,552	9,104	18,208	45,520	91,041
	AIRAformer-A (Stretched)	14	18	4,176	8,352	16,703	41,758	83,517
	AIRAformer-B (Stacked)	14	18	4,176	8,352	16,703	41,758	83,517
	AIRAformer-C (Stacked)	33	15	2,879	5,758	11,516	28,791	57,581
	AIRAformer-C (Stretched)	33	15	2,879	5,758	11,516	28,791	57,581
	AIRAformer-D (Stacked)	33	15	2,879	5,758	11,516	28,791	57,581
	AIRAformer-D (Stretched)	33	15	2,879	5,758	11,516	28,791	57,581
3B	Composite (Stacked)	17	34	1,651	3,302	6,605	16,512	33,024
	Composite (Stretched)	18	36	1,559	3,119	6,238	15,595	31,190
	AIRAformer-A (Stretched)	25	31	1,504	3,008	6,015	15,038	30,076
	AIRAformer-B (Stacked)	21	32	1,589	3,178	6,356	15,889	31,778
	AIRAformer-C (Stacked)	56	20	1,108	2,216	4,432	11,081	22,161
	AIRAformer-C (Stretched)	47	23	1,203	2,406	4,812	12,030	24,061
	AIRAformer-D (Stacked)	56	20	1,108	2,216	4,432	11,081	22,161
	AIRAformer-D (Stretched)	47	23	1,203	2,406	4,812	12,030	24,061

Table 9: Training steps required to reach target isoFLOP budgets across 12 architectures and 3 parameter scales. We report the count of Mamba (Mb), MLP (M), and Attention (A) layers to contextualize computational intensity. Calculations assume a constant batch size of 524,288 tokens per step.

					Target FLOP Budget				
Scale	**Architecture**	**Mb**	**M**	**A**	**2e19**	**4e19**	**8e19**	**2e20**	**4e20**
350M	Nemotron-H (Approx.)	8	8	2	15,402	30,804	61,608	154,020	308,040
	Nemotron-2 (Approx.)	8	8	1	16,672	33,344	66,688	166,720	333,440
	Composer (2Mb-M-3A)	10	4	12	15,331	30,662	61,324	153,310	306,620
	AIRAhybrid-A (Stretched)	16	4	0	17,660	35,320	70,640	176,600	353,200
	AIRAhybrid-B (Stretched)	10	6	4	14,826	29,652	59,304	148,260	296,520
	AIRAhybrid-B (Stacked)	10	6	4	14,826	29,652	59,304	148,260	296,520
	AIRAhybrid-C (Stretched)	9	4	16	8,416	16,832	33,664	84,160	168,320
	AIRAhybrid-D (Stretched)	7	8	3	12,521	25,042	50,084	125,210	250,420
	AIRAhybrid-D (Stacked)	7	8	3	12,521	25,042	50,084	125,210	250,420
	AIRAhybrid-E (Stretched)	6	8	6	12,521	25,042	50,084	125,210	250,420
	AIRAhybrid-E (Stacked)	6	8	6	12,521	25,042	50,084	125,210	250,420
1B	Nemotron-H (Approx.)	12	13	3	5,732	11,464	22,928	57,320	114,640
	Nemotron-2 (Approx.)	13	13	3	5,596	11,192	22,384	55,960	111,920
	Composer (2Mb-M-3A)	16	8	21	4,894	9,788	19,576	48,940	97,880
	AIRAhybrid-A (Stretched)	26	6	0	6,351	12,702	25,404	63,510	127,020
	AIRAhybrid-B (Stretched)	16	10	6	4,922	9,844	19,688	49,220	98,440
	AIRAhybrid-B (Stacked)	16	10	6	4,922	9,844	19,688	49,220	98,440
	AIRAhybrid-C (Stretched)	13	6	21	4,033	8,066	16,132	40,330	80,660
	AIRAhybrid-D (Stretched)	12	14	6	4,841	9,682	19,364	48,410	96,820
	AIRAhybrid-D (Stacked)	12	14	6	4,841	9,682	19,364	48,410	96,820
	AIRAhybrid-E (Stretched)	10	12	10	4,841	9,682	19,364	48,410	96,820
	AIRAhybrid-E (Stacked)	10	12	10	4,841	9,682	19,364	48,410	96,820
3B	Nemotron-H (Approx.)	16	17	4	1,978	3,956	7,912	19,780	39,560
	Nemotron-2 (Approx.)	17	17	3	1,986	3,972	7,944	19,860	39,720
	Composer (2Mb-M-3A)	20	10	27	1,882	3,764	7,528	18,820	37,640
	AIRAhybrid-A (Stretched)	36	8	0	2,033	4,066	8,132	20,330	40,660
	AIRAhybrid-B (Stretched)	22	13	8	1,881	3,762	7,524	18,810	37,620
	AIRAhybrid-B (Stacked)	22	13	8	1,881	3,762	7,524	18,810	37,620
	AIRAhybrid-C (Stretched)	20	9	32	1,361	2,722	5,444	13,610	27,220
	AIRAhybrid-D (Stretched)	15	17	7	1,703	3,406	6,812	17,030	34,060
	AIRAhybrid-D (Stacked)	15	17	7	1,703	3,406	6,812	17,030	34,060
	AIRAhybrid-E (Stretched)	14	16	14	1,703	3,406	6,812	17,030	34,060
	AIRAhybrid-E (Stacked)	14	16	14	1,703	3,406	6,812	17,030	34,060

D Disentangling Agent Reasoning from Statistical Aggregation

AIRA-Compose combines agentic exploration of primitive arrangements with statistical aggregation of promising proxy architectures before scaling. It can be argued that the successful discovery of neural architectures is mainly derived from the latter rather than genuine agentic capabilities. To decouple their respective contributions, we study three cases:

- changing the input quality while keeping aggregation fixed;
- scaling candidates with and without aggregation;
- tracking search and aggregation over time;

The analysis is performed on the 2-primitive search experiments over the MAD dataset proxy and employing N_0 aggregation.

Aggregated performance derives from input quality. We apply the same aggregation operator to uniformly sampled, random architectures, the bottom-400 agent discoveries, and a top-ranked agent pool, yielding AIRAformer-B. All three rely on rankings shown in Fig.19. Table 10 reports the resulting proxy performance.

Table 10: Input-quality control with a fixed aggregation operator. Each character denotes one layer. The top-ranked pool yields AIRAformer-B.

Input pool	Output architecture	Accuracy ↑
Uniform random	Varies across runs	0.579 ± 0.053
Bottom-400 agent discoveries	AMAAMAMAAAAMAMAA	0.525
Top-ranked agent discoveries	AAAMAMAMAMAMMMMM	0.675

Applying the same operator to low-quality inputs does not reproduce the performance of the top-ranked pool. These controls establish that aggregation performance depends on candidate-pool quality, consistent with agentic search supplying useful architectural arrangements. They do not, by themselves, isolate the contribution of the agent's internal reasoning from the search scaffold, evaluation feedback, or ranking procedure.

Aggregation works as a denoiser useful for scaling. We compare aggregated AIRAformer patterns with the highest-scoring individual proxy architecture scaled without aggregation. Table 11 reports the base patterns, MAD proxy accuracies where available, and validation loss and perplexity after 1B-parameter pretraining with fixed 37.5B-token budget.

Table 11: Aggregated patterns and an individual top proxy at the 1B scale. AIRAformer-A aggregates discoveries across MAD, BabiStories, and DCLM, so it has no single MAD proxy score.

Architecture	Base pattern	Proxy acc. ↑	1B loss ↓	1B ppl ↓
AIRAformer-A (Str.)	AMAAMMAMAMAMAMMM	—	2.752	15.67
AIRAformer-B (St.)	AAAMAMAMAMAMMMMM	0.675	2.752	15.67
AIRAformer-C (St.)	AAMAMAMAMAAMAAAA	0.635	2.751	15.66
AIRAformer-C (Str.)	AAMAMAMAMAAMAAAA	0.635	2.750	15.64
AIRAformer-D (St.)	AAMAAMAAMAAMAAMA	0.669	2.745	15.56
AIRAformer-D (Str.)	AAMAAMAAMAAMAAMA	0.669	**2.734**	**15.39**
Top proxy, no aggregation	MMMMMMMMAAAMMMMA	**0.733**	2.792	16.31

The individual top proxy achieves the highest MAD accuracy, but its 1B validation loss (2.792) is worse than that of every aggregated pattern (2.734–2.752). AIRAformer-D (Str.) reduces validation loss by 0.058 and perplexity by 0.92: maximizing the score of a single small-scale architecture does not necessarily identify the best pattern for larger-scale language modeling.

This comparison supports aggregation as a means of extracting patterns that transfer across scales despite lower peak proxy performance. It is consistent with reducing sensitivity to proxy-specific effects, although one individually scaled control cannot establish that all unaggregated candidates transfer poorly or quantify the average causal benefit of aggregation. The difference between stacked and stretched AIRAformer-D also shows that the scaling prescription contributes to the final performance.

The final architectures are discovered, not synthesized. Table 12 tracks GPT-5 search over 20 seeds. At each checkpoint, we report the best-so-far proxy accuracy and the accuracy of the architecture obtained by aggregating candidates evaluated up to that point. The two metrics measure different objects: one summarizes individual discoveries, while the other evaluates the pattern produced from the accumulated pool.

Table 12: Search and aggregation dynamics for GPT-5 on MAD over 20 seeds. The aggregate score follows the improvements deriving from agentic search, and the aggregated architecture converges to AIRAformer-D pattern at step 21.

Search step	Best-so-far ↑	Aggregate acc. ↑	Aggregated architecture
1	0.600	0.642	AAMAMMAMAMAMAAMM
5	0.670	0.666	AAMAMMAMAMAAMAMA
9	0.677	0.636	AAMAMAMAAMAMAAMA
13	0.680	0.651	AAMAMAMAAMAMAMAA
17	0.680	0.669	AAMAAMAAMAMAAMAA
21	0.683	0.669	AAMAAMAAMAAMAAMA (AIRAformer-D)
25–49	$0.684 \rightarrow 0.688$	0.669	AAMAAMAAMAAMAAMA (AIRAformer-D)

By step 21, aggregation selects AIRAformer-D and retains that pattern through step 49 even as the individual best-so-far score continues to improve. Aggregation therefore provides a stable summary of an evolving discovery pool driven by the agent and its harness, rather than simply reproducing its highest-scoring member.

Inspection of the search logs further shows that often reported AIRAformer base architecture is an exact architecture evaluated by an agent during search. AIRAformer-D was independently rediscovered in runs using o3-mini, GPT-4o, and GPT-5. These observations establish that the final reported arrangements occurred in agent exploration, even though majority voting could, in principle, construct a previously unevaluated combination. Repeated discovery across different agents provides additional evidence that AIRAformer-D is not specific to a single trajectory.

Summary. Changing the candidate pool while fixing the aggregation operator changes the quality of the resulting architecture; the search logs show that the reported final aggregated patterns were already evaluated by agents; and the 1B comparison shows that an aggregated pattern can transfer better than the highest-scoring individual proxy. Aggregation stabilizes recurring layer choices, while agentic exploration and its evaluation scaffold supply the pool from which those choices are extracted. The evidence attributes the gains to the combined search, selection, and aggregation pipeline at the evaluated scales. It does not separately identify internal LLM reasoning as the sole cause, establish that majority voting cannot synthesize arrangements, or guarantee performance under substantially larger parameter or token budgets. Isolating reasoning from the scaffold and measuring transfer over a broader set of unaggregated candidates remain useful directions for further ablation.

E AIRA-Compose Task Artifacts

E.1 `project_description.md`

```
# Overview

## Task Description

### Overview
This is a Machine Learning project and your goal is to complete the project's TASK following the instructions provided below.

**TASK:** Design and submit a hybrid language model architecture that achieves the **highest average accuracy** across the **6 MAD (Mechanistic Architecture Design) benchmark tasks**.

This is a Neural Architecture Search (NAS) task for discovering optimal hybrid Language Model architectures. Modern language models typically use a single repeated pattern (e.g., Transformer blocks with alternating attention and feedforward layers). However, recent research suggests that **hybrid architectures** - models that combine different types of computational primitives in varied arrangements - can achieve better performance.

Your goal is to **design and submit a hybrid language model architecture** (a sequence of computational primitives) that achieves the **highest average accuracy** across the 6 MAD (Mechanistic Architecture Design) synthetic benchmark tasks. You should produce a string called `model_string` (a space-separated sequence of primitive names) to be saved in `submission.csv`.

**Important Note:** The **only thing you should change** is the layer architecture definition (i.e., the sequence of primitives: `mh-attention` and `mlp`). All other training parameters are fixed:
- Number of primitives: **16 primitives** (required)
- Optimizer: AdamW (default)
- Training epochs: 200 epoch per task
- Batch size: 128 (default)
- Model dimension: 128 (default)
- **Note:** lr, min-lr, and weight-decay parameters use task defaults and are NOT specified in the training command

This constraint ensures that improvements in accuracy are solely due to architectural choices.
```

```
23
24  **The search space is constrained:**
25  - You can choose from **2 available computational primitives**: `mh-attention` and `mlp`
26  - You must arrange exactly **16 primitives** to form your architecture
27  - Total possible architectures: **2^16 = 65,536** possibilities
28
29  Your task is to intelligently select an architecture from this space. You should:
30  - Determine the optimal ratio of attention to MLP layers
31  - Consider the ordering/pattern of these primitives (e.g., alternating, clustered, etc.)
32  - Think about what capabilities the task requires (memory, transformation, etc.)
33  - Draw insights from machine learning literature on Transformer architectures
34
35  ### Architecture Representation
36
37  An architecture is a **space-separated string of primitive names**. For example:
38  - "mh-attention mlp mh-attention mlp" (4 primitives)
39
40  **CRITICAL: Architecture Evaluation Details**
41
42  The submitted architecture will be evaluated by **sequentially stacking the specified
        primitives in the exact order provided**, with no additional modifications. The training
        framework will NOT apply:
43  - Layer normalization or batch normalization between layers
44  - Skip connections or residual connections
45  - Dropout or other regularization techniques beyond what's built into each primitive
46  - Any architectural modifications or enhancements
47
48  Your task is to **reason about and justify the optimal sequence of primitives** based on their
         inherent capabilities and interactions. Do not attempt to implement or test the
        architecture yourself with additional tricks, as these will not be present during
        evaluation. Focus solely on the intelligent arrangement of primitives.
49
50  ### Available Computational Primitives
51
52  You have **2 available computational primitives** to choose from:
53
54  1. **`mh-attention`** - Multi-head Attention (standard Transformer mechanism)
55     - Enables the model to attend to different positions in the sequence
56     - Provides the memory and context-awareness capabilities
57
58  2. **`mlp`** - Multi-Layer Perceptron (feedforward network)
59     - Provides non-linear transformations and feature processing
60     - Complements attention with point-wise computation
61
62  All primitives are defined in: `composer/mad_lab/configs/layers/` as `.yaml` files. You must
        not modify their definitions.
63
64  ### Output Metric
65
66  Your goal is to maximize the **average accuracy** across all 6 MAD benchmark tasks. Results of
         each training are stored under `/tmp/logs/<run_details>/`, with `<run_details>` being
        the string generated by the `mad_lab/train.py` script corresponding to the current
        architecture, task, and optimization parameters.
67
68  In the logs directory for each task, you will find:
69  - `results.csv` - Contains the final test accuracy score under the `test_accuracy` column
70  - `metrics.csv` - Contains the training evolution with metrics logged at each step (useful for
         monitoring training progress)
71
72  The final score is calculated as the **average of the `test_accuracy` values across all 6
        tasks**.
73
74  ## Submission File
75
76  - You must write Python code that generates a file named `submission.csv`.
77  - The file must contain a **single line** with your model architecture string.
78  - **Your model string must consist of exactly 16 primitives, separated by spaces.**
79  - Submissions with more or fewer than 16 primitives will be rejected.
```

```
- When generating your model string, please include comments in your Python code explaining the reasoning behind your choice of primitives and their specific arrangement in the architecture.

## CRITICAL: How to Evaluate Architectures

**YOU MUST USE the provided `evaluate_val.py` script to get REAL accuracy scores. You must not modify `evaluate_val.py`, `train.py`, or any other script provided to you** The evaluation script runs a single train + validation step on the 6 MAD tasks with reduced dataset sizes for faster iteration. An example of a valid solution is given below. Your job is to come up with an original and sensible arrangement of computational primitives.

```python
import subprocess

# Define your architecture (exactly 16 primitives) and justify the arrangement
model_string = "mh-attention mlp mh-attention mlp mh-attention mlp mh-attention mlp mh-attention mlp mh-attention mlp mh-attention mlp"

# Save to submission.csv
with open("submission.csv", "w") as f:
    f.write(model_string)

# Run validation
result = subprocess.run(
    ["python", "data/evaluate_val.py", "--submission-file", "submission.csv"],
    capture_output=True,
    text=True
)
print(result.stdout)
print(result.stderr)
```

DO NOT:
- Generate random scores or simulate validation
- Write your own training loops for this task
- Assume or hallucinate what the accuracy might be

DO:
- Write your `model_string` to `submission.csv`
- Run `python data/evaluate_val.py` to get the validation accuracy

Files in This Directory

- `evaluate_val.py` - Validation script that trains your architecture on all 6 MAD tasks and returns validation average accuracy
- `composer/` - Contains the MAD codebase and training infrastructure
  - `mad_lab/train.py` - Training script (DO NOT MODIFY)
  - `mad_lab/configs/tasks/` - Task configuration files
  - `mad_lab/configs/layers/` - Layer/primitive configuration files

Dataset: MAD (Mechanistic Architecture Design)

The MAD dataset is a set of synthetic tasks designed to provide quick and cheap performance estimates for given LLM architectures. The dataset consists of simple pretext token manipulations that LLMs should be capable of performing.

MAD consists of six categories of tasks:

1. **In-context Recall**: Tests the model's ability to recall key-value associations from context
2. **Fuzzy In-context Recall**: Tests recall with motifs (multi-token keys/values)
3. **Noise In-context Recall**: Tests recall in the presence of noise tokens
4. **Selective Copying**: Tests the model's ability to selectively copy specific tokens
5. **Compression**: Tests the model's ability to compress information (autoencoder task)
6. **Memorization**: Tests the model's ability to memorize key-value mappings
```

```

**Data Source:** The dataset is loaded from the global shared data directory (`NAS/benchmark/MAD/data`) and made available to the evaluation container.

**Training Configuration:**
- Each task: 800 training examples, 1280 test examples
- Sequence lengths and vocabulary sizes vary per task (see task configs in `composer/mad_lab/configs/tasks/`)
- All tasks trained for 200 epoch each

**Evaluation Metric:**
- For each candidate hybrid architecture, the model is trained across 800 samples per task
- Then evaluated on 1280 samples to obtain accuracy for each task
- The **final score is the average accuracy across all six tasks**

## Evaluation Function
The `evaluate.py` script will automatically launch training using `mad_lab/train.py` with your specified architecture. **You must not modify the training script `train.py`**. The `evaluate.py` script reads your `model_string` from `submission.csv`, enforces that the submission contains **exactly 16 primitives**, launches the training process, and returns the test accuracy metric. If `submission.csv` is missing, empty, or does not contain exactly 16 space-separated primitives, `evaluate.py` will exit with a clear error message.

**Note:** Training logs are written to `/tmp/logs` because the composer directory is read-only in the evaluation container. Results are automatically retrieved from this location.
```

E.2 `metadata.yaml`

```yaml
metric_lower_is_better: No
file_export_globs:
  - submission.csv
container_python_requirements:
  - torch==2.6.0
  - torchvision==0.21.0
  - torchaudio==2.6.0
  - --extra-index-url
  - https://download.pytorch.org/whl/cu126
  - ninja
  - datasets==4.0.0
  - scikit-learn
  - multiprocess
  - ax-platform
  - submitit
  - blobfile
  - numpy
  - pandas
  - einops
  - seaborn
  - torchmetrics
  - pytorch-lightning
  - opt-einsum
  - ray
  - scipy
  - absl-py
  - tiktoken
  - pyyaml
  - tqdm
  - packaging
  - transformers
  - tokenizers
  - huggingface-hub
  - safetensors
  - filelock
  - fsspec
```

```
    - triton==3.1.0
    - --no-build-isolation
    - flash-attn==2.8.3
    - causal-conv1d==1.5.3.post1
    - mamba-ssm==2.2.6.post3
  evaluate_container_python_requirements:
    - torch==2.6.0
    - torchvision==0.21.0
    - torchaudio==2.6.0
    - --extra-index-url
    - https://download.pytorch.org/whl/cu126
    - ninja
    - datasets==4.0.0
    - scikit-learn
    - multiprocess
    - ax-platform
    - submitit
    - blobfile
    - numpy
    - pandas
    - einops
    - seaborn
    - torchmetrics
    - pytorch-lightning
    - opt-einsum
    - ray
    - scipy
    - absl-py
    - tiktoken
    - pyyaml
    - tqdm
    - packaging
    - transformers
    - tokenizers
    - huggingface-hub
    - safetensors
    - filelock
    - fsspec
    - triton==3.1.0
    - --no-build-isolation
    - flash-attn==2.8.3
    - causal-conv1d==1.5.3.post1
    - mamba-ssm==2.2.6.post3
  logging_info:
    name: NeuralArchitectureSearchMAD2PrimitivesAccuracy
    config: MAD
    dataset: NAS/benchmark
    category: NLP
    research_problem: Neural Architecture Search
    output_type: Text Generation
    train_split: train
    test_split: test
    input_columns: n/a
    scoring_column: "model_string"
    shape: [1,]
    custom_gold_labels: Yes
    custom_rad_class: No
    metric: Accuracy
```

E.3 `prepare.py`

```python
import argparse
import os
import sys
import logging
```

```python
import shutil

# Configure logger with custom prefix
logger = logging.getLogger('dataset_code')
handler = logging.StreamHandler()
formatter = logging.Formatter('[Running provided `dataset_code`] %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.INFO)

def main(global_shared_data_dir: str, agent_data_mount_dir: str, agent_log_dir: str) -> None:
    """
    Main function for processing data directories.

    :param global_shared_data_dir: Path to the AIRS-Bench raw data directory.
    :param agent_data_mount_dir: Path to the agent data mount directory.
    :param agent_log_dir: Path to an agents log directory.
    """

    # Load MAD datasets from the raw data directory
    # Source: global_shared_data_dir/NAS/benchmark/MAD/
    mad_source_path = os.path.join(global_shared_data_dir, 'NAS/benchmark/MAD')

    # Copy to 'data' subdirectory in agent_data_mount_dir
    # This matches the default data_path='./data' in configs.py
    data_dest_dir = os.path.join(agent_data_mount_dir, 'data')
    os.makedirs(data_dest_dir, exist_ok=True)

    # Copy MAD train and test directories
    # After reorganization, MAD has structure: MAD/train/task_dirs and MAD/test/task_dirs
    if os.path.exists(mad_source_path):
        # Copy train and test directories
        for split in ['train', 'test']:
            source_split = os.path.join(mad_source_path, split)
            dest_split = os.path.join(data_dest_dir, split)

            if os.path.exists(source_split):
                if os.path.exists(dest_split):
                    shutil.rmtree(dest_split)
                shutil.copytree(source_split, dest_split)
                logger.info(f"Copied {split} directory from {source_split} to {dest_split}")
            else:
                logger.error(f"{split} directory not found at {source_split}")
                raise FileNotFoundError(f"{split} directory not found at {source_split}")

        logger.info(f"MAD data copied from {mad_source_path} to {data_dest_dir}")
    else:
        logger.error(f"MAD data not found at {mad_source_path}")
        raise FileNotFoundError(f"MAD data not found at {mad_source_path}")

    logger.info("MAD dataset preparation completed.")

    # Copy the composer directory to the agent_data_mount_dir
    # This allows the agent to run training and validation with evaluate_val.py
    script_dir = os.path.dirname(os.path.abspath(__file__))
    composer_src = os.path.join(script_dir, 'composer')
    composer_dst = os.path.join(agent_data_mount_dir, 'composer')

    if os.path.exists(composer_src):
        if os.path.exists(composer_dst):
            shutil.rmtree(composer_dst)
        shutil.copytree(composer_src, composer_dst)
        logger.info(f"Copied composer directory to {composer_dst}")
    else:
        logger.error(f"composer directory not found at {composer_src}")
        raise FileNotFoundError(f"composer directory not found at {composer_src}")

    # Copy precompile_cuda_extensions.py (required by evaluate_val.py)
```

```
    precompile_src = os.path.join(script_dir, 'precompile_cuda_extensions.py')
    precompile_dst = os.path.join(agent_data_mount_dir, 'precompile_cuda_extensions.py')
    if os.path.exists(precompile_src):
        shutil.copy2(precompile_src, precompile_dst)
        logger.info(f"Copied precompile_cuda_extensions.py to {precompile_dst}")
    else:
        logger.warning(f"precompile_cuda_extensions.py not found at {precompile_src} (optional
     )")

    # Copy evaluate_val.py for validation during architecture search
    evaluate_val_src = os.path.join(script_dir, 'evaluate_val.py')
    evaluate_val_dst = os.path.join(agent_data_mount_dir, 'evaluate_val.py')
    if os.path.exists(evaluate_val_src):
        shutil.copy2(evaluate_val_src, evaluate_val_dst)
        logger.info(f"Copied evaluate_val.py to {evaluate_val_dst}")
    else:
        logger.error(f"evaluate_val.py not found at {evaluate_val_src}")
        raise FileNotFoundError(f"evaluate_val.py not found at {evaluate_val_src}")

    logger.info("prepare.py completed successfully.")


if __name__ == "__main__":
    # Boilerplate code from prepare_boilerplate.py

    parser = argparse.ArgumentParser(
        description="Ensure the script is run with the required data directory arguments."
    )
    parser.add_argument(
        "--global-shared-data-dir",
        required=True,
        help="Path to the sharded raw data directory, e.g at /checkpoint/maui/shared/airsbench
     -raw-data"
    )
    parser.add_argument(
        "--agent-data-mount-dir",
        required=True,
        help="Path to the agent data mount directory, e.g ~/aira-dojo/data"
    )
    parser.add_argument(
        "--agent-log-dir",
        required=False,
        help="Path to a specific agents log directory, e.g /checkpoint/maui/shared/agent-log-
     dirs/agent-428391. This can be used e.g to pull in data created by a previous agent."
    )

    args = parser.parse_args()

    # Validate that the provided directories exist
    for path in [args.global_shared_data_dir, args.agent_data_mount_dir, args.agent_log_dir]:
        if path is not None and not os.path.isdir(path):
            print(f"Error: Directory not found: {path}", file=sys.stderr)
            sys.exit(1)

    logger.info(f"AirsBench raw data directory: {args.global_shared_data_dir}")
    logger.info(f"Agent data mount directory: {args.agent_data_mount_dir}")
    if args.agent_log_dir:
        logger.info(f"Agent log directory: {args.agent_log_dir}")
    else:
        logger.info("No agent log directory provided.")

    main(
        global_shared_data_dir=args.global_shared_data_dir,
        agent_data_mount_dir=args.agent_data_mount_dir,
        agent_log_dir=args.agent_log_dir,
    )
```

E.4 evaluate_val.py

```python
#!/usr/bin/env python3
"""
Validation script for architecture search.
Uses smaller dataset sizes for faster iteration during architecture search.
Performs a single train + validation run (similar to evaluate.py but with reduced data).
"""
import argparse
import json
import os
import subprocess
import numpy as np
import pandas as pd
import sys
import shutil

# ============================================================================
# APPLY CUDA ENVIRONMENT FIXES **BEFORE** IMPORTING TORCH OR ANY CUDA LIBS
# ============================================================================

def apply_cuda_fixes_early():
    """Apply CUDA environment fixes before any CUDA libraries are loaded."""

    if not os.environ.get('CUDA_HOME'):
        os.environ['CUDA_HOME'] = '/usr/local/cuda'

    triton_cache_dir = os.path.expanduser("~/.triton/cache")
    if os.path.exists(triton_cache_dir):
        shutil.rmtree(triton_cache_dir)
    os.makedirs(triton_cache_dir, mode=0o755, exist_ok=True)

    current_ld_path = os.environ.get('LD_LIBRARY_PATH', '')
    path_parts = [p for p in current_ld_path.split(':') if p]

    priority_paths = [
        '/.singularity.d/libs',
        '/usr/local/cuda/compat',
        '/usr/lib/x86_64-linux-gnu',
    ]

    for priority_path in priority_paths:
        if priority_path in path_parts:
            path_parts.remove(priority_path)

    final_paths = priority_paths + path_parts
    final_paths = [p for p in final_paths if p and 'stubs' not in p]
    os.environ['LD_LIBRARY_PATH'] = ':'.join(final_paths)
    os.environ['TRITON_DISABLE_LINE_INFO'] = '1'

    import ctypes
    try:
        libcuda = ctypes.CDLL('/.singularity.d/libs/libcuda.so.1', mode=ctypes.RTLD_GLOBAL)
    except:
        try:
            libcuda = ctypes.CDLL('libcuda.so.1', mode=ctypes.RTLD_GLOBAL)
        except:
            pass

apply_cuda_fixes_early()

# ============================================================
# BEST PRACTICE: PRE-COMPILE CUDA EXTENSIONS BEFORE TRAINING
# ============================================================
print("\n[EVALUATE_VAL] ========================================")
print("[EVALUATE_VAL] PRE-COMPILING CUDA EXTENSIONS")
print("[EVALUATE_VAL] ========================================")
```

```
66
67  script_dir = os.path.dirname(os.path.abspath(__file__))
68  precompile_script = os.path.join(script_dir, "precompile_cuda_extensions.py")
69
70  if not os.path.exists(precompile_script):
71      print(f"[EVALUATE_VAL] [WARN] WARNING: Precompile script not found at {precompile_script}"
        )
72      print(f"[EVALUATE_VAL]   Extensions will be JIT-compiled during training (may fail)")
73  else:
74      print(f"[EVALUATE_VAL] Running precompilation script: {precompile_script}")
75
76      env = os.environ.copy()
77      env['CUDA_LAUNCH_BLOCKING'] = '0'
78      env['TORCH_USE_CUDA_DSA'] = '0'
79
80      precompile_result = subprocess.run(
81          [sys.executable, precompile_script],
82          cwd=script_dir,
83          capture_output=True,
84          text=True,
85          env=env
86      )
87
88      if precompile_result.returncode == 0:
89          print("[EVALUATE_VAL] [OK] CUDA extensions pre-compiled successfully!")
90      else:
91          print(f"[EVALUATE_VAL] [FAIL] WARNING: Pre-compilation failed (exit code {
       precompile_result.returncode})")
92          print("[EVALUATE_VAL]   STDOUT:")
93          print(precompile_result.stdout)
94          print("[EVALUATE_VAL]   STDERR:")
95          print(precompile_result.stderr)
96
97  print("[EVALUATE_VAL] ========================================")
98
99
100 def evaluate_validation(model_string, seed=42):
101     """
102     Evaluate architecture using a single train + validation run.
103     Uses smaller dataset sizes for faster validation while maintaining
104     similar proportions to the full evaluate.py.
105
106     evaluate.py uses: 800 train, 1280 test per task
107     evaluate_val.py uses: 600 train, 200 val per task (75% train, 25% val ratio)
108     """
109     script_dir = os.path.dirname(os.path.abspath(__file__))
110     composer_dir = os.path.join(script_dir, "composer")
111
112     mad_lab_path = os.path.join(composer_dir, "mad_lab")
113     if mad_lab_path not in sys.path:
114         sys.path.insert(0, mad_lab_path)
115
116     from mad.registry import layer_registry, task_registry
117
118     layers = model_string.strip().split()
119     if len(layers) != 16:
120         raise ValueError(f"Must have exactly 16 primitives, got {len(layers)}")
121
122     logs_base_path = "/tmp/logs_val"
123     if os.path.exists(logs_base_path):
124         shutil.rmtree(logs_base_path)
125     os.makedirs(logs_base_path, exist_ok=True)
126
127     # Dataset sizes for validation (smaller than evaluate.py for faster iteration)
128     # evaluate.py: 800 train, 1280 test
129     # evaluate_val.py: 600 train, 200 val
130     num_train = 600
131     num_val = 200
```

```
    mad_tasks = [
        {'name': 'in-context-recall', 'vocab_size': 16, 'seq_len': 128, 'extra_args': []},
        {'name': 'fuzzy-in-context-recall', 'vocab_size': 16, 'seq_len': 128, 'extra_args': ['
     --k-motif-size', '3', '--v-motif-size', '3']},
        {'name': 'noisy-in-context-recall', 'vocab_size': 32, 'seq_len': 128, 'extra_args': ['
     --noise-vocab-size', '16', '--frac-noise', '0.2']},
        {'name': 'selective-copying', 'vocab_size': 16, 'seq_len': 256, 'extra_args': ['--num-
     tokens-to-copy', '16']},
        {'name': 'compression', 'vocab_size': 16, 'seq_len': 32, 'extra_args': []},
        {'name': 'memorization', 'vocab_size': 256, 'seq_len': 32, 'extra_args': []},
    ]

    env = os.environ.copy()
    env['CUDA_HOME'] = '/usr/local/cuda'
    env['TRITON_DISABLE_LINE_INFO'] = '1'
    env['TORCH_CUDA_ARCH_LIST'] = '9.0'
    env['PYTHONHASHSEED'] = str(seed)

    current_ld_path = env.get('LD_LIBRARY_PATH', '')
    path_parts = [p for p in current_ld_path.split(':') if p and 'stubs' not in p]
    priority_paths = ['/.singularity.d/libs', '/usr/local/cuda/compat', '/usr/local/nvidia/
     lib64', '/usr/local/cuda/lib64']
    final_paths = [p for p in priority_paths if os.path.exists(p)] + [p for p in path_parts if
      p not in priority_paths]
    env['LD_LIBRARY_PATH'] = ':'.join(final_paths)

    for candidate in ['/.singularity.d/libs/libcuda.so.1', '/usr/local/nvidia/lib64/libcuda.so
     .1']:
        if os.path.exists(candidate):
            env['LD_PRELOAD'] = candidate
            break

    model_id = '-'.join(layer_registry[l]['shorthand'] for l in layers)
    task_results = []

    print(f"\n{'='*60}")
    print(f"VALIDATION RUN")
    print(f"Architecture: {model_string}")
    print(f"Seed: {seed}")
    print(f"Train samples: {num_train}, Val samples: {num_val}")
    print(f"{'='*60}\n")

    for task_idx, task_config in enumerate(mad_tasks, 1):
        task_name = task_config['name']

        print(f"  Task {task_idx}/6: {task_name}")
        sys.stdout.flush()

        cmd_parts = [
            "python", "mad_lab/train.py",
            "--task", task_name,
            "--layers", *layers,
            "--seq-len", str(task_config['seq_len']),
            "--vocab-size", str(task_config['vocab_size']),
            "--num-train-examples", str(num_train),
            "--num-test-examples", str(num_val),
            "--log-base-path", logs_base_path,
            "--seed", str(seed),
        ]
        cmd_parts.extend(task_config['extra_args'])

        triton_cache = os.path.expanduser("~/.triton/cache")
        cmd = f"rm -rf {triton_cache} && mkdir -p {triton_cache} && " + " ".join(cmd_parts)

        task_accuracy = 0.0
        try:
            result = subprocess.run(cmd, cwd=composer_dir, env=env, shell=True,
```

```
                                    capture_output=True, text=True)

            if result.returncode == 0:
                task_shorthand = task_registry[task_name]['shorthand']
                for entry in os.listdir(logs_base_path):
                    entry_path = os.path.join(logs_base_path, entry)
                    if os.path.isdir(entry_path) and f"t-{task_shorthand}_" in entry and f"
    model-{model_id}" in entry:
                        result_file = os.path.join(entry_path, "results.csv")
                        if os.path.exists(result_file):
                            df = pd.read_csv(result_file)
                            task_accuracy = df['test_acc'].iloc[0]
                            print(f"    Accuracy: {task_accuracy:.4f}")
                        break
        except Exception as e:
            print(f"    Error: {e}")

        task_results.append(task_accuracy)

    mean_acc = sum(task_results) / len(task_results)

    print(f"\n{'='*60}")
    print(f"RESULTS")
    print(f"Validation Accuracy: {mean_acc:.6f}")
    print(f"{'='*60}\n")

    return mean_acc


def _cli():
    p = argparse.ArgumentParser()
    p.add_argument("--submission-file", default="submission.csv")
    p.add_argument("--seed", type=int, default=42)
    a = p.parse_args()

    with open(a.submission_file, 'r') as f:
        model_string = f.read().strip()

    if len(model_string.split()) != 16:
        print(f"ERROR: Must have exactly 16 primitives, got {len(model_string.split())}")
        sys.exit(1)

    avg_acc = evaluate_validation(model_string, seed=a.seed)
    print(json.dumps({"Accuracy": avg_acc}, indent=2))


if __name__ == '__main__':
    _cli()
```

E.5 `evaluate.py`

```
#!/usr/bin/env python3
import argparse
import json
import os
import subprocess
import glob
import pandas as pd
import sys

# ============================================================================
# APPLY CUDA ENVIRONMENT FIXES **BEFORE** IMPORTING TORCH OR ANY CUDA LIBS
# This must happen before torch imports to ensure CUDA is detected properly
# ============================================================================

```

```python
def apply_cuda_fixes_early():
    """Apply CUDA environment fixes before any CUDA libraries are loaded."""

    # Fix 1: Set CUDA_HOME
    if not os.environ.get('CUDA_HOME'):
        os.environ['CUDA_HOME'] = '/usr/local/cuda'

    # Fix 2: Clear and recreate Triton cache directory
    # CRITICAL: Delete any old cached cuda_utils.so that was compiled with wrong
     LD_LIBRARY_PATH
    import shutil
    triton_cache_dir = os.path.expanduser("~/.triton/cache")
    if os.path.exists(triton_cache_dir):
        # Remove old broken cache
        shutil.rmtree(triton_cache_dir)
    # Create fresh cache directory
    os.makedirs(triton_cache_dir, mode=0o755, exist_ok=True)

    # Fix 3: Prioritize real CUDA driver in LD_LIBRARY_PATH
    # CRITICAL: Put Singularity driver FIRST, before any other paths
    current_ld_path = os.environ.get('LD_LIBRARY_PATH', '')
    path_parts = [p for p in current_ld_path.split(':') if p]

    # Paths to prioritize (in order)
    priority_paths = [
        '/.singularity.d/libs',  # Real driver (35MB) - MUST BE FIRST!
        '/usr/local/cuda/compat',  # CUDA compat layer (28MB)
        '/usr/lib/x86_64-linux-gnu',  # Standard location (even if stubs)
    ]

    # Remove priority paths from their current positions
    for priority_path in priority_paths:
        if priority_path in path_parts:
            path_parts.remove(priority_path)

    # Add priority paths at the beginning
    final_paths = priority_paths + path_parts

    # CRITICAL FIX: Remove stub library paths that could interfere with linking
    # Filter out any paths containing 'stubs' as they only have stub libraries for compilation
    final_paths = [p for p in final_paths if p and 'stubs' not in p]

    # Set the new LD_LIBRARY_PATH
    os.environ['LD_LIBRARY_PATH'] = ':'.join(final_paths)

    # Fix 4: Force PyTorch/Triton to use compatible implementations
    # Disable Triton's rotary implementation which is causing issues
    os.environ['TRITON_DISABLE_LINE_INFO'] = '1'  # Reduce Triton debug overhead

    # Fix 5: Pre-load the real CUDA driver library to ensure it's available
    # This helps ctypes and dynamic linker find the right library
    import ctypes
    try:
        # Try loading from Singularity first (the real driver)
        libcuda = ctypes.CDLL('/.singularity.d/libs/libcuda.so.1', mode=ctypes.RTLD_GLOBAL)
        print("Successfully pre-loaded CUDA driver from /.singularity.d/libs/libcuda.so.1")
    except:
        try:
            # Fallback to standard library search
            libcuda = ctypes.CDLL('libcuda.so.1', mode=ctypes.RTLD_GLOBAL)
            print("Successfully pre-loaded CUDA driver via libcuda.so.1")
        except Exception as e:
            print(f"Warning: Could not pre-load CUDA driver: {e}")

# Apply fixes immediately, before any imports
apply_cuda_fixes_early()


```

```python
def evaluate_model_string(model_string, data_dir=None):
    """
    Launches training with the given model_string and returns the test accuracy.

    For MAD benchmark, trains on all 6 MAD tasks and returns the average test accuracy.

    Args:
        model_string: Space-separated string of layer primitives
        data_dir: Optional override for data directory path

    Returns:
        dict: {"Accuracy": average_test_accuracy_across_6_mad_tasks}
    """
    # Get absolute path to composer directory
    script_dir = os.path.dirname(os.path.abspath(__file__))
    composer_dir = os.path.join(script_dir, "composer")

    # Add mad_lab to Python path so we can import the registry
    mad_lab_path = os.path.join(composer_dir, "mad_lab")
    if mad_lab_path not in sys.path:
        sys.path.insert(0, mad_lab_path)

      # Now we can import the layer and task registries
    from mad.registry import layer_registry, task_registry

    # Determine data path - for MAD tasks, point to the MAD data directory
    if data_dir is None:
        # Default: assume container environment with mounted data at ./data/MAD
        # Convert to absolute path since train.py runs from composer/ directory
        data_path = os.path.abspath("./data/MAD")
    else:
        # Use provided absolute path
        data_path = os.path.abspath(data_dir)

    # Enforce exactly 16 primitives in the model string
    layers = model_string.strip().split()
    if len(layers) != 16:
        raise ValueError(f"Submission must contain exactly 16 primitives, but got {len(layers)
     }: '{model_string}'")

    # CRITICAL: Clear /tmp/logs to prevent checkpoint reuse between evaluations
    # This ensures each evaluation trains from scratch without loading old checkpoints
    import shutil
    logs_base_path = "/tmp/logs"
    if os.path.exists(logs_base_path):
        print(f"[EVALUATE] Clearing {logs_base_path} to prevent checkpoint reuse...")
        shutil.rmtree(logs_base_path)
    os.makedirs(logs_base_path, exist_ok=True)
    print(f"[EVALUATE] Clean logs directory created at {logs_base_path}")

    # Define the 6 MAD tasks with their specific parameters
    mad_tasks = [
        {
            'name': 'in-context-recall',
            'vocab_size': 16,
            'seq_len': 128,
            'epochs': 200,
            'num_train_examples': 800,
            'num_test_examples': 1280,
            'extra_args': []
        },
        {
            'name': 'fuzzy-in-context-recall',
            'vocab_size': 16,
            'seq_len': 128,
            'epochs': 200,
            'num_train_examples': 800,
            'num_test_examples': 1280,
```

```
            'extra_args': ['--k-motif-size', '3', '--v-motif-size', '3']
        },
        {
            'name': 'noisy-in-context-recall',
            'vocab_size': 32,
            'seq_len': 128,
            'num_train_examples': 800,
            'num_test_examples': 1280,
            'extra_args': ['--noise-vocab-size', '16', '--frac-noise', '0.2']
        },
        {
            'name': 'selective-copying',
            'vocab_size': 16,
            'seq_len': 256,
            'num_train_examples': 800,
            'num_test_examples': 1280,
            'extra_args': ['--num-tokens-to-copy', '16']
        },
        {
            'name': 'compression',
            'vocab_size': 16,
            'seq_len': 32,
            'num_train_examples': 800,
            'num_test_examples': 1280,
            'extra_args': []
        },
        {
            'name': 'memorization',
            'vocab_size': 256,
            'seq_len': 32,
            'num_train_examples': 800,
            'num_test_examples': 1280,
            'extra_args': []
        },
    ]

    print(f"\n{'='*80}")
    print(f"RUNNING MAD BENCHMARK: 6 TASKS")
    print(f"Model Architecture: {' '.join(layers)}")
    print(f"Data Path: {data_path}")
    print(f"{'='*80}\n")

    # Prepare environment for subprocess with CUDA fixes
    env = os.environ.copy()

    # Ensure our CUDA environment fixes are explicitly passed to subprocess
    env['CUDA_HOME'] = '/usr/local/cuda'
    env['TRITON_DISABLE_LINE_INFO'] = '1'

    # CRITICAL FIX: Set deterministic CUDA compilation flags to prevent NaN
    env['TORCH_CUDA_ARCH_LIST'] = '9.0'  # H200 architecture
    env['CUDA_LAUNCH_BLOCKING'] = '0'  # Async launch for performance
    env['TORCH_USE_CUDA_DSA'] = '0'  # Disable device-side assertions for consistency

    # ============================================================
    # BEST PRACTICE: PRE-COMPILE CUDA EXTENSIONS BEFORE TRAINING
    # ============================================================
    print("\n[EVALUATE] ========================================")
    print("[EVALUATE] PRE-COMPILING CUDA EXTENSIONS")
    print("[EVALUATE] ========================================")

    precompile_script = os.path.join(script_dir, "precompile_cuda_extensions.py")

    if not os.path.exists(precompile_script):
        print(f"[EVALUATE] [WARN] WARNING: Precompile script not found at {precompile_script}"
    )
        print(f"[EVALUATE]   Extensions will be JIT-compiled during training (may fail)")
    else:
```

```
        print(f"[EVALUATE] Running precompilation script: {precompile_script}")

        precompile_result = subprocess.run(
            [sys.executable, precompile_script],
            cwd=script_dir,
            capture_output=True,
            text=True,
            env=env
        )

        if precompile_result.returncode == 0:
            print("[EVALUATE] [OK] CUDA extensions pre-compiled successfully!")
        else:
            print(f"[EVALUATE] [FAIL] WARNING: Pre-compilation failed (exit code {
 precompile_result.returncode})")
            print("[EVALUATE]   STDOUT:")
            print(precompile_result.stdout)
            print("[EVALUATE]   STDERR:")
            print(precompile_result.stderr)

    print("[EVALUATE] ========================================")

    # Reconstruct LD_LIBRARY_PATH with real driver first
    current_ld_path = env.get('LD_LIBRARY_PATH', '')
    path_parts = [p for p in current_ld_path.split(':') if p and 'stubs' not in p]

    priority_paths = [
        '/.singularity.d/libs',
        '/usr/local/cuda/compat',
        '/usr/local/nvidia/lib64',
        '/usr/local/nvidia/lib',
        '/usr/local/cuda/lib64',
    ]

    final_paths = []
    for p in priority_paths:
        if os.path.exists(p) and p not in final_paths:
            final_paths.append(p)

    for p in path_parts:
        if p not in final_paths:
            final_paths.append(p)

    env['LD_LIBRARY_PATH'] = ':'.join(final_paths)

    # Pre-load the real CUDA driver library using LD_PRELOAD
    libcuda_candidates = [
        '/.singularity.d/libs/libcuda.so.1',
        '/usr/local/nvidia/lib64/libcuda.so.1',
        '/usr/local/cuda/compat/libcuda.so.1',
    ]

    libcuda_to_preload = None
    for candidate in libcuda_candidates:
        if os.path.exists(candidate):
            libcuda_to_preload = candidate
            print(f"Found CUDA driver for LD_PRELOAD: {candidate}")
            break

    if libcuda_to_preload:
        env['LD_PRELOAD'] = libcuda_to_preload
        print(f"Setting LD_PRELOAD={libcuda_to_preload}")
    else:
        print("WARNING: Could not find libcuda.so.1 to preload")

    print(f"\nSubprocess LD_LIBRARY_PATH: {env['LD_LIBRARY_PATH']}")
    print(f"Subprocess CUDA_HOME: {env['CUDA_HOME']}")
    print(f"Subprocess LD_PRELOAD: {env.get('LD_PRELOAD', 'NOT SET')}")
```

```
    # Build the model ID from the layer names using the registry
    model_id = '-'.join(layer_registry[l]['shorthand'] for l in layers)

    # Run training for each MAD task and collect test accuracies
    # If a task fails, assign it 0.0 accuracy
    task_results = []  # List of (task_name, accuracy, failed) tuples

    for task_idx, task_config in enumerate(mad_tasks, 1):
        task_name = task_config['name']
        vocab_size = task_config['vocab_size']
        seq_len = task_config['seq_len']
        num_train = task_config['num_train_examples']
        num_test = task_config['num_test_examples']
        extra_args = task_config['extra_args']

        print("\n" + "="*80)
        print(f"TASK {task_idx}/6: {task_name}")
        print(f"  vocab_size={vocab_size}, seq_len={seq_len}")
        print(f"  num_train={num_train}, num_test={num_test}")
        print("="*80 + "\n")
        sys.stdout.flush()

        # Build training command for this specific task
        training_cmd_parts = [
            "python", "mad_lab/train.py",
            "--task", task_name,
            "--layers", *layers,
            "--data-path", data_path,
            "--seq-len", str(seq_len),
            "--vocab-size", str(vocab_size),
            "--num-train-examples", str(num_train),
            "--num-test-examples", str(num_test),
            "--log-base-path", logs_base_path,
        ]

        # Add task-specific extra arguments
        training_cmd_parts.extend(extra_args)

        # Create shell command that clears Triton cache first, then runs training
        triton_cache_path = os.path.expanduser("~/.triton/cache")
        cmd = f"rm -rf {triton_cache_path} && mkdir -p {triton_cache_path} && " + " ".join([f"
'{part}'" if ' ' in part else part for part in training_cmd_parts])

        print(f"Running training command (with cache clear): {cmd}")
        print(f"Working directory: {composer_dir}")

        # Run training with output streaming directly to parent stdout/stderr
        task_accuracy = 0.0  # Default to 0 if task fails
        task_failed = False

        try:
            result = subprocess.run(
                cmd,
                cwd=composer_dir,
                env=env,
                shell=True,
                stdout=None,  # Inherit parent's stdout (makes output visible)
                stderr=None   # Inherit parent's stderr (makes errors visible)
            )

            if result.returncode != 0:
                print(f"\n[WARNING] Training failed for task '{task_name}' with return code: {
result.returncode}")
                print(f"[WARNING] Assigning 0.0 accuracy for this task")
                task_failed = True
            else:
                print(f"\n[SUCCESS] Task '{task_name}' completed successfully")
```

```
                # Find and read results for this task
                # Get the task shorthand (e.g., "R" for "in-context-recall")
                task_shorthand = task_registry[task_name]['shorthand']

                matching_dirs = []
                for entry in os.listdir(logs_base_path):
                    entry_path = os.path.join(logs_base_path, entry)
                    # Look for directories containing task shorthand and model ID
                    # Format: t-{task_shorthand}_..._model-{model_id}
                    if os.path.isdir(entry_path) and f"t-{task_shorthand}_" in entry and f"model-{model_id}" in entry:
                        matching_dirs.append(entry_path)

                if not matching_dirs:
                    print(f"[WARNING] No log directory found for task '{task_name}' and model '{model_id}' in {logs_base_path}")
                    print(f"[WARNING] Assigning 0.0 accuracy for this task")
                    task_failed = True
                else:
                    # Get the most recent matching directory
                    latest_dir = max(matching_dirs, key=os.path.getmtime)
                    result_file = os.path.join(latest_dir, "results.csv")

                    if not os.path.exists(result_file):
                        print(f"[WARNING] results.csv not found at {result_file}")
                        print(f"[WARNING] Assigning 0.0 accuracy for this task")
                        task_failed = True
                    else:
                        print(f"Reading results from: {result_file}")

                        # Read results
                        results_df = pd.read_csv(result_file)
                        task_accuracy = results_df['test_acc'].iloc[0]
                        print(f"Task '{task_name}' Test Accuracy: {task_accuracy}")

            except Exception as e:
                print(f"\n[WARNING] Exception occurred while processing task '{task_name}': {e}")
                print(f"[WARNING] Assigning 0.0 accuracy for this task")
                task_failed = True

            # Record the result (either actual accuracy or 0.0 if failed)
            task_results.append((task_name, task_accuracy, task_failed))

        # Calculate average test accuracy across all 6 tasks (including 0s for failures)
        task_accuracies = [accuracy for _, accuracy, _ in task_results]
        average_accuracy = sum(task_accuracies) / len(task_accuracies)

        print("\n" + "="*80)
        print("MAD BENCHMARK COMPLETED")
        print("="*80)
        print(f"\nAll {len(mad_tasks)} tasks processed")
        print("\nIndividual Task Results:")

        # Show results for all tasks
        for idx, (task_name, accuracy, failed) in enumerate(task_results, 1):
            status = " (FAILED - assigned 0.0)" if failed else ""
            print(f"  {idx}. {task_name:30s}: {accuracy:.6f}{status}")

        print(f"\nAverage Test Accuracy across all {len(mad_tasks)} tasks: {average_accuracy:.6f}")
        print("="*80 + "\n")

        return {"Accuracy": average_accuracy}


def _cli():
    p = argparse.ArgumentParser(
```

```
        description="Evaluate model architecture by training and measuring accuracy"
    )
    p.add_argument("--submission-file", default="submission.csv",
                   help="Path to text file containing model_string.")
    p.add_argument("--data-dir", default=None,
                   help="Optional: Override data directory path. If not provided, uses ./data/MAD")
    a = p.parse_args()

    print(f"Loading model_string from: {a.submission_file}")
    try:
        with open(a.submission_file, 'r') as f:
            model_string = f.read().strip()
        if not model_string:
            raise ValueError("submission.csv is empty")
        # Enforce exactly 16 primitives in submission.csv
        if len(model_string.split()) != 16:
            p.error(f"submission.csv must contain exactly 16 primitives, but got {len(model_string.split())}: '{model_string}'")
        print(f"Model string: {model_string}")
    except FileNotFoundError:
        p.error(f"Submission file not found: {a.submission_file}")
    except Exception as e:
        p.error(f"Error loading submission_file: {e}")

    print("Evaluating model architecture...")
    result = evaluate_model_string(model_string, data_dir=a.data_dir)

    print("\n--- EVALUATION RESULT ---")
    print(json.dumps(result, indent=2))

if __name__ == '__main__':
    _cli()
```

E.6 `train.py`

```
import os
import argparse
import shutil
import torch
import random

import numpy as np
import pandas as pd
import pytorch_lightning as pl

from torch.utils.data import DataLoader

from mad.configs import MADConfig, MADModelConfig
from mad.paths import make_log_path
from mad.data import generate_data
from mad.model import PLModelWrap
from mad.registry import task_registry, layer_registry


def get_args():
    parser = argparse.ArgumentParser()

    # task settings:
    parser.add_argument('--task', type=str, default='in-context-recall', choices=list(task_registry.keys()), help='task to train model on')
    parser.add_argument('--vocab-size', type=int, default=16, help='size of token vocabulary')
    parser.add_argument('--seq-len', type=int, default=128, help='length of input sequences')
    parser.add_argument('--num-train-examples', type=int, default=12_800, help='number of training examples')
```

```
    parser.add_argument('--num-test-examples', type=int, default=1_280, help='number of test examples')
    parser.add_argument('--frac-noise', type=float, default=0., help='fraction of input sequence that is noise')
    parser.add_argument('--noise-vocab-size', type=int, default=0, help='size of noise token vocabulary')
    parser.add_argument('--num-tokens-to-copy', type=int, default=0, help='number of tokens to copy in selective-copying')
    parser.add_argument('--k-motif-size', type=int, default=1, help='number of adjacent tokens that together form a key in fuzzy in-context recall')
    parser.add_argument('--v-motif-size', type=int, default=1, help='number of adjacent tokens that together form a value in fuzzy in-context recall')
    parser.add_argument('--multi-query', action=argparse.BooleanOptionalAction, default=True, help='if True, multi-query variant of in-context recall tasks is used')

    # model settings:
    parser.add_argument('--layers', nargs='+', default=['mh-attention', 'swiglu', 'mh-attention', 'swiglu'], help='layers of model')
    parser.add_argument('--backbone', type=str, default='language-model', help='model backbone used for layers')
    parser.add_argument('--dim', type=int, default=128, help='width of the model (will be enforced for all layers)')

    # training settings:
    parser.add_argument('--batch-size', type=int, default=128, help='batch size for training and evaluation')
    parser.add_argument('--epochs', type=int, default=200, help='maximum number of epochs to train')
    parser.add_argument('--optimizer', type=str, default='adamw', choices=['adamw', 'sgd'], help='optimizer used for training')
    parser.add_argument('--scheduler', type=str, default='cosine', choices=['plateau', 'cosine', 'none'], help='learning rate scheduler used for training')
    parser.add_argument('--early-stop', action=argparse.BooleanOptionalAction, default=False, help='if True, training is stopped when validation accuracy does not improve anymore for training epochs specified by stop-patience')
    parser.add_argument('--stop-patience', type=int, default=20, help='patience for early stopping (in training epochs)')
    parser.add_argument('--plateau-patience', type=int, default=5, help='patience for plateau scheduler (in training epochs)')
    parser.add_argument('--plateau-factor', type=float, default=0.9, help='learning rate reduce factor for plateau scheduler')
    parser.add_argument('--accelerator', type=str, default='cuda', help='accelerator used for training')
    parser.add_argument('--devices', type=int, default=1, help='number of devices to use for training')
    parser.add_argument('--precision', type=str, default='bf16', help='precision of the model (see PyTorch Lightning Trainer docs for details)')

    # optimizer settings:
    parser.add_argument('--lr', type=float, default=5e-4, help='learning rate for optimizer')
    parser.add_argument('--min-lr', type=float, default=1e-4, help='minimum learning rate for cosine learning rate scheduler')
    parser.add_argument('--weight-decay', type=float, default=0., help='weight decay for optimizer')

    # logging:
    parser.add_argument('--log-base-path', type=str, default='./logs', help='path where training logs are stored')
    parser.add_argument('--log-to-csv', action=argparse.BooleanOptionalAction, default=True, help='if True, metrics are stored locallly in a csv file in the log directory')
    parser.add_argument('--log-to-wandb', action=argparse.BooleanOptionalAction, default=False, help='if True, metrics are logged to Weights & Biases')
    parser.add_argument('--wandb-project', type=str, default='MAD', help='name of the Weights & Biases project to log to')
    parser.add_argument('--save-checkpoints', action=argparse.BooleanOptionalAction, default=True, help='if True, final and best model checkpoints are saved in the log directory')

    # data:
```

```
    parser.add_argument('--data-path', type=str, default='./data', help='path where generated data are stored')
    parser.add_argument('--num-data-workers', type=int, default=0, help='number of workers for data generation and data loading')
    parser.add_argument('--persistent-data-workers', action=argparse.BooleanOptionalAction, default=True, help='if True, data workers are kept alive between epochs')
    parser.add_argument('--tokenizer-path', type=str, default='./tokenizer', help='path of the tokenizer model, only used for dclm task')

    # misc:
    parser.add_argument('--seed', type=int, default=12345, help='random seed for reproducibility')
    parser.add_argument('--target-ignore-index', type=int, default=-100, help='ignore index for target in loss function')

    args = vars(parser.parse_args())

    # make sure we select the correct model backbone!
    if args['task'] in {'compression'} and args['backbone'] != 'autoencoder':
        print(f'Setting model backbone to "autoencoder", which is required for the compression task!')
        args['backbone'] = 'autoencoder'

    return args


# train model according to mad_config:

def train(
    model: torch.nn.Module,
    mad_config: MADConfig,
    log_path: str,
    log_to_csv: bool = True,
    log_to_wandb: bool = False,
    wandb_project: str = 'MAD',
    save_checkpoints: bool = True,
    data = None
) -> pd.DataFrame:
    """
    Train a model with given configuration and log results.

    Args:
        model (nn.Module): model to train
        mad_config (MADConfig): MAD configuration
        log_path (str): path to logs
        log_to_csv (bool): if True, log results to csv in log_path
        log_to_wandb (bool): if True, log results to Weights & Biases
        wandb_project (str): name of Weights & Biases project to log to
        save_checkpoints (bool): if True, save model checkpoints

    Returns:
        results_df (pd.DataFrame): results of training
    """

    # Check if results exist already.

    if os.path.exists(log_path):
        path_results_df = os.path.join(log_path, 'results.csv')
        if os.path.exists(path_results_df):
            results_df = pd.read_csv(path_results_df)
            print(f'Log path "{log_path}" exists, retrieved results from there...')
            return results_df
        else:
            shutil.rmtree(log_path)


    # PyTorch Lightning Model Wrap.

```

```
    model_wrapped = PLModelWrap(model=model, mad_config=mad_config)

    # Make Dataloaders.

    if mad_config.task == 'dclm':
        shuffle = False # IterableDatasets cannot be shuffled
    else:
        shuffle = True

    train_dl = DataLoader(
        dataset=data['train'],
        batch_size=mad_config.batch_size,
        shuffle=shuffle,
        num_workers=mad_config.num_data_workers,
        persistent_workers=mad_config.persistent_data_workers and mad_config.num_data_workers
     >0
    )

    test_dl = DataLoader(
        dataset=data['test'],
        batch_size=mad_config.batch_size,
        shuffle=shuffle,
        num_workers=mad_config.num_data_workers,
        persistent_workers=mad_config.persistent_data_workers and mad_config.num_data_workers
     >0
    )

    # Make Loggers & Callbacks.

    early_stop = pl.callbacks.EarlyStopping(
        monitor='test/Accuracy_epoch',
        min_delta=0.00,
        stopping_threshold=0.999,
        patience=mad_config.stop_patience if mad_config.early_stop else mad_config.epochs,
        verbose=True,
        mode='max'
    )
    callbacks = [early_stop]

    if save_checkpoints and log_path is not None:
        checkpoint_best = pl.callbacks.ModelCheckpoint(
            save_top_k=1,
            monitor="test/Perplexity_epoch",
            mode="min",
            dirpath=os.path.join(log_path, 'checkpoints'),
            filename="best",
        )
        checkpoint_last = pl.callbacks.ModelCheckpoint(
            save_top_k=1,
            monitor="epoch",
            mode="max",
            dirpath=os.path.join(log_path, 'checkpoints'),
            filename="last",
        )
        callbacks += [checkpoint_best, checkpoint_last]

    loggers = []
    if log_to_csv and log_path is not None:
        loggers.append(
            pl.loggers.CSVLogger(
                save_dir=log_path,
                name='logs',
                version=''
            )
        )

    if log_to_wandb:
        # We import wandb here so it doesn't create any random directories in /tmp
```

```
        # when not used
        import wandb
        wandb.init(
            project=wandb_project,
            name=os.path.basename(log_path) if log_path is not None else None
        )
        loggers.append(pl.loggers.WandbLogger())

    # set default precision of float32 matrix multiplications:
    torch.set_float32_matmul_precision('high')

    # Make trainer.
    # Use 'auto' strategy instead of forcing DDP to avoid OOM errors with single GPU
    parallelism_strategy = 'auto'
    trainer = pl.Trainer(
            max_epochs=mad_config.epochs,
            accelerator=mad_config.accelerator if torch.cuda.is_available() else 'cpu',
            devices=mad_config.devices,
            logger=loggers,
            enable_checkpointing=mad_config.save_checkpoints,
            callbacks=callbacks,
            precision=mad_config.precision,
            detect_anomaly=False,
            limit_train_batches=(mad_config.num_train_examples // mad_config.batch_size) if
     mad_config.task == "dclm" else None,
            limit_val_batches=(mad_config.num_test_examples // mad_config.batch_size) if
     mad_config.task == "dclm" else None,
            strategy=parallelism_strategy
        )

    # Train.
    trainer.fit(model_wrapped, train_dl, test_dl)

    # Evaluate Final Performance.

    results_train = trainer.validate(dataloaders=train_dl)[0]
    results_test = trainer.validate(dataloaders=test_dl)[0]
    results_df = pd.DataFrame({
        # training data:
        'train_acc': results_train['test/Accuracy_epoch'], # its called "test/..." because we
     compute results with trainer.validate
        'train_ppl': results_train['test/Perplexity_epoch'],
        'train_loss': results_train['test/Loss_epoch'],
        # test data:
        'test_acc': results_test['test/Accuracy_epoch'],
        'test_ppl': results_test['test/Perplexity_epoch'],
        'test_loss': results_test['test/Loss_epoch'],
    }, index=[0])
    results_df.to_csv(os.path.join(log_path, 'results.csv'), index=False)

    # Done!

    return results_df


if __name__ == '__main__':

    # get cli args:

    args = get_args()

    # Set random seed before model init to enable reproducible results
    random.seed(args['seed'])
    np.random.seed(args['seed'])
    torch.manual_seed(args['seed'])

    # create MAD config:

```

```
    mad_config = MADConfig()
    mad_config.update_from_kwargs(args)

    # create model config:

    model_config = MADModelConfig()
    model_config.update_from_kwargs(args)

    # IMPORTANT: Set max_length to seq_len for tasks with specific sequence lengths
    # This ensures the AutoEncoder model uses the correct sequence length
    model_config.max_length = args['seq_len']

    # Make Data.

    data = generate_data(
        instance_fn=mad_config.instance_fn,
        instance_fn_kwargs=mad_config.instance_fn_kwargs,
        train_data_path=mad_config.train_dataset_path,
        test_data_path=mad_config.test_dataset_path,
        tokenizer_path=args['tokenizer_path'],
        num_train_examples=mad_config.num_train_examples,
        num_test_examples=mad_config.num_test_examples,
        num_workers=mad_config.num_data_workers,
    )

    # Update the vocab size from the tokenizer for the dclm task
    if mad_config.task == "dclm":
        model_config.vocab_size = data['vocab_size']

    model = model_config.build_model_from_registry()
    model_id = '-'.join(layer_registry[l]['shorthand'] for l in model_config.layers)

    # train model:

    log_path = make_log_path(
        base_path=args['log_base_path'],
        mad_config=mad_config,
        model_id=model_id
    )
    train(
        model=model,
        mad_config=mad_config,
        log_path=log_path,
        log_to_csv=args['log_to_csv'],
        log_to_wandb=args['log_to_wandb'],
        wandb_project=args['wandb_project'],
        save_checkpoints=args['save_checkpoints'],
        data=data
    )
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction claim that agent-discovered architectures outperform baselines and Composer-found alternatives under isotoken and isoFLOP experiments. These claims are directly validated in Tables 2–3 and Figures 1, 4–18 with quantitative results. The methodological advantage of agentic search over rule-based NAS is exemplified in Figure 3.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 7 explicitly includes a dedicated Limitations and Future Work paragraph discussing reliance on small-scale proxies, non-agentic aggregation/extrapolation and the restricted search space (arrangement only). We pair limitations listed with actionable next steps.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: This paper is purely empirical and does not include theoretical results or proofs.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We specify all task definitions (Table 1), datasets employed (Section 3), agent configurations (Section 4), LLMs used, scaffold types, seed counts, compute budgets (Sections 5) aggregation methods, and scaling procedures (Section 6, Appendix C). The underlying harness (AIRA-DOJO), task template (AIRS-BENCH) and NAS framework (Composer) have also been publicly documented in prior works.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: The code associated with our experiments is currently undergoing internal review in accordance with our organization's open-source policies. We intend to release the full codebase, including task definitions and pretraining scripts, upon receiving approval.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4 and Section 5 detail seed counts, GPU allocation, runtime budgets, and evaluation metrics. Section 3 specifies datasets, splits, and proxy training sizes. Appendix C provide exact model configurations at small and large scale, as well as exact FLOP budgets and step counts at each scale.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

Justification: The search phase uses 10–20 seeds per agent, and results relative to the search all have clear error bars and confidence interval (see figures in Appendix A. For the 2-primitive isotoken experiments, we report mean $\pm$ standard deviation across 3 random seeds, and report error intervals for all 3 main metrics (Table 2, Figure 15). Due to computational limitation and good agreement among random seeds in the 2-primitive case, the 3-primitive results are single-seed and explicitly noted as such (Table 3).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section 5 states each agent uses one H200 GPU with 24h (or 60h) runtime budgets. The IsoFLOP experiments require 8 H200 GPUs for 350M experiments and 16 H200 GPUs for 1B and 3B experiments. The environment employed for agentic-search experiments and pretraining experiments will be released with the code.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: This work involves automated architecture search using popular, publicly documented datasets and does not raise ethical concerns regarding human subjects, privacy, or dual-use risks. We follow every aspect of the NeurIPS Code of Ethics.

   Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    This is foundational ML research on architecture search methodology. It does not introduce capabilities for misuse beyond what existing LLM training pipelines already enable. There is no societal impact of the work performed.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: We do not release model weights or new datasets. The contribution is a search methodology and architectural patterns, which pose no direct misuse risk.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All datasets (MAD, BabiStories, DCLM), frameworks (Composer, AIRA-dojo, AIRS-Bench), and models are properly cited. DCLM is used under its open license; MAD and BabiStories are open-source research datasets.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: This paper does not release new datasets, models, or code packages.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.

- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are the core component of our method, since they drive the NAS pipeline: 11 agents powered by 6 distinct LLMs perform the architecture search. Their usage, scaffolding, and configurations are extensively documented in Sections 4–5 and throughout the manuscript.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.