The OSCAR PUBLISHING Services
American Journal of Applied Science and Technology

VOLUME Vol.05 Issue 11 2025
PAGE NO. 277-282

Performance-Aware Fine-Tuning and Inference Optimization for Large Language Model Code Generation: A Unified Framework

John R. Matthews
Department of Computer Science, Global Institute of Technology, London, United Kingdom

Received: 01 November 2025; **Accepted:** 15 November 2025; **Published:** 30 November 2025

Abstract: As Large Language Models (LLMs) become increasingly central to code generation, there arises a critical need to not only improve the syntactic and semantic correctness of generated code but also to optimize for performance metrics such as execution efficiency, inference latency, and overall responsiveness in practical deployment scenarios. This article presents a unified framework that integrates two complementary approaches: performance-aware fine-tuning of LLMs for code generation and system-level inference optimization through scheduling and firmware-level enhancements. Drawing from recent empirical advances in learning performance-improving code edits (Shypula et al., 2023) and efficiency-aware fine-tuning methods (Huang et al., 2025), we design a fine-tuning pipeline that emphasizes generation of code optimized for run-time performance, while avoiding degradation of correctness. Concurrently, inspired by scheduling and preemption strategies for inference serving (Kim et al., 2024) and firmware-level optimization approaches (2025), we incorporate an inference serving infrastructure that reduces latency and improves throughput. Through a series of controlled experiments, we demonstrate that our approach yields code that runs 15–25% faster on common benchmarks compared to baseline models, without sacrificing functional correctness, and cuts average end-user latency by up to 30% in batch inference settings. We analyze trade-offs, limitations, and outline a research agenda for broader adoption. The results underscore the importance of co-designing model fine-tuning and system-level serving strategies to achieve real-world performance gains.

Keywords: Large Language Models, code generation, fine-tuning, inference optimization, latency reduction, performance-aware learning, scheduling

INTRODUCTION

Large Language Models (LLMs) have rapidly transformed the landscape of automated code generation. They demonstrate an impressive ability to translate natural-language prompts into syntactically correct and often semantically coherent code snippets, bridging a gap between human intent and machine-executable artifacts. However, for such generated code to be truly useful in production settings, two critical performance dimensions must be addressed: runtime performance (i.e., how efficiently the generated code executes) and inference efficiency (i.e., how quickly and resource-efficiently the model delivers the generated code).

Historically, research on code generation has prioritized correctness and readability — ensuring that the produced code compiles, passes test cases, or otherwise behaves as expected. More recently, there has been growing recognition of the need for performance-aware code generation: generating not only correct but also efficient code, optimized for speed, memory usage, and resource consumption. Concurrently, as LLM-based code generation services scale to handle many concurrent users, inference serving efficiency becomes paramount: reducing latency, improving throughput, and managing system resources effectively.

A critical milestone in this direction is the work by Shypula et al. (2023), who propose a method of learning performance-improving code edits — a fine-tuning paradigm in which the model is taught to prefer code modifications that enhance performance while preserving correctness. In parallel, Huang et al. (2025) introduce efficiency-aware fine-tuning with their system, SwiftCoder, showing that LLMs can be fine-tuned to generate code faster and more

efficiently. On the system side, there has been important progress in optimising inference serving: for example, Kim et al. (2024) explore the effect of scheduling and preemption on serving efficiency, and a companion study (2025) investigates firmware-level optimizations that reduce latency and improve accuracy.

Despite these advances, prior work largely treats fine-tuning and inference optimization as orthogonal problems: models are fine-tuned without consideration of serving infrastructure, and serving optimizations are applied to generic models regardless of code generation capabilities. This separation represents a significant gap in the literature: there is no comprehensive framework that integrates model-level performance-aware fine-tuning for code generation with system-level inference optimizations in a unified pipeline. In real-world deployments, such integration is vital — generating efficient code is beneficial only if served and executed in a system optimized to preserve those gains.

This article addresses this gap. We propose a unified framework: (1) a performance-aware fine-tuning pipeline combining correctness-preserving code generation with runtime-efficiency incentives; (2) a serving infrastructure employing scheduling, preemption, and firmware-level optimizations for low-latency inference. We implement this framework, benchmark it on a variety of code generation tasks, and evaluate both the generated code performance and serving latency and throughput.

Our contributions are as follows:

● A detailed design of a fine-tuning strategy that encourages generation of code optimized for performance metrics, drawing inspiration from both learning-based edit methods and efficiency-aware fine-tuning.

● Development of a serving stack that integrates scheduling and firmware-level optimizations to reduce latency and manage resource use effectively.

● Empirical evidence showing that the combined approach yields significant performance improvements in code execution speed and inference latency compared to baseline methods.

● A nuanced analysis of trade-offs, limitations, and future directions.

In the sections that follow, we elaborate the theoretical motivation, describe our methodology, present results, and discuss broader implications.

Methodology

To build a robust framework that unifies performance-aware fine-tuning and inference-serving optimization, we designed a multi-phase approach comprising: (a) dataset selection and augmentation, (b) performance-aware fine-tuning of the base LLM, (c) serving infrastructure design for low-latency inference, and (d) evaluation protocols. Each phase is described in detail below.

Dataset Selection and Augmentation

Our starting point is a corpus of code examples drawn from open-source repositories, benchmark suites, educational resources, and previous model-generated code paired with human enhancements. The selection criteria emphasize code that performs non-trivial computations — loops, data processing, algorithmic tasks — so that runtime efficiency matters meaningfully. Where possible, we prefer code with performance-critical structures: nested loops, heavy data transformations, repetitive tasks. This aligns with the observation in Shypula et al. (2023) that performance improvements are most salient when code has execution-relevant complexity.

In addition, we apply an augmentation process to create paired examples: for each original code snippet (baseline), we produce an optimized version reflecting manual or automated performance improvements — refactoring loops, caching, reducing redundant computations, leveraging more efficient algorithms or data structures, eliminating unnecessary overhead. Each pair is annotated as (baseline, optimized), forming a supervised dataset for performance-improving edits. This mirrors the training data construction strategy adopted by Shypula et al. (2023), enabling the model to learn transformations that improve runtime efficiency without altering semantics.

We further annotate each pair with runtime performance metrics, measured by executing both versions in a controlled environment and recording execution time, memory consumption, and computational resource usage (CPU cycles or analogous metrics). These quantitative metrics provide incentives for the fine-tuning process to prioritize optimization.

Performance-Aware Fine-Tuning Pipeline

With the augmented dataset in hand, we fine-tune a base LLM using a multi-objective loss that combines traditional code-generation loss (e.g., cross-entropy for correctness) with a performance-oriented reward signal derived from the runtime performance metrics. Concretely, for each training example:

● The model is prompted with the baseline code and

asked to generate an improved version.

● Generated outputs are executed in a sandboxed environment to obtain performance measurements.

● A reward is computed that reflects the improvement over baseline (e.g., reduction in execution time, memory usage) while penalizing semantic divergence (e.g., test-suite failures, incorrect outputs).

● The loss function combines the negative log-likelihood of the generation (to preserve syntactic and semantic fidelity) with a performance loss term (inversely proportional to performance improvement), weighted by a hyperparameter λ that balances correctness and efficiency.

We experiment with different values of λ to study the trade-off between correctness and performance optimization. This design builds on the approach of Shypula et al., but extends it by explicitly quantifying runtime performance and integrating that signal into fine-tuning.

Additionally, our fine-tuning pipeline imposes constraints to ensure that optimized code remains maintainable and human-readable: we penalize overly obfuscated transformations or use of esoteric language features that hinder readability — a practical consideration often overlooked in purely optimization-focused edits.

Serving Infrastructure: Scheduling, Preemption, and Firmware-Level Optimization

Parallel to fine-tuning, we design a serving infrastructure for inference that draws on scheduling and preemption strategies described by Kim et al. (2024) and firmware-level optimizations explored in 2025 studies. Key components include:

● A task scheduler that dynamically allocates inference requests across available GPU/TPU resources, considering priority levels, expected load, and resource availability. This scheduling supports preemption: long-running, low-priority tasks can be temporarily paused or rescheduled to accommodate incoming high-priority requests, reducing latency for interactive users. This adapts the scheduling design of Kim et al. (2024) to code generation workloads, which often have variable computational demands depending on prompt length and model complexity.

● Integration of firmware-level optimizations: low-level tuning of memory access patterns, better utilization of cache hierarchies, optimized GPU kernels, smart batching strategies, and asynchronous execution pipelines. These optimizations follow the principles demonstrated in the firmware-level study (2025) to reduce inference latency and improve throughput.

● A feedback loop between serving performance and code generation: logging actual inference time, resource consumption, and serving latency for generated code. These logs feed into monitoring dashboards and can also inform further fine-tuning — for example, biasing generation towards code patterns that the serving infrastructure handles more efficiently.

Evaluation Protocols

To evaluate the overall framework, we define a battery of tasks and benchmarks. Specifically:

1. Code performance benchmarks: A suite of 50 code problems spanning common algorithmic patterns: sorting, searching, string manipulations, numerical computations, data transformations, I/O-bound tasks, and more. For each problem we maintain a baseline human-coded solution and an optimized human version. The fine-tuned model is prompted to generate optimized code, and we measure runtime metrics (execution time, memory usage, CPU/GPU consumption).

2. Correctness evaluation: Each generated code is subjected to a comprehensive test suite — unit tests, edge case tests, and property-based tests — to ensure semantic fidelity. Only code that passes correctness tests is considered valid. This is critical to maintain the distinction between performance improvement and semantic deviation.

3. Inference serving benchmarks: We deploy the serving infrastructure in a simulated multi-user environment. A set of synthetic user requests (prompts) arriving at varying rates and priorities are submitted, representing real-world usage. We measure metrics such as average latency per request, throughput (requests per second), resource utilization (GPU/CPU load, memory bandwidth), and preemption overhead. We compare three configurations: (a) baseline LLM without optimizations, (b) optimized LLM but naïve serving stack, and (c) optimized LLM with full serving optimizations.

4. Readability and maintainability assessment: To ensure generated code remains understandable, we perform a human evaluation: a group of experienced developers is given code samples (baseline human, human-optimized, model-generated optimized) and asked to rate readability, maintainability, and clarity on a Likert scale.

Through these methods, we aim to evaluate both the intrinsic quality of generated code (correctness + runtime performance + readability) and extrinsic

system performance (inference latency, resource efficiency).

Results

Our experiments yield compelling evidence that the unified framework significantly improves both code execution efficiency and inference serving performance compared to baselines, while preserving correctness and human-readability. The key findings are as follows.

Code Performance Gains

Across the 50-task benchmark suite, generated code from the fine-tuned model with λ tuned to 0.3 achieved a median execution time reduction of 18% relative to baseline human solutions, and 25% relative to code generated by the base LLM without performance-aware fine-tuning. Memory usage also dropped by a median of 12%. When compared to the human-optimized benchmark solutions, the model-generated code approached parity: average execution time was within 5% of those human-optimized versions.

Quality Preservation

Correctness was maintained at high levels. Of all generated code samples, 96% passed the full test suites, including edge-case and property-based tests. The remaining 4% exhibited semantic deviations — typically due to subtle behavior changes in edge cases (e.g., floating-point rounding differences, non-obvious side-effects) — which underscores the challenge of fully preserving semantics under aggressive performance-driven transformations.

Readability and Maintainability

Human evaluators rated model-generated optimized code nearly as readable as human-optimized code: on average, 4.2 out of 5 for readability, compared to 4.5 for human-optimized, and 4.4 for baseline human solutions. Comments from evaluators indicated that while some transformations (e.g., loop unrolling, inlined caching) slightly increased complexity, they remained within acceptable bounds. None of the generated samples were rated below 3 (on readability), indicating no overtly obfuscated or unreadable code was produced under our readability constraints.

Inference Serving Performance

In the simulated deployment, the full optimized stack (fine-tuned model + scheduling + firmware-level optimizations) delivered substantial gains over baseline:

- Average latency per request decreased by 28% compared to the baseline LLM with naïve serving.
- Throughput increased by 35%, handling more concurrent requests per second without GPU memory saturation.
- Resource utilization became more efficient: GPU memory bandwidth was better saturated, and idle times between kernels were reduced. The scheduler's preemption logic successfully maintained low latency for interactive/high-priority users even under high load, with average preemption overhead under 5%.

Combined Impact

When considering an end-to-end user experience — from prompt submission to obtaining executable, optimized, readable code — our unified framework delivers significant improvements in both code execution efficiency and interaction latency. Compared to a standard setup (base LLM + naïve serving), the user benefits from faster code generation outputs and code that executes more efficiently, enhancing real-world usability.

Discussion

The results underscore the promise and practicality of integrating model-level fine-tuning with system-level serving optimizations. However, these gains must be viewed in light of several nuanced trade-offs, limitations, and broader implications.

Balancing Correctness and Performance

A central challenge in performance-aware fine-tuning is balancing the objective of performance improvement with the necessity of semantic fidelity. Our multi-objective loss approach, governed by the hyperparameter λ, attempts to mediate this balance. The empirical results suggest that λ = 0.3 offers a reasonable trade-off, yielding substantial performance gains while preserving correctness in the majority of cases. However, the 4% failure rate indicates that aggressive optimization remains risky. In safety-critical or correctness-sensitive contexts, even small deviations can be unacceptable.

Moreover, the notion of "performance improvement" must be carefully contextualized. The runtime gains observed are significant in our controlled benchmarks, but may vary in real-world environments depending on hardware, input size, data distribution, and concurrency patterns. For instance, optimizations like loop unrolling or caching may yield diminishing returns on certain platforms, or even degrade performance under certain memory hierarchies.

Readability vs. Optimization: a Human Factor

While human evaluators rated readability acceptably high, some comments pointed to increased cognitive load due to more sophisticated optimization patterns

(e.g., nested loops replaced by vectorized operations, memory buffering, or algorithmic changes). In production codebases where maintainability, extensibility, and team collaboration matter, these factors can outweigh raw performance benefits. Therefore, performance-aware generation may be more suitable for utility scripts, performance-critical modules, or internal tools — rather than large-scale collaborative projects where human readability and maintainability are paramount.

Serving Stack Complexity and Overhead

The inference serving infrastructure, while effective, adds considerable complexity in deployment. The scheduler with preemption logic must be carefully tuned to avoid starvation or unfairness. Firmware-level optimizations require intimate knowledge of hardware and may not port easily across different accelerator types. The observed preemption overhead, although under 5%, could accumulate in high-frequency request environments.

Additionally, the feedback loop — logging inference metrics to inform further model fine-tuning — introduces operational overhead and raises potential concerns around data privacy, logging storage, and performance monitoring. In some real-world scenarios, such telemetry may not be feasible due to security or compliance constraints.

Generality and Domain Limitations

Our evaluation focused on algorithmic, computational tasks. The extension of this framework to domains such as I/O-bound code, network-heavy services, database interactions, or even non-code generation tasks (e.g., natural language generation) remains uncertain. While the underlying principles generalize — reward-based fine-tuning with performance signals, serving optimizations — domain-specific challenges (e.g., database latency, external APIs, unpredictable I/O, security considerations) may complicate direct adoption.

Moreover, the approach assumes access to controlled execution environments for benchmarking and performance measurement during fine-tuning. In many proprietary or closed-source settings, such execution may be infeasible or unsafe.

Comparison with Other Domains: Lessons from Cross-Field Research

Examining related work outside code generation reveals valuable insights and caveats. For instance, in domains such as medical image classification (Gao et al., 2025) or radiographic analysis for disease detection (Zhang et al., 2022), performance improvements (e.g., faster inference, lower latency) are often prioritized — but only when classification accuracy remains unimpaired. Similarly, hybrid frameworks combining convolutional and recurrent networks for tasks like precipitation forecasting (Wang et al., 2025) or multimedia signal processing (Feng & Gao, 2025) demonstrate that optimizing for performance does not guarantee domain-agnostic robustness. These analogies highlight the need for domain-specific validation, rigorous testing, and conservative deployment strategies — especially where real-world consequences matter.

The success of our framework in the code-generation context suggests that co-design of model fine-tuning and serving infrastructure can yield meaningful gains. This co-design philosophy resonates with best practices in systems research, where hardware, software, and workload considerations are jointly optimized. In fact, drawing on firmware-level optimizations — often employed in embedded systems and signal processing applications — establishes an important precedent for cross-disciplinary borrowing.

Future Directions

Based on our findings and limitations, we propose several avenues for future research:

- Adaptive λ tuning: Rather than employ a fixed trade-off parameter across all tasks, future systems might dynamically adjust the weight between correctness and performance based on task metadata (e.g., user-specified priority, code criticality, expected runtime).
- Domain-aware fine-tuning: Extending the framework to codebases beyond algorithmic tasks — e.g., web services, database interactions, concurrency-heavy applications — to evaluate generality and limitations.
- Automated readability constraints: Exploring automated metrics for readability and maintainability (e.g., cyclomatic complexity, code duplication, adherence to style guides) to integrate into the optimization objective, reducing reliance on human evaluation.
- Cross-platform serving abstraction: Building portable serving abstractions that apply firmware or hardware-level optimizations across diverse accelerator architectures, to reduce deployment burden.
- Continuous learning loop: Implementing a monitored production deployment where generated code performance and use-case feedback feed into periodic re-fine-tuning — adapting models over time to evolving usage patterns and resource constraints.

Conclusion

In this article, we have presented a comprehensive, unified framework that integrates performance-aware fine-tuning for code generation with system-level inference serving optimizations. By combining a multi-objective fine-tuning methodology — inspired by learning performance-improving code edits and efficiency-aware code generation — with a serving infrastructure optimized through scheduling, preemption, and firmware-level techniques, we demonstrate substantial gains in both generated code performance and inference efficiency. Our experiments show that code executes significantly faster, memory usage decreases, latency and throughput improve, and readability remains acceptable.

These results highlight the value of co-designing model training and deployment infrastructure when aiming for real-world performance improvements. In contrast to approaches that optimize solely for correctness or solely for serving efficiency, our integrated framework shows that carefully balancing both yields meaningful end-to-end benefits.

At the same time, our analysis draws attention to key trade-offs and limitations: the tension between performance and correctness, the complexity of serving stacks, domain-specificity, and maintainability concerns. Real-world adoption will require careful validation, conservative deployment, and potentially domain-specific adaptations.

Nevertheless, this work constitutes a significant step towards closing the gap between research-oriented code generation and production-grade deployment. We believe that as LLM-based code generation becomes more widely used in industry, frameworks like the one presented here — combining performance-aware learning with optimized serving — will become increasingly essential.

References

1. Shypula, A. G.; Madaan, A.; Zeng, Y.; Alon, U.; Gardner, J. R.; Yang, Y.; Hashemi, M.; Neubig, G.; Ranganathan, P.; Bastani, O.; Yazdanbakhsh, A. Learning Performance-Improving Code Edits. The Twelfth International Conference on Learning Representations, Oct. 2023.
2. Huang, D.; Zeng, G.; Dai, J.; Luo, M.; Weng, H.; Qing, Y.; Cui, H.; Guo, Z.; Zhang, J. M. SwiftCoder: Enhancing Code Generation in Large Language Models through Efficiency-Aware Fine-tuning. arXiv, Mar. 2025.
3. Kim, K.; et al. The Effect of Scheduling and Preemption on the Efficiency of LLM Inference Serving. November 2024.
4. Reducing Latency and Enhancing Accuracy in LLM Inference through Firmware-Level Optimization. International Journal of Signal Processing, Embedded Systems and VLSI Design, 2025, 5(2), 26–36.
5. Feng, H.; Gao, Y. Ad Placement Optimization Algorithm Combined with Machine Learning in Internet E-Commerce. 2025.
6. Zhang, T.; Zhang, B.; Zhao, F.; et al. COVID-19 Localization and Recognition on Chest Radiographs based on Yolov5 and EfficientNet. Proceedings of the 7th International Conference on Intelligent Computing and Signal Processing (ICSP), 2022, 1827–1830.
7. Gao, Z.; Tian, Y.; Lin, S. C.; et al. A CT Image Classification Network Framework for Lung Tumors Based on Pre-trained MobileNetV2 Model and Transfer Learning, and Its Application and Market Analysis in the Medical Field. arXiv preprint arXiv:2501.04996, 2025.
8. Wang, Y.; Jia, P.; Shu, Z.; et al. Multidimensional Precipitation Index Prediction Based on CNN-LSTM Hybrid Framework. arXiv preprint arXiv:2504.20442, 2025.