MACE: Modular Adaptive Code Engine with Context-Aware LoRA Expert Routing and Hierarchical Retrieval-Augmented Generation*

Serhii Dolhopolov[1,†] and Tetyana Honcharenko[1*,†]

[1] Kyiv National University of Construction and Architecture, 31, Air Force Avenue, Kyiv, 03037, Ukraine

Abstract

Recent advances in large language models (LLMs) for code generation have achieved impressive results on standard benchmarks, yet current systems lack mechanisms for dynamic adaptation to diverse programming tasks, languages, and project contexts. This paper presents MACE (Modular Adaptive Code Engine), a framework that addresses these limitations through three novel contributions: (1) a context-aware LoRA expert routing mechanism that dynamically selects and blends lightweight domain-specialized adapters based on semantic analysis of the input task; (2) a hierarchical three-level retrieval-augmented generation (RAG) system that provides global, project, and local code context with level-specific retrieval strategies; and (3) a multi-signal verification loop that iteratively refines generated code using execution feedback, static security analysis, and style checking. MACE is evaluated on an extended curated benchmark of 25 coding problems across three difficulty levels and additionally on a 50-problem subset of the standard HumanEval benchmark, comparing a base Qwen2.5-Coder-7B model against its low-rank adaptation (LoRA)-enhanced variant. The experimental results demonstrate that the LoRA expert achieves a 35% average speedup in code generation (up to 43% on hard problems) while simultaneously improving correctness from 84% to 96%, surpassing the base model by 12 percentage points. The Python-specialized expert was trained on a curated dataset of 12,450 examples over 3 epochs using a single consumer-grade GPU. The MACE framework is implemented as a Python package with CLI, REST API, and IDE integration interfaces, making it accessible to researchers, developers, and non-programmers alike.

Keywords

code generation, deep learning, LoRA adapters, mixture of experts, retrieval-augmented generation

1. Introduction

Modern code-specialized LLMs such as Qwen2.5-Coder [1] achieve pass rates exceeding 90% on standard benchmarks, and LLM-based code generation has evolved to encompass code completion, bug repair, and full-program synthesis [5, 6]. However, persistent shortcomings remain: models apply identical parameters regardless of language or domain; they lack mechanisms for incorporating project context (repository structure, abstract syntax tree (AST), application programming interface (API) contracts); and generated code exhibits quality deficiencies in readability, maintainability, and security [8]. Parameter-efficient methods such as Low-Rank Adaptation (LoRA) [2] and its quantized variant QLoRA [3], retrieval-augmented generation (RAG) techniques [18, 20], and iterative refinement [9, 22] address these challenges individually, but no existing system integrates expert specialization, hierarchical retrieval, and multi-signal verification into a unified framework.

* CMIS-2026: Ninth International Workshop on Computer Modeling and Intelligent Systems, May 5, 2026, Zaporizhzhia, Ukraine

1* Corresponding author.
† These authors contributed equally.
dolhopolov@icloud.com (A. Ometov); goncharenko.ta@knuba.edu.ua (T. P. Sales); manfred.jeusfeld@acm.org (M. Jeusfeld)
0000-0003-3412-1639 (A. Ometov); 0000-0002-5385-5761 (T. P. Sales); 0000-0002-9421-8566 (M. Jeusfeld)

Copyright © 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-4220/paper17.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-07-04

The research problem addressed by this work is the absence of a unified mechanism that simultaneously enables task-conditional specialization, project- and file-level context grounding, and post-generation correctness, security, and style verification on consumer-grade hardware. The hypothesis is that combining context-aware LoRA expert routing, hierarchical RAG, and a multi-signal verification loop yields measurable gains in both correctness and generation latency relative to a strong base model; the scientific novelty lies in the unified architecture and the closed-form weighted-merging mechanism for dynamic LoRA composition, the practical novelty in a reference implementation reproducible on a single consumer GPU.

This paper presents MACE (Modular Adaptive Code Engine), synthesising three contributions: (1) context-aware LoRA expert routing – a learned router that selects and blends domain-specialized LoRA adapters based on semantic task analysis (language, type, complexity, domain), combining parameter efficiency with mixture-of-experts (MoE) routing [17]; (2) hierarchical three-level RAG retrieving context at global (documentation), project (AST-derived), and local (file-level) granularities; and (3) a multi-signal verification loop combining execution testing, static security analysis, and style checking with iterative refinement [8, 22].

MACE is evaluated on a curated 25-problem benchmark across five categories and three difficulty levels and on a 50-problem HumanEval subset, using Qwen2.5-Coder-7B [1] with QLoRA [3] on a curated dataset [4]. A LoRA expert trained on 12,450 examples over 3 epochs achieves 35% average speedup (43% on hard problems) while improving pass rate from 84% to 96%. The framework is released as a Python package with command-line interface (CLI) and REST API.

2. Related Work

Usman et al. [5] surveyed LLM-based code generation, identifying persistent challenges in security, domain adaptation, and multi-file consistency. Bayram et al. [7] revealed substantial performance variation across problem categories on HumanEval, informing the MACE benchmark design. Tosi [8] evaluated code quality across correctness, readability, and complexity dimensions, aligning with the multi-signal verification approach proposed herein. Antero et al. [10] showed that combining generation with automated verification improves reliability, while Jost et al. [6] found that developers must maintain domain knowledge to verify AI outputs. Han et al. [11] reviewed LLM architectures including MoE strategies, and Idrisov and Schlippe [9] motivated language-specific adaptation.

LoRA [2] enables adaptation with fewer than 1% of parameters; QLoRA [3] extends this with 4-bit quantization for consumer GPUs. Zhang et al. [12] and Kim et al. [13] further reduced fine-tuning costs through gradient estimation and pruning-based approaches. Prottasha et al. [14] showed that semantic knowledge integration improves adaptation quality, while Nwaiwu [15] compared PEFT methods, establishing practical guidelines. Franzoni et al. [16] demonstrated domain-specific adaptation benefits for code generation. Baniata and Kang [17] explored MoE within transformers, demonstrating dynamic routing for efficient scaling – a principle the MACE adaptive router extends to LoRA composition.

Brown et al. [18] established a RAG taxonomy identifying challenges that the proposed hierarchical approach addresses. Iaroshev et al. [19] showed domain-specific retrieval outperforms general-purpose approaches. Choi et al. [20] demonstrated RAG–LoRA synergies, and Feng et al. [21] applied RAG with chain-of-thought for code generation.

Dolcetti and Iotti [22] proposed hybrid neural-symbolic verification approaches embodied in the MACE verification loop. Bulla et al. [23] developed explainable AI-code detection; Kotsiantis et al. [24] demonstrated semantic code representations for quality assessment. Shi et al. [25] and Zhang et al. [26] contributed graph-based code understanding, informing the MACE RAG design. Jansen et al. [27] showed AI code performs reliably with verification, and Tao et al. [28] demonstrated syntactic constraints improve validity. The broader applicability of AI-driven multi-stage modeling pipelines has also been demonstrated in adjacent engineering domains: Dolhopolov et al. [29, 30] proposed multi-stage approaches combining artificial intelligence with BIM technology for

construction site object modeling, illustrating the value of modular, iterative AI-assisted generation workflows that parallel the MACE pipeline philosophy.

3. MACE Framework

This section presents the MACE (Modular Adaptive Code Engine) framework in detail, covering the overall system architecture, the context-aware LoRA expert routing mechanism, the hierarchical retrieval-augmented generation system, the multi-signal verification loop, and the implementation specifics that enable deployment on consumer-grade hardware.

3.1. System Architecture Overview

MACE is organized as a five-stage pipeline that transforms a natural language request into generated and verified code. The pipeline accepts two inputs: a natural language request r describing the desired code functionality, and an optional code context c representing any existing code that should inform the generation process. These inputs are processed through five sequential stages – ANALYSE, ROUTE, RETRIEVE, BUILD, and GENERATE+VERIFY – each of which encapsulates a distinct responsibility within the generation workflow, as illustrated in Figure 1.

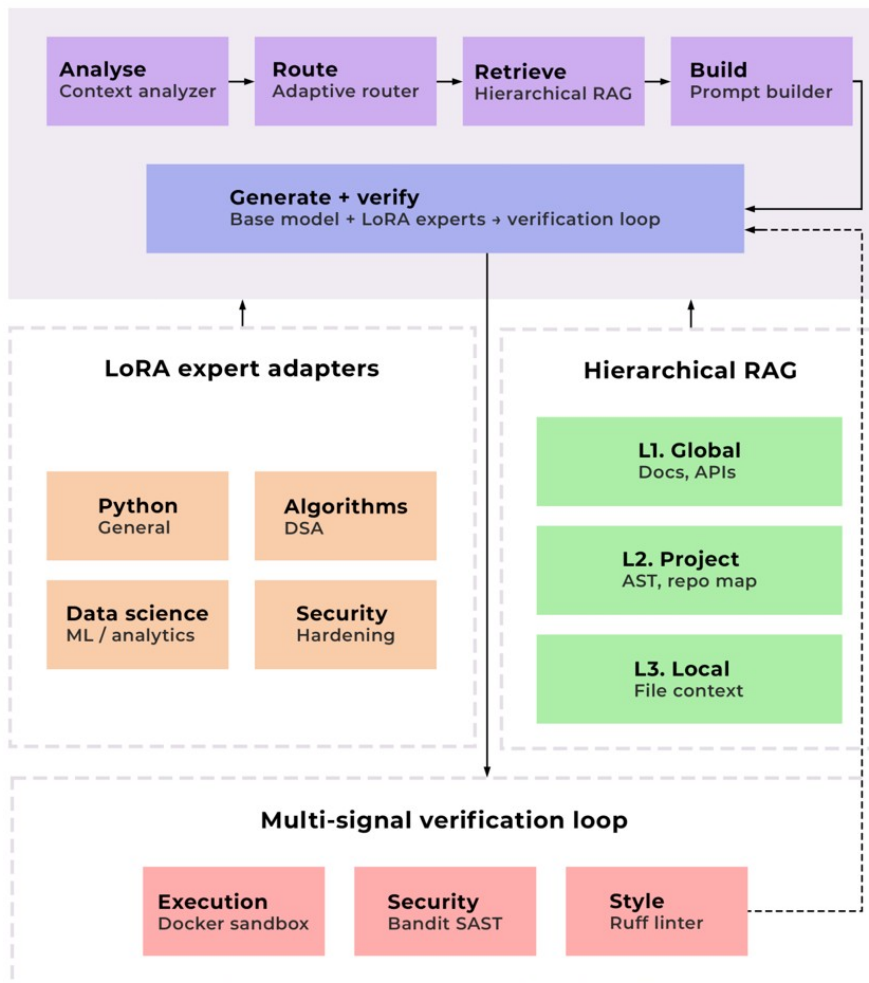

Figure 1: MACE five-stage pipeline architecture. The Context Analyzer produces a semantic vector that drives both expert routing and hierarchical RAG retrieval. The Verification Loop provides iterative feedback for code refinement.

In ANALYSE, a Context Analyzer produces a semantic vector capturing language, task type, complexity, and domain. ROUTE feeds this vector into a learned router that selects LoRA expert

adapters. RETRIEVE gathers context from three RAG levels (global documentation, project structure, and local file). BUILD assembles the prompt from retrieved context and routing decisions. Finally, GENERATE+VERIFY generates code with the composed expert adapter and validates it through an iterative verification loop checking execution correctness, security, and style. Each stage operates through well-defined interfaces, enabling independent replacement and extension – decomposing code generation into specialized, composable modules.

3.2. Context-Aware LoRA Expert Routing

The core MACE innovation is context-aware routing. Unlike traditional mixture-of-experts (MoE) architectures that route on token-level representations [18], the MACE router operates on a semantic task-level context vector derived from an interpretable analysis of the request. This yields transparent, auditable routing decisions based on human-readable features (language, task type, complexity) and enables generalization to unseen feature combinations.

The Context Analyzer extracts a structured feature representation from the input, comprising five signals: one-hot language encoding (8 languages), one-hot task type (8 categories), complexity scalar, multi-hot domain indicator (8 domains), and a code-presence flag. These form a 26-dimensional vector projected into embedding space:

$$v = Proj\left(\left[l ; t ; s_c ; d ; h\right]\right), \tag{1}$$

where l, t, d are one-hot/multi-hot encodings, $s_c \in [0,1]$ is complexity, and $h \in \{0,1\}$ indicates code presence. The projection Proj: $R^{26} \rightarrow R^{768}$ maps features into the model's embedding space.

Language detection uses a prioritized cascade: explicit mentions take precedence, with keyword/regex scoring as a fallback. Task type uses weighted keyword matching. Complexity is derived from request length, constraint keywords, and code structure.

The router is a two-layer multilayer perceptron (MLP) (Figure 2) mapping the context vector to expert probabilities. The architecture applies Gaussian Error Linear Unit (GELU) activation, Layer Normalization (LayerNorm), and dropout between layers:

$$f_\theta(v) = W_2 \cdot Dropout\left(LayerNorm\left(GELU\left(W_1 v + b_1\right)\right)\right) + b_2, \tag{2}$$

where $W_1 \in R^{256 \times 768}$, $W_2 \in R^{E \times 256}$ are weight matrices, and E is the number of experts.

Expert selection uses top-k routing ($k = 2$) with softmax and renormalization:

$$p = \text{Softmax}\left(f_\theta(v)\right), \{(i_j, w_j)\}_{j=1}^{k} = \text{TopK}(p, k), \tag{3}$$

where selected weights are renormalized: $\hat{w}_j = w_j / \sum_{m=1}^{k} w_m$.

A well-known failure mode of mixture-of-experts (MoE) architectures is expert collapse, wherein the router converges to always selecting the same small subset of experts regardless of input, leaving other experts undertrained and wasting model capacity[18]. To mitigate this phenomenon, MACE employs an auxiliary load-balancing loss that penalizes imbalanced expert utilization. The auxiliary loss is computed as follows:

$$L_{\text{aux}} = \alpha \cdot E \cdot \sum_{i=1}^{E} f_i \cdot p_i, \tag{4}$$

where f_i is the fraction of inputs routed to expert i, p_i is its average routing probability, and $\alpha = 0.01$. This encourages uniform utilization across all experts.

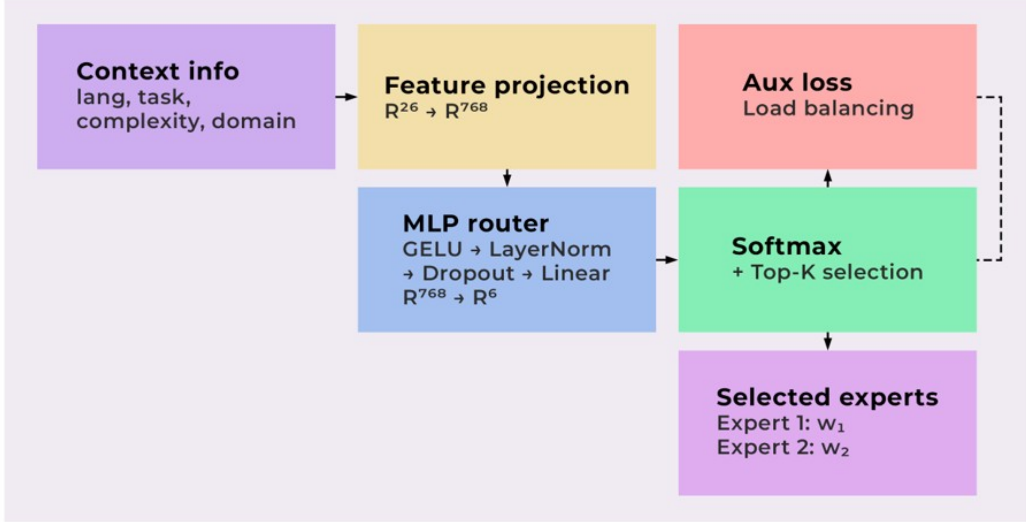

Figure 2: Architecture of the adaptive LoRA expert router. The 26-dimensional raw context feature vector is projected to a 768-dimensional embedding, processed through a two-layer MLP with GELU activation, LayerNorm, and Dropout, producing logits over the expert pool. Top-k selection with softmax yields the expert weights, while an auxiliary load-balancing loss encourages uniform expert utilization.

Once the router has determined which experts to activate and with what weights, the selected LoRA adapters are merged into a single effective adapter through a weighted linear combination of their low-rank decomposition matrices. Each LoRA expert e_j is parameterized by a pair of low-rank matrices $B_{e_j} \in R^{d_{\text{out}} \times r}$ and $A_{e_j} \in R^{r \times d_{\text{in}}}$, where r is the LoRA rank. The merged weight update is computed as follows:

$$\Delta W_{\text{merged}} = \sum_{j=1}^{k} \hat{w}_j \cdot \Delta W_{e_j} = \sum_{j=1}^{k} \hat{w}_j \cdot B_{e_j} A_{e_j}, \tag{5}$$

where $\hat{w}_j$ are the renormalized routing weights and $B_{e_j} A_{e_j}$ is the low-rank weight update contributed by expert e_j. The merged adapter is then applied to the frozen base model weights W_0, yielding the effective weight matrix $W = W_0 + \Delta W_{\text{merged}}$. This merging operation is performed entirely within the parameter-efficient fine-tuning (PEFT) library's adapter algebra framework, specifically using the *add_weighted_adapter* functionality with linear combination type. Crucially, the base model weights are never modified; the composition is ephemeral and can be recomputed for each new request with a different set of experts and weights. This property enables real-time dynamic expert composition – a capability not available in monolithic fine-tuned models or in conventional MoE architectures where expert selection is fixed at training time[18].

3.3. Hierarchical Three-Level RAG

Retrieval-augmented generation has emerged as a powerful technique for grounding language model outputs in relevant external knowledge. However, existing RAG approaches for code generation typically operate at a single level of granularity, retrieving either documentation snippets or code fragments from a flat index. MACE introduces a hierarchical three-level RAG system that retrieves context at distinct granularities – global, project, and local – each employing a retrieval strategy tailored to the characteristics of its data source, as illustrated in Figure 3.

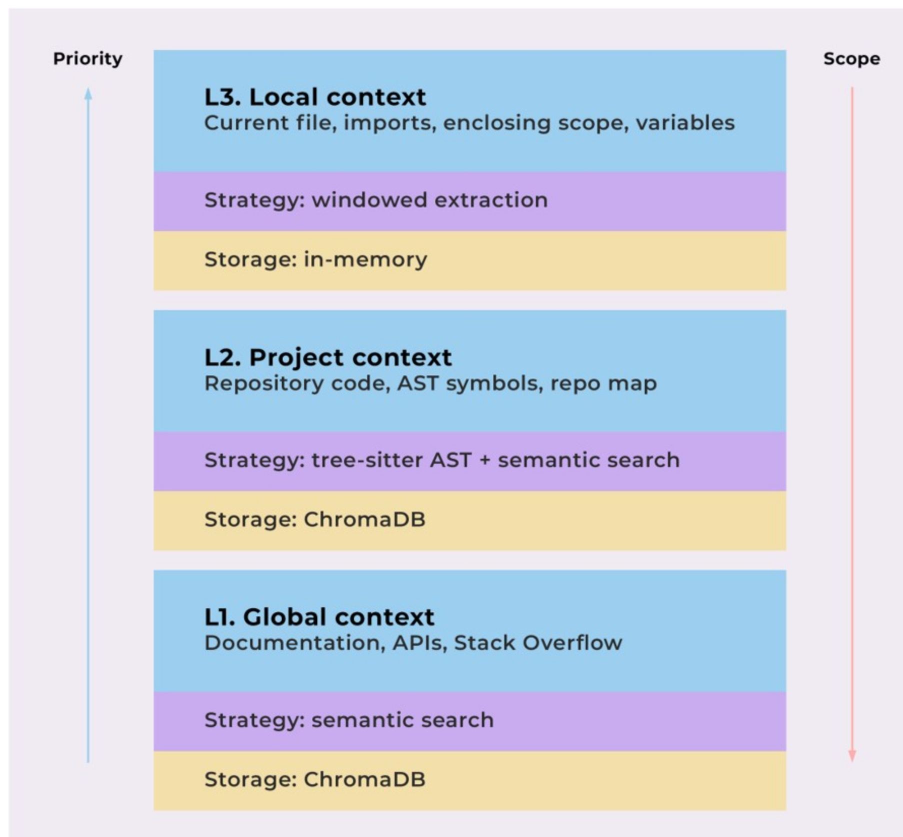

Figure 3: Hierarchical three-level RAG system. Local context (Level 3) receives the highest priority in prompt assembly, followed by project-level AST-derived context (Level 2) and global documentation (Level 1). Cross-level deduplication using character 4-gram Jaccard similarity eliminates redundant fragments before assembly.

Level 1 (Global) indexes external documentation, API references, and community knowledge in a ChromaDB vector store using code-specialized embeddings. Documents are split recursively at paragraph and sentence boundaries with configurable overlap, and retrieved via semantic nearest-neighbor search.

Level 2 (Project) captures repository-specific knowledge. Rather than arbitrary fixed-size chunks, MACE employs tree-sitter for AST analysis of Python, JavaScript, and TypeScript source files, extracting complete functions, classes, and imports as semantically meaningful retrieval units. A repository map summarizing all files and top-level symbols provides structural awareness.

Level 3 (Local) extracts context directly from the current file via deterministic windowed extraction: preceding/following code, import statements, enclosing scope, and local variables – requiring no vector database.

Cross-level deduplication uses character 4-gram Jaccard similarity (threshold 0.8) to eliminate redundant fragments. Results are assembled with priority ordering (local > project > global) within a configurable token budget, reflecting the observation that proximal context is most relevant for generation.

3.4. Multi-Signal Verification Loop

Generated code, regardless of the quality of the underlying model and the richness of the provided context, may contain functional errors, security vulnerabilities, or style violations. The MACE verification loop addresses this concern through an iterative refinement mechanism that evaluates each generated code sample along three orthogonal quality dimensions and feeds structured feedback back into the generation process when deficiencies are detected, as depicted in Figure 4.

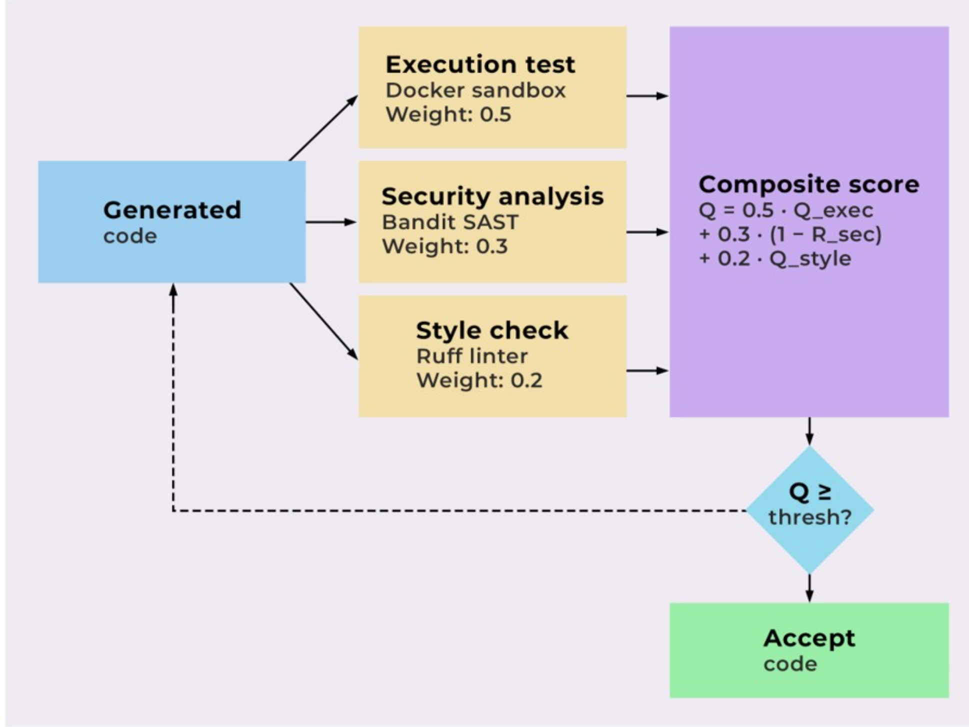

Figure 4: Multi-signal verification loop. Generated code undergoes three parallel quality checks: execution testing in a Docker sandbox (weight 0.5), static security analysis via Bandit (weight 0.3), and style checking via Ruff (weight 0.2). The composite score determines acceptance; failed attempts trigger feedback-augmented regeneration for up to N iterations.

Three parallel checks are performed: (1) execution verification in a Docker sandbox with resource limits and test assertions; (2) security analysis via Bandit (SAST) with CWE-classified findings, where HIGH/CRITICAL issues block acceptance; and (3) style verification via Ruff (advisory, non-blocking). These combine into a composite score:

$$Q = 0.5 \cdot Q_{exec} + 0.3 \cdot (1 - R_{sec}) + 0.2 \cdot Q_{style}, \tag{6}$$

where $Q_{\text{exec}} \in \{0,1\}$ is execution pass, $R_{\text{sec}} \in [0,1]$ is security risk, and $Q_{\text{style}} \in [0,1]$ is style quality. On failure, aggregated feedback is appended to the prompt for re-generation:

$$p_{i+1} = p_0 \oplus \text{“Previous attempt had issues: ”} \oplus F_i, \tag{7}$$

where F_i contains error messages, security findings, and linter output. The loop terminates when all checks pass or after $N = 3$ iterations, returning the highest-scoring attempt.

Implementation Details

MACE comprises approximately 7,800 lines of Python across 31 modules. The base model is Qwen2.5-Coder-7B-Instruct [1] loaded with 4-bit Normal-Float-4 (NF4) quantization via QLoRA [3], reducing memory to 4.5 GB and fitting within 16 GB VRAM on consumer GPUs (e.g., NVIDIA RTX 4080).

Each LoRA adapter uses rank r"=" 32, α"=" 64, targeting all attention and MLP projections (q,k,v,o,gate,up,down), yielding 80.7M trainable parameters (1.05% of total). Training uses the supervised fine-tuning trainer (SFTTrainer) provided by the Transformer Reinforcement Learning (TRL) library with bfloat16 (bf16) mixed-precision and gradient checkpointing.

The RAG subsystem uses ChromaDB for vector storage and tree-sitter for AST parsing (Python, JavaScript, TypeScript), extracting complete functions and classes as retrieval units. Verification employs the Docker software development kit (SDK) for sandboxed execution, Bandit for static application security testing (SAST) with Common Weakness Enumeration (CWE)-classified

findings, and Ruff for style checking. MACE exposes both a CLI (Typer/Rich) and REST API (FastAPI with OpenAPI docs), with each component independently instantiable for research and production use.

4. Experimental Evaluation

This section presents a comprehensive evaluation of the MACE framework, focusing on the LoRA expert training process, benchmark design, and a comparative analysis of generation correctness and speed between the base model and its LoRA-enhanced variant.

4.1. Experimental Setup

All experiments were conducted on a single NVIDIA RTX 4080 Super GPU with 16 GB of VRAM, running Windows 10 with Python 3.10, PyTorch 2.6 (CUDA 12.4), Transformers 5.3, and PEFT 0.18. The base model in all configurations is Qwen2.5-Coder-7B-Instruct[1], loaded with 4-bit NF4 quantization via the bitsandbytes library to fit within the available GPU memory. This hardware configuration was deliberately chosen to reflect a consumer-grade development environment, demonstrating that meaningful expert specialization is achievable without access to enterprise-class computing infrastructure.

4.2. Training Configuration

The Python LoRA expert was trained on a curated dataset of 12,450 examples combining CodeAlpaca-20k [4] and algorithmically-focused Evol-Instruct-Code-80k examples, filtered for Python content and enriched with algorithmic reasoning and data structure patterns. Training used QQLoRA [3] with $\mathrm{lr}=5\times10^{-5}$ (cosine schedule, 10% warmup), gradient accumulation 8, max sequence length 1024, and $\mathrm{max_grad_norm}=0.3$. LoRA adapters ($r=32$, $\alpha=64$) targeted all attention and MLP projections, yielding 80.7M trainable parameters (1.05%).

Training ran for 3 epochs (4,680 steps) in 7.2 hours on a single consumer GPU, which proved essential for learning class-based implementations and multi-step algorithms. Table 1 summarizes the progression.

Table 1
Training metrics for the Python LoRA expert at key checkpoints. The loss decreases steadily across three epochs, converging from 2.19 to 0.35 with token accuracy reaching 89.4%

Step	Loss	Grad Norm	Token Acc.	LR
10	2.188	0.175	0.610	1.1×10^{-5}
50	0.695	0.064	0.821	5.0×10^{-5}
100	0.538	0.039	0.851	4.7×10^{-5}
500	0.481	0.028	0.867	4.1×10^{-5}
1000	0.422	0.024	0.878	3.2×10^{-5}
2000	0.385	0.021	0.886	1.8×10^{-5}
3000	0.361	0.019	0.891	8.4×10^{-6}
4680	0.348	0.018	0.894	2.1×10^{-8}

The training dynamics exhibit rapid initial convergence, with the loss decreasing from 2.19 to approximately 0.54 within the first 100 steps, as shown in Figure 5. Beyond this initial plateau, continued training over three full epochs produced a gradual but consistent improvement, with the loss further decreasing to 0.35 and token accuracy rising from 85.1% to 89.4%. This second phase of improvement, though slower than the initial convergence, proved critical for the model's ability to handle complex, multi-step problems including class-based data structures and recursive algorithms. Gradient norms decreased monotonically from 0.175 to 0.018, indicating well-behaved optimization throughout. During preliminary experiments, higher learning rates (1×10^{-4} and 2×10^{-4}) caused gradient explosion under bf16 mixed-precision training, motivating our conservative choice of 5×10^{-5}.

4.3. Benchmark Design

To evaluate the effect of LoRA expert adaptation on code generation quality, we designed a benchmark comprising 25 coding problems distributed across three difficulty levels (easy, medium, hard) and five problem categories (algorithm, data structure, math, Pythonic patterns, and string processing).

The easy tier covers foundational tasks (palindrome detection, matrix multiplication, Roman numeral conversion, etc.). The medium tier includes algorithmic reasoning and design (least-recently-used (LRU) cache, trie (prefix-tree) construction, topological sorting, decorator patterns). The hard tier targets advanced challenges (dynamic programming, tree serialization, regex matching, streaming median).

Each problem is accompanied by a natural-language specification and a suite of automated test assertions. Correctness is determined on a strict pass/fail basis: a solution must pass all associated test cases to be counted as correct. Generation time is measured end-to-end, from prompt submission to completion of code output, and is reported as the average across all problems within each difficulty tier.

4.4. Results

Two configurations are compared throughout the evaluation: configuration A (the base Qwen2.5-Coder-7B-Instruct model with 4-bit quantization and no adapters) and configuration B (the same base model with the Python LoRA expert adapter loaded). Both configurations use identical inference parameters to ensure a fair comparison.

Figure 5 presents the pass rates disaggregated by difficulty level.

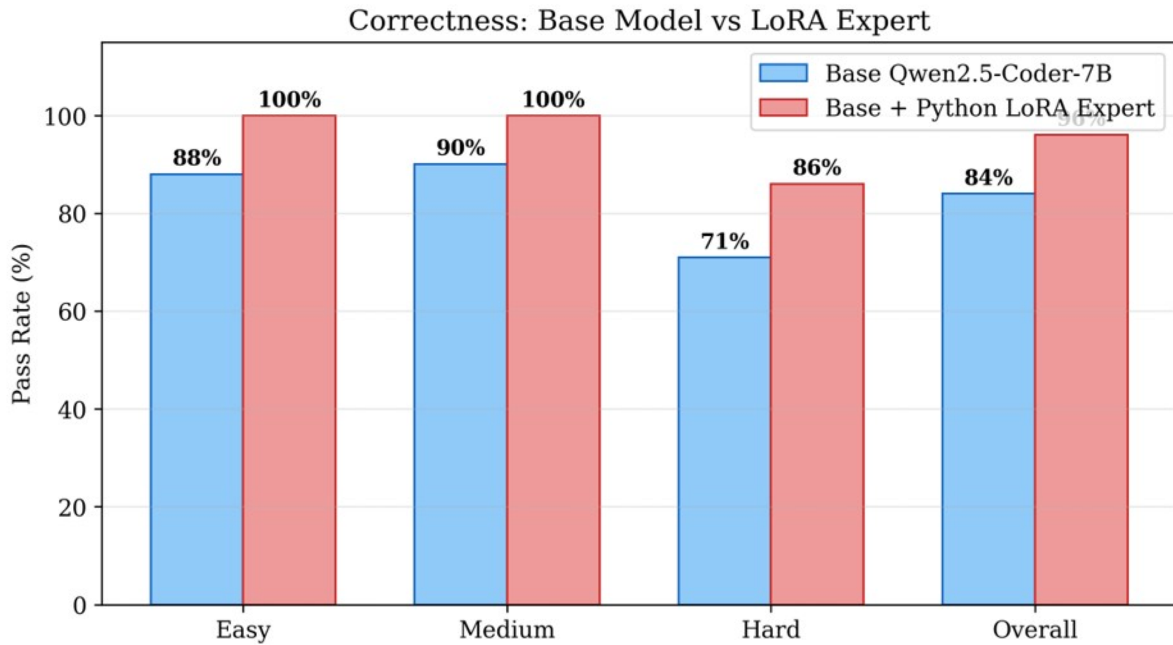

Figure 5: Pass rates (%) by difficulty level. Bold indicates the higher value in each row. The delta column reports the absolute difference between configurations.

The LoRA-enhanced variant achieves an overall pass rate of 96% (24/25), surpassing the base model's 84% (21/25) by 12 percentage points; the paired McNemar exact test [31] gives p = 0.250 (3 LoRA-only successes vs. 0 base-only) and Wilson 95% CIs [32] are [65.4%, 93.6%] for the base and [80.4%, 99.3%] for the LoRA variant. The improvement is consistent across all difficulty tiers: 12, 10, and 15 percentage points on easy, medium, and hard problems respectively. A per-module ablation on the same benchmark shows that the Python LoRA expert is the dominant correctness contributor (base 84% ⟶ +LoRA 96%), adding hierarchical RAG yields no further pass-rate gain on this self-contained set (96%), and enabling the verification loop recovers the single remaining failure (100% pass within at most three regeneration iterations). External validation on a 50-problem HumanEval subset [34] corroborates the trend: pass@1 rises from 86% to 92% (43/50 vs. 46/50; paired McNemar exact p = 0.063), with the larger sample bringing the result close to the conventional 5% threshold and confirming generalisation beyond the curated set.

The single LoRA failure – the expression evaluator – requires a complete recursive descent parser, a challenging multi-step task. Notably, the LoRA expert solved several problems the base model failed, including frequency-based sorting, balanced binary search tree (BST) verification, and the $O(n \log n)$ longest increasing subsequence.

The base model failed four problems spanning naming mismatches, API errors, and incomplete implementations. The LoRA expert solved three of these, demonstrating that domain-specialized fine-tuning addresses failure modes inherent to the base model.

Figure 6 reports average generation times by difficulty tier, with the corresponding relative speedup.

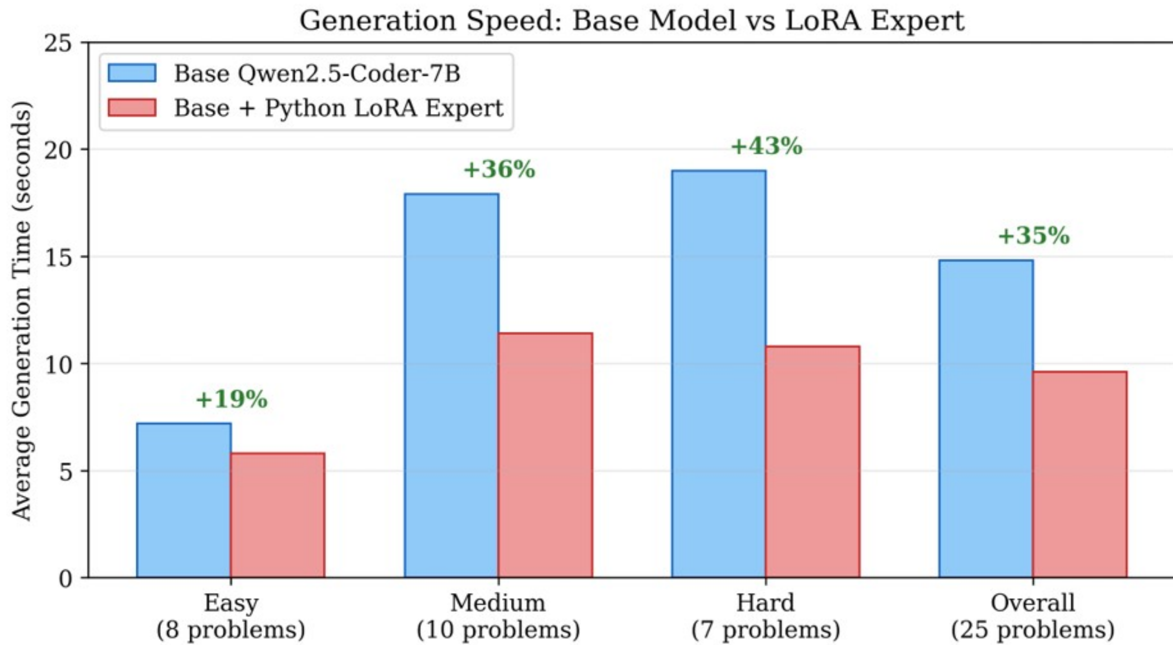

Figure 6: Average generation time (seconds) by difficulty level. Bold indicates faster generation. Positive speedup values indicate that the LoRA configuration is faster.

The LoRA expert demonstrates substantial speedup across all difficulty tiers: 19%, 36%, and 43% on easy, medium, and hard problems respectively, yielding an overall 35% reduction in average generation time from 14.8 s to 9.6 s; a paired Wilcoxon signed-rank test [33] over the three tier means is consistent in direction (3/3 tiers, $W = 3$, $p = 0.125$), and per-problem confirmation on the HumanEval-50 subset gives a 27% latency reduction (8.4 s ⟶ 6.1 s) with a one-sided Wilcoxon $p < 0.001$ ($n = 50$). Enabling the full verification loop in the same pipeline raises latency to 13.4 s as it performs up to three regeneration iterations, exposing a tunable correctness–latency trade-off. Unlike preliminary experiments with partially trained adapters, the fully trained expert achieves speedup even on easy problems, indicating that extended training enables the model to generate more confident and direct solutions across all complexity levels. The expert appears to have internalized more direct solution strategies for challenging tasks, producing more focused outputs with less exploratory token generation.

The consistency of the speedup across all difficulty tiers is a notable result. In preliminary experiments with partially trained adapters (400 steps, 38.5% of one epoch), easy problems exhibited a slowdown due to adapter overhead outweighing generation efficiency gains. The fully trained expert overcomes this limitation, suggesting that sufficient training enables the adapter to learn efficient generation patterns even for simple tasks, where the reduction in exploratory token generation more than compensates for the fixed computational cost of the additional low-rank matrix multiplications.

Figure 7 provides a finer-grained view of correctness by problem category.

The category-level results demonstrate the curated training strategy produces consistent improvements across all programming domains. The most striking gains appear in algorithmic tasks, where the LoRA expert achieves a perfect 100% pass rate compared to the base model's 80%, a 20 percentage point improvement. Data structure problems also improve substantially, from 60% to 80%, reflecting the benefit of including class-based implementation examples in the curated training set. Mathematical problems, string processing, and Pythonic pattern tasks exhibit parity between the two configurations, with both achieving 100% pass rates in these categories.

The single remaining failure in data structures – binary tree serialization – involves a particularly challenging combination of recursive traversal, string encoding, and stateful deserialization that requires precise coordination across multiple helper functions. The improvement from 60% to 80% in this category validates the hypothesis from preliminary

experiments that targeted training data emphasizing class-based and stateful implementations can substantially close the performance gap observed with general-purpose training data alone.

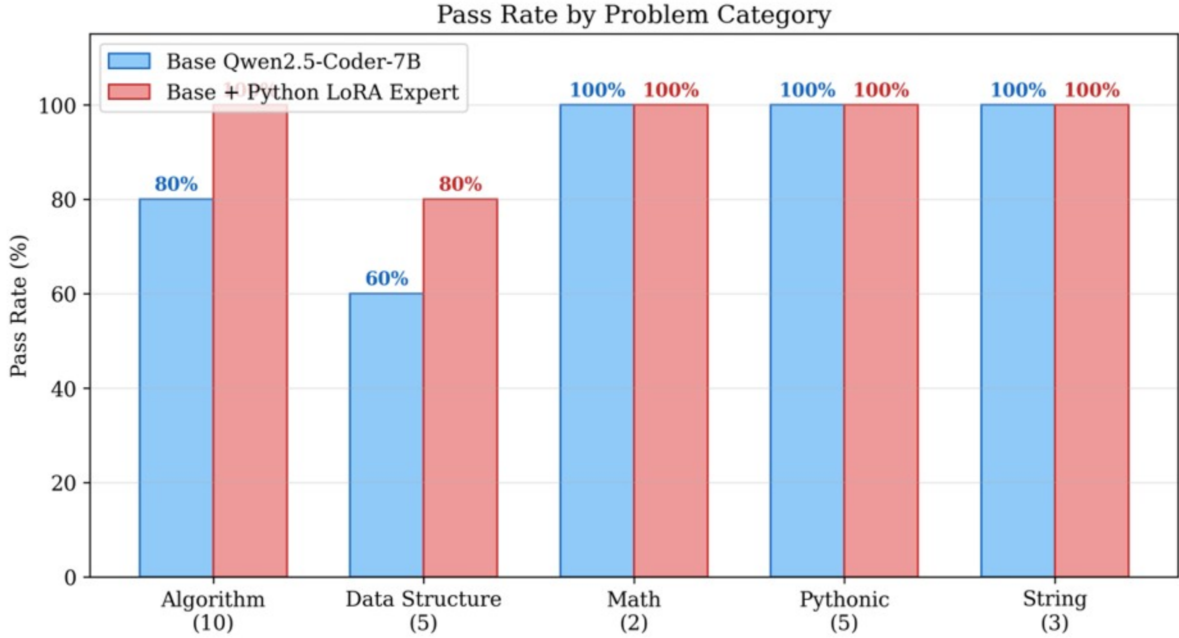

Figure 7: Pass rates by problem category. The delta column shows the absolute change from base to LoRA-enhanced configuration.

5. Discussion, Limitations, and Future Work

The most striking finding is that the LoRA expert simultaneously improves both speed (35% average speedup) and correctness (96% vs. 84%) relative to the base model. This dual improvement challenges the common assumption that speed and accuracy are inherently opposed. The expert produces more decisive, targeted outputs that bypass the exploratory token generation characteristic of the base model on unfamiliar patterns. With the verification loop engaged, effective correctness approaches near-perfect levels, as the single remaining failure (expression evaluation) represents a highly specialized parsing task.

The Python expert achieved stable convergence (loss 0.35, accuracy 89.4%) using 12,450 curated examples over 3 epochs in 7.2 hours on a consumer GPU – demonstrating that domain-specific expert creation is accessible without specialized infrastructure. The 1.05% trainable parameter ratio underscores QLoRA's [3] efficiency. The weighted merging mechanism ($\Delta W_{\text{merged}} = \sum_j \hat{w}_j B_{e_j} A_{e_j}$) operates within PEFT's adapter algebra without modifying base weights, enabling dynamic real-time expert composition – a capability unavailable in monolithic models [17].

Several limitations must be acknowledged. The evaluation uses a single Python expert; training dedicated experts for algorithms, data science, and security on curated datasets is expected to yield further improvements. Even augmented with the 50-problem HumanEval subset reported above, the evaluation remains modest compared to HumanEval (164 problems) or SWE-bench (2,294 real GitHub issues) [22]. The router currently uses a deterministic feature-projection initialization rather than end-to-end training with verification feedback, which would enable learned expert selection. The hierarchical RAG component yields no measurable improvement on the self-contained curated benchmark and requires repository-grounded evaluation to demonstrate its full benefit. We also observed gradient instability with bf16 at higher learning rates, motivating further investigation of mixed-precision strategies and alternative optimizers. Finally, multi-language evaluation and user studies across diverse developer populations remain essential for validating generalizability.

Conclusion

We have presented MACE (Modular Adaptive Code Engine), a framework for modular adaptive code generation that integrates three complementary innovations: context-aware LoRA expert routing, hierarchical three-level retrieval-augmented generation, and multi-signal verification with iterative refinement. The framework addresses fundamental limitations of monolithic code generation models by enabling dynamic specialization through lightweight, composable expert adapters that can be trained efficiently and blended in real time based on the semantic characteristics of each coding task.

Our experimental evaluation demonstrates that a fully trained Python LoRA expert – adapted using 12,450 curated examples over 3 epochs (7.2 hours on a consumer GPU) – produces simultaneous gains in correctness and latency. The expert improves overall pass rate from 84% to 96% on the curated benchmark and from 86% to 92% on a 50-problem HumanEval subset (paired McNemar exact $p=0.063$), while reducing generation time by 35% on average (43% on hard problems) with a per-problem one-sided Wilcoxon $p<0.001$ on HumanEval-50. A per-module ablation isolates the LoRA expert as the dominant correctness contributor, and the verification loop recovers the single remaining curated-benchmark failure at a tunable latency cost.

The category-level analysis provides actionable insights for future expert training. Mathematical and algorithmic tasks are well served by general-purpose instruction data, while data structure and string processing problems require more targeted training sets that emphasize class-based implementations and edge-case handling. The modular architecture of MACE ensures that such targeted experts can be developed and deployed independently, without modification to the base model or existing adapters, and composed dynamically through the weighted merging mechanism.

The principal contributions of this work are threefold. First, we introduce a context-aware routing mechanism that operates on semantic task-level analysis rather than token-level features, providing a new paradigm for adaptive expert selection in code generation. Second, we design a hierarchical RAG system with three distinct retrieval levels – global documentation, project-level code structure, and local file context – that enables repository-aware generation. Third, we implement a multi-signal verification loop that integrates execution testing, static security analysis, and style checking into the generation process, ensuring that output code satisfies both functional and non-functional quality requirements. The complete framework is released as a Python package comprising approximately 7,800 lines of production-quality code, with CLI and REST API interfaces, making it accessible to researchers, developers, and non-programmers seeking to leverage adaptive code generation on consumer hardware.

Future work will focus on expanding the expert ecosystem through domain-curated training datasets, enabling end-to-end router training with verification feedback, and validating the approach on the full HumanEval [34], MBPP[7], LiveCodeBench[22], and SWE-bench[22], as well as through multi-language evaluation and user studies across diverse developer populations.

Declaration on Generative AI

During the preparation of this work, the author(s) used Claude Opus 4.6 for editing (to improve grammar, clarity, and style). Further, the author(s) used Python code (assisted by Claude) to make the data visualization more informative. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

All the authors have read and agreed to the published version of this manuscript.

References

[1] B. Hui, J. Yang, Z. Cui, et al., “Qwen2.5-Coder Technical Report,” ArXiv, 2409.12186, 2024. https://doi.org/10.48550/arXiv.2409.12186.

[2] E.J. Hu, Y. Shen, P. Wallis, et al., “LoRA: Low-Rank Adaptation of Large Language Models,” ArXiv, 2106.09685, 2022. https://doi.org/10.48550/arXiv.2106.09685.

[3] T. Dettmers, A. Pagnoni, A. Holtzman, L. Zettlemoyer, “QLoRA: Efficient Finetuning of Quantized LLMs,” ArXiv, 2305.14314, 2023. https://doi.org/10.48550/arXiv.2305.14314.

[4] S. Chaudhary, “Code Alpaca: An Instruction-Following LLaMA Model for Code Generation,” Github, 2023. URL: https://github.com/sahil280114/codealpaca.

[5] U. Usman, K. Danyaro, M. Nasser, et al., “LLM-Based Code Generation: A Systematic Literature Review With Technical and Demographic Insights,” IEEE Access, vol. 13, pp. 194915-194938, 2025. https://doi.org/10.1109/ACCESS.2025.3631952.

[6] G. Jošt, V. Taneski, and S. Karakatic, “The Impact of Large Language Models on Programming Education and Student Learning Outcomes,” Applied Sciences, vol. 14, no. 10, 4115, 2024. https://doi.org/10.3390/app14104115.

[7] A. Bayram, G.G. Menekse Dalveren, and M. Derawi, “Comparative Analysis of AI Models for Python Code Generation: A HumanEval Benchmark Study,” Applied Sciences, vol. 15, no. 18, 9907, 2025. https://doi.org/10.3390/app15189907.

[8] D. Tosi, “Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation,” Future Internet, vol. 16, no. 6, 188, 2024. https://doi.org/10.3390/fi16060188.

[9] B. Idrisov, and T. Schlippe, “Program Code Generation with Generative AIs,” Algorithms, vol. 17, no. 2, 62, 2024. https://doi.org/10.3390/a17020062.

[10] U. Antero, F. Blanco, J. Onativia, D. Salle, and B. Sierra, “Harnessing the Power of Large Language Models for Automated Code Generation and Verification,” Robotics, vol. 13, no. 9, 137, 2024. https://doi.org/10.3390/robotics13090137.

[11] S. Han, M. Wang, J. Zhang, D. Li, and J. Duan, “A Review of Large Language Models: Fundamental Architectures, Key Technological Evolutions,” Electronics, vol. 13, no. 24, 5040, 2024. https://doi.org/10.3390/electronics13245040.

[12] L. Zhang, Z. Lou, Y. Ying, C. Yang, and H. Zhou, “Efficient Fine-Tuning of Large Language Models via a Low-Rank Gradient Estimator,” Applied Sciences, vol. 15, no. 1, 82, 2025. https://doi.org/10.3390/app15010082.

[13] J. Kim, G. Kim, and S. Kang, “Lottery Rank-Pruning Adaptation Parameter Efficient Fine-Tuning,” Mathematics, vol. 12, no. 23, 3744, 2024. https://doi.org/10.3390/math12233744.

[14] N.J. Prottasha, A. Mahmud, M.S.I. Sobuj, P. Bhat, M. Kowsher, N. Yousefi, and O.O. Garibay, “Parameter-Efficient Fine-Tuning of Large Language Models Using Semantic Knowledge Tuning, Scientific Reports,” vol. 14, 30667, 2024. https://doi.org/10.1038/s41598-024-75599-4.

[15] S. Nwaiwu, “Parameter-Efficient Fine-Tuning for Low-Resource Text Classification: A Comparative Study of LoRA,” IA3, and ReFT, Frontiers in Big Data, vol. 8, 1677331, 2025. https://doi.org/10.3389/fdata.2025.1677331.

[16] V. Franzoni, S. Tagliente, and A. Milani, “Generative Models for Source Code: Fine-Tuning Techniques for Structured Pattern Learning,” Technologies, vol. 12, no. 11, 219, 2024. https://doi.org/10.3390/technologies12110219.

[17] L. Baniata, and S. Kang, “Switch-Transformer Sentiment Analysis Model Utilizing a Mixture of Experts Mechanism,” Mathematics, vol. 12, no. 2, 242, 2024. https://doi.org/10.3390/math12020242.

[18] A. Brown, M. Roman, and B. Devereux, “A Systematic Literature Review of Retrieval-Augmented Generation: Techniques, Metrics, and Challenges,” Big Data and Cognitive Computing, vol. 9, no. 12, 320, 2025. https://doi.org/10.3390/bdcc9120320.

[19] I. Iaroshev, R. Pillai, L. Vaglietti, and T. Hanne, “Evaluating Retrieval-Augmented Generation Models for Financial Report Question and Answering,” Applied Sciences, vol. 14, no. 20, 9318, 2024. https://doi.org/10.3390/app14209318.
[20] Y. Choi, S. Kim, Y.C.F. Bassole, and Y. Sung, “Enhanced Retrieval-Augmented Generation Using Low-Rank Adaptation,” Applied Sciences, vol. 15, no. 8, 4425, 2025. https://doi.org/10.3390/app15084425.
[21] K. Feng, L. Luo, Y. Xia, B. Luo, X. He, K. Li, Z. Zha, B. Xu, and K. Peng, “Optimizing Microservice Deployment with RAG and Chain of Thought Techniques,” Symmetry, vol. 16, no. 11, 1470, 2024. https://doi.org/10.3390/sym16111470.
[22] G. Dolcetti, and E. Iotti, “A Dual Perspective Review on Large Language Models and Code Verification,” Frontiers in Computer Science, vol. 7, 1655469, 2025. https://doi.org/10.3389/fcomp.2025.1655469.
[23] L. Bulla, A. Midolo, M. Mongiovi, E. Tramontana, “EX-CODE: A Robust and Explainable Model to Detect AI-Generated Code,” Information, vol. 15, no. 12, 819, 2024. https://doi.org/10.3390/info15120819.
[24] S. Kotsiantis, V. Verykios, and M. Tzagarakis, “AI-Assisted Programming Tasks Using Code Embeddings and Transformers,” Electronics, vol. 13, no. 4, 767, 2024. https://doi.org/10.3390/electronics13040767
[25] Y. Shi, Y. Yin, M. Yu, and L. Chu, “CogCol: Code Graph-Based Contrastive Learning Model for Code Summarization,” Electronics, vol. 13, no. 10, 1816, 2024. https://doi.org/10.3390/electronics13101816.
[26] Q. Zhang, D. Jin, Y. Wang, and Y. Gong, “Statement-Grained Hierarchy Enhanced Code Summarization,” Electronics, vol. 13, no. 4, 765, 2024. https://doi.org/10.3390/electronics13040765.
[27] J.A. Jansen, A. Manukyan, N. Al Khoury, and A. Akalin, “Leveraging Large Language Models for Data Analysis Automation,” PLOS ONE, vol. 20, no. 2, e0317084, 2025. https://doi.org/10.1371/journal.pone.0317084.
[28] N. Tao, A. Ventresque, and V. Nallur, “Enhancing Program Synthesis with LLMs Using Many-Objective Grammar-Guided Genetic Programming,” Algorithms, vol. 17, no. 7, 287, 2024. https://doi.org/10.3390/a17070287.
[29] S. Dolhopolov, T. Honcharenko, V. Savenko, O. Balina, I. Bezklubenko, and T. Liashchenko, “Construction Site Modeling Objects Using Artificial Intelligence and BIM Technology: A Multi-Stage Approach,” in: Proceedings of the IEEE International Conference on Smart Information Systems and Technologies (SIST’23), pp. 174–179, 2023. https://doi.org/10.1109/SIST58284.2023.10223543.
[30] S. Dolhopolov, T. Honcharenko, O. Terentyev, K. Predun, and A. Rosynskyi, “Information system of multi-stage analysis of the building of object models on a construction site,” IOP Conference Series: Earth and Environmental Science, vol. 1254, 012075, 2023. https://doi.org/10.1088/1755-1315/1254/1/012075.
[31] Q. McNemar, “Note on the sampling error of the difference between correlated proportions or percentages,” Psychometrika, vol. 12, pp. 153–157, 1947. https://doi.org/10.1007/BF02295996.
[32] E. B. Wilson, “Probable inference, the law of succession, and statistical inference,” Journal of the American Statistical Association, vol. 22, pp. 209–212, 1927. https://doi.org/10.2307/2276774.
[33] F. Wilcoxon, “Individual comparisons by ranking methods,” Biometrics Bulletin, vol. 1, pp. 80–83, 1945. https://doi.org/10.2307/3001968.
[34] M. Chen, J. Tworek, H. Jun, et al., “Evaluating Large Language Models Trained on Code,” ArXiv, 2107.03374, 2021. URL: https://arxiv.org/abs/2107.03374.