NNGPT: Rethinking AutoML with Large Language Models

Anonymous CVPR-NAS26 submission

Paper ID

Abstract

Building self-improving AI systems remains a fundamental challenge in machine learning. We present NNGPT, an open-source framework that turns a large language model (LLM) into a closed-loop AutoML system for neural network development, primarily for computer vision. Unlike prior frameworks, NNGPT expands a corpus of executable networks by generating new models and continuously fine-tuning the LLM within a loop of generation, assessment, and improvement. It integrates five LLM-based pipelines into a single workflow: zero-shot architecture synthesis, hyperparameter optimization (HPO), code-aware accuracy and early-stop prediction, retrieval-augmented synthesis of scope-closed PyTorch blocks (NN-RAG), and reinforcement learning. Built on the LEMUR dataset as an audited corpus with reproducible metrics, NNGPT emits from a single prompt a complete specification of network architecture, preprocessing code, and hyperparameters; it validates and executes the specification end-to-end and learns from the resulting traces. A PyTorch adapter keeps NNGPT framework-agnostic, and LLM-driven synthesis avoids fixed NAS search spaces. Empirically, we provide component-level evidence across the shared execution stack: NN-RAG attains 73% executability on 1,289 targets, and hash-based deduplication eliminates hundreds of redundant runs. In iterative refinement cycles, mean one-epoch CIFAR-10 accuracy increases from 28.06% to 50.99% (best 63.98%), and the training corpus expands from 1,698 to 2,154 prompt–code pairs. One-shot hyperparameter specifications achieve accuracy comparable to search-based AutoML with fewer trials; on LEMUR, HPO reaches RMSE 0.60 (vs. Optuna 0.64), and the code-aware predictor reaches RMSE 0.14 with Pearson r=0.78. The system has already produced over 5,000 validated models, supporting NNGPT as a practical systems-level step toward self-improving, LLM-driven AutoML. Upon acceptance, we will release code, prompts, and checkpoints to support reproducibility and community use.

1. Introduction

A longstanding goal in machine learning is to build systems that design and improve themselves. In practice, assembling a competitive neural pipeline still requires substantial expert effort: choosing architectures, writing transforms, specifying losses and metrics, tuning hyperparameters, and running many training jobs. Classical AutoML, for example Optuna [3] with Tree-structured Parzen Estimator (TPE) [6], automates parts of this process but relies on black-box search that requires hundreds of trials and ignores the semantics of the underlying code. A central limitation of many NAS-style approaches is that they optimize within a predefined and often narrow search space (e.g., cell templates and restricted operator sets), which can constrain architectural diversity and transfer across tasks.

Large language models (LLMs) have demonstrated significant advancements across various domains [18, 29, 56], with models such as GPT-4 [49], Code Llama [54], and DeepSeek [13] excelling in code generation, which has motivated attempts to let them propose architectures, hyperparameters, or training scripts. Most prior work, however, remains at design time: the LLM emits fragments or templates that are only loosely tied to execution, logging, or learning from outcomes, so the feedback loop from generation to training to improved generation is rarely closed. In contrast, a code-generating LLM can, in principle, propose architectures beyond a fixed NAS search space by drawing on a broad prior captured in its parameters and by conditioning on large corpora of executable models.

We ask whether a single LLM, embedded in an online loop with executable code and training feedback, can serve as the core of a self-improving AutoML system for neural networks. Answering this requires representing full training pipelines as executable specifications, validating and running them at scale, predicting performance early, and using logs to continuously improve the LLM. Crucially, this loop must be grounded in execution: invalid or hallucinated generations should be filtered automatically so that only runnable outcomes contribute to further learning.

We introduce **NNGPT**, an open-source framework that, to our knowledge, is the first to unify zero-shot model gen-

Method	LLM	Arch.	HPO	Full train	One-shot	Closed loop	Acc. pred.	Open source
Optuna [3]	✗	✗	✓	✗	✗	✗	✗	✓
Hyperopt [7]	✗	✗	✓	✗	✗	✗	✗	✓
Google Vizier [19]	✗	✗	✓	✗	✗	✗	✗	✓
ENAS [53]	✗	✓	✓	✗	✗	✗	✗	✓
DARTS [37]	✗	✓	✓	✗	✗	✗	✗	✓
AutoKeras [28]	✗	✓	✓	✗	✗	✗	✗	✓
EvoPrompting [8]	✓	✓	✓	✗	✗	✓	✗	✗
LLAMBO [40]	✓	✗	✓	✗	✗	✓	✗	✗
LLMatic [47]	✓	✓	✓	✗	✗	✓	✗	✗
GPT-NAS [65]	✓	✓	✓	✗	✗	✓	✗	✗
Self-Programming AI [58]	✓	✓	✓	✗	✗	✓	✗	✗
Grammar / schema prompting [60]	✓	✗	✗	✗	✓	✗	✗	✗
NNGPT (ours)	✓	✓	✓	✓	✓	✓	✓	✓

Table 1. Feature comparison between NNGPT and representative AutoML and LLM-based methods. Columns indicate whether a method uses an LLM, generates architectures (Arch.), performs hyperparameter optimization (HPO), synthesizes full training specifications, operates in a one-shot fashion, supports a closed feedback loop, includes accuracy prediction, and is open source.

eration and editing, hyperparameter recommendation, accuracy and early-stop prediction, retrieval-augmented code synthesis, and a reinforcement learning (RL)-based improvement loop into a single closed AutoML cycle driven by prompt-based LLM inference. From one high-level prompt, NNGPT produces an executable training specification (model, transforms, metrics, optimizer, schedule), executes it, logs code and metrics, and uses these logs to fine-tune and reinforce the underlying LLMs. In our current deployment, hyperparameter recommendation and execution/logging form the core production loop, while zero-shot synthesis primarily expands the executable corpus that improves subsequent fine-tuning and retrieval.

NNGPT is built on LEMUR [20], which we treat as a corpus of executable neural programs with audited implementations, unified preprocessing, and reproducible metrics. This grounds one-shot generation in runnable PyTorch [51] code rather than abstract templates. The framework remains PyTorch-agnostic via a thin adapter that exposes model factories, data transforms, and metric bindings. Although the retrieval set is optional in the interface, in practice LEMUR currently provides the most complete audited execution corpus available; other PyTorch codebases can be integrated through the same adapter. In practice, NNGPT has generated and trained over 10,000 distinct models (over 5,000 through its self-improving loop with LLMs), with training statistics incorporated into LEMUR as verified outcomes.

The closed-loop view is as follows. A fine-tuned LLM emits network architectures, preprocessing code, and hyperparameters; an execution engine compiles, trains, and logs; a code-aware predictor forecasts final accuracy and a stopping epoch from model code plus early metrics; a retrieval module patches or synthesizes PyTorch blocks; and an RL layer updates LoRA adapters [26] using rewards derived from executability and downstream performance.

We evaluate NNGPT as a unified system through component-level experiments on the key stages of this loop, all implemented in a shared execution stack. Across these experiments, we show that the core modules are effective and interoperable in practice, supporting NNGPT as a systems-level step toward fully automated, LLM-driven AutoML. Because NNGPT targets a broad portion of the neural pipeline design space, we view this work as a first integrated implementation that demonstrates feasibility at scale; deeper analysis of bias, long-horizon overfitting, and cross-domain transfer remains important future work.

1.1. Related Work

Prior work falls into two strands: (i) search-based AutoML/NAS and (ii) LLM-driven generation.

Search-based AutoML and NAS. Classical HPO frameworks such as Optuna [3], Hyperopt [7], and Google Vizier [19] rely on Bayesian optimization or TPE to iteratively probe the space of hyperparameters. These tools are strong baselines but typically require hundreds of trials, treat training code as an opaque black box, and do not synthesize *complete* train specs (data, metrics, optimizer). NAS systems, including ENAS [53], DARTS [37], and AutoKeras [28], extend search to architectures (with many stabilized variants), yet remain compute-intensive and slow to adapt to new tasks and codebases.

LLM-driven generation and closed loops. Recent methods integrate LLMs into design loops: EvoPrompting [8] uses an LLM as a mutation operator in evolutionary search; LLAMBO [39] couples LLM suggestions with Bayesian optimization; LLMatic [47], GPT-NAS [65], and Self-Programming AI [58] generate code or fill templates. Grammar/schema prompting [60] and retrieval help with validity. However, most systems either require repeated LLM queries, cover only fragments of the pipeline (e.g., mutations without training integration), or lack an online mechanism to learn from executed runs.

Performance prediction. Predicting final accuracy and

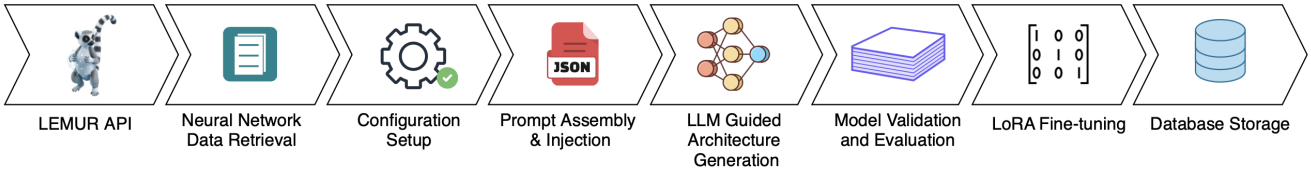

Figure 1. Overview of the NNGPT Pipeline: starting with a query to the LEMUR API to retrieve a neural network entry and its metadata, the system constructs a prompt, uses an LLM to generate the model code, and trains it. Artifacts are logged in a structured database, while LoRA fine-tuning continuously updates the LLM based on training results, creating a self-improving AutoML loop.

early stopping has been studied via curve fitting from early learning curves [2, 16], architecture- or text-based surrogates [41, 62], and graph-level representations [15]. These approaches often abstract away implementation details (losses, custom layers, metric code) that strongly affect convergence, and they rarely integrate with executable pipelines.

Position of NNGPT. Tab. 1 situates NNGPT among these lines. Unlike prior art, NNGPT (1) *one-shot* generates schema-validated *full* training specifications (architecture & HPO) that run immediately; (2) replaces costly search for many tasks with LLM-based hyperparameter recommendation; (3) includes a *code-aware* predictor that consumes executable code and early metrics to forecast final accuracy and a stopping epoch; and (4) closes the loop via LoRA/RL updates using execution logs - within a single open-source framework. A comprehensive literature review with extended comparisons is provided in the Supplementary Material, Sec. 2.

2. NNGPT Framework

2.1. System Overview

NNGPT exposes LLMs as end-to-end configuration agents for neural networks. Given a high-level task description, the system constructs a structured prompt, calls an LLM once, validates the structured output, executes the resulting training pipeline, logs all artifacts, and feeds the trace back into its training data.

At the core of NNGPT is a shared execution substrate built on the LEMUR dataset framework, which provides executable PyTorch models, standardized training loops, and reproducible evaluation metrics. To avoid hard coupling to LEMUR, the framework defines a thin adapter interface for arbitrary PyTorch codebases that maps a configuration object to a model constructor, data transforms, and metric definitions. Any model that implements this interface can be driven by the same LLM generated specification.

A typical pass through the system proceeds as follows. First, the configuration module aggregates task level information such as dataset, objective, resource limits, and optional seed architectures. Next, a prompt builder converts this state into a JSON-like instruction block that encodes the desired output schema. The LLM backend then emits a structured, machine-readable specification with explicit fields for architecture, hyperparameters, and training procedure. A schema layer based on Pydantic validates the output and either autocorrects minor issues or requests a single regeneration in case of structural violations. Valid configurations are handed off to a distributed training engine, which runs the experiment, logs metrics and metadata in an SQLite store, and optionally exports models back into the LEMUR corpus. Finally, dedicated modules use these traces to fine tune the LLM with LoRA, to train an accuracy and early stopping predictor, or to update a RL signal. This shared infrastructure underpins all pipelines described below.

2.2. Problem Formulation

We cast NNGPT as closed-loop neural pipeline synthesis problem. Given a task prompt P (task, dataset, metric, optional budget) and an optional retrieval set R (code and metadata from LEMUR or external projects), the goal is to produce an executable training specification C and a horizon T that maximize downstream performance under a compute budget.

An LLM generator G_θ maps

$$(P, R) \;\mapsto\; \tilde{C} \in \mathcal{C},$$

where $\tilde{C}$ is a YAML/JSON spec that defines the model, data transforms, optimization schedule, loss, and nominal epochs. A validator V (schema + type/shape checks) enforces structural constraints, auto-fixes minor issues or triggers a single re-prompt; only $V(\tilde{C}) = \text{pass}$ is promoted to C.

Given C and T, the trainer E executes the pipeline,

$$(\mathbf{m}, \mathbf{u}) = E(C, T),$$

producing validation metrics over epochs $\mathbf{m}$ and auxiliary logs $\mathbf{u}$ (loss curves, learning rate, runtime, hardware). Code and traces are persisted in a structured log.

To cut wasted compute, a code-aware predictor H_ϕ estimates

$$(\widehat{\text{acc}}_*, \widehat{t}_*) = H_\phi(C, R, \mathbf{m}_{1:t_0}),$$

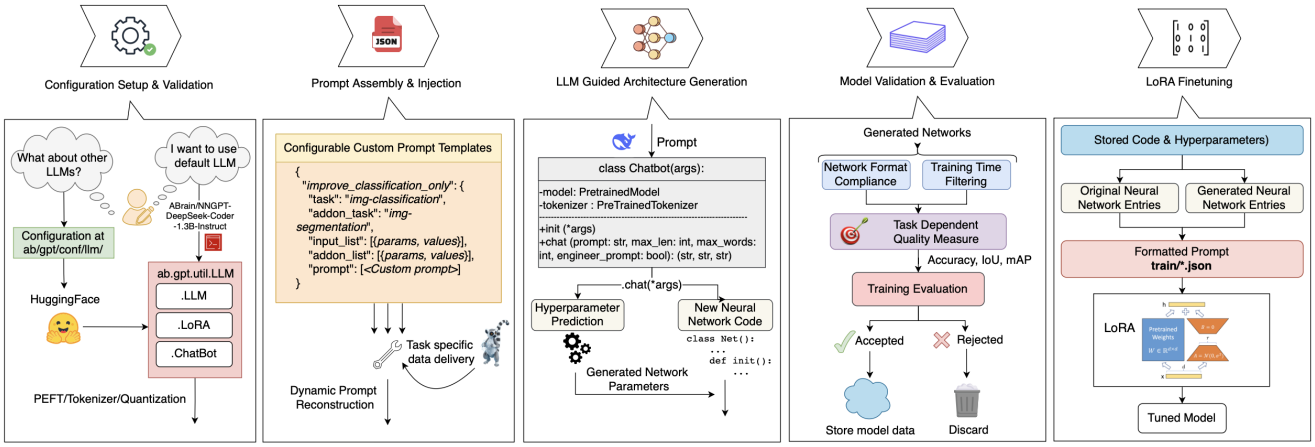

Figure 2. Detailed view of pipeline stages 3–7 in NNGPT. Left to right: Configuration Setup & Validation (default or HF-loaded LLM, PEFT/quantization); Prompt Assembly & Injection (templated prompts with task data); LLM-Guided Architecture Generation (one-shot code and hyperparameters); Model Validation & Evaluation (format checks, training, logging); LoRA Fine-tuning (update adapters from `train/*.json`). All pipelines reuse this stack; variation mainly occurs in the Prompt stage.

where $\widehat{\mathrm{acc}}_*$ approximates the final accuracy and $\widehat{t}_*$ proposes an early-stopping epoch from the first t_0 epochs. This induces an adaptive horizon $T' = \widehat{t}_*$ and enables early termination and priority scheduling.

Over time the log

$$\mathcal{D}_{\log} = \{(P^{(k)}, R^{(k)}, C^{(k)}, \mathbf{m}^{(k)}, \mathbf{u}^{(k)})\}_{k=1}^{K}$$

updates θ and ϕ: G_θ is refined with LoRA fine-tuning and lightweight policy-gradient updates using rewards from executability and downstream performance, while H_ϕ is trained to minimize prediction error on final accuracy and stopping epoch. This closes the loop: generate, validate, execute, log, update, under an explicit compute budget.

2.3. Pipelines in NNGPT

On top of the common backbone, NNGPT implements five concrete pipelines that target different stages of the model development process while reusing the same prompting, validation, and execution stack. As summarized in Fig. 1, all pipelines follow one seven-step template: LEMUR retrieval → configuration setup/validation → prompt assembly & injection → LLM-guided generation → model validation/training → LoRA fine-tuning → database logging. In our current deployment, hyperparameter recommendation and execution/logging form the core production loop, while zero-shot synthesis and NN-RAG primarily serve to expand and repair the executable corpus that the system learns from over time; the RL loop is optional and currently used as a diagnostic extension. The pipeline-specific behavior is concentrated in the *Prompt* stage (steps 3–4): the prompt schema, retrieved context, and expected outputs differ across zero-shot generation/editing, hyperparameter recommendation, accuracy/early-stop prediction, NN-RAG patching, and the RL loop. Implementation details of the shared stages are expanded in Fig. 2.

2.3.1. Zero-shot model generation and editing

The first pipeline turns LLMs into generators of complete architectures and training specifications. Given a task prompt and a target dataset, NNGPT queries LEMUR for high-performing models, selects one *reference* implementation and n additional *supporting* examples, and assembles a Few-Shot Architecture Prompting (FSAP) prompt that concatenates the task description, dataset specs, and full PyTorch classes for these models. The LLM (typically a LoRA-fine-tuned DeepSeek-Coder-7B) is instructed to emit a new scope-closed architecture class plus a YAML configuration rather than a local patch. A whitespace-normalized MD5 hash validator removes duplicate programs in < 1 ms, preventing redundant training of architectures that differ only in formatting. Generated specifications are checked by the schema layer and executed by the training engine; successful runs are logged and added back to LEMUR as new entries. See Sec. 3.4 for the experimental setup and evaluation.

2.3.2. Hyperparameter recommendation

The second pipeline treats hyperparameter tuning as conditional text generation over a fixed schema. Training data are drawn from LEMUR runs that link each model to its dataset, transforms, target accuracy, epoch budget, and the hyperparameters that achieved this performance. Prompts expose the model name, `nn_code`, `transform_code`, task, dataset, and either a target accuracy or a "maximize accuracy" instruction; the LLM is asked to output only the values for a predefined set of hyperparameters (learning rate, momentum, weight decay, batch size, etc.) in a strict order.

We fine-tune several DeepSeek and CodeLlama variants with LoRA on this corpus and integrate the best checkpoints (DeepSeek-Coder-1.3B-Base and DeepSeek-R1-Distill-Qwen-7B) as plug-and-play recommenders inside NNGPT. At inference time, the hyperparameter pipeline runs as a one-shot replacement for search-based HPO: a single LLM call produces a configuration that is executed by the same backend as Optuna, and outcomes are logged back to LEMUR. Section 3.3 shows that these one-shot recommendations reach accuracy comparable to TPE-based Optuna on LEMUR, while eliminating thousands of optimization trials.

2.3.3. Accuracy and early-stop prediction

The third pipeline uses an LLM as a multi-task predictor of training dynamics. Each training run in LEMUR is converted into a structured prompt that fuses three modalities: (i) structured metadata (task, dataset, maximum epochs), (ii) executable code snippets for the model, metric, and data transforms, and (iii) early-epoch validation accuracies (typically at epochs 1 and 2). A DeepSeek-Coder-1.3B backbone is adapted with 4-bit QLoRA, reducing memory by $\sim 4\times$ while keeping full context, and fine-tuned to regress both the final best validation accuracy and the epoch at which it occurs under a constrained JSON/YAML output schema. Stratified group splitting by (task, dataset, architecture) prevents leakage between similar models and enforces out-of-architecture generalization.

Within NNGPT, the predictor runs alongside training: after a few epochs, it estimates $(\widehat{\mathrm{acc}}_*, \widehat{t}_*)$ and can trigger early stopping when predicted accuracy is low, or prioritize promising runs in the job scheduler. As shown in Supplementary Sec. 5.1, the baseline model achieves RMSE ≈ 0.145 and Pearson $r \approx 0.78$ over 1,200+ runs, making it practical for compute-aware filtering and scheduling.

2.3.4. NN-RAG: retrieval augmented patching

The fourth pipeline augments generation with a retrieval system for reusable PyTorch blocks. NN-RAG scans curated repositories for non-abstract `nn.Module` subclasses that implement `forward()`, round-trips their source through LibCST [34] to preserve concrete syntax, and indexes parse trees and import relations in an SQLite-backed store with SHA-1 content addressing. Given a target architecture and a textual or code query (e.g., "multi-head attention block"), NNGPT retrieves candidate blocks and uses a scope-sensitive resolver to compute the minimal transitive closure of dependencies that respects Python's LEGB and import semantics. Definitions are reordered via topological sort to satisfy definition-before-use constraints, and the resulting scope-closed module is passed through AST parsing, bytecode compilation, and sandboxed import. On a corpus of 1,289 target blocks, this validator admits 941 modules (73% executability), covering attention, convolutions, transformer blocks [59], losses, and higher-level architectures. These validated blocks form a library that NNGPT can splice into generated models, enabling retrieval-augmented patching while preserving executability guarantees.

2.3.5. Reinforcement learning based closed loop

We add an RL layer on top of supervised fine-tuning, treating the LLM as a policy over architecture code and configurations. From LEMUR we build a masked-architecture corpus where layer definitions (e.g., `self.conv1 = nn.Conv2d(...)`) are replaced by placeholders; the model learns to reconstruct these blocks, acquiring realistic priors. At RL time it proposes full architectures that are executed via `check_nn`, which tests compilation, forward correctness, and a short CIFAR-10 [30] run. A scalar reward mixes syntactic validity, runtime feasibility, and validation accuracy, and a GRPO-style policy-gradient update (inspired by DeepSeek-R1 [12]) is applied to LoRA adapters.

In parallel, a channel-mutation engine performs deterministic width edits through five steps: `torch.fx` tracing, mapping graph nodes to source, planning shape-preserving changes, AST rewriting, and re-validation. Both RL-generated and mutated models pass through the same validation and training stack, and successful configurations are added to LEMUR, closing a performance-aware loop that explores and reinforces architectures that train well in practice.

Across all five pipelines, accepted configurations, logs, and artifacts are exported back into LEMUR dataset or external repositories, turning raw LLM generations into a continually expanding corpus of executable neural programs that subsequent NNGPT runs can query, modify, and build upon.

3. Experiments

Experiments run on a single 24GB GPU (Geforce RTX 3090/4090) using PyTorch and 8-bit or 4-bit LoRA adapters on code-capable LLMs within the AI Linux environment on a Kubernetes cluster. NNGPT interfaces with LEMUR via a fixed API (model factories, transforms, metrics), while the same adapter can target any PyTorch codebase.

We report component-wise results by design: each pipeline optimizes a distinct objective and metric (e.g., executability, validity, RMSE), avoiding misleading apples-to-apples comparisons with NAS/HPO baselines that cover only parts of the pipeline. All modules nonetheless share the same execution loop (schema validation $\rightarrow$ execution $\rightarrow$ logging $\rightarrow$ feedback), so results reflect interoperable parts of one closed-loop system.

For short-horizon evaluation, we adopt the true first-epoch validation accuracy as the primary performance metric [17], in lieu of zero-cost NAS proxies, which remain

Source	Prompt–code pairs
LEMUR (deduplicated, chat-format train split)	1,698
Self-generated (22 cycles, novel, $\geq$ 40% acc.)	455
Total by cycle 22	2,153

Table 2. **Closed-loop corpus growth for architecture refinement.** The training set is initialized from deduplicated LEMUR prompt–code pairs and expanded with self-generated architectures that pass executability, proxy-accuracy (one-epoch CIFAR-10), and novelty filtering.

Cycle	Valid (%)	Best (%)	Mean (%)	$\geq$40% (%)	Unique	Total train
1	44.0	47.78	28.06	2.04	1	1698
5	32.0	49.13	29.88	6.82	9	1724
10	53.8	55.48	37.70	38.04	18	1785
15	**66.8**	58.60	47.40	80.70	34	1911
18	59.1	**63.98**	**50.99**	**96.81**	**38**	2025
22	41.8	57.62	49.48	92.86	30	**2154**

Table 3. **Closed-loop refinement dynamics (CIFAR-10, one-epoch proxy).** Selected cycles from the multi-cycle architecture synthesis loop. The valid generation rate, the best and mean first-epoch accuracy, the fraction above a 40% proxy threshold, the number of structurally unique selected models, and the cumulative training-set size are reported.

indirect measures with reported rank correlation coefficients ($\rho \approx 0.5$–0.82) [1, 10, 35, 45, 61]. We evaluate performance on a diverse suite of mid-scale vision and language benchmarks, spanning image classification (MNIST, CelebA-Gender, CIFAR-10, CIFAR-100, SVHN, ImageNette, and Places365), COCO-based object detection and semantic segmentation, and text generation, following standardized LEMUR evaluation protocols. While individual experiments target specific benchmarks, select evaluations span the full dataset suite, enabling both focused and broad empirical coverage. Full implementation details, CLI tooling, and prompt specifications are deferred to the Supplementary Material (Sec. 3).

3.1. Closed-Loop Architecture Refinement

This study investigates the main pipeline of multi-cycle closed-loop deep model generation by LLM: NNGPT generates candidate architectures, filters them by executability, novelty, and one-epoch CIFAR-10 validation accuracy, and periodically fine-tunes the generator on accepted prompt–code pairs. Across 22 cycles, both proxy quality and training data coverage improve. Mean first-epoch accuracy increases from 28.06% (cycle 1) to 50.99% (cycle 18), with best one-epoch accuracy reaching 63.98% (Tab. 3). In parallel, the fraction of generated models exceeding a 40% proxy threshold rises from 2.04% to 96.81%, and the training corpus grows from 1,698 to 2,153 prompt–code pairs (Tab. 2). These trends support the NNGPT design principle that execution-grounded filtering and iterative fine-tuning improve architecture generation quality over time.

Category	Count	Examples
Attention	180	`MultiHeadAttention`, `SelfAttention`
Convolutions	220	`Conv2d`, `DeformConv`
Transformer blocks	150	`BertLayer`, `SwinTransformerBlock`
Pooling ops	110	`MaxPool2d`, `AdaptiveAvgPool2d`
Normalization	100	`BatchNorm2d`, `LayerNorm`
Losses	90	`FocalLoss`, `DiceLoss`
Architectures	150	`ResNet`, `EfficientNet`
Utilities	289	`DropPath`, `PatchEmbed`
Total	**1,289**	–

Table 4. **NN-RAG evaluation corpus by category.** Distribution of the 1,289 target PyTorch blocks that NN-RAG attempts to reconstruct as scope-closed modules. *Count* is the number of distinct block names per category; *Examples* list typical modules retrieved in each group.

To compare quality across datasets with different sample sizes, we report dataset-balanced means (Table **??**). The balanced mean accuracy peaks at $n = 3$ (`alt-nn3`, 53.1%) compared to 51.5% for the $n = 1$ baseline, while larger contexts ($n = 4, 5$) degrade performance to 47.3% and 43.0%. Gains are most pronounced on harder tasks such as CIFAR-100, where `alt-nn3` improves over `alt-nn1` by +11.6 percentage points after a single epoch. In aggregate, this pipeline contributes roughly 1.9k new architectures (and over 3k trained runs) to LEMUR, demonstrating that NNGPT can continuously grow its own training corpus through LLM-driven zero-shot generation.

3.2. NN-RAG: Executability and Correctness

We evaluate NN-RAG as a retrieval-augmented system for extracting reusable, scope-closed PyTorch blocks. The pipeline scans repositories for non-abstract `nn.Module` classes with `forward()`, round-trips sources through LibCST to preserve syntax, and indexes parse artifacts and import relations in an SQLite store with content hashing. For each target name, a scope-aware resolver computes the minimal transitive closure of dependencies (respecting Python's LEGB and import rules), orders definitions topologically, and emits a self-contained module that preserves original imports and aliases. Candidates then pass AST parsing, compilation, and sandboxed import to catch unresolved symbols and import-time failures.

We test on 1,289 target blocks from widely used PyTorch projects including: attention, convolutions, transformer components, pooling, normalization, losses, architectures, and utilities (Tab. 4). NN-RAG reconstructs 941 blocks as executable scope-closed modules, a 73.0% executability rate without manual intervention. Major libraries (torchvision [43], timm [63], transformers [64], OpenMMLab [50]) contribute many targets, while a long tail of smaller repos supplies the rest. This yields a large, validated library of import-complete building blocks that NNGPT can

Exp.	Model	Ep.	Correct /500	Failed	Pct. (%)
1	DeepSeek-Coder-V2-Lite-Base	15	8	492	1.60
2	DeepSeek-Coder-1.3b-base	10	435	65	87.00
3	**DeepSeek-Coder-1.3b-base**	**15**	**442**	**58**	**88.40**
4	DeepSeek-Coder-1.3b-base	25	0	500	0.00
5	DeepSeek-Coder-1.3b-base	35	4	496	0.80
6	DeepSeek-R1-Distill-Qwen-7B	15	457	43	91.40
7	**DeepSeek-R1-Distill-Qwen-7B**	**20**	**465**	**35**	**93.00**
8	DeepSeek-R1-Distill-Qwen-7B	35	176	324	35.20
9	DeepSeek-R1-Distill-Qwen-7B	35	268	232	53.60
10	DeepSeek-Coder-7b-base-v1.5	15	127	373	25.40
11	DeepSeek-Coder-7b-base-v1.5	20	114	386	22.80
12	DeepSeek-Coder-7b-base-v1.5	35	0	500	0.00
13	DeepSeek-Math-7b-base	15	214	286	42.80
14	DeepSeek-Math-7b-base	35	1	499	0.20
15	**DeepSeek-Coder-7b-Instruct-v1.5**	15	368	132	73.60
16	DeepSeek-Coder-7b-Instruct-v1.5	20	368	132	73.60
17	DeepSeek-Coder-7b-Instruct-v1.5	35	23	477	4.60
18	**CodeLlama-7b-Python-hf**	**21**	**460**	40	**92.00**

Table 5. **Structured hyperparameter generation on 500 held-out LEMUR configs.** Each row is a fine-tuned checkpoint (model, epochs). For every config, the LLM must output a complete hyperparameter set that matches a strict schema; a prediction is counted as *Correct* if it passes all schema/value checks (else *Failed*). Percentage is Correct/500. Bold = best per family and overall.

safely retrieve and assemble during LLM-driven synthesis.

3.3. Hyperparameter Generation Quality

We evaluate how well fine-tuned DeepSeek and CodeLlama models generate *valid* hyperparameter sets from structured prompts. Given a specification describing model architecture, dataset, task, transform pipeline, and desired accuracy, the LLM must output a complete hyperparameter configuration (names and values) conforming to the expected schema (Listing 5). Evaluation is performed on 500 held-out neural configurations sampled from LEMUR; a generation is counted as correct if it passes schema and value-type checks.

Table 5 reports the number of correct generations. The best overall result is obtained by DeepSeek-R1-Distill-Qwen-7B [12] (20 epochs), with 465 valid outputs (93.00%), slightly outperforming our fine-tuned CodeLlama-7b-Python-hf [54] (460/500, 92.00%), which has previously been used as a strong baseline for this task [29]. Smaller and domain-specialized DeepSeek variants also perform well: DeepSeek-Coder-1.3b-base [21] reaches 88.40% after 15 epochs, and DeepSeek-Coder-7b-Instruct-v1.5 [21] achieves a stable 73.60% at 15–20 epochs. At the same time, several runs with prolonged fine-tuning collapse to near-zero valid generations (e.g., 25–35 epochs for DeepSeek-Coder-1.3b-base and DeepSeek-Math-7b-base [57]), highlighting the sensitivity of structured generation tasks to overfitting and overtraining. This pipeline builds on our earlier study [29], which showed that a LoRA-tuned CodeLlama [55] can produce competitive one-shot hyperparameter configurations

17 CV Models (as in [29])			
Method	**RMSE**	**SE**	**95% CI**
Optuna*	0.589	0.004	[0.581, 0.597]
CodeLlama-7b-Python-hf*	**0.563**	0.004	[0.556, 0.570]
DeepSeek-Coder-1.3b-base	0.651	0.032	[0.588, 0.714]
DeepSeek-R1-Distill-Qwen-7B	0.659	0.037	[0.586, 0.732]
CodeLlama-7b-Python-hf	0.642	0.031	[0.554, 0.729]
LEMUR Dataset			
Method	**RMSE**	**SE**	**95% CI**
Optuna	0.636	0.002	[0.633, 0.640]
DeepSeek-Coder-1.3b-base	0.649	0.062	[0.526, 0.771]
DeepSeek-R1-Distill-Qwen-7B	0.652	0.029	[0.595, 0.709]
CodeLlama-7b-Python-hf	**0.603**	0.031	[0.543, 0.663]

Table 6. **Downstream training with LLM-generated hyperparameters vs. Optuna.** We compare one-shot LLM recommendations (no iterative search; 35–50 predictions per model) against Optuna (>20k trials) on two evaluation sets. Metric: RMSE↓ of $\epsilon_i = 1 - a_i$, where a_i is the observed validation accuracy when trained with predicted HPs. We report SE and normal-approx. 95% CI. Rows * reproduced from [29].

Method	Trial	RMSE↓	σ	95% Conf. Int.
Optuna	All	0.589	0.219	[0.581, 0.597]
Optuna	Best	0.416	0.115	[0.375, 0.456]
LLM (CodeLlama)	Fine-tuning 1	0.563	0.182	[0.556, 0.570]
LLM (CodeLlama)	Fine-tuning 2	0.567	0.159	[0.563, 0.572]
LLM (CodeLlama)	Best	0.404	0.118	[0.358, 0.480]
LLM (CodeLlama)	One-shot	0.533	0.162	[0.470, 0.596]

Table 7. **Prior evidence for one-shot HPO with LLMs.** RMSE, standard deviation (σ), and 95% confidence intervals reported in [29] for Optuna and a LoRA-tuned CodeLlama on the 17-model CV benchmark. "All" reports RMSE over all Optuna trials, while "Best" reports RMSE using only the best-achieved accuracies. For the LLM, we report results after two fine-tuning cycles, the best-achieved accuracies, and one-shot predictions.

relative to Optuna-style search while requiring far fewer trials; for example, on a 17-model CV benchmark, CodeLlama achieved RMSE 0.563 versus Optuna's 0.589 (Tab. 7). Our earlier HPO study also covers non-vision settings, including NLP [29].

To assess downstream effectiveness, we use the generated hyperparameters to train models on two evaluation sets: (i) the 17-model CV benchmark from [29] and (ii) the LEMUR dataset. For each configuration, we compute an error term $\epsilon_i = 1 - a_i$, where a_i is the observed accuracy under the predicted hyperparameters, and report RMSE over ϵ_i along with standard errors and 95% confidence intervals (Tab. 7).

On the 17-model benchmark, prior results for CodeLlama-7b-Python-hf [29] achieve an RMSE of 0.563, statistically outperforming Optuna [4] (0.589). Our DeepSeek-based models yield higher RMSEs and wider intervals on this heterogeneous testbed, consistent with their sensitivity to overfitting. On the LEMUR distribution, where schemas and transforms align more closely with

Variant	RMSE	R^2	r	within 5%	within 10%
Exp. 1 (baseline)	**0.1449**	**0.55**	**0.7766**	**50.0%**	**75.0%**
Exp. 2 (balanced + reg.)	0.2567	0.2885	0.6409	12.3%	45.0%

Table 8. **Global results over all datasets.** Baseline (Exp. 1) is used in NNGPT; Exp. 2 underperforms under added balancing/regularization.

training, DeepSeek-Coder-1.3b-base and DeepSeek-R1-Distill-Qwen-7B obtain RMSEs of 0.649 and 0.652, close to Optuna's 0.636, while our fine-tuned CodeLlama-7b-Python-hf achieves the best RMSE of 0.603. Although LLM-based methods show wider confidence intervals (partly because each model is evaluated on only 35–50 one-shot predictions compared with over 20 000 trials for Optuna), they approach or exceed Optuna's accuracy while avoiding iterative search, demonstrating that one-shot LLM hyperparameter generation can serve as a competitive, low-cost alternative in the NNGPT loop.

3.4. Few-Shot Architecture Prompting and LEMUR Growth

To quantify NNGPT's ability to expand LEMUR, we instantiate Few-Shot Architecture Prompting (FSAP). For each base model and dataset, the LLM receives a task description, one reference implementation, and $k \in \{1, \ldots, 6\}$ supporting architectures from strong LEMUR entries, and must synthesize a new PyTorch model that follows the required API. We denote the variants as `alt-nn`k. Generated code is schema-validated and deduplicated via a whitespace-normalized MD5 hash before training.

Table 9 summarizes yield and accuracy. With k=1 (`alt-nn1`), NNGPT produces 3,394 valid architectures; yield drops to 306 for k=2 and $\approx$ 100 for k=3–5, consistent with higher failure under longer contexts. At k=6 (`alt-nn6`), only 7 models are produced (99.8% failure), indicating context overflow. Overall, we obtain 4,033 candidates, of which $\sim$1,900 are unique after deduplication. `alt-nn3` achieves the highest balanced mean (53.1%), while accuracy generally degrades for k>3.

Additional results for code-aware accuracy, early-stop predictor, and the initial reinforcement-learning study are reported in the Supplementary Material (Sect. 5.1 and 5.2).

Limitations. Our deployment on a single 24 GB GPU limits prompt length, context window, and schema complexity. Experiments focus on mid-scale vision; large-scale datasets such as ImageNet-1k [14] and LVIS [23] are left for future work. We do not yet enforce architectural novelty or explicitly optimize for hardware efficiency. More broadly, NNGPT spans a wide design space, so we position this work as a first integrated systems implementation that demonstrates feasibility and interoperability; deeper analysis of bias, long-horizon overfitting, and cross-domain transfer remains future work.

Variant (k)	**Models**	**Success Rate**	**Bal. Mean (%)**
`alt-nn1`	**3,394**	100%	51.5 ± 29.5
`alt-nn2`	306	9.0%	49.8 ± 32.3
`alt-nn3`	103	3.0%	**53.1** ± 26.9
`alt-nn4`	102	3.0%	47.3 ± 32.5
`alt-nn5`	121	3.6%	43.0 ± 33.1
`alt-nn6`	7	0.2%†	—
Total	**4,033**	–	–

Table 9. **FSAP yield and accuracy vs. number of example architectures (k).** Each variant `alt-nn`k receives k example architecture codes as context. *Models*: schema-validated, deduplicated architectures; *Success rate*: yield relative to `alt-nn1` (100%); *Bal. Mean*: top-1 accuracy after one SGD+momentum epoch, macro-averaged across datasets with equal weight to correct for uneven sample counts. `alt-nn6` excluded from accuracy due to insufficient sample size (n=7). Total: 4,033 candidates ($\sim$1.9k unique).

† Severe context overflow (k=6).

4. Conclusion

We introduced **NNGPT**, a self-improving AutoML framework that turns an LLM into an online engine for neural network development. From a single prompt, NNGPT emits an executable training specification, validates and runs it, logs outcomes, and updates its generators and predictors from execution traces. The framework unifies five pipelines in one loop: zero-shot model generation and editing, hyperparameter recommendation, accuracy and early-stop prediction, retrieval-augmented code synthesis, and reinforcement learning updates. Unlike classical NAS, NNGPT avoids a fixed, template-limited search space by using LLM-driven synthesis grounded in an executable corpus and automated validation.

Experiments demonstrate practical effectiveness: zero-shot generation added over 5,000 trained models and about 1.9k unique architectures to LEMUR; NN-RAG reconstructs 941 of 1,289 target blocks (73% executability); one-shot HPO reaches 93% schema-valid outputs and RMSE close to Optuna; and the code-aware predictor achieves RMSE 0.1449 with Pearson $r = 0.7766$ across more than 1,200 runs, supporting early termination and scheduling. Initial RL results are reported in the Supplementary Material as a diagnostic feasibility study. Overall, NNGPT reduces AutoML cost and latency by replacing many-trial search with one-shot, schema-validated generation while maintaining transparent, reproducible logs. Code, prompts, and checkpoints are released to support replication and extension.

References

[1] Mohamed S. Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D. Lane. Zero-cost proxies for lightweight NAS. In *International Conference on Learning Representations*, 2021. 6

[2] Adriaensen et al. Efficient bayesian learning curve extrapolation using prior-data fitted networks. *Advances in Neural Information Processing Systems*, 36:19858–19886, 2023. 3, 2

[3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, page 2623–2631, New York, NY, USA, 2019. Association for Computing Machinery. 1, 2

[4] Akiba et al. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019. 7, 1

[5] Amir Bassamzadeh and Samira Methani. Enhancing llm output quality for specialized apis: Fine-tuning versus retrieval. *arXiv preprint arXiv:2401.01234*, 2024. 1

[6] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, page 2546–2554, Red Hook, NY, USA, 2011. Curran Associates Inc. 1

[7] Bergstra et al. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. *SciPy*, 13: 20, 2013. 2, 1

[8] Angelica Chen, David M. Dohan, and David R. So. Evoprompting: Language models for code-level neural architecture search. *arXiv preprint arXiv:2302.14838*, 2023. 2, 1

[9] Mark et al. Chen. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 1

[10] Wuyang Chen, Xinyu Gong, and Zhangyang Wang. Neural architecture search on ImageNet in four GPU hours: A theoretically inspired perspective. In *International Conference on Learning Representations*, 2021. 6

[11] Xiangning Chen and Cho-Jui Hsieh. Stabilizing differentiable architecture search via perturbation-based regularization. In *International conference on machine learning*, pages 1554–1565. PMLR, 2020. 1

[12] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. 5, 7

[13] DeepSeek-AI et al. DeepSeek-V3 technical report, 2025. 1

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. 8

[15] Ding et al. Architecture-aware learning curve extrapolation via graph ordinary differential equation. *arXiv preprint arXiv:2412.15554*, 2024. 3, 2

[16] Domhan et al. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, pages 3460–3468, 2015. 3, 2

[17] Romain Egele, Felix Mohr, Tom Viering, and Prasanna Balaprakash. The unreasonable effectiveness of early discarding after one epoch in neural network hyperparameter optimization. *Neurocomputing*, 597:127964, 2024. 5

[18] Mohamed Gado, Towhid Taliee, Muhammad Danish Memon, Dmitry Ignatov, and Radu Timofte. Vist-gpt: Ushering in the era of visual storytelling with llms? *arXiv preprint arXiv:2504.19267*, 2025. 1

[19] Golovin et al. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*, pages 1487–1495. ACM, 2017. 2, 1

[20] Arash Torabi Goodarzi, Roman Kochnev, Waleed Khalid, Furui Qin, Tolgay Atinc Uzun, Yashkumar Sanjaybhai

Dhameliya, Yash Kanubhai Kathiriya, Zofia Antonina Bentyn, Dmitry Ignatov, and Radu Timofte. LEMUR Neural Network Dataset: Towards Seamless AutoML, 2025. 2

[21] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. 7

[22] Guo et al. DeepSeek-Coder: When the large language model meets programming - the rise of code intelligence, 2024. 1

[23] Agrim Gupta, Piotr Dollár, and Ross Girshick. Lvis: A dataset for large vocabulary instance segmentation, 2019. 8

[24] He et al. Designing network algorithms via large language models. *arXiv preprint arXiv:2404.01617*, 2024. 2

[25] Jeremy Howard. Imagenette: A smaller subset of ImageNet for quick experimentation. https://github.com/fastai/imagenette, 2019. 6

[26] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models, 2021. 2

[27] Hu et al. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR 2022)*, 2022. 1

[28] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1946–1956, 2019. 2, 1

[29] Roman Kochnev, Arash Torabi Goodarzi, Zofia Antonina Bentyn, Dmitry Ignatov, and Radu Timofte. Optuna vs Code Llama: Are LLMs a New Paradigm for Hyperparameter Tuning? In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, pages 5664–5674, 2025. 1, 7

[30] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). 5, 6, 7

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1097–1105, 2012. 7

[32] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. MNIST. 6, 7

[33] Li et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. 1

[34] LibCST Contributors. Libcst: Concrete syntax tree parser and toolkit for python. https://libcst.readthedocs.io/en/latest/, 2025. Accessed: Nov. 13, 2025. 5

[35] Ming Lin, Pichao Wang, Zhenhong Sun, Hesen Chen, Xiuyu Sun, Qi Qian, Hao Li, and Rong Jin. Zen-NAS: A zero-shot NAS for high-performance image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12265–12275, 2021. 6

[36] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015. 6

[37] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018. 2, 1

[38] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *ICCV*, 2015. CelebA dataset. 6

[39] Liu et al. Large language models to enhance bayesian optimization. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. 2

[40] Liu et al. Large language models to enhance bayesian optimization, 2024. 2

[41] S. Long, Y. Wang, and H. Zhang. Performance prediction based on neural architecture features. *Cognitive Computation and Systems*, 2(2):80–83, 2020. 3, 2

[42] Machida et al. Msr-darts: Minimum stable rank of differentiable architecture search. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9. IEEE, 2022. 1

[43] TorchVision maintainers and contributors. "torchVision: PyTorch's computer vision library", "2016". 6, 5

[44] Mangrulkar et al. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022. 3

[45] Joe Mellor, Jack Turner, Amos Storkey, and Elliot J. Crowley. Neural architecture search without training. In *Proceedings of the 38th International Conference on Machine Learning*, pages 7588–7598. PMLR, 2021. 6

[46] Yash Mittal, Dmitry Ignatov, and Radu Timofte. Preparation of fractal-inspired computational architectures for advanced large language model analysis. *arXiv preprint arXiv:2511.07329*, 2025. 3

[47] Nasir et al. Llmatic: Neural architecture search via large language models and quality-diversity optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '24)*, page 1–9, Melbourne, VIC, Australia, 2024. ACM. 2, 1

[48] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011. SVHN. 6, 7

[49] OpenAI et al. GPT-4 technical report, 2024. 1

[50] OpenMMLab Contributors. Openmmlab: Open-source computer vision toolbox ecosystem. https://openmmlab.com/, 2023. 6

[51] Paszke et al. PyTorch: An imperative style, high-performance deep learning library, 2019. 2, 3

[52] Peng et al. The impact of AI on developer productivity: Evidence from GitHub Copilot, 2023. 1

[53] Pham et al. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR, 2018. 2, 1

[54] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code, 2024. 1, 7

[55] Rozière et. al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2024. 7, 1

[56] Bhavya Rupani, Dmitry Ignatov, and Radu Timofte. Exploring the collaboration between vision models and llms for enhanced image classification. *Dimensions*, 27(1), 2025. doi:10.13140/RG.2.2.14615.69284. 1

[57] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. 7

[58] Alex Sheng and Shankar Padmanabhan. Self-programming artificial intelligence using code-generating language models. *arXiv preprint arXiv:2205.00167*, 2022. 2, 1

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. 5, 6

[60] Wang et al. Grammar prompting for domain-specific language generation with large language models. In *Advances in Neural Information Processing Systems*, 2023. 2, 1

[61] 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. 6

[62] White et al. How powerful are performance predictors in neural architecture search? In *NeurIPS*, pages 28454–28469, 2021. 3, 2

[63] Ross Wightman. Pytorch image models (timm). https://github.com/rwightman/pytorch-image-models, 2019. 6

[64] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. HuggingFace's Transformers: State-of-the-art Natural Language Processing, 2020. 6, 3

[65] Yu et al. GPT-NAS: Generative pre-trained models for neural architecture search. *arXiv preprint arXiv:2310.12345*, 2023. 2, 1

[66] Yu, Xi and others. Cyclic differentiable architecture search. *IEEE transactions on pattern analysis and machine intelligence*, 45(1):211–228, 2022. 1

[67] Li Zhang, Ruizhi Chen, and Pengfei Yu. Llm-assisted hyperparameter tuning: A prompt-based approach. *arXiv preprint arXiv:2308.04214*, 2023. 2

[68] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(6):1452–1464, 2018. Places365. 6

NNGPT: Rethinking AutoML with Large Language Models

Supplementary Material

1. Demo Video: TuneNNGen in Action

We include a time-lapse screen recording that demonstrates the end-to-end workflow of `TuneNNGen.py`: prompt assembly, one-shot YAML/code generation, schema validation, training and logging, and LoRA updates. It also shows typical failure modes (e.g., schema violations, missing functions) and how NNGPT surfaces them through the validator and executor stack. Refer to the supplementary video for the full run, including intermediate console outputs and per-model evaluation messages.

2. Extended Related Work and Detailed Comparisons

Automated neural network configuration has long been a focus of research, with prior work falling broadly into two categories: classical search-based optimization and recent LLM-driven generative approaches. Below we review key methods from both domains, focusing on the limitations addressed by NNGPT.

Search-Based Hyperparameter Optimization. Frameworks like Optuna [3], Hyperopt [7], and Google Vizier [19] use Bayesian optimization and Tree-structured Parzen Estimators (TPE) [6] to iteratively tune hyperparameters. Though effective, these methods demand significant compute, often requiring hundreds of training runs, and act as black-box optimizers, offering limited transparency into the configuration semantics or model behavior beyond their final scores.

AutoML and Neural Architecture Search (NAS). AutoML systems such as Optuna [4] and Hyperopt treat tuning as black-box optimization, while NAS frameworks like ENAS [53], AutoKeras [28], and DARTS [37] (plus variants like CDARTS, SDARTS, iDARTS) expand this to architecture generation via reinforcement learning or differentiable search. These methods, however, remain resource-intensive and adapt slowly to new tasks.

Efforts to stabilize NAS, such as RC-DARTS [66], NDARTS [11], and MSR-DARTS [42], still rely on costly infrastructure. NNGPT bypasses iterative search entirely: a single LLM call produces a complete, schema-valid configuration that is directly executable and logged for traceability. This design reduces computational cost while maintaining competitive accuracy, as shown by our DeepSeek-based results.

LLMs for Code and Configuration Generation. Recent advances in code-oriented LLMs such as Codex [9], Code Llama [55], and DeepSeek-Coder [22] enable automatic generation of hyperparameters, architectures, and training scripts. Tools like GitHub Copilot[52] and AlphaCode [33] further highlight their utility for programming tasks.

Most existing systems, however, lack integration with execution pipelines. Outputs are rarely validated empirically or tested for reproducibility. EvoPrompting [8] embeds LLMs into evolutionary search but requires repeated inference and lacks schema control.

NNGPT employs a one-shot generation strategy: a fine-tuned LLM produces a schema-compliant YAML configuration that is immediately validated and executed. Structured output enforcement (via Pydantic) and empirical feedback support reliable AutoML automation.

Prediction of Neural Network Accuracy. Despite advances in LLMs, many methods rely on high-level descriptions that miss implementation-specific patterns, or predict isolated metrics instead of full training dynamics. Yet performance prediction is vital for tuning efficiency and resource use in AutoML.

We address this via a multi-modal predictor that processes both structured metadata (e.g., task, dataset) and executable code (e.g., `nn_code`, `metric_code`). Key contributions include: (1) joint modeling of code and metadata, (2) efficient 4-bit QLoRA fine-tuning [27] to predict accuracy and stopping points, and (3) stratified data splits that prevent leakage while preserving balance.

LLM-Driven Neural Network Generation. LLMs are increasingly integrated into NAS pipelines. EvoPrompting uses a pretrained LLM as a mutation operator, requiring multiple inference calls and soft prompt-tuning [8]. LLMatic replaces hand-written mutations with CodeGen-generated code [47]. GPT-NAS [65] completes partial architectures, and Self-Programming AI [58] revises its own configs.

While effective, these methods rely on expensive iterative queries. In contrast, NN-GPT performs one-shot generation: a single LoRA-tuned LLM call yields an executable, schema-compliant configuration with no further iteration required.

Schema-Constrained Generation. LLMs frequently generate syntactically invalid code in structured tasks. Grammar Prompting mitigates this by embedding formal grammars (e.g., Backus–Naur Form) into prompts [60], while RAG methods reduce errors by referencing external examples [5]. NN-GPT adopts a different approach: it employs a strict Pydantic schema tailored to YAML configurations. Minor issues are auto-corrected; major ones trigger a single re-prompt, ensuring valid, executable outputs without relying on grammars or retrieval.

LLMs for Pipeline Optimization. LLMs have also been explored for broader pipeline automation. LLAMBO [39] frames Bayesian optimization as an LLM-driven dialogue, where the model proposes new candidates based on previous evaluations, outperforming classical BO in sparse regimes. Prompt-based HPO uses GPT-4 to refine hyperparameters from task descriptions[67], and NADA[24] generates full control algorithms, filtering them through simulation. In contrast, NN-GPT focuses on end-to-end generation: a single LLM call outputs complete, runnable training recipes — data loaders, models, and optimization logic, without requiring iterative refinement.

Training Performance Prediction. Efforts to predict training outcomes fall into three main groups, each with key limitations. Statistical curve fitting [2, 16] extrapolates final metrics from early trends but ignores architecture and implementation, reducing generalizability. These approaches treat training as a numeric process, overlooking structural factors that influence convergence behavior.

Architecture-Based Performance Predictors. Methods like [41, 62] use textual summaries or engineered features to estimate model performance. However, such abstractions overlook critical implementation-level details, limiting their ability to capture the nuanced behaviors encoded in executable code.

Graph-Based Approaches. Techniques such as [15] represent models via explicit computational graphs, sometimes enhanced with differential modeling. Although more expressive than text, these methods require costly manual graph construction and still miss fine-grained code-level patterns such as custom regularization or dynamic scheduling, which strongly affect training outcomes.

Multimodal Prediction via Executable Code. While prior approaches either discard architectural context [2, 16] or rely on lossy text abstractions [41, 62], our method directly processes executable code snippets (`nn_code`, `metric_code`, `transform_code`) within the LLM prompt. This enables the model to learn implementation-specific patterns, e.g., custom regularization, branching, or metric logic, that strongly influence convergence but are inaccessible via abstract representations.

In contrast to graph-based methods [15] requiring manual topology construction, we combine three complementary inputs: (i) structured metadata, (ii) raw executable code, and (iii) early training dynamics. This multimodal context allows the model to align syntactic code patterns with numerical trends, revealing complex interactions that are missed by unimodal models.

Moreover, our LLM jointly predicts final accuracy and optimal stopping points within a shared latent space, capturing their intrinsic correlation — unlike prior works that treat them separately. This unified structure enhances generalization across tasks and architectures.

Table 1 situates NNGPT among classical AutoML toolkits and recent LLM-driven approaches. Bayesian HPO and NAS frameworks provide strong baselines for search over hyperparameters and architectures but rely on iterative evaluation and do not synthesize full training programs. Recent LLM-based methods introduce generation and closed-loop ideas, yet typically cover only parts of the pipeline or lack robust execution and prediction components. In contrast, NNGPT combines one-shot generation of executable specifications, code-aware accuracy prediction, and a closed feedback loop inside a single open-source framework.

3. Implementation

The NNGPT framework is implemented in pure Python and provided as a pip-installable package. It requires Python 3.10 or higher and CUDA for GPU-based training. The core components are accessible via top-level command-line interfaces such as `NNAlter*.py`, `NNEval.py`, `TuneNNGen*.py`, `TuneAccPrediction.py`, `TuneHyperparameters.py` or `TuneRAG.py`. These tools together compose the full pipeline introduced in Section 4.

3.1. Repository Structure

The codebase is organized into modular subdirectories. The `ab/gpt` directory contains the main scripts. Prompt templates, training configurations, and YAML schema definitions are defined in `ab/gpt/conf`, while LoRA fine-tuning utilities and other helper functions reside in `ab/gpt/util`.

3.2. Command-Line Interface

The script `NNAlter.py` performs seed model alteration using architectures from the LEMUR corpus. Each modified model is trained for a small number of epochs (default: 8, configurable via the `-e` flag). This utility relies on the `ab.gpt.util.AlterNN.alter` method and defaults to the deepseek/ai/DeepSeek-R1-Distill-Qwen-7B model.

The `TuneNNGen*.py` scripts handle one-shot LLM-based generation, YAML schema validation, full model training, and optional LoRA fine-tuning. A fast iteration mode can be enabled by passing the `-s` flag, which skips the alteration phase.

Additionally, `NNEval.py` offers standalone evaluation of generated architectures without invoking LoRA, enabling targeted testing of LLM outputs.

All required CUDA and MPI dependencies are prepackaged in the public Docker image.

3.3. Prompting and Inference Flow

Prompt assembly begins with encoding task specifications, dataset, metric, and compute constraints into a structured JSON block. This prompt is then transformed into strict

YAML by the LLM. The DeepSeek 7B model is loaded via `transformers.AutoModelForCausalLM` in 8-bit precision, with LoRA adapters applied as needed.

3.4. Distributed Training Back-End

Model training is executed using `PyTorch` [51] and `torch.distributed`. MPI support is configured at install time using system-level dependencies. During training, per-epoch statistics such as loss, accuracy, mAP/IoU, GPU utilization, and Git SHA are recorded in a local SQLite database via SQLAlchemy. If an optional statistics module is available, the database can be exported to Excel-compatible dashboards using a single API call.

3.5. LoRA Fine-Tuning Loop

LoRA adapters are fine-tuned after every 50 new model training runs. The fine-tuning procedure consists of three epochs using a linear warm-up and cosine decay schedule. The resulting adapter checkpoints are published to the framework's Hugging Face namespace, making them available for the next generation cycle.

3.6. Consistent Dependencies

NNGPT ensures consistency and reproducibility through a pre-built Docker image and a pinned `requirements.txt` file that specifies fixed versions of key libraries such as `torch` [51], `transformers` [64], and `peft` [44].

3.7. Prompt Generation Pipeline

Prompt generation is template-driven. A JSON file defines the structure of the prompt, including optional insertion points for code samples. This enables dynamic construction of prompts containing one or two models sampled from the database, under a given task. For training data generation, prompts may be further constrained to ensure shared or divergent characteristics between samples, improving task alignment while maintaining flexibility.

3.8. Offline Initialization of the Training Set

During early experiments, the raw DeepSeek-Coder-1.3B model frequently refused to modify input code, either claiming sufficiency or producing rule-violating changes. Introducing stricter task constraints often led to outputs that ignored parts of the prompt or simply returned unmodified code. These observations motivated the need for a bootstrapped training set.

To address this, the larger DeepSeek-R1-Distill-Qwen-7B model was used to generate initial examples. During 20 offline epochs, this model produced modified architectures by applying simple transformations, such as changing channel widths or digits in `Conv2d` layers, without requiring accuracy improvements. The first prompt template is shown in Listing 1. To diversify the training set, additional prompts requested the model to modify 2 to 4 digits randomly. Since DeepSeek-R1 7B reliably introduced at least one valid change, its outputs served as the initial training corpus for fine-tuning the smaller model.

During prompt engineering, both one-shot and few-shot examples were evaluated. However, neither significantly improved task fidelity on the raw models. In fact, DeepSeek-Coder-1.3B and DeepSeek-R1-Distill-Qwen-7B often lost track of task requirements or confused multiple examples. Zero-shot prompting consistently yielded more predictable behavior and was therefore adopted.

```
Define two `nn.Conv2d()` layers as a neighboring
    pair
    when there are only non-Conv2d layers between
         them.

Your task:
      - Randomly pick and modify 1 or 2
          neighboring pairs
        of `nn.Conv2d()` layers based on the
            rules below.
      - Return ONLY the full result.

Instructions:
      1. For each picked neighboring pair:
           - Change the out-channels of the
               former `nn.Conv2d()`
             and the in-channels of the latter to
                  the same new integer.
      2. Modify the numbers directly in the code.
         DO NOT introduce new methods or
             parameters.
      3. Do NOT pick the first neighboring pair;
          select randomly.
      4. You MUST return the FULL CODE, including
           all classes - even if unmodified.
      5. If no such pair exists:
           - Randomly change 2 digits in the code
                to new values.
      6. Do not make any changes beyond the rules
           above.

Here is the input code for you to modify:
    '''\n{nn_code}\n'''
```

Listing 1. Prompt instructing the LLM to modify neighboring nn.Conv2d() layers under strict rules, ensuring in-place edits and structural preservation of PyTorch code.

3.9. Automated LEMUR Dataset Extension

The extended dataset was generated using prompt templates,such as those of FractalNet [46], similar to those used during fine-tuning (e.g., Listing 2). Generated samples were stored alongside their originating code and task context to ensure proper evaluation. Certain datasets such as COCO required specific preprocessing (e.g., fixed `Resize` operations), and segmentation tasks mandated the use of IoU rather than accuracy.

```
1. Change the layers in the following
    implementation to improve accuracy on image
    classification tasks.

2. Strictly follow these constraints:
      - Modify ONLY the layers in the code.
      - DO NOT introduce new methods or
          parameters.

3. Your response MUST:
      - Include the full code (including
          unchanged classes or functions).
      - Indicate the modified class explicitly.
      - Actually make at least one layer
          modification.
      - Not worry about decreasing accuracy - it'
          s acceptable

Input code: '''\n{nn_code}\n'''
```

Listing 2. Prompt directing the LLM to modify existing layers in a neural network implementation under strict constraints, requiring full-code output and explicit identification of edits.

To extract generated code, regular expressions were used to match code blocks wrapped in triple backticks. To avoid infinite loops in parsing, the number of backticks was counted before applying regex-based extraction. Evaluation was performed via the LEMUR API. If parsing or execution errors occurred, the failing code and exception trace were returned to the LLM for repair. Each sample was given up to two correction attempts. Validated models were saved in structured directories compatible with the LEMUR dataset format and integrated into the local repository for downstream use.

3.10. Fine-Tuning for Accuracy Prediction

The model Deepseek-coder-1.3b-Instruct was fine-tuned via supervised learning to enable the prediction of validation accuracy changes resulting from architectural modifications. To align with the offline generation procedure described earlier, the fine-tuning objective was formulated as a code-to-code transformation task, where the model is presented with an original implementation, its evaluated accuracy, and the accuracy of a known improved variant. The expected output is the modified architecture that corresponds to the target accuracy, drawn from the LEMUR dataset.

The prompt, shown in Listing 3, encodes this training setup explicitly. It instructs the model to alter only layer definitions in order to achieve the desired accuracy gain, without introducing new methods or structural components. The fields {accuracy} and {addon_accuracy} represent the original and improved accuracy, respectively, and {nn_code} contains the complete source code of the base model. The expected output is the full code of the improved model, including unchanged components, so the model also learns to preserve contextually irrelevant structures.

```
1. Change the layers in the following
    implementation to improve accuracy from {
    accuracy} to {addon_accuracy} on image
    classification tasks.
2. Strictly modify ONLY the layers in the code
    and STRICTLY DO NOT INTRODUCE NEW METHODS OR
    PARAMETERS
3. You should answer not only the class you
    modified but with FULL CODES INCLUDING
    EVERYTHING (Other classes, etc.) NOT CHANGED
4. You have to reply will full codes, don't be
    afraid of actually reducing the accuracy.
Strictly check and make sure at least one layer
    being modified.
```

Listing 3. Prompt template used to generate training data by requesting constrained layer modifications aimed at improving model performance.

After eight epochs of supervised fine-tuning, the model consistently produced syntactically valid outputs that adhered to the required formatting and structural constraints. It successfully learned to apply targeted architectural modifications that align with observed changes in validation accuracy, and reliably preserved all components mandated by the LEMUR API specification.

In addition to replicating transformations observed during training, the model demonstrated the ability to compose valid variants across different architecture families. It consistently selected effective transformation strategies for given task-model pairs, producing high-quality outputs with minimal prompt engineering. This behavior reflects an emerging structural prior that favors stable and empirically grounded edits.

Moreover, the model preserved functional integrity even in cases where only subtle changes were required, maintaining coherence across unchanged code regions. These results confirm that the fine-tuned model captures not only the mapping between code edits and accuracy shifts, but also the appropriate output syntax and modular consistency necessary for downstream execution and evaluation.

3.11. Fine-Tuning for Hyperparameter Generation

In addition to accuracy prediction, NNGPT also supports automated generation of numeric hyperparameter values tailored to a given neural network, task, and dataset. To enable this capability, we fine-tuned a diverse collection of DeepSeek models using supervised learning on entries from the LEMUR dataset. The selected set spans a wide range of model scales and pretraining strategies, including both lightweight and instruction-tuned architectures. This diversity allowed us to examine how model size and initialization influence hyperparameter reasoning ability.

Each model was fine-tuned using the LoRA method to reduce memory overhead while preserving gradient flow across transformer layers. LoRA rank and alpha were selected from 16, 32, 64, with a dropout of 0.05. All ex-

periments employed a consistent setup that included the AdamW optimizer, a cosine learning rate scheduler, a fixed learning rate of 3×10^{-5}, gradient accumulation, checkpointing, and a training duration of up to 35 epochs depending on convergence behavior.

Training prompts were constructed to align with the LEMUR schema and encode structured conditioning variables: architecture code, target accuracy, number of training epochs, and required transformations. The input template, shown in Listing 4, requests the LLM to predict the values of specific hyperparameters that would lead the given model to achieve a known accuracy level.

```
### Input:
    "Generate only the values (do not provide
        any explanation) of the hyperparameters
        ({prm_names}) of a given model:
        {entry['nn']} for the task:
        {entry['task']} on dataset:
        {entry['dataset']}, with transformation:
        {entry['transform_code']}, so that the
        model achieves accuracy =
        {entry['accuracy']} with number of
        training epochs = {entry['epoch']}. Code
        of that model: {entry['nn_code']}"

### Response:
    "Here are the hyperparameter values for
        which the model will achieve the
        specified accuracy: {prm_values}."
```

Listing 4. Prompt template used for fine-tuning LLMs to generate hyperparameter values based on model code, task, dataset, transformations, target accuracy, and training duration.

During evaluation, a modified version of the prompt (Listing 5) asked the model to produce hyperparameters aimed at maximizing accuracy, thereby testing generalization to unseen combinations.

```
### Input:
"Generate only the values (do not provide any
    explanation) of the hyperparameters
    ({prm_names}) of a given model:
    {entry['nn']} for the task: {entry['task']}
    on dataset: {entry['dataset']}, with
    transformation: {entry['transform_code']},
    so that the model achieves the HIGHEST
    accuracy with number of training epochs =
    {entry['epoch']}. Code of that model:
    {entry['nn_code']}"
```

Listing 5. Prompt for generating hyperparameter values that maximize accuracy, conditioned on model code, task, dataset, transformation logic, and training epochs.

All fine-tuned models were subsequently integrated into the NNGPT pipeline. Specifically, two of the best performing checkpoints, DeepSeek-Coder-1.3b-Base and DeepSeek-R1-Distill-Qwen-7B, were published to Hugging Face and are now directly accessible within the framework. These models can be used as plug-and-play components for automated hyperparameter suggestion in both evaluation and generation modes.

By extending the NNGPT workflow with fine-tuned LLMs for hyperparameter prediction, we close the loop between architecture generation and training configuration, offering a unified and fully automated AutoML solution. Experimental results evaluating the quality and effectiveness of these generated hyperparameters are presented in Section 3.3.

4. Methodology

NNGPT reconceptualizes LLMs as fully automated configuration agents for neural networks. Given a high-level task specification, the system follows the eight-stage pipeline depicted in Figure 1, encompassing architecture retrieval, prompt generation, model evaluation, and iterative self-improvement. Below, we detail each stage.

4.1. Stage 1: LEMUR API

NNGPT starts by querying the LEMUR Dataset API for executable architectures and metadata with standardized training and evaluation. LEMUR provides full implementations, uniform preprocessing, and reproducible metrics, so one-shot generation targets runnable code and uses LEMUR as a reliable metric oracle. Unlike torchvision [43], it adds reproducible benchmarking, native metric logging, and direct access to architecture code, making it well suited for generative AutoML.

4.2. Stage 2: Neural Network Data Retrieval

Following the API query, candidate architectures and associated metadata are retrieved from the LEMUR corpus. A dedicated script, `NNAlter*.py`, then applies a controlled set of architectural perturbations - including insertion/removal of layers, changes to channel widths, and alterations to residual connections. Each altered model undergoes a lightweight convergence check via a short warm-up training run (with the `-e` flag), ensuring that the resulting design remains viable. The resulting population of diverse, training-validated models is archived in an SQLite-backed repository, which serves as training data for LLM fine-tuning and architecture prediction.

4.3. Stage 3: Configuration Setup

After retrieval, a configuration scaffold is constructed that encapsulates the task, dataset, desired metric, and (optionally) a compute budget. This configuration is used to parameterize the prompt generator, enabling reproducible and deterministic experiment assembly. The prompt generation code is modular and extensible, supporting dynamic reconfiguration and alternate LLM backbones through Hugging Face-compatible interfaces.

4.4. Stage 4: Prompt Assembly & Injection

The core input to the LLM is a structured prompt, assembled from the configuration data. This includes a JSON object describing the task, dataset, metric, and resource constraints. The prompt is formatted with system-level instructions that force the model to emit a strict YAML block beginning with `-yaml` and following a predefined schema.

The generative pass itself is carried out using a DeepSeek-V2 16B model equipped with 8-bit LoRA adapters fine-tuned on the altered-model corpus. Sampling is configured using $T = 0.2$, $p = 0.9$, and top-1 selection to minimize unnecessary exploration. Output is parsed with `ruamel.yaml` and validated using `pydantic`. Minor schema violations are auto-corrected, while major issues trigger a re-prompt with an embedded diagnostic message. Only configurations that pass validation proceed to execution.

4.5. Stage 5: LLM-Guided Architecture Generation

Once validated, the YAML output is interpreted as an experiment specification that includes a full architecture graph, optimizer parameters, loss functions, and data transformations. This file serves as the input to the training engine. Generated architectures may differ from their seed counterparts not only in topology but also in auxiliary properties such as normalization strategies or metric implementations. This stage integrates architectural creativity with empirical feasibility, ensuring that every configuration is both novel and executable.

4.6. Stage 6: Model Validation and Evaluation

Training is conducted using a distributed PyTorch engine (`ab/gpt/TuneNNGen*.py`), which uses `mpi4py` and `torch.distributed` for multi-GPU support. All datasets are accessed through the LEMUR API to ensure consistent preprocessing and evaluation. During training, metrics such as loss, accuracy, mAP, and IoU are logged per epoch alongside GPU utilization and configuration hashes. These logs are persisted in the same SQLite database as the architecture graph, enabling traceability and post-hoc analysis via an internal visualization suite designed for model benchmarking.

In parallel, a lightweight accuracy prediction module is used to estimate final model performance based on early training signals and static features. The predictor processes three input modalities: (i) structured metadata, (ii) executable code (model, transform, metric), and (iii) early accuracy values. A stratified split strategy ensures robust generalization and prevents data leakage across similar architectures.

4.7. Stage 7: LoRA Fine-tuning

After $k = 50$ successful training runs, a new fine-tuning iteration is triggered. LoRA adapters are retrained on the extended dataset of architecture–performance pairs using three epochs of linear warm-up and cosine decay. This updates the LLM's internal representation of the architecture space, biasing future generations toward successful configurations. The updated checkpoints are immediately used in subsequent pipeline passes.

4.8. Stage 8: Database Storage

All artifacts, including trained weights, YAML specifications, metric logs, and architectural source code, are stored in a structured SQLite-based repository designed for long-term reproducibility. This supports future reanalysis, dataset expansion, and model distillation. The continuous accumulation of structured training data reinforces the NNGPT pipeline, transforming raw LLM generations into a curated and executable AutoML knowledge base.

4.9. AutoML Loop

Once all eight stages complete, the pipeline automatically restarts using updated model checkpoints, prompt templates, and database entries. This iterative process enables NNGPT to function as a self-improving AutoML cycle, where each generation, validation, and fine-tuning pass enhances the quality and diversity of future outputs.

5. Additional Experimental Results

5.1. Accuracy and Early-Stop Prediction

We instantiate the predictor H_ϕ as a code-aware LLM fine-tuned with 4-bit QLoRA on 1,200+ NNGPT/LEMUR runs spanning CNNs and Transformers [59] on CelebA-Gender [38], CIFAR-10/100 [30], ImageNette [25], MNIST [32], Places365 [68], SVHN [48], and COCO [36]. Each run is converted into a structured prompt with `nn_code`, `transform_code`, `metric_code`, hyperparameters, dataset/task metadata, max epochs, and validation accuracies at epochs 1–2; the model predicts the final best validation accuracy and its epoch. To prevent leakage, we use stratified group splits by (task, dataset, architecture). Per-dataset outcomes are summarized in Tab. 1.

We report two variants. The baseline (Exp. 1) is a QLoRA-tuned code LLM without additional balancing/strong regularization; it achieves RMSE $= 0.1449$, $r = 0.7766$, and $R^2 \approx 0.55$ globally (see Tab. 8). Correlation varies by dataset, ranging from moderate on CIFAR-10/SVHN to strong on COCO, while roughly 50% and 75% of predictions fall within 5% and 10% of the true final accuracy, respectively. The dataset-wise heterogeneity is visible in Tab. 1. We use the predictions both as a proxy for final

Dataset	RMSE [95% CI]	Pearson r (p)	Notes
CelebA-Gender	0.0497 [0.044, 0.056]	0.2125 ($p = 2.38 \times 10^{-7}$)	Low correlation despite low RMSE
CIFAR-10	0.1589 [0.150, 0.168]	0.3472 ($p = 3.77 \times 10^{-30}$)	Moderate correlation
CIFAR-100	0.1668 [0.155, 0.178]	-0.0276 (*ns*)	Very poor ranking ability
ImageNette	0.2439 [0.221, 0.266]	0.4973 ($p = 1.25 \times 10^{-36}$)	High error, moderate r
MNIST	0.0331 [0.025, 0.040]	0.4224 ($p = 1.04 \times 10^{-22}$)	Excellent precision
Places365	0.1672 [0.135, 0.193]	-0.0297 (*ns*)	Weak generalization
SVHN	0.0605 [0.058, 0.063]	0.3587 ($p = 4.08 \times 10^{-20}$)	Low r, tolerable RMSE
COCO	0.1494 [0.138, 0.160]	0.8173 ($p < 10^{-4}$)	Strongest dataset correlation
Overall	**0.1449**	**0.7766**	Captures trends; uneven by dataset

Table 1. **Per-dataset accuracy prediction (baseline, Exp. 1).** RMSE reflects regression error for final accuracy; Pearson r measures ranking quality.

Metric	Value
RMSE	0.2567 [0.248–0.265]
MAE	0.1709
R^2	0.2885
Pearson correlation	0.6409 ($p = 1.25 \times 10^{-204}$)
Within 5% tolerance	12.30%
Within 10% tolerance	45.01%
Samples	1764

Table 2. **Experiment 2 (balanced + regularized) – global metrics.** Aggregate performance of the alternative predictor across all datasets; RMSE shows 95% CI.

performance and to propose an early-stop epoch $T' = \widehat{t}_*$ for low-potential runs.

An experiment 2 applies inverse-frequency balancing and stronger regularization (higher dropout/weight decay, fewer epochs). It underperforms: RMSE rises to 0.2567, R^2 drops to 0.2885, r to 0.6409, and only 12.3%/45.0% of predictions fall within 5%/10% tolerance (Tab. 2; also compared side-by-side in Tab. 8). We therefore adopt the baseline predictor in the full NNGPT loop.

The baseline is already useful for early-stop and priority scheduling, but (i) correlation varies across datasets; (ii) aggressive balancing/regularization can suppress adaptation in LoRA-tuned LLMs; and (iii) adding uncertainty estimates is a promising next step for safer automated stopping.

Dataset	Baseline avg	RL avg (n)	Δ (avg)	Baseline best	RL best
CIFAR-100	0.0461	0.0163 (5)	-0.0298	0.2079	0.0253
MNIST	0.7088	0.9876 (13)	0.2788	0.9906	0.9921
SVHN	0.2526	0.8148 (14)	0.5622	0.9032	0.9068

Table 3. **RL one-epoch results vs LEMUR baselines.** Baseline avg and best come from LEMUR *AlexNet (ast-dimension)*; RL avg is over generated *AirNet* models (n shown). Δ is RL avg minus baseline avg.

5.2. Reinforcement Learning Setup

We report our first RL results by generating 32 executable AirNet models and training each for a single epoch on MNIST [32], SVHN [48], and CIFAR-100 [30]. As a non-RL baseline we use one-epoch averages and best scores from the LEMUR AlexNet [31] (ast-dimension) family; the baseline values are embedded in Tab. 3. RL-generated models match or exceed the baseline on the simpler datasets: on MNIST the average and best accuracies are 0.9876 and 0.9921, and on SVHN they are 0.8148 and 0.9068. Performance on CIFAR-100 is weak with an average of 0.0163 and a best of 0.0253, while the LEMUR best reaches 0.2079. These initial results indicate that the RL loop reliably produces runnable architectures and can achieve strong one-epoch accuracy on easier datasets; harder regimes appear to require more iterations, longer training, and reward shaping. We treat this study as a diagnostic feasibility evaluation due to small per-dataset sample sizes and single-epoch budgets.