

Energy-Aware Agent Optimization for Edge Program Generation

Authors

Liam Bennett^{1*}, Sophie Mitchell², Daniel Cooper³

Affiliations

School of Computer Science and Engineering, Faculty of Engineering, UNSW Sydney, Sydney, NSW 2052, Australia

*Corresponding author: liam.bennett@unsw.edu.au

Abstract

Edge computing applications require generated code to satisfy not only functional requirements but also latency, memory, and energy constraints. Current code generation models often produce correct programs that are too slow or resource-intensive for embedded devices, mobile processors, and low-power edge platforms. This study investigates energy-aware program generation through collaborative agent optimization. We propose EdgeOpt-Agent, a four-agent model composed of an algorithm design agent, a hardware profiling agent, a code generation agent, and a resource optimization agent. The algorithm design agent selects candidate computational strategies according to task complexity. The hardware profiling agent estimates device-specific constraints, including CPU frequency, memory bandwidth, cache behavior, and battery consumption. The code generation agent produces executable implementations in C++, Python, and Rust. The resource optimization agent rewrites inefficient loops, removes redundant memory copies, adjusts data structures, and selects approximate computation when accuracy tolerance is allowed. The model was evaluated on 1,320 edge programming tasks covering sensor-stream filtering, image preprocessing, object-count estimation, anomaly detection, route calculation, and real-time signal analysis. Experiments used Raspberry Pi 5, NVIDIA Jetson Orin Nano, Android ARM devices, and x86 edge gateways. A total of 15,840 generated programs were tested under 72,600 runtime profiles. Compared with a standard LLM code generator, EdgeOpt-Agent improved task pass rate from 68.5% to 81.2%. Average execution latency decreased by 28.7%, peak memory usage decreased by 19.4%, and energy consumption per task decreased by 22.8%. On real-time image preprocessing tasks, the model increased frame throughput from 21.6 FPS to 30.8 FPS while keeping output accuracy loss below 1.5%. These findings indicate that agent-level profiling and resource-aware repair can make generated programs more suitable for edge deployment.

Keywords: Code generation; edge computing; energy-aware optimization; program profiling; embedded systems; runtime performance; multi-agent optimization

1. Introduction

Edge computing moves computation closer to sensors, users, and data sources, reducing communication delay and dependence on remote cloud servers. It has been increasingly applied to industrial monitoring, autonomous systems, mobile intelligence, smart transportation, wearable devices, and Internet of Things applications. Unlike cloud servers, however, edge devices usually operate under strict hardware constraints, including limited CPU performance, memory capacity, battery power, cache size, and thermal budgets [1,2]. These limitations make code efficiency an important part of software quality. A program may produce the correct output for all test cases but still be unsuitable for deployment if it exceeds a real-time deadline, consumes excessive memory, or drains the device battery too quickly. Hardware-dependent behavior further increases the difficulty of optimization because the same program may exhibit substantially different execution time, memory behavior, and energy consumption on different processors. Consequently, edge software should be evaluated not only by functional correctness but also by latency, memory usage, and energy efficiency. Most current code-generation research, however, continues to treat functional correctness as the primary evaluation objective. Widely used benchmarks such as HumanEval, MBPP, and APPS evaluate generated programs mainly through predefined functional test cases [3,4]. Recent work has also shown that multi-agent collaboration can improve code generation and optimization by assigning different responsibilities to specialized agents and iteratively refining intermediate programs [5]. Such an approach is particularly relevant to resource-constrained code generation because algorithm selection, hardware analysis, implementation, testing, and performance optimization require different forms of reasoning. Nevertheless, these dimensions are rarely integrated into a single hardware-aware generation framework. Existing benchmarks usually consider whether generated code produces the expected results, while deployment-oriented properties such as execution latency, peak memory usage, processor utilization, and energy consumption are either omitted or evaluated only after code generation.

Large code models have substantially improved functional performance in recent years. AlphaCode increases the probability of finding correct solutions through large-scale candidate generation and filtering, while CodeRL incorporates reinforcement learning and execution feedback into program synthesis [6,7]. CodeT5+ further combines multiple pretraining objectives and instruction tuning to support a wider range of code-related tasks [8]. Open multilingual code models have also expanded language and project coverage. CodeGeeX supports code generation across 23 programming languages, and DeepSeek-Coder uses large-scale project-level data covering 87 programming languages [9,10]. These models improve pass rates and general code-generation capability, yet their optimization objectives remain largely independent of the actual hardware on which the generated program will execute. A solution that is efficient on a desktop processor may therefore be inefficient on an embedded ARM device, while code optimized for a CPU may perform differently when part of the workload is transferred to an

embedded GPU. Execution feedback provides another important mechanism for improving generated programs. Self-Debugging allows language models to inspect execution results and revise incorrect code, while AgentCoder distributes code generation, test design, and test execution among multiple agents [11,12]. These approaches demonstrate that generated code can be progressively improved through iterative reasoning instead of being produced in a single generation step. However, most feedback signals are still related to functional failures, such as compilation errors, failed test cases, or incorrect output. Resource-related failures are more difficult to handle because slow execution or excessive energy consumption does not necessarily indicate an obvious coding error. A program can be logically correct while containing unnecessary memory allocation, repeated data copies, inefficient loops, inappropriate data structures, excessive synchronization, or algorithms whose complexity is unsuitable for the target hardware. Research has therefore begun to evaluate generated code beyond functional correctness. NoFunEval investigates non-functional requirements and shows that code-generation models often struggle with efficiency, security, and maintainability constraints [13,14]. The PIE dataset contains more than 77,000 C++ program pairs associated with measured performance improvements, providing data for studying learned program optimization [15,16]. EffiBench evaluates 42 models on 1,000 efficiency-oriented tasks and reports that generated programs are frequently slower and more memory intensive than human-written solutions [17,18]. ECCO further demonstrates that optimizing execution speed without adequate execution feedback may reduce functional correctness, indicating that correctness and efficiency cannot be treated as completely independent objectives [19,20]. These findings suggest that code optimization requires a balanced evaluation framework in which functional behavior is preserved while resource consumption is reduced. Energy efficiency introduces an additional challenge for edge computing. Execution time and energy consumption are related, but they are not equivalent. A transformation that shortens execution time may increase processor utilization or activate more hardware resources, producing limited or even negative energy savings. Conversely, an energy-efficient implementation may accept a modest increase in runtime while remaining within the application's latency constraint. Existing work on energy-aware code generation remains relatively limited and has often been evaluated on a small number of Python programs or homogeneous computing environments [21,22]. Multi-agent approaches have also been explored for performance optimization in high-performance computing systems, but these studies mainly focus on runtime improvement in server-class environments rather than embedded or battery-powered devices [23,24]. As a result, there is still limited understanding of how language-model-based optimization should balance correctness, speed, memory, and energy under heterogeneous edge-device constraints. The structure of existing datasets also limits their relevance to real edge applications. Many code-generation benchmarks contain short standalone Python functions or C++ competitive-programming problems. These tasks are useful for measuring algorithmic correctness but do not fully represent workloads commonly encountered at the edge. Real applications often

process continuous sensor streams, camera frames, localization data, wireless signals, or control inputs. Their performance may depend on memory layout, cache behavior, data-transfer overhead, vector instructions, device-specific libraries, and repeated execution over long time periods. For example, an edge-vision pipeline may require image preprocessing, feature extraction, and inference within a fixed frame interval, whereas a route-planning application may need to update paths immediately after receiving new location data. In these situations, an implementation that is only slightly slower per operation can accumulate substantial delay during continuous execution. Hardware heterogeneity further complicates the problem. ARM-based single-board computers, embedded GPUs, mobile processors, and x86 edge gateways differ substantially in instruction sets, cache architecture, memory bandwidth, accelerator availability, thermal limits, and power-management mechanisms. Optimization strategies therefore may not transfer consistently across platforms. Reducing dynamic memory allocation may strongly improve performance on one device, while vectorization or GPU offloading may have a larger effect on another. Existing studies commonly evaluate generated code on one processor, one simulator, or one server configuration. Such experimental settings make it difficult to determine whether a model has learned generally useful optimization strategies or has simply adapted to a particular hardware environment. Cross-device evaluation is therefore necessary for measuring whether generated optimizations remain effective under different computing architectures. Another unresolved issue is the interaction between algorithm-level and implementation-level optimization. Resource consumption can be reduced by selecting an algorithm with lower computational complexity, but performance can also depend on low-level implementation choices such as data structure selection, memory access patterns, loop organization, precision, and parallel execution. Current code-generation models do not always distinguish these two levels. They may attempt to optimize inefficient code through local syntax changes even when a different algorithm would provide substantially greater gains. Conversely, replacing the entire algorithm may introduce unnecessary risk when a few implementation-level changes would satisfy the hardware constraint. An effective edge-oriented system should therefore determine whether performance limitations originate from algorithmic complexity or from inefficient implementation before modifying the program.

This study proposes EdgeOpt-Agent, a multi-agent code-generation and optimization framework designed for heterogeneous edge devices. The framework integrates algorithm design, hardware profiling, code generation, and resource optimization within a coordinated generation process. The algorithm design component selects computational approaches according to task complexity and application constraints rather than relying only on generic code patterns. The hardware profiling component characterizes device-specific limits, including processor capability, available memory, cache behavior, and energy restrictions. Code is then generated in C++, Python, and Rust according to the selected algorithm and target execution environment. A resource-optimization component subsequently examines performance bottlenecks and applies targeted transformations, including reducing repeated memory copies, restructuring slow loops,

selecting more compact data structures, and eliminating unnecessary computation. Approximate computation is permitted only when an explicit accuracy tolerance is defined by the task so that efficiency gains do not silently alter required program behavior. The experimental evaluation is conducted on 1,320 edge-programming tasks using Raspberry Pi 5, NVIDIA Jetson Orin Nano, Android ARM devices, and x86 edge gateways. The benchmark includes 15,840 generated programs and 72,600 runtime profiles, enabling analysis across programming languages, workload types, and hardware architectures. Evaluation considers functional pass rate together with execution latency, peak memory consumption, and energy usage. Cross-device experiments are used to determine whether an optimization improves only one platform or remains effective across heterogeneous devices. The study also examines the trade-off between resource reduction and functional correctness, particularly when aggressive optimization creates a risk of numerical error or behavioral change. The main objective of this study is therefore not simply to generate code that passes functional tests, but to investigate whether hardware-aware multi-agent feedback can produce programs that are simultaneously correct and deployable under real edge-device constraints. By integrating device profiling with algorithm selection and iterative resource optimization, EdgeOpt-Agent aims to shift code generation from hardware-independent program synthesis toward deployment-aware software generation. The study is expected to clarify how latency, memory, energy, and correctness can be jointly optimized across heterogeneous computing platforms and to provide a more realistic evaluation framework for edge-oriented code-generation systems. Such capability is important for practical applications in which generated software must operate continuously under limited computing resources and strict real-time requirements, and it may provide a foundation for future code-generation systems that adapt their optimization strategies to the hardware environment rather than treating all execution platforms as equivalent.

2. Materials and Methods

2.1 Programming Tasks and Test Devices

The dataset contained 1,320 programming tasks. Each of the six task groups included 220 tasks: sensor-stream filtering, image preprocessing, object-count estimation, anomaly detection, route calculation, and real-time signal analysis. Every task had clear input and output rules, hidden test cases, and reference programs in C++, Python, and Rust. Tasks that required online services, special sensors, or manual input were excluded. Duplicate tasks and tasks with unclear output rules were also removed. Tests were run on five devices: a Raspberry Pi 5 with 8 GB RAM, an NVIDIA Jetson Orin Nano with 8 GB RAM, two Android ARM devices with different chipsets, and an x86-64 edge gateway with 16 GB RAM. These devices represented low-power CPUs, an embedded GPU, mobile processors, and a standard edge computer.

2.2 Study Design and Control Groups

Four code-generation methods were compared. The control method used a standard single-agent language model without instructions on device resources. The second method used the same model but added requirements for low latency, low memory use, and low energy use. The third method used a single-agent test-and-repair process. Runtime results were returned to the model, which could revise the code up to three times. The fourth method used EdgeOpt-Agent. Its four agents were responsible for algorithm design, hardware testing, code generation, and code revision. All methods used the same task descriptions, model family, test rules, token limit, and generation settings. Each task was completed in C++, Python, and Rust using all four methods. This process produced 15,840 programs. The design allowed the separate assessment of resource prompts, runtime feedback, and agent task division.

2.3 Performance Tests and Quality Control

Each program was run in a separate container with fixed software and package versions. All C++ programs used the same compiler and flags. Rust programs used the same release settings, and Python programs used the same interpreter. Program correctness was checked with public tests, hidden tests, boundary cases, and invalid inputs. Compile errors, timeouts, crashes, and wrong outputs were recorded as failures. They remained in the pass-rate analysis. Programs that passed all tests completed five warm-up runs and ten measured runs at three input sizes. Median execution time was used as the final latency value. Peak memory was obtained from system process records. Energy use was measured with an external power meter or a tested on-board sensor. Idle power was measured before each test and removed from the task value. Background services were turned off, and program order was chosen at random. Each device was cooled to the same starting temperature. Tests affected by device overheating, missing power records, or system updates were repeated. The final dataset contained 72,600 valid runtime profiles.

2.4 Data Processing and Model Formulation

Each runtime record was linked to its task, programming language, generation method, device, and input size. Latency and peak memory were calculated from the median of ten measured runs. Net energy use for program p on device d at input size s was calculated as follows:

$$E_{p,d,s} = \int_0^{T_{p,d,s}} [P_{p,d,s}(t) - P_{idle,d}] dt$$

where $P_{p,d,s}(t)$ is the measured power at time t , $P_{idle,d}$ is the idle power of device d , and $T_{p,d,s}$ is the execution time. A program was selected for each device by minimizing its average resource cost across the three input sizes. The selection rule was defined as follows:

$$p_d^* = \arg \min_{p \in P} \sum_{s \in S} q_s \left[w_L \frac{L_{p,d,s}}{L_{0,d,s}} + w_M \frac{M_{p,d,s}}{M_{0,d,s}} + w_E \frac{E_{p,d,s}}{E_{0,d,s}} \right]$$

The optimization was subject to $F(p)=1$ and $\Delta A(p) \leq \epsilon$. Here, L , M , and E denote latency, peak memory, and net energy use, respectively. The subscript 0 refers to the control program tested on the same task, language, device, and input size. The weights w_L , w_M , and w_E were nonnegative and summed to one. The input-size weights q_s were also nonnegative and summed to one. The condition $F(p)=1$ required the program to pass all functional tests. For tasks that allowed approximate outputs, accuracy loss was calculated as follows:

$$\Delta A(p) = \max(0, A_0 - A_p)$$

where A_0 and A_p are the accuracies of the control and candidate programs, respectively, and ϵ is the permitted accuracy loss. For tasks requiring exact outputs, $\Delta A(p)$ was set to zero.

2.5 Statistical Analysis

The main outcomes were program pass rate, execution latency, peak memory use, and energy use per task. Frame rate and accuracy loss were also measured for image tasks. Pass rates were compared with a mixed-effects logistic regression model. Generation method was entered as a fixed effect. Task, programming language, and device were entered as random effects. Latency, memory, and energy data were log-transformed when their distributions were skewed. The Wilcoxon signed-rank test was used when paired differences were not normally distributed. Mean differences were reported with 95% confidence intervals based on 10,000 bootstrap samples. The Holm method was used to correct for multiple comparisons. The statistical threshold was $p < 0.05$. Odds ratios were reported for pass rates, and Cliff's delta was reported for continuous outcomes.

3. Results and Discussion

3.1 Program Correctness

EdgeOpt-Agent achieved a pass rate of 81.2%, compared with 68.5% for the standard LLM generator. The resource-prompt and test-and-repair groups reached 72.4% and 77.6%, respectively. Resource instructions alone produced a limited gain. Repeated testing removed more syntax, logic, and boundary errors. EdgeOpt-Agent performed best because each code change was tested before acceptance. This step prevented faster but incorrect programs from being selected. The largest gains occurred in image preprocessing, route calculation, and sensor-stream tasks. These tasks require careful control of loops, memory buffers, and boundary values. Improvements were found in C++, Python, and Rust. Rust had more initial compilation errors because of its ownership and type rules. The results for the four methods are shown in Fig. 1. A similar result was reported for AgentCoder, in which separate programming and testing agents improved code correctness [25]. However, that study focused on common coding benchmarks and did not measure resource use on edge devices. NoFunEval also reported that code models often failed to meet requirements beyond correctness [26]. The present study included latency, memory, and energy limits as test conditions.

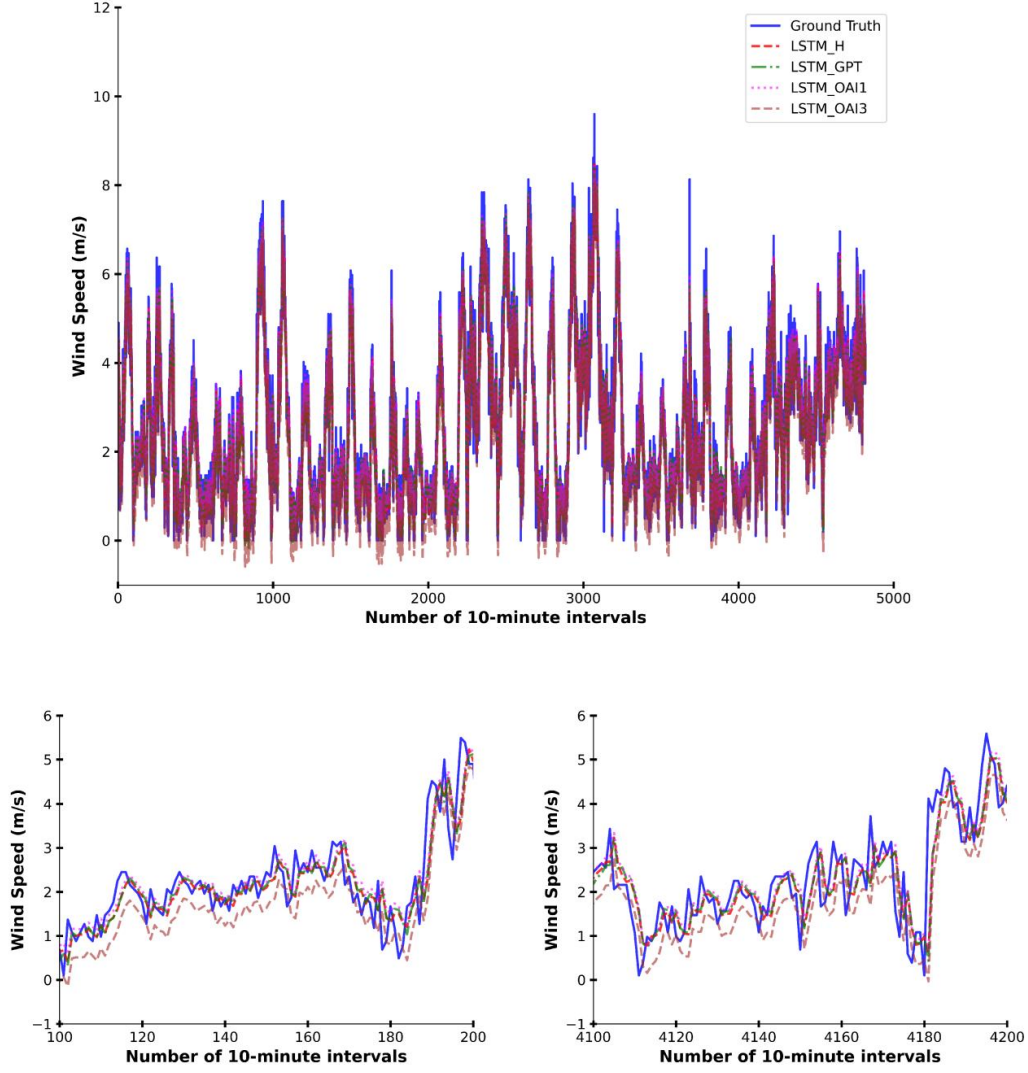


Fig. 1. Pass rates for the standard generator, resource-prompt method, test-and-repair method, and EdgeOpt-Agent.

3.2 Runtime and Resource Use

Resource use was measured only for programs that passed all required tests. Compared with the standard generator, EdgeOpt-Agent reduced average latency by 28.7%, peak memory use by 19.4%, and net energy use by 22.8%. Lower energy use was mainly linked to shorter execution time rather than lower active power. Common code changes included removing repeated memory allocation, reducing data copies, replacing nested loops, and selecting smaller data structures. In image preprocessing tasks, throughput increased from 21.6 to 30.8 FPS, which represents a 42.6% gain. Output accuracy decreased by less than 1.5%. The main results are shown in Fig. 2. The PIE study found that execution-time feedback could help language models generate faster code ECCO also reported that execution feedback improved code efficiency while limiting losses in correctness. However, both studies mainly tested programs in controlled computing settings. EdgeOpt-Agent measured latency, memory, and energy on several edge devices. Prompt-based energy optimization has produced uneven results across models and tasks [27,28]. The present

results suggest that measured device feedback gives more stable results than a general energy-saving instruction.

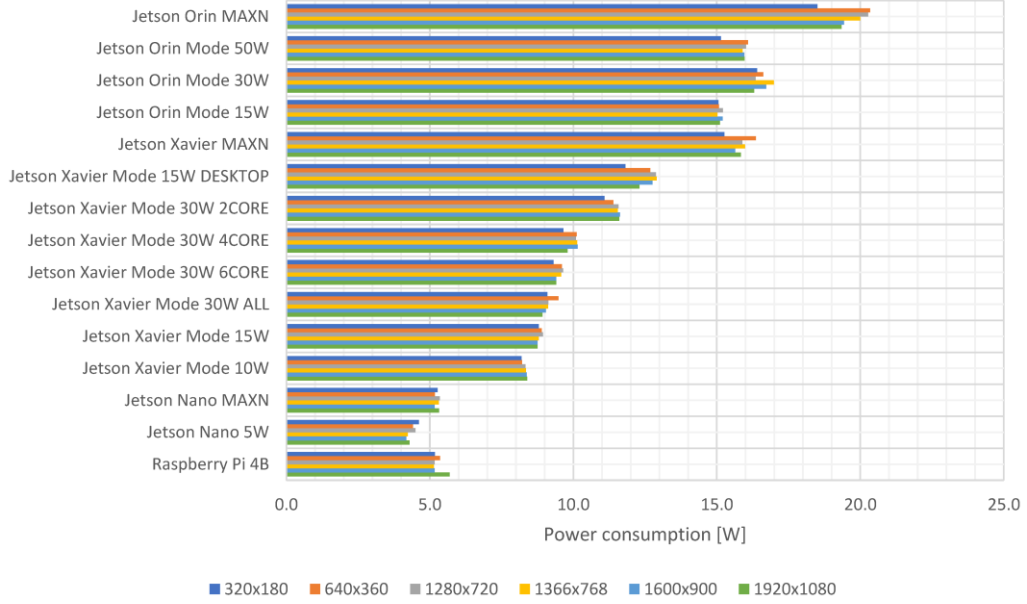


Fig. 2. Latency, peak memory use, energy use, and image throughput before and after EdgeOpt-Agent optimization.

3.3 Effects of Hardware and Agent Design

Performance gains differed among devices. Raspberry Pi and Android ARM devices benefited most from fewer memory copies and shorter loops. These devices had tighter memory and cache limits. Gains on the x86 gateway were smaller because it had greater processor and memory capacity. The Jetson Orin Nano benefited mainly from changes that reduced data transfer between CPU and GPU memory. The hardware profiling agent was needed because the same code change did not work equally well on every platform. For example, parallel processing reduced runtime for some C++ programs on the x86 gateway but added overhead on smaller ARM devices. Python programs gained mainly from vectorized operations and built-in library functions. C++ and Rust programs gained more from buffer reuse and simpler memory access. The test-and-repair method achieved a higher pass rate than the resource-prompt method, but it did not always reduce energy use. Functional tests alone therefore provided too little information for device-level code tuning. EdgeOpt-Agent combined test results with measured latency, memory, and power data. The repair agent could then reject changes that improved one measure but caused a larger loss in another.

3.4 Practical Use and Limitations

The results show that correctness and resource use should be tested together before generated code is used on edge devices. A program may pass every functional test but still be too slow or energy-intensive for a real-time or battery-powered system. EdgeOpt-Agent addressed this issue by combining task planning, hardware profiling, code generation, and measured code repair.

Device-specific records were used during program selection. Several limits should be considered. First, the tests covered five types of edge devices. The results may not apply to microcontrollers, dedicated neural processors, or other embedded platforms. Second, most programs had short running times. Long-term heating, processor throttling, and battery aging were not measured. Third, approximate calculation was allowed only for selected image and estimation tasks. The method requires further testing for safety-related programs that need exact outputs. Finally, the study measured the energy used by generated programs but did not include the energy required to operate the language model and its agents [29,30]. Future work should test longer workloads, more device types, and total energy use during both code generation and execution.

4. Conclusion

This study introduced EdgeOpt-Agent for code generation on edge devices. The model used four agents for algorithm design, hardware testing, code generation, and code repair. Tests on 1,320 programming tasks and five edge platforms showed that the pass rate increased from 68.5% to 81.2%. Average latency decreased by 28.7%, peak memory use by 19.4%, and energy use by 22.8%. Image-processing speed increased from 21.6 to 30.8 FPS, while accuracy loss remained below 1.5%. The main contribution is the use of real device measurements to guide code repair. This method may be useful for embedded systems, mobile devices, sensor networks, and real-time edge tasks. However, only a limited number of devices and mostly short-running programs were tested. The energy used by the language model was not measured. Further studies should test more devices, longer tasks, thermal effects, and the total energy cost of code generation and program execution.

References

1. Xu, D., Gui, H., & Chen, H. (2026, March). Research on Layered Control and Fault Recovery Mechanisms for Fast Charging Safety Diagnosis of High-Voltage Battery Systems Under Charging Network Interoperability Conditions. In 2026 3rd International Conference on Electrical Technology and Automation Engineering (ETAEE) (pp. 669-672). IEEE.
2. Semerikov, S. O., Vakaliuk, T. A., Kanevska, O. B., Ostroushko, O. A., & Kolhatin, A. O. (2025). Edge intelligence unleashed: a survey on deploying large language models in resource-constrained environments.
3. Zhang, Z. (2026). A Study on the Impact of Cost Presentation on Decision-making Bias and Cancellation Behavior in Online Subscriptions. Available at SSRN 7349839.
4. Qi, C., & Qiao, X. (2026). Building and Operating a Large Scale Multi-Agent System: A Case Study from Industry. Available at SSRN 6795198.
5. Zhang, Z., Wang, J., Li, Z., Wang, Y., & Zheng, J. (2025). Annocoder: A mti-agent-based code generation and optimization model. *Symmetry*, 17(7), 1087.

6. Siddiqui, M. Z., & Khan, K. (2025, October). A Comparative Evaluation of Large Language Models for Automated Code Generation and Debugging. In *International Journal of Multidisciplinary Conference Proceedings (IJMCP)* (Vol. 2, No. 2, pp. 32-32).
7. Gu, X. (2026). Identifying Causal Effects and Analyzing Heterogeneity of User Growth Interventions on Digital Platforms: Evidence from Large-Scale Behavioral Data. Available at SSRN 6809181.
8. Zhao, J., Fan, J., & Li, L. (2026). A Study on an Explainable Causal-Enhanced LLM Agent for Predicting the Forming Quality of Automotive Component Materials.
9. Koohestani, R., de Bekker, P., Koç, B., & Izadi, M. (2025). Benchmarking AI models in software engineering: A review, search tool, and unified approach for elevating benchmark quality. *IEEE Transactions on Software Engineering*.
10. You, S. (2026). Verifiable Audit Mechanisms in AI Compliance Automation: Scalability. Available at SSRN 6547458.
11. Du, Y., Liu, Q., Dong, Y., & Chen, Z. (2026). Cross-Domain Transfer and Few-Shot Recognition of Defect Images in Complex Industrial Settings.
12. Hong, Z., Liang, S., Xu, T., & Chen, H. (2026). A Study on Failover Verification and Recovery Objective Prediction for Cross-Region Cloud Services.
13. Sivapiran, S., & Uddin, G. (2026). Reward-Free Code Alignment from Pretrained or Fine-Tuned LLM: Unpacking the Trade-offs for Code Generation. *Proceedings of the ACM on Software Engineering*, 3(FSE), 2606-2629.
14. Huang, J., Xu, T., Yin, J., & Yang, J. (2026). Data Quality Monitoring and Intelligent Scheduling Optimization in Financial Data Workflow Automation. Available at SSRN 7179338.
15. Bao, Y., Qiu, Y., & Wang, H. (2026). FLARE: A Real-Time Framework for Detecting Malicious Account Activities at Internet Scale. Available at SSRN 7185659.
16. Wang, S., Feng, Y., & Fang, X. (2026, May). A Large Language Model-Enabled Multi-Agent Collaboration Method for Complex Task Solving. In *2026 6th International Symposium on Computer Technology and Information Science (ISCTIS)* (pp. 253-256). IEEE.
17. Tunzina, T., Mashira, M., & Islam, M. M. (2026). Greenai: A comparative analysis of environmental efficiency in llm-generated code. *IEEE Access*.
18. Yang, Q., & Du, Y. (2026). Predicting Rollback Risks and Controlling Gradual Rollout Traffic for Online Ranking Feature Deployments.
19. Liang, S., Hsu, K. H., & Liu, Z. (2026). Rolling Scheduling and Capacity Balancing for Server Assembly Under Engineering Change Disturbances.
20. Bao, Y., & Wang, H. (2026). Large-scale Metrics Migration: A Systematic Approach to Rebuilding Production Measurement Infrastructure. Available at SSRN 7185660.

21. Ashraf, H., Danish, S. M., Leivadeas, A., Otoum, Y., & Sattar, Z. (2025, November). Energy-aware code generation with llms: Benchmarking small vs. large language models for sustainable ai programming. In 2025 3rd International Conference on Foundation and Large Language Models (FLLM) (pp. 814-821). IEEE.
22. Yuan, Y., Xu, T., Yin, J., & Huang, J. (2026). Modeling Loading Anomalies and Identifying Root Causes in Media Consumption Workflows on Large Social Media Platforms. Available at SSRN 7393318.
23. Zhao, Z., & Welsch, R. E. (2026). Point-in-Time Financial RAG with Frozen LLMs and Market-Feedback Adaptive Retrieval. arXiv preprint arXiv:2605.31201.
24. Menon, H., Jekel, C. F., Korner, K., Gunnarson, B., Brown, N. K., Stees, M., ... & Belof, J. L. (2026). Multi-Agent Collaboration for Automated Design Exploration on High Performance Computing Systems. arXiv preprint arXiv:2603.11515.
25. Yuan, Y., Huang, J., Yin, J., & Xu, T. (2026). Confidence Calibration and Semantic Error Analysis for Ambiguous Coreference Resolution in User-generated Social Media Text. Available at SSRN 7393238.
26. Jain, N., Gu, A., Li, W. D., Yan, F., Zhang, T., Wang, S., ... & Stoica, I. (2025, May). Livecodebench: Holistic and contamination free evaluation of large language models for code. In International Conference on Learning Representations (Vol. 2025, pp. 58791-58831).
27. Zheng, J., & Makar, M. (2022). Causally motivated multi-shortcut identification and removal. *Advances in Neural Information Processing Systems*, 35, 12800-12812.
28. Jiao, Y., Shi, T., Zhao, B., & Wang, A. (2026). Retrieval-Guided Structured Reasoning and Interpretable Representation Learning for Large-Scale Video–Language Models.
29. El Makroum, R., Zwickl-Bernhard, S., & Kranzl, L. (2026). Agentic AI home energy management system: A large language model framework for residential load scheduling. *Results in Engineering*, 109857.
30. Liang, S., Du, Y., Chen, W., & Liu, Z. (2026). Order Allocation and Emergency Transportation Decisions Amid Multi-Source Procurement Disruptions.