From Memorization to Creativity: LLM as a Designer of Novel Neural Architectures

Waleed Khalid Dmitry Ignatov Radu Timofte

Computer Vision Lab, CAIDAS & IFI, University of Würzburg, Germany

Supplementary Material

This document provides supplementary details for the main paper. Section references of the form "Section X" refer to the main paper; supplementary sections are labelled A–L.

A. Additional Method Details

A.1. MinHash/LSH Configuration for Near-Duplicate Detection

Tokenization and shingling. We perform lexer-based tokenization of Python/PyTorch source code into a sequence of syntactic tokens (e.g., keywords, identifiers, literals, operators, and delimiters). We then construct token-level shingles using a shingle size of $k = 10$ (i.e., contiguous 10-grams over the token stream), yielding a set representation for each architecture.

MinHash signatures. Each shingle set is mapped to a MinHash signature using $N_{\text{perm}} = 256$ permutations (hash functions). MinHash signatures are used to efficiently approximate Jaccard similarity between shingle sets.

LSH candidate retrieval. To accelerate retrieval, we index MinHash signatures using locality-sensitive hashing (LSH) with a retrieval threshold of 0.85, producing a candidate set of potentially similar architectures via band collisions. Candidates are subsequently verified using the MinHash-estimated Jaccard similarity.

Acceptance threshold for near-duplicates. A pair of architectures is marked as a near-duplicate if the estimated Jaccard similarity exceeds $\tau = 0.90$. We use the same τ consistently for lexical/structural duplicate checks, including dataset curation and novelty filtering during sampling.

A.2. Operational Novelty Filtering During Sampling

Order of evaluation. Novelty filtering is applied only after a candidate satisfies execution validity (successful parse, instantiation, and forward pass) and completes one epoch of training. This ordering avoids expending LSH queries on invalid candidates.

Cycle-local archive. Within each sampling cycle, we maintain an in-memory archive of MinHash signatures for all previously accepted candidates in that cycle to enable efficient computation of $J^{(c)}_{\text{gen}}$.

Rejection accounting. We record the number of rejected candidates encountered before accepting a non-duplicate architecture (`rejection_count`), quantifying the propensity of the generator to propose near-duplicates under fixed decoding settings.

B. LEMUR Corpus Deduplication and Conversion

The reduction from 109,913 raw LEMUR records to 1,065 unique records reflects the high degree of text-level redundancy in the raw export: many entries are minor variants of the same architecture differing only in hyperparameter values, comments, or variable names. The MinHash–LSH deduplication retains one representative per near-duplicate cluster, preserving architectural diversity while removing redundant instances that would otherwise bias fine-tuning toward overrepresented families. The 216 records dropped during chat-format conversion were excluded due to missing required interface components (e.g., absent `forward` method signatures, incompatible constructor arguments, or malformed metadata fields) that would have produced syntactically invalid training targets; including them would introduce noise into the supervised signal.

While this reduction inevitably narrows coverage relative to the full LEMUR corpus, the retained 849 records span a diverse range of architecture families—including standard convolutional networks, residual architectures, depthwise-separable and lightweight designs, and multi-branch topologies—across multiple image-classification task types as catalogued in the LEMUR metadata [**?**], providing sufficient diversity to initialize the generator without strong family-level bias. We acknowledge that architectures well-represented in dropped or deduplicated clusters may be underrepresented in the initial prior, and that this constitutes a potential source of coverage bias that future work could address by applying more permissive conversion or by supplementing with additional corpora.

C. Dual-Task Training Pair Construction

For each converted record, two training examples are constructed from the same assistant code but with user messages describing the task as either MNIST (28×28 grayscale) or CIFAR-10 (32×32 RGB) classification. This duplication serves two purposes: first, it encourages the LLM to associate the same architectural pattern with multiple input specifications, promoting generalization of learned design priors across related image-classification tasks rather than overfitting to a single dataset description; second, it doubles the number of gradient updates over each architectural pattern during fine-tuning, which stabilizes learning on a corpus of fewer than 900 unique examples without introducing new architectural information. The code is identical across both variants by design: the intent is not to generate dataset-specific architectures but to train the generator to respond to task descriptions as a flexible interface while producing architectures that respect the shared structural constraints. This duplication does not inflate the number of *unique* architectural patterns in the training set, which remains 849.

D. Stopping Criterion

The synthesis loop was run for 22 cycles, a number determined by the convergence behavior of the valid generation rate and mean first-epoch accuracy rather than by an arbitrary practical cutoff. Specifically, both metrics exhibit clear plateau behavior after approximately cycle 18 (Section 4 of the main paper), consistent with the diminishing-returns regime documented in iterative self-training literature [**?** **?**]. Four additional cycles (19–22) were run beyond this apparent plateau to confirm stabilization rather than temporary fluctuation, yielding a total of 22 cycles.

In practice, we recommend the following data-driven stopping criterion for future applications of this loop: terminate when the improvement in mean first-epoch accuracy across three consecutive cycles falls below a threshold ϵ (e.g., $\epsilon = 0.5$ percentage points), and the proportion of models exceeding the accuracy threshold has stabilized within its Wilson confidence interval across those same cycles. Under this criterion applied retrospectively to our data, termination would have occurred at cycle 18–19, yielding results within one percentage point of the 22-cycle endpoint at approximately 18% lower computational cost.

E. Justification of First-Epoch Accuracy as Performance Proxy

We adopt true validation accuracy after the first training epoch as our primary performance signal, rather than relying on zero-shot NAS proxies [**?**]. Although training-free indicators have been shown to correlate with fully-trained accuracy [**?** **?** **?**], they remain indirect. Even the strongest reported Spearman rank-correlation coefficients ($\rho \approx 0.50$–0.82) on standard benchmarks such as NAS-Bench-101 and NAS-Bench-201 correspond to a coefficient of determination of at most $R^2 \approx 0.67$, leaving substantial variance unexplained [**?** **?**]. Moreover, recent work has established that early discarding after a single epoch constitutes an effective and principled strategy in automated model selection [**?**], providing theoretical grounding for our proxy choice. Unlike zero-cost indicators, first-epoch accuracy is a *direct measurement* of how well an architecture supports learning on the target task and dataset, making it both interpretable and actionable as a selection criterion within the iterative synthesis loop.

F. Detailed Ablation Analysis

This section expands on the ablation results summarized in Table 4 of the main paper.

Removing the novelty filter. The novelty filter is disabled while the 40% accuracy threshold and LoRA fine-tuning remain unchanged. Under this condition, any architecture that compiles, trains for one epoch, and exceeds the 40% first-epoch accuracy threshold becomes eligible for inclusion in the training corpus, even if its code closely matches models already present. Across cycles, the valid generation rate and first-epoch accuracy trends remain qualitatively similar to the full method, and the generator still transitions into a regime where a large fraction of valid models exceed the 40% threshold. However, the composition of the training corpus changes substantially: without the novelty constraint, the corpus accumulates repeated motifs and near-duplicate architectures, and the number of genuinely distinct code patterns added per cycle is noticeably lower than in the full method. Generated models therefore retain acceptable accuracy but exhibit reduced code-level diversity, indicating that the novelty filter is central for sustaining exploration of new regions of the design space rather than repeatedly reinforcing a narrow family of designs.

Removing the accuracy threshold. The MinHash-based novelty filter is retained, but the 40% first-epoch accuracy threshold is removed. In this variant, all architectures that (i) compile and train for one epoch and (ii) are non-duplicates with respect to the accepted set and earlier generations are added to the training corpus regardless of early-epoch performance. Improvements in valid generation rate are still observed as the LLM is exposed to more executable code, and the generator continues to produce architectures with reasonable first-epoch accuracy. Nonetheless, the shift in the overall accuracy distribution is clearly weaker than in the full method. The training set contains a broader mix-

ture of low- and high-performing architectures, and low-accuracy yet valid models are regularly promoted into the corpus. Consequently, the fraction of models exceeding the 40% accuracy threshold increases more slowly and stabilizes at a lower level than when performance filtering is applied. This outcome indicates that novelty alone does not suffice to steer refinement toward rapidly learning architectures under the early-epoch proxy; performance-based selection is required to align corpus growth with empirical learning behavior.

Removing iterative fine-tuning. The iterative refinement mechanism is removed entirely. The base LLM is fine-tuned once on the initial LEMUR-derived training split, and architectures are generated from this fixed model without adding self-generated examples back into the training corpus. The evaluation protocol is unchanged, but the feedback loop is disabled. Under this non-iterative baseline, the valid generation rate and first-epoch accuracies match the behavior observed in early cycles of the full loop and do not exhibit the progressive improvements obtained under cycle-wise updates. The proportion of models surpassing the 40% accuracy threshold remains substantially below the levels achieved in later cycles, and the distribution of architectures does not shift into the regime where high-accuracy models dominate. Although code-level novel architectures can still be produced, these discoveries do not influence future generations, leaving the generator effectively static.

G. Additional Results

G.1. First-Epoch Accuracy Trends

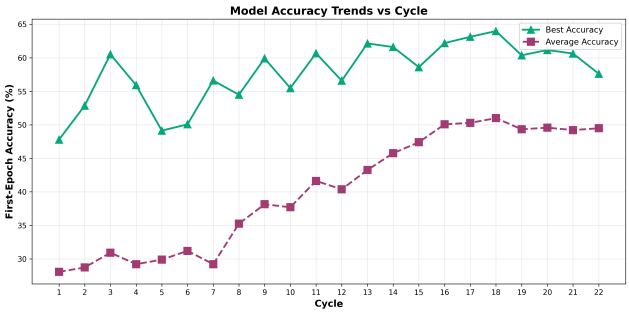

Figure 1. Best, mean, and median first-epoch accuracy per cycle on CIFAR–10 (see Section 4 of the main paper).

G.2. Proportion Exceeding the 40% First-Epoch Accuracy Threshold

Late-cycle saturation. After cycle 18, both best and mean first-epoch accuracies plateau and fluctuate within a narrow band, and the fraction of models above 40% stabilizes. This saturation suggests diminishing returns from

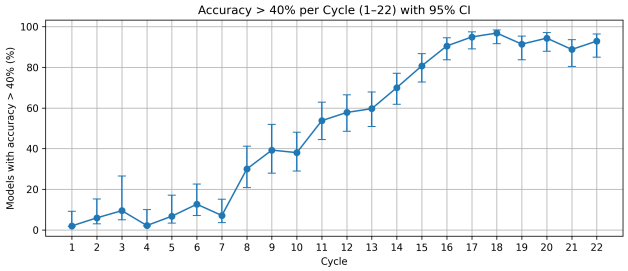

Figure 2. Proportion of models with first-epoch accuracy $\geq 40\%$ per cycle (see Section 4 of the main paper).

continued self-training and is consistent with known failure modes of iterative self-training pipelines, where selection bias and distributional narrowing can limit further gains when the training set becomes increasingly dominated by model-generated samples [**? ?**].

G.3. On Pre-training Bias and Architectural Exploration

A natural concern is whether the LLM's generative behavior remains constrained by architectural patterns encountered during pre-training, effectively limiting exploration to known motifs such as VGG-style stacks or ResNet-like blocks. We argue this concern, while legitimate, is precisely what the iterative fine-tuning loop is designed to address, and that our results provide empirical evidence against severe constraint.

First, the base model (DeepSeek-Coder-7B) was pre-trained on general code corpora rather than curated neural architecture datasets; architectural patterns constitute a small and heterogeneous fraction of its pre-training distribution. The dominant inductive bias is therefore toward syntactically valid Python and PyTorch idioms, not toward any particular architectural family.

Second, and more directly, the MinHash–Jaccard novelty filter explicitly rejects candidates whose token-level shingle representation matches any architecture in the LEMUR-derived training corpus above threshold $\tau = 0.90$. Since the LEMUR corpus itself spans a broad range of architectural families, any architecture that passes this filter is by construction dissimilar to known patterns at the code level. The 455 architectures admitted over 22 cycles on CIFAR-10 therefore represent structures that are not near-duplicates of the 109,913 LEMUR records—including the original pre-deduplicated corpus—providing a lower bound on the degree to which the generator escapes its initial prior.

Third, the monotonic increase in both the number of novel models admitted per cycle (1 in cycle 1 to 40 in cycle 17) and the diversity of structural fingerprints suggests that the feedback loop actively expands rather than contracts the explored region of design space. The cross-

dataset experiments reinforce this conclusion: the loop admits novel architectures at comparable rates on CIFAR-100 and SVHN, demonstrating that exploration is not confined to CIFAR-10-specific motifs.

H. Qualitative Analysis of Discovered Architectures

To complement the quantitative metrics reported in the main paper, we examine four representative architectures produced by the generator, selected to illustrate the structural diversity of the output distribution. Table 1 summarizes each architecture.

Structural diversity. The four examples illustrate that the generator produces architectures spanning a substantial range of structural complexity. At one extreme, Architecture C is an ultra-compact design consisting of a single convolutional feature block followed immediately by global average pooling and a linear classifier—a minimal architecture that nonetheless achieves 58.1% first-epoch accuracy. At the other extreme, Architecture A is a deep multi-stage network with four `make_layer` stages, each containing multiple Conv–BN–ReLU blocks with progressive channel widening ($64 \rightarrow 512$) and spatial downsampling, achieving 62.8%.

Emergence of residual connections. Architecture B is particularly notable: the generator independently produces a `Block` class implementing identity shortcuts with conditional downsampling, structurally analogous to a ResNet basic block. The forward pass explicitly computes `out += identity` with a separate downsampling branch when spatial dimensions or channel counts change. This residual pattern, combined with a deep 3-stage layout (3/4/6 blocks) and a two-layer classifier with dropout regularization, represents a sophisticated design that passes the novelty filter—indicating it is not a near-duplicate of any architecture in the LEMUR corpus despite its conceptual similarity to ResNet. This suggests the generator is capable of re-deriving well-known effective patterns through the iterative feedback loop rather than simply copying them from training data.

Variation in design philosophy. Architecture D adopts a VGG-style philosophy—no skip connections, uniform dual-Conv–BN–ReLU blocks per stage—but replaces the traditional flattened fully-connected classifier with global average pooling followed by a single linear layer, a more modern and parameter-efficient choice. The contrast between Architectures B and D (residual vs. plain, complex vs. uniform) illustrates that the generator maintains meaningful design diversity even among models with similar accuracy levels (61.4% vs. 58.0%).

Summary. The examples above demonstrate that the generator produces architectures exhibiting both combinatorial novelty (different arrangements and depths of known building blocks) and operational diversity (e.g., the emergence of residual identity shortcuts in Architecture B, dropout regularization strategies, and the choice between global average pooling and fully-connected classifiers). These variations go beyond superficial code-level edits: they reflect distinct computational graphs with different gradient-flow properties and capacity–efficiency trade-offs, all synthesized within the constrained design space and confirmed as structurally novel by the MinHash–Jaccard filter.

I. Prompt and API Contract Specification

Each generated model is required to implement a fixed API contract: a class `Net(nn.Module)` with methods `__init__`, `forward`, `train_setup`, and `learn`, together with a module-level function `supported_hyperparameters()` returning {`"lr"`, `"momentum"`}. Prompts require `import torch` and `import torch.nn as nn` and instruct the model to output a single `nn.Module` definition without data loading or training loops. No pretrained weights or external feature extractors are permitted; only standard convolutional, pooling, normalization, and activation operations are allowed. An implicit edge-friendly latency target accompanies the 500,000-parameter budget. The system message casts the model as an expert PyTorch architecture designer optimizing for first-epoch accuracy under these constraints; the user message specifies the dataset name, input/output shapes, parameter budget, and the API contract listed above. This prompt template is held fixed across all 22 cycles.

J. Full Hyperparameter Specifications

LoRA fine-tuning. Fine-tuning uses DeepSeek-Coder-7B-Instruct-v1.5 with LoRA applied to attention projections (q/k/v/o) and MLP projections (`up_proj`, `down_proj`, `gate_proj`) in all 24 Transformer layers (0–23). LoRA hyperparameters are fixed across cycles: rank r=32, LoRA α=32, dropout 0.05. The task is causal language modeling over chat-format prompt–response pairs. Each cycle runs for 5 epochs with learning rate 1×10^{-5}, per-device batch size 1, gradient accumulation 4 (effective batch size 4), paged AdamW in 8-bit, cosine schedule with 20 warmup steps, weight decay 0.01, max gradient norm 1.0, and `bfloat16` mixed precision.

Decoding configuration. Generation uses a chat interface with fixed decoding across all cycles: temperature 0.20, top-k 50, nucleus p=0.9, `do_sample=True`, and maximum new tokens 2,048 (padded with EOS). These settings are

Arch.	1-ep. Acc.	Architecture summary	Key features
A	62.8%	Conv_{64}→ReLU→MaxPool→ 4×`make_layer` (64→128→256→512), each with Conv→BN→ReLU blocks and MaxPool→AdaptiveAvgPool(6,6)→Linear	Multi-stage, Kaiming init, no skip connections
B	61.4%	Conv_{32}→BN→ReLU→MaxPool→ 3 stages via `make_layer` using `Block` class (32→64→128; 3/4/6 blocks)→ AvgPool→Dropout→Linear(4096)→ Dropout→Linear	Residual identity shortcuts, conditional downsampling, deep classifier head
C	58.1%	`build_features`: Conv_{32}→BN→ReLU →AdaptiveAvgPool(1,1)→Linear(32, C)	Ultra-compact single-block, parameterized num. classes
D	58.0%	Conv_{32}→ReLU→MaxPool→ 3×Sequential (32→64→128), each with dual Conv→BN→ReLU →AdaptiveAvgPool(1,1)→Linear(128)	VGG-style, Kaiming init, global avg pooling

Table 1. Representative generated architectures spanning different design families. All architectures were generated within the 22-cycle synthesis loop on CIFAR-10 and passed the MinHash–Jaccard novelty filter (i.e., each is structurally dissimilar to both the LEMUR seed corpus and other generated models at the token-shingle level).

deliberately held constant so that changes in generator behavior are attributable solely to the evolving training corpus.

K. Statistical Confidence Intervals

Let $N_{\text{gen}}^{(c)}$ denote the number of candidate architectures sampled in cycle c, and $N_{\text{valid}}^{(c)}$ the subset that compile and train successfully. The valid generation rate is $p_{\text{valid}}^{(c)} = N_{\text{valid}}^{(c)}/N_{\text{gen}}^{(c)}$.

For all valid models in cycle c, let $\mathcal{A}^{(c)} = \{A(m) : m \in \mathcal{M}_{\text{valid}}^{(c)}\}$. For the sample mean $\bar{A}^{(c)}$ and standard deviation $s^{(c)}$ computed over $n_c = |\mathcal{A}^{(c)}|$, we report t-based 95% confidence intervals:

$$\bar{A}^{(c)} \pm t_{0.975,\, n_c-1} \frac{s^{(c)}}{\sqrt{n_c}}. \tag{1}$$

For any proportion $\hat{p} = k/n$ (e.g., $p_{\text{valid}}^{(c)}$ or the proportion with $A(m) \geq \tau$), we report Wilson score confidence intervals [**?**].

L. Computational Complexity & Accessibility

To evaluate the practical accessibility of our framework, we benchmarked the computational requirements on consumer-grade hardware. Utilizing a single NVIDIA GeForce RTX 4090 (24GB), candidate generation averages 0.7–1.5 GPU-minutes per sample, while the low-fidelity proxy evaluation (1-epoch training) requires 1–5 GPU-minutes per valid model. Parameter-efficient fine-tuning via LoRA (rank 32, 5 epochs) over the terminal corpus of 2,153 pairs is completed in approximately 2–4 hours. We estimate the cumulative compute budget for the entire 22-cycle evolution at 90–266 GPU-hours, demonstrating that NNGPT enables sophisticated neural architecture design without requiring enterprise-scale clusters.

M. Ethical Considerations (Extended)

Data Usage and Privacy. The initial training corpus is derived from the publicly available LEMUR Neural Network Dataset. The dataset consists exclusively of source code and associated technical metadata and does not contain personally identifiable information (PII) or sensitive user data. Architectures generated during the iterative synthesis process are programmatically produced code artifacts within an isolated execution environment. As a result, no private, confidential, or user-generated data are introduced into the training or evaluation loop.

Security and Integrity of Generated Code. The LLM is used to generate executable source code as part of the architecture synthesis process. While syntactic and semantic validity checks are applied during experimentation, we note that any real-world deployment or reuse of LLM-generated code would necessitate comprehensive security reviews. Such audits would be required to mitigate risks related to unsafe coding practices or the inadvertent reproduction of vulnerable code patterns.

Transparency and Reproducibility. To support transparency and reproducibility, we provide a detailed account of the experimental setup, including the base LLM, fine-tuning strategy (LoRA), data filtering mechanisms (MinHash-Jaccard novelty filtering), and evaluation protocols. This level of documentation is intended to facilitate independent verification of the results and to enable replication or extension of the proposed methodology by the research community.