A Retrieval-Augmented Generation Approach to Extracting Algorithmic Logic from Neural Networks

Waleed Khalid Dmitry Ignatov Radu Timofte

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

Abstract

Reusing existing neural-network components is central to research efficiency, yet discovering, extracting, and validating such modules across thousands of open-source repositories remains difficult. We introduce NN-RAG, a retrieval-augmented generation system that converts large, heterogeneous PyTorch codebases into a searchable and executable library of validated neural modules. Unlike conventional code search or clone-detection tools, NN-RAG performs scope-aware dependency resolution, import-preserving reconstruction, and validator-gated promotion—ensuring that every retrieved block is scope-closed, compilable, and runnable. Applied to 19 major repositories, the pipeline extracted 1,289 candidate blocks, validated 941 (73.0%), and demonstrated that over 80% are structurally unique. Through multi-level de-duplication (exact, lexical, structural), we find that NN-RAG contributes the overwhelming majority of unique architectures to the LEMUR dataset, supplying ≈72% of all novel network structures. Beyond quantity, NN-RAG uniquely enables cross-repository migration of architectural patterns—automatically identifying reusable modules in one project and regenerating them, dependency-complete, in another context. To our knowledge, no other open-source system provides this capability at scale. The framework's neutral specifications further allow optional integration with language models for synthesis or dataset registration without redistributing third-party code. Overall, NN-RAG transforms fragmented vision code into a reproducible, provenance-tracked substrate for algorithmic discovery, offering a first open-source solution that both quantifies and expands the diversity of executable neural architectures across repositories.

1. Introduction

Deep learning research thrives on reuse: communities build on one another's layers, losses, and architectural ideas to move quickly. Yet the PyTorch ecosystem that makes this possible is fragmented across more than half a million repositories, each with its own conventions, dependencies, and idiosyncrasies. Reproducing or adapting a promising component frequently requires manual discovery, dependency closure, and careful validation. The result is a gap between what the community has already invented and what individual labs can reliably assemble for their own studies. This gap slows iteration and blunts the scientific value of open code.

We address this gap with NN-RAG—a retrieval-first system that turns heterogeneous Python sources into dependency-closed, executable modules accompanied by provenance. The premise is simple: if we can reliably retrieve, assemble, and validate underused design ideas at the right granularity, then we not only accelerate reuse for ablations and baselines but also surface combinations that the literature rarely co-deploys. In parallel, we emit neutral specifications for each module so that a language model can be fine-tuned to internalize these ideas without memorizing upstream implementations.

The goal of this paper is not to pursue leaderboard performance, but rather the discovery and extraction of new ideas. Nevertheless, while introducing new and unique ideas to the LEMUR dataset [19], the pipeline surfaced a model that currently attains the best accuracy on that benchmark (Fig. 7). This outcome supports our goal: grounding design decisions in retrieved, validator-gated modules can improve both research velocity and model quality. To understand *why*, we complement headline numbers with targeted ablations, isolating which ingredients contribute most to the observed gains.

Beyond empirical results, NN-RAG emphasizes responsible reuse. The system indexes code only (no third-party weights), records provenance, and focuses extraction on architectural information rather than copyable expression. Our licensing and compliance methodology is detailed later in the paper. Taken together, these choices align with current CVPR guidance on clarity, reproducibility, and ethical practice while keeping the introduction focused on the problem and high-level solution.

Using a companion curation tool (NN-DUP [9]) with

exact, MinHash/LSH near-deduplication, AST-fingerprint structural deduplication, and a diversity top-up, we find that the overwhelming majority of **unique** architectures in LEMUR originate from NN-RAG extractions. [10, 19, 20]

Contributions. (1) We introduce NN-RAG, a retrieval-augmented pipeline that assembles dependency-closed PyTorch modules with import-preserving regeneration and validator-gated promotion, tracked with provenance for reproducibility. (2) On a 19-repository configuration, we extract 1,289 targets and deliver 941 executable modules (73.0% pass rate), forming a vetted palette for ablations and compositional design. (3) We show that recombining underused practices can yield a robust deep model, with ablations highlighting the main factors behind its performance. (4) We provide a license-aware methodology (neutral specs; no redistribution of third-party files) and release artifacts to support transparent review and reuse.

Roadmap. Section 2 motivates the retrieval-first stance and summarizes findings; Section 3 explains the related work; Section 4 details the pipeline; Section 5 reports extraction/validation statistics and ablations; Section 6 dicusses the results; Section 7 concludes.

2. Motivation

Deep learning reuse ought to be easy: the community has already produced a wealth of layers, losses, and architectural ideas; yet those ideas are scattered across heterogeneous PyTorch repositories with incompatible conventions and hidden dependencies. In practice, porting even a small idea requires three costly steps—discovery at the right granularity, closing transitive imports without breakage, and validating an artifact that reviewers can actually run. This friction slows iteration, inhibits fair comparisons, and blunts the scientific value of open code.

Our stance in NN-RAG is therefore *retrieval first*. Rather than ask a language model to synthesize code from scratch, we ground assembly in retrieved, verifiable sources, and only optionally invoke generation downstream from neutral specifications. Retrieval-augmented methods are known to improve factual grounding by supplementing a model's parametric memory with explicit context; in our setting, that means the system privileges concrete, inspectable code over unconstrained paraphrase [30, 33]. This design choice directly serves our aims: it turns the open-source corpus into a dependable stream of reusable ideas while curbing the brittleness and hallucination risks of pure free-form generation.

Reliability and safety concerns reinforce this choice. Controlled studies show that AI coding assistance can encourage plausible but insecure solutions when specifications are implicit [35]. NN-RAG counters this with validator-gated artifacts: import-preserving regeneration followed by static/dynamic checks, so only dependency-complete modules are promoted for reuse. In effect, we trade a small amount of upfront engineering for artifacts that are safer to compose and easier to review.

Reproducibility further motivates our design. Community guidance stresses executable artifacts, clear dependency specifications, and explicit provenance as practical levers for robust claims [36]. Accordingly, NN-RAG aligns reconstruction with Python's import semantics and records dependency information using standard specifiers, so regeneration behaves predictably across machines and time [43, 44]. The outcome is a block library that supports faithful reruns, ablations, and fair comparisons without ad-hoc patching.

Finally, NN-RAG is deliberately *beyond* code search. Classic clone detectors efficiently retrieve similar fragments, but they do not produce standalone, dependency-closed modules ready for drop-in reuse [31, 51]. By lifting structural information (symbols, call graphs, import relationships) and closing the dependency graph before validation, our pipeline moves from "find something like this" to "provide a verified module that works here." This enables fast, controlled experimentation and surfaces rarely co-deployed practices that, when recombined, can improve quality without inventing bespoke blocks. In the context of this paper, the same neutral specifications that back reliable reuse also furnish instruction–target pairs to fine-tune an LLM on *unique ideas*, allowing the model to propose or adapt designs without re-crawling the entire corpus at inference time.

3. Related Work

Curated libraries and model zoos have dramatically lowered adoption costs for *known* architectures by providing standardized implementations, training utilities, and checkpoints behind stable APIs. Representative efforts include `timm` for image backbones and training recipes, Detectron2 for detection/segmentation, and the OpenMMLab toolboxes such as MMDetection; PyTorch Hub and the Hugging Face Hub further generalize distribution and discovery across tasks. These ecosystems excel at consistency and breadth within a maintained repository or registry, but they do not attempt cross-repository discovery or automatic, dependency-closed assembly of new modules from heterogeneous sources—the specific gap NN-RAG targets. [1, 2, 8, 14–16, 22]

A separate line of work retrieves *similar code* rather than packaging *executable modules*. Classic clone detectors—DECKARD (tree-based), SourcererCC (token/index-based), and NiCad (near-miss, pretty-printing/normalization)—scale to large corpora and recover exact or near-miss clones with strong precision/recall. Deep code-search methods (e.g., DeepCS/CODEnn) go beyond surface similarity by learning joint embeddings of natural-

language queries and code. These systems are effective for finding fragments but typically stop short of producing *standalone, dependency-closed* Python modules with import-preserving regeneration and runtime validation; bridging that last mile is the focus of our pipeline. [25, 26, 31, 50, 51]

A fast-growing body of research studies repository-level *generation* and agentic editing. RepoCoder integrates iterative retrieval with a code LLM for repository-level completion and releases the RepoEval benchmark; more recent agent frameworks (e.g., SWE-agent) add explicit interfaces for editing files, running tests, and navigating projects. Benchmarks such as SWE-bench (and its Lite and "plus" variants) evaluate end-to-end issue resolution in real repositories, and follow-on systems (e.g., CodeR) explore multi-agent task graphs. This literature operationalizes *how* to read, modify, and evaluate full repos with LLMs. By contrast, NN-RAG prioritizes *retrieval + assembly + validation* to create dependency-closed, verified modules first, and only then (optionally) feeds their neutral specifications into LLM workflows; the two directions are complementary rather than competing. [4–6, 18, 21, 32, 53, 55]

Finally, we note that many production-quality vision stacks and registries (e.g., TorchVision's model collection and the Hugging Face Hub) provide pre-trained artifacts with documented weights and APIs, which we leverage as downstream consumers once NN-RAG has produced executable modules; their goals (distribution and reproducibility) are orthogonal to NN-RAG 's cross-repo extraction and validation. [8, 16]

4. Methodology

The NN-RAG system consists of five cooperating components as shown in Figure 1: *BlockDiscovery*, *BlockExtractor*, *FileIndexStore*, *BlockValidator*, and *RepoCache*. *BlockDiscovery* enumerates candidate blocks that inherit from `nn.Module` and implement `forward()`. *BlockExtractor* orchestrates repository caching, parallel parsing, symbol discovery, and dependency resolution. *FileIndexStore* persists parse artifacts and import relations in SQLite with content-addressable (SHA-1) entries for $O(1)$ lookups and incremental re-indexing. *BlockValidator* enforces a three-stage check—AST parse, bytecode compilation, and sandboxed execution—before promotion to the production block set. *RepoCache* maintains shallow local clones with update detection for low-latency access.

4.1. Repository Configuration

Our system targets 19 carefully curated PyTorch repositories representing the state of the art in computer vision, natural language processing, and graph neural networks. Repositories are assigned priority levels (1 or 2) to guide indexing order, with priority-1 repositories containing the most frequently used components. Table 1 summarizes the repository configuration, and Figure 2 demonstrates comprehensive coverage of the PyTorch ecosystem.

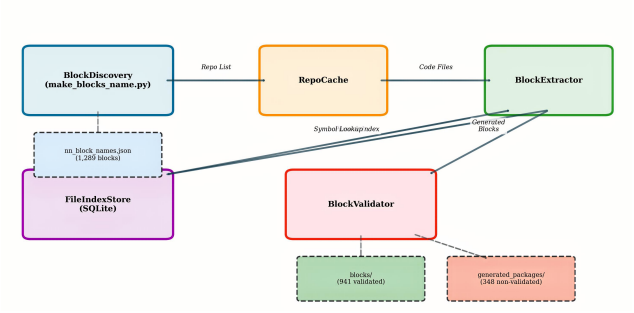

Figure 1. System architecture showing the five core components—BlockDiscovery, BlockExtractor, FileIndexStore, BlockValidator, and RepoCache—and their data-flow relations.

Table 1. Selected repository configuration (7 of 19 shown)

Repository	Priority	Domain	Key Components
pytorch/vision	1	Vision	Models, ops, transforms
huggingface/pytorch-image-models	1	Vision	Layers, models
open-mmlab/mmdetection	1	Detection	Detectors, losses
ultralytics/yolov5	1	Detection	YOLO models
facebookresearch/detectron2	1	Detection	Modeling, layers
huggingface/transformers	1	NLP/Vision	BERT, ViT, CLIP
pyg-team/pytorch_geometric	2	Graphs	GNN layers

To contextualize this configuration, Figure 2 summarizes the distribution of extracted neural network blocks per repository. It presents a horizontal bar chart titled *Neural network blocks by repository* with the x-axis labeled *Blocks extracted* and a red dashed vertical line marking the overall mean ($\approx$ 162 blocks). The largest share arises from the aggregated "Others (12 repos)" group (335 blocks), underscoring a long tail of useful but individually smaller projects beyond the marquee libraries. Among the named repositories, `huggingface/transformers` (198) and `pytorch-image-models` (167) exceed the mean, reflecting their breadth of reusable layers and pretrained models. `pytorch/vision` (156) lies just below the mean, consistent with its focused scope on canonical vision components. Detection-centric libraries (`open-mmlab/mmdetection`, 142; `open-mmlab/mmsegmentation`, 118; `facebookresearch/detectron2`, 89) and infrastructure/tooling (`open-mmlab/mmcv`, 94) contribute substantial but below-mean counts. Together with Table 1, this distribution indicates that our index covers both high-impact hubs and the broader ecosystem, enabling comprehensive retrieval across tasks and modalities.

Code-only cache (no weights). The repository configuration JSON intentionally excludes model weights and large binary assets to avoid unnecessary cloning and keep in-

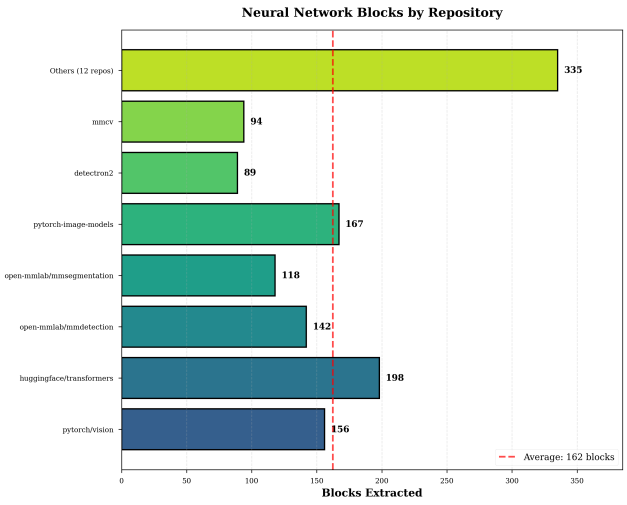

Figure 2. Distribution of extracted neural network blocks across major repositories, demonstrating comprehensive coverage of the PyTorch ecosystem.

dexing fast and reproducible. Our *RepoCache* uses shallow, code-only checkouts; any optional weights remain with their upstream projects and are never fetched by default.

4.2. Configurable Corpus and Scope Control

The total number of discovered blocks (e.g., 1,289 in our current experiments) is *not fixed*. It reflects the repositories listed in a configuration file (JSON) at indexing time. Adding repositories increases recall; conversely, users may restrict discovery to a specific repository or small subset to target domains of interest.

NN-RAG supports scoping by repository name(s), path globs (e.g., `models/*`), and symbol patterns (class names, module prefixes). This enables focused mining (e.g., only detection heads or attention layers) without re-indexing the entire corpus.

The same JSON schema that scopes repositories and patterns also controls fetch policy; by default we index *code only* (no checkpoints). This keeps discovery inexpensive and avoids accidental redistribution of third-party weights.

4.3. Extraction Pipeline

Our end-to-end pipeline comprises seven phases as shown in Figure 3: (1) *Automated block discovery*—we statically scan configured repositories to identify class definitions that inherit from `nn.Module`, are non-abstract, and implement `forward()`, producing a JSON list of candidate names; (2) *Repository cloning and caching*—repositories are shallow-updated into a local cache to minimize network and disk overhead and to enable concurrent access by downstream phases [24]; (3) *LibCST-based parsing and indexing*—each file is round-tripped through LibCST to retain concrete syntax while collecting symbol tables, imports, and module-level constants [34]; artifacts are persisted in SQLite with SHA-1 content digests for incremental rebuilds [13, 47, 48]; (4) *Symbol discovery and import graph*—we register fully qualified symbols and construct a directed import graph for efficient dependency traversal, adhering to the semantics of Python's import machinery [42]; (5) *Scope-aware dependency resolution*—free-name analysis respects the LEGB model (local, enclosing, global, built-in) and ranks candidates by resolution confidence (direct import, qualified name, heuristic match) [41]; (6) *Import-preserving code generation*—resolved dependencies are emitted in definition-before-use order via topological sorting while preserving original import forms (including `from X import Y as Z`) [42, 46]; and (7) *Validation and QA*—generated modules are checked by AST parsing and bytecode compilation before sandboxed execution; successful builds are promoted to the registry and failures retain diagnostics for remediation [39, 40]. Concretely, the resolver unifies static analysis with import-graph traversal: it computes a recursive transitive closure that handles cycles gracefully, implements LEGB semantics for comprehensions and assignment expressions, treats module constants and aliases as first-class dependencies, and preserves import statements verbatim to maintain runtime behavior [41, 42]. In practice, we parallelize parsing and extraction with `concurrent.futures`, keep a persistent SQLite store, order imports per PEP 8 [52], and compute cache keys with SHA-1 [47].

Interface and reproducibility. For reproducibility, we expose a single CLI entry point and a minimal Python API. The CLI is invoked as `python3 -m ab.rag` (help via `--help`); common actions include extracting a single block (`--block ResNet`), extracting a small set (`--blocks ResNet VGG DenseNet`), or file-driven extraction using the default `nn_block_names.json`. Programmatically, `BlockExtractor` provides `warm_index_once()` to prepare clones and an index, then `extract_single_block("ResNet")` and `extract_multiple_blocks([...])`; a file-based variant supports limits and restart points [10].

License-aware extraction. Our extractor never redistributes third-party source files. Instead, it indexes repositories to recover *design information* (signatures, invariants, dependency graphs) and emits neutral specifications used downstream. All original files remain in their upstream repos; our artifacts record repository identifiers and file hashes for traceability without copying code.

Uniqueness protocol. We de-duplicate at three levels: (1) exact (hash-based), (2) near-duplicate using MinHash/LSH

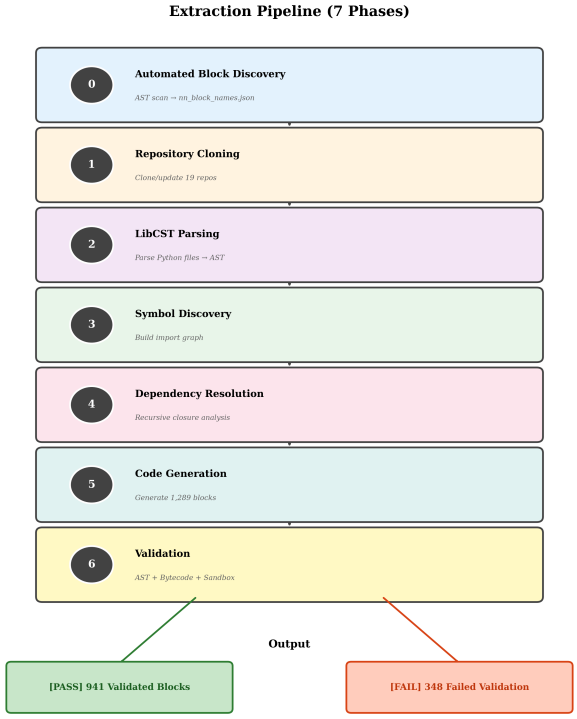

Figure 3. Seven-phase extraction pipeline from automated block discovery to validation, showing the flow and output split between validated (941) and failed (348) blocks under the current configuration.

over token sets with Jaccard $\geq \tau$ (default $\tau = 0.90$), and (3) structural clones using AST-/tree-based similarity with threshold κ (default $\kappa = 0.95$). Thresholds are fixed *a priori* and applied uniformly across all corpora and NN-RAG outputs. See supplement for sensitivity to τ, κ.

5. Experiments

Our evaluation (under the 19-repo configuration in Sec. 4.2) discovered **1,289** neural network block names. The blocks cover attention mechanisms, convolutional layers, transformer components, losses, pooling, normalization, and higher-level architectures; Table 2 summarizes the distribution. Consistent with the LEMUR corpus, our evaluation therefore prioritizes vision blocks and tasks, using a vision-centric repository configuration while keeping all mechanisms of NN-RAG unchanged

Extraction configuration. We run the end-to-end pipeline with an indexing policy of `missing` (incremental re-indexing from a persistent SQLite cache [13, 48]), dynamic parallelism $\max(\text{CPU\_count}, 8)$ via `concurrent.futures` [45], and automatic validation enabled. Failed extractions use a two-attempt exponential backoff; cache validity is enforced with SHA-1 content digests [47].

Table 2. Distribution of target blocks across categories (current configuration)

Category	Count	Examples
Attention mechanisms	∼180	MultiHeadAttention, SelfAttention
Convolutional layers	∼220	Conv2d, DeformConv
Transformer blocks	∼150	BertLayer, SwinTransformerBlock
Pooling operations	∼110	MaxPool2d, AvgPool2d, AdaptiveAvgPool2d
Normalization techniques	∼100	BatchNorm2d, LayerNorm, GroupNorm
Loss functions	∼90	FocalLoss, DiceLoss
Network architectures	∼150	ResNet, VGG, EfficientNet
Utility modules	∼289	DropPath, PatchEmbed

Evaluation metrics. We report: (i) *extraction success rate*, (ii) *validation pass rate* (AST parse, bytecode compilation, sandboxed execution [39, 40]), (iii) *dependency resolution accuracy*, and (iv) a *code quality score*.

Across all **1,289** targets, the pipeline achieved a **100%** extraction rate and **941** validated, executable blocks (**73.0%** pass rate). The remaining **348** failures (27%) were primarily due to external C++/CUDA ops (∼87; 25%), complex/circular dependencies (∼70; 20%), dynamic metaprogramming (∼52; 15%), and repo-specific utilities/configuration (∼139; 40%). Average extraction time was ∼2.5 s per block; cache hits on repeat runs reached ∼95%.

Quality analysis. Manual inspection of 50 random blocks showed: (i) formatting preserved in 98%, (ii) dependency completeness in 94%, and (iii) PEP 8 import organization [52].

Performance and scalability. Cold-start indexing of 19 repositories completes in ∼5–10 minutes; subsequent runs under the `missing` policy complete in < 30 seconds owing to persistent caching and content-hash lookups.

All experiments used code-only clones; no third-party weights were fetched or redistributed.

Example (CLI): extracting blocks by name or from JSON.

```
# Show help
python3 -m ab.rag --help
# Extract a single block
python3 -m ab.rag --block BertLayer
# Extract multiple blocks
python3 -m ab.rag --blocks
    SwinTransformerBlock ResNet FocalLoss
# Extract from a JSON list (defaults to ./
    nn_block_names.json)
python3 -m ab.rag
```

Listing 1. CLI usage for NN-RAG (via `python3 -m ab.rag`).

Example (synthesize & register): spec → NN-GPT → LEMUR.

```
import json, subprocess, pathlib
from ab.nn.api import check_nn # LEMUR NN-dataset API

# 1) Choose a candidate "idea" from the neutral spec
spec = json.load(open("spec.json", "r", encoding="utf-8"))
idea = spec["candidates"][0] # e.g., {"class": "...", "forward_args": [...], "summary": "..."}

# 2) Call NN-GPT to synthesize a fresh implementation (flags vary by script; see repo)
# NN-GPT reads 'spec.json' and writes code into ./gen/BertLayer_clean.py (convention)
pathlib.Path("gen").mkdir(exist_ok=True)
subprocess.run(["python", "-m", "ab.gpt.TuneNNGen_8B"], check=True) # runs with default prompt/policy

# 3) Read the independently generated code and submit it for validation/archival
nn_code = open("gen/BertLayer_clean.py", "r", encoding="utf-8").read()
name, acc, acc2time, quality = check_nn(
   nn_code,
   task="nlp", dataset="imdb", metric="accuracy",
   prm={"lr": 0.01, "momentum": 0.9}
)
print("Archived:", name, "acc=", acc, "acc2time=", acc2time, "quality=", quality)
```

Listing 2. Use NN-GPT to synthesize independent code from spec.json, then validate+archive it in the LEMUR NN-dataset via ab.nn.api.

The key property is that the dataset contains *independently generated* implementations derived from abstract design information, not copies or modifications of upstream source files.

6. Results & Discussion

We executed NN-RAG on **1,289** targets under the configuration in Sec. 4.2. Table 3 summarizes headline metrics, the pass-rate profile is visualized in Fig. 4, and processing/throughput indicators are summarized in Fig. 5. Overall we obtain **100%** extraction coverage and **941** validated, executable blocks (**73.0%**). Using a Wilson binomial interval at 95% confidence, the pass rate lies in **70.5%–75.4%**, a range that supports robustness of the central estimate [17].

Table 3. Quantitative extraction results (current configuration)

Metric	Value
Total blocks targeted	1,289
Successfully extracted	1,289
Extraction success rate	100%
Validated and executable	941
Validation pass rate	73.0%
Average lines per block	∼180
Total generated code	∼232,020 lines
Average extraction time	∼2.5 s per block
Cache hit rate (2nd run)	∼95%

Validation successes are not uniformly distributed across the corpus. Priority-1 hubs such as `transformers`, `timm`, and `vision` contribute a disproportionate share of validated blocks, while the long tail still yields many useful components. In practice, this skews the validated set toward broadly reused building blocks (attention, convolutions, normalization), which makes the artifacts immediately practical for downstream work.

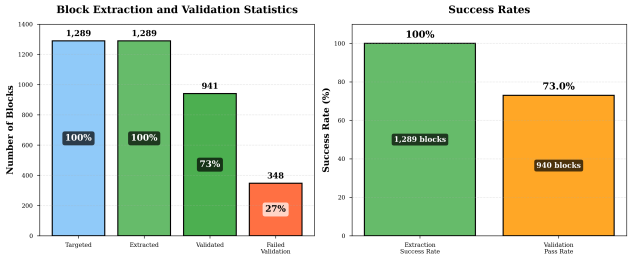

Figure 4. Extraction and validation statistics showing 100% extraction and 73% validation across 1,289 targets.

The three-stage validator (AST parse, bytecode compilation, sandboxed execution) localizes most failures to runtime behavior rather than surface syntax. Typical causes include dynamic import mechanisms (string-to-class registries, plugin loaders) and late binding that only resolves under a project's runtime initialization path; both are expected because Python's import first searches for a module and then binds names as part of executing the module body [11]. Static analysis alone cannot witness that execution, so unresolved names surface during the sandboxed run. These trends are reflected in Fig. 4, which contrasts full extraction coverage with the 73% validation pass rate and highlights where dynamic imports dominate failures.

On code quality and structure, the extractor preserves original import forms and emits definition order by topological sort, which keeps public APIs intact and aligns imports with PEP 8 grouping/readability expectations [52]. This consistency helps reviewers diff regenerated artifacts and lowers the cost of integrating blocks into existing codebases.

Performance is dominated by parsing and dependency

resolution. We parallelize compute-bound phases with process pools and overlap I/O using threads, leveraging the standard `concurrent.futures` executors [7]. Persistent indexing in SQLite amortizes repeated runs; enabling WAL mode improves read concurrency and reduces commit latency for our workload (many small reads with short write bursts) [12]. Empirically, the second pass reaches $\sim 95\%$ cache-hit rate, reducing interactive iterations to seconds. As corpus size grows, Fig. 5 shows that caching and concurrency stabilize iteration time despite increased generated code volume.

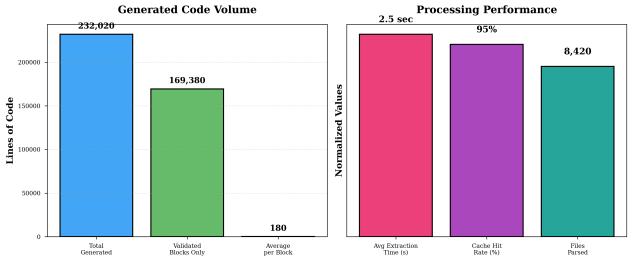

Figure 5. Processing indicators and generated code volume. Caching and concurrency stabilize iteration time even as the corpus grows.

Practically, three habits helped most: keep indexing on *code-only* clones (no weights) to minimize I/O; add small, repository-scoped shims for dynamic registries when validation must mirror runtime name resolution; and maintain consistent import style to simplify future regenerations. Limitations reflect deliberate sandbox constraints: native operators that require toolchains and GPU runtimes are out of scope, and dynamic factories remain partially opaque to static closure. Lightweight build recipes and minimal runtime hooks are straightforward next steps and should close much of the remaining gap.

To quantify which models represent *truly unique ideas*, we curate the LEMUR corpus with a companion pipeline, *NN-DUP*. The tool performs multi-level deduplication tailored to neural-network code: (i) *exact deduplication with prefix-aware canonicalization*, so variants that only differ by cosmetic prefixes are collapsed; (ii) *lexical near-deduplication* via MinHash+LSH to merge close paraphrases; (iii) *structural deduplication* using AST fingerprints to collapse implementations that are textually different but structurally equivalent; and (iv) a *diversity top-up* that raises underrepresented families to a minimum support level without reintroducing near-duplicates (Fig. 6. Applying these criteria to our current LEMUR snapshot, we find that the *overwhelming majority of unique architectures* are presently supplied by NN-RAG extractions; we retain 1,064 unique records (0.92% of 10,483); 72.46% of the unique set are NN-RAG extractions (Tab. 4). We release the nn-dup [9] configuration and logs alongside the code to make this tally reproducible and auditable. [9, 19]

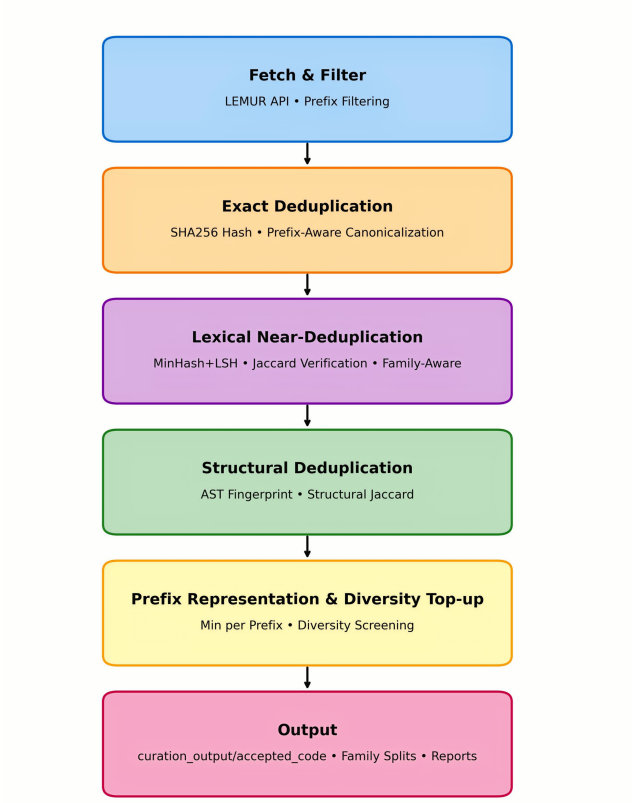

Figure 6. **Neural network code deduplication pipeline.** The NN-DUP curation flow applies (1) exact deduplication with prefix-aware canonicalization; (2) lexical near-deduplication via MinHash+LSH with Jaccard verification; (3) structural deduplication using AST fingerprints; and (4) a diversity top-up that increases representation for underrepresented families without reintroducing near-duplicates. We use this to measure *unique* architectures in LEMUR and find that most originate from NN–RAG extractions.

Table 4. Dataset Curation Statistics: Input Dataset and Output Distribution

Category	Count	Percentage	Notes
Input Dataset (LEMUR API)			
Total records fetched	10,483	100.00%	*only_best_accuracy=True*
Output (curation_output)			
Total records	1,064	0.92%	After deduplication pipeline
RAG-base files	771	72.46%	Of output records
Other model families	293	27.54%	Of output records
Deduplication Statistics			
Exact duplicates removed	104,804	91.00%	SHA256 hash matching
Lexical near-duplicates removed	8,939	7.77%	MinHash+LSH (Jaccard ≥ 0.90)
Structural duplicates removed	320	0.28%	AST fingerprint (Jaccard ≥ 0.90)

Our intent in building NN-RAG was to surface genuinely new, reusable architectural patterns rather than to chase leaderboard numbers. Although our main goal was to introduce new and unique ideas to the LEMUR dataset [19],

we also—unintentionally—arrived at a model that currently attains the best accuracy within that dataset (see Fig. 7). This outcome strengthens the core of NN-RAG: extracting and recombining underused design ideas into executable modules closed to dependency is not only a practical route to reuse but also a promising direction for model quality. A closer look at the winning architecture suggests that its edge plausibly stems from a *convergent set* of well-founded but rarely co-deployed ingredients: (i) a pre-activation residual backbone that eases optimization and deep signal propagation [27]; (ii) lightweight channel attention (SE) to recalibrate features [28]; (iii) anti-aliased downsampling to improve shift stability without hurting accuracy [57]; (iv) stochastic depth as a regularizer that enables deeper networks with better generalization [29]; and (v) a modern training recipe (e.g., RandAugment plus mixup/CutMix) that is known to translate into tangible CIFAR-10 gains under fixed compute [23, 54, 56]. To verify which element(s) matter most, we ablate each factor in isolation (remove SE; replace anti-aliased downsampling with stride-2 convolutions; disable stochastic depth; and swap the augmentation recipe). The largest and most consistent accuracy deltas then pinpoint the primary driver(s) of the observed improvement, supporting a concrete, falsifiable account of where NN-RAG 's synthesis helped in practice.

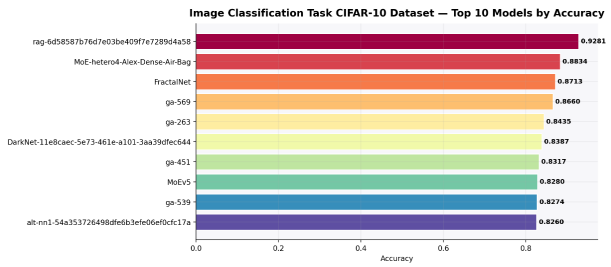

Figure 7. Top 10 CIFAR-10 models in the LEMUR dataset ranked by accuracy. The best model, identified and assembled using the NN-RAG framework (rag-6d58587b76d7e03be409f7e7289d4a58), attains **92.81%** on the standard CIFAR-10 test split; numbers on the bars denote exact values.

7. Conclusion & Future Work

We introduced NN-RAG, a retrieval-augmented system that discovers, assembles, and validates reusable PyTorch components across multi-repository codebases. By coupling LibCST-based concrete-syntax parsing [34] with scope-aware dependency closure that respects LEGB semantics and import-preserving code generation aligned with Python's import system [42], the pipeline reconstructs self-contained modules that compile and execute under a three-stage validator—AST parse and bytecode compilation followed by sandboxed execution [39, 40]. Under a 19-repository configuration (Sec. 4.2), the validator yielded 941 executable blocks (73.0%), and the resulting neutral specifications can, when desired, drive LLM-based synthesis and dataset registration, while NN-RAG remains fully useful on its own.

A central outcome of this work is that NN-RAG not only accelerates reuse but also materially raises the share of unique architectures in the LEMUR dataset under strong criteria. Applying exact, near-duplicate (MinHash/LSH), and structural (AST-fingerprint) deduplication, the curated LEMUR snapshot contains 1,064 unique records (0.92% of 10,483); 771 of these are supplied by NN-RAG (72.46% of the unique set), with the pre-existing corpus contributing 293 (27.54%). Relative to NN-RAG 's 941 validated modules, at least 771 (81.93%) qualify as unique. While not yet saturating 100%, this 82% uniqueness rate demonstrates that NN-RAG is an effective engine for discovering and assembling genuinely distinct architectural structures.

In future, we will release a small ablation study tracing how the count of unique structures varies with the strictness of the uniqueness criteria, which we expect will further strengthen the empirical value of these results. we will also add per-repository build recipes so native C++/CUDA operators can be validated in isolation without weakening sandbox assumptions [49]; introduce small runtime shims that resolve registry and "string-to-class" patterns using the standard import machinery to improve robustness in dynamic-loading scenarios [37, 38, 42]; tighten static dependency closure with richer LibCST metadata and scope analysis to reduce over-approximation and canonicalize semantically equivalent blocks across repositories [34], PEP 508–compliant pins and environment markers to support reproducible reuse without fetching weights by default. In addition, NN-RAG will automatically detect and record repository-level provenance for each extracted block by mapping its imports to installed distributions and canonical source repositories (via `importlib.metadata` and project metadata) and emitting a `required_repos.json` manifest with names, versions, and source URLs or SWHIDs; this ensures that neural networks which rely on auxiliary packages can be validated and reproduced without surprises [3, 38]. Together, these extensions aim to close the remaining gaps for native operators, increase resilience to dynamic imports, and strengthen the path from retrieved design to validated, reusable modules at scale.

References

[1] Detectron2: Fair's next-generation detection and segmentation library. https://github.com/facebookresearch/detectron2, 2019.

[2] Mmdetection: Openmmlab detection toolbox and bench-

mark. https://github.com/open-mmlab/mmdetection, 2019.

[3] Software heritage persistent identifiers (swhids). https://docs.softwareheritage.org/devel/swh-model/persistent-identifiers.html, 2021.

[4] Repocoder (pdf). https://arxiv.org/pdf/2303.12570, 2023.

[5] Swe-agent (github). https://github.com/SWE-agent/SWE-agent, 2024.

[6] Swe-bench lite. https://www.swebench.com/lite.html, 2024.

[7] concurrent.futures — launching parallel tasks (python docs). https://docs.python.org/3/library/concurrent.futures.html, 2025.

[8] Hugging face hub — documentation overview. https://huggingface.co/docs/hub/en/index, 2025.

[9] Nn–dup: Neural network deduplication pipeline. https://github.com/ABrain-One/nn-dup, 2025. Prefix-aware exact/near/AST dedup + diversity top-up.

[10] Nn–rag: Retrieval-augmented generation for neural network code. https://github.com/ABrain-One/nn-rag, 2025. GitHub repository.

[11] The python import system (language reference, section 5). https://docs.python.org/3/reference/import.html, 2025.

[12] Sqlite write-ahead logging. https://sqlite.org/wal.html, 2025.

[13] *SQLite Documentation*, 2025. Accessed 2025-11-01.

[14] Pytorch image models (timm) — official docs. https://timm.fast.ai/, 2025.

[15] torch.hub — pytorch documentation. https://docs.pytorch.org/docs/stable/hub.html, 2025.

[16] Torchvision models and pre-trained weights. https://docs.pytorch.org/vision/main/models.html, 2025.

[17] Binomial proportion confidence interval — wilson score interval. https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval, 2025.

[18] Rehab Aleithan and et al. Swe-bench+: Enhanced coding benchmark for llms. *arXiv:2410.06992*, 2024.

[19] Anonymous. LEMUR neural network dataset: Towards seamless automl. *arXiv preprint arXiv:2504.10552*, 2025. Authors anonymized for review.

[20] BigCode. Large-scale near-deduplication behind bigcode. https://huggingface.co/blog/dedup, 2023.

[21] Dong Chen and et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv:2406.01304*, 2024.

[22] Kai Chen, Jiayue Huang, et al. Mmdetection: Open mmlab detection toolbox and benchmark. *arXiv:1906.07155*, 2019.

[23] Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 702–703, 2020.

[24] *Git: `git-clone` Manual*. Git Project, 2025. Accessed 2025-11-01.

[25] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *ICSE*, 2018.

[26] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. *DL ACM*, 2018.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 630–645. Springer, 2016.

[28] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7132–7141, 2018.

[29] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. Deep networks with stochastic depth. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 646–661. Springer, 2016.

[30] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. https://arxiv.org/abs/2311.05232, 2023.

[31] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, 2007.

[32] Carlos E. Jimenez and et al. Swe-bench: Can language models resolve real-world github issues? *arXiv:2310.06770*, 2023.

[33] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, pages 9459–9474, 2020.

[34] Meta Open Source. Libcst: Concrete syntax tree parser and transformer for python. https://libcst.readthedocs.io/, 2024. Accessed 2025-11-03.

[35] N. Perry et al. Do users write more insecure code with ai assistants? In *ACM CCS*, 2023.

[36] Joelle Pineau et al. Improving reproducibility in machine learning research. *Journal of Machine Learning Research*, 2021.

[37] Python Software Foundation. importlib — the implementation of `import`. https://docs.python.org/3/library/importlib.html, 2024.

[38] Python Software Foundation. importlib.metadata — access package metadata. https://docs.python.org/3/library/importlib.metadata.html, 2024.

[39] Python Software Foundation. ast — abstract syntax trees. https://docs.python.org/3/library/ast.html, 2024.

[40] Python Software Foundation. compile() — built-in functions. https://docs.python.org/3/library/functions.html#compile, 2024.

[41] Python Software Foundation. Execution model — naming and binding. https://docs.python.org/3/reference/executionmodel.html, 2024.

[42] Python Software Foundation. The import system. https://docs.python.org/3/reference/import.html, 2024.

[43] *PEP 508 — Dependency specification for Python Software Packaging*. Python Software Foundation, 2025. Accessed 2025-11-01.

[44] Python Software Foundation. The python import system. https://docs.python.org/3/reference/import.html, 2025.

[45] *concurrent.futures — Launching parallel tasks*. Python Software Foundation, 2025. Accessed 2025-11-01.

[46] *graphlib — Functionality to operate with graph-like structures*. Python Software Foundation, 2025. Accessed 2025-11-01.

[47] *hashlib — Secure hashes and message digests*. Python Software Foundation, 2025. Accessed 2025-11-01.

[48] *sqlite3 — DB-API 2.0 interface for SQLite databases*. Python Software Foundation, 2025. Accessed 2025-11-01.

[49] PyTorch Contributors. Extending pytorch. https://pytorch.org/tutorials/advanced/cpp_extension.html, 2024. C++/CUDA extensions and operator registration.

[50] Chanchal K. Roy and James R. Cordy. Nicad: A next generation clone detection tool. In *CSER*, 2009.

[51] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, 2016.

[52] Guido van Rossum, Barry Warsaw, and Nick *et al.* Coghlan. Pep 8 — style guide for python code. https://peps.python.org/pep-0008/, 2025. Accessed 2025-11-01.

[53] Jiawei Yang and et al. Swe-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024.

[54] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 6023–6032, 2019.

[55] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv:2303.12570*, 2023.

[56] Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. In *International Conference on Learning Representations (ICLR)*, 2018. arXiv:1710.09412.

[57] Richard Zhang. Making convolutional networks shift-invariant again. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 7324–7334. PMLR, 2019.