Evaluating Large Language Models for Arduino Code Generation

Sardar K. Jabrw and Qusay I. Sarhan†

Department of Computer Science, College of Science, University of Duhok,
Duhok, Kurdistan Region – F.R. Iraq

***Abstract*—Large language models (LLMs), also known as generative AI, have transformed code generation by translating natural language prompts into executable code. Yet, their capabilities in generating code for resource-constrained devices such as Arduino, which are used in the Internet of Things and embedded systems, remained underexplored. This study evaluates six state-of-the-art LLMs for generating correct, efficient, and high-quality Arduino code. The evaluation was performed across five dimensions, namely functional correctness, runtime efficiency, memory usage, code quality, similarity to human-written code, and multi-round error correction. The results reveal that ChatGPT-4o achieves the highest zero-shot functional correctness and aligns closely with human code in readability and similarity. On the other hand, Gemini 2.0 Flash generates faster-executing code but at the cost of higher code complexity and lower similarity. DeepSeek-V3 balances correctness with superior flash memory optimization, whereas Claude 3.5 Sonnet struggles with prompt adherence. Finally, multi-round error correction improves correctness across all six models. Overall, the findings underscore that none of the evaluated LLMs consistently outperforms all evaluation criteria. Hence, model choice must align with project priorities; as shown, ChatGPT-4o excels in functional correctness, whereas Gemini 2.0 excels in execution time, and DeepSeek-V3 in memory efficiency. This study provides a systematic evaluation of code generated with LLMs for Arduino, which, to the best of our knowledge, has not been previously studied across multiple models and performance metrics, thereby establishing a foundation for future research and contributing to enhancing the trustworthiness and effectiveness of LLM-generated code.**

***Index Terms*—Large language models, Arduino, Code generation, Internet of Things, Code performance.**

I. Introduction

Large language models (LLMs) have rapidly advanced the field of code generation by enabling the automatic translation of natural language prompts into syntactically correct and executable code (Jiang, et al., 2024). These models have shown considerable promise in supporting developers with code generation (Koubaa, et al., 2023) (Sharma, 2024), documentation (Coello, Alimam and Kouatly, 2024) (Hou, et al., 2024), debugging (Nazir and Wang, 2023), and code translation (Rai, et al., 2024). While LLMs have shown promising results in general-purpose programming languages, their effectiveness in generating code for resource-constrained devices such as Arduino remains underexplored.

Arduino is a cheap, low-power, and small-sized computing device (Nayyar and Puri, 2016). It has been used for over a decade in a wide range of systems and applications such as Internet of Things (IoT), robotics, smart homes, and real-time control, including monitoring systems (Kim, Choi and Suh, 2020). Arduino is currently one of the most popular development platforms for prototyping and implementing IoT systems due to its simplicity, flexibility, affordability, and large community support (Yusro, Guntoro and Rikawarastuti, 2021). However, these devices have limited computing power and resources. Therefore, it is essential to program them efficiently when developing time-critical applications where every microsecond matters. In addition, efficient programming helps reduce energy consumption, which is an essential factor for battery-powered systems used primarily in IoT and wireless sensor networks. Moreover, optimized code allows developers to integrate more features and functionalities without exceeding the device's constraints.

To better understand the capabilities and limitations of LLMs for code generation, the study evaluates and investigates their capabilities in generating correct, efficient, and high-quality Arduino code. To this end, we evaluate six state-of-the-art models: ChatGPT-4o, Gemini 2 Flash, DeepSeek-V3, Claude 3.5 Sonnet, GitHub Copilot, and LLaMA-3 using 31 subject programs of Arduino coding tasks covering various aspects of coding capabilities, including data types, functions, structures, loops, arrays, and more, with a focus on code performance. We structure our analysis around five Research Questions (RQs), explained in Section III, that target the performance of the selected LLMs in terms of Overall Correctness, Code Performance, Multi-round Error Correction, Code Complexity, and Code Similarity. Our evaluation follows recent recommendations for multi-metric machine learning assessment (Abdullah, et al., 2025), considering correctness, performance, and complexity.

The results of the study reveal that ChatGPT-4o and DeepSeek-V3 offer a strong balance of correctness,

ARO-The Scientific Journal of Koya University
Vol. XIV, No. 1 (2026), Article ID: ARO.12344. 11 pages
DOI: 10.14500/aro.12344
Received: 11 June 2025; Accepted: 16 November 2025
Regular research paper; Published: 05 February 2026
†Corresponding author's e-mail: qusay.sarhan@uod.ac

maintainability, and adaptability to iterative refinement, whereas Gemini 2.0 Flash stands out in raw performance but at the cost of code readability and alignment with human coding conventions. In contrast, LLMs such as Claude 3.5 Sonnet demonstrate the importance of precise prompt interpretation, and GitHub Copilot, which has been trained on a large corpus of code, highlights that code-focused models do not guarantee superior output. The findings indicate that none of the evaluated LLMs outperform all evaluation criteria; consequently, trade-offs among functional correctness, performance, and code maintainability must be balanced. The findings inform several real-world applications. Specifically, automated code generation for resource-constrained devices such as Arduino is highly relevant to IoT, enabling rapid prototyping and robotics development, smart home automation, and industrial control through efficient and reliable integration of sensors and actuators. Moreover, the results guide developers in selecting models that balance code quality and development time, while highlighting limitations that inform future improvements to enhance the trustworthiness and effectiveness of LLM-generated code.

The key contributions of this paper are outlined as follows:

- A quantitative evaluation of the capabilities of six leading LLMs in generating Arduino code. To the best of our knowledge, it is the first study to evaluate LLMs' performance in generating correct, efficient, and quality code for Arduino across multiple models and performance metrics.
- The study introduces subject programs consisting of 31 optimized Arduino programs, covering various tasks to evaluate LLMs across various coding dimensions such as data types, functions, structures, loops, and arrays, focusing on code performance. The subject programs are publicly available on GitHub[1].
- It also contributes to advancing the potential knowledge and understanding capabilities of LLMs in improving automated code generation, with a particular focus on Arduino code generation, and establishing a foundation for future research in this swiftly developing field.

The organization of this paper is structured as follows: Section II presents an overview of the key related works. Section III describes the research methodology. Section IV presents the study's results and findings. Section V provides a discussion across different directions. Finally, the study's conclusions are provided in Section VI.

II. Related Works

This section briefly reviews the most relevant literature publications on the topic, and Table I summarizes them. The authors (Petrovic, Konicanin and Suljovic, 2023) explored the application of ChatGPT in embedded systems by integrating it with the Arduino platform. They conducted two case studies: They first used ChatGPT to perform classification tasks on sensor data collected through Arduino, and they secondly employed ChatGPT to generate template code for typical Arduino use cases automatically. The study highlighted ChatGPT's potential to accelerate development, though limitations in context retention and code correctness were noted. However, their work is limited to a single model (ChatGPT) and a small number of case studies, lacking the systematic, multi-model, and multi-metric comparative analysis required to understand the broader landscape of LLM capabilities for this domain. The authors (Su, et al., 2023) proposed a comprehensive evaluation framework to evaluate the code generation capabilities of LLMs, focusing on six dimensions: validity, correctness, complexity, reliability, security, and readability. They introduced the LLMC dataset, consisting of 45 Python-based coding problems, and applied it to four LLMs. Similarly, authors (Bucaioni, et al., 2024) conducted an empirical study to evaluate ChatGPT's capability in solving general programming problems using C++ and Java. They created a dataset covering various categories and difficulty levels, and through a structured experimental setup, they evaluated the correctness, runtime efficiency, and memory usage of ChatGPT-generated solutions, comparing them to those produced by human programmers. Moreover, authors (Miah and Zhu, 2024) evaluated ChatGPT's effectiveness as an R code generation tool using a user-centric approach. They evaluated code quality (accuracy, readability, and conciseness) and user experience. ChatGPT showed strong performance in generating accurate and readable code with clear explanations, but scored lower on conciseness. In a more recent study, authors (Palla and Slaby, 2025) conducted a comparative study evaluating LLMs for Python code generation. Their framework evaluated models such as OpenAI's GPT series, Google's Gemini, Meta's LLaMA, and Anthropic's Claude across 10 programming tasks of varying complexity, with evaluation criteria including syntax correctness, accuracy, reliability across multiple iterations, response time, cost, and exception handling. While these studies provide valuable methodologies for evaluating code quality, their focus on general-purpose languages such as Python, Java, and R means their findings are not directly transferable to the unique constraints of resource-constrained IoT devices, where memory usage and execution speed are critical. The authors (Kok, Demirci and Ozdemir, 2024) examined the integration of IoT and LLMs, outlining key applications in smart homes, healthcare, transportation, manufacturing, and environmental monitoring. They highlighted how LLMs enhance IoT systems through natural language interfaces and improved decision-making. The paper also addressed challenges such as resource limitations, latency, and privacy concerns, and suggested edge-cloud collaboration and model optimization as potential solutions. However, their work remains conceptual, outlining challenges like resource limitations without offering an empirical evaluation of code generation for resource-constrained IoT devices. The authors (DeLorenzo, Gohil and Rajendran, 2024) proposed CreativEval, a framework for assessing the creativity of LLM-generated HDL (Hardware Description

1 https://github.com/LLMsRes-ch/arduino-subject-programs/

TABLE I
SUMMARY OF RELATED WORKS ON LARGE LANGUAGE MODELS (LLMS) FOR CODE GENERATION

Paper	Target languages	Models evaluated	Datasets used	Prompting types	Metrics/Methods used
(Mirjalili, et al., 2025)	N/A	LLaMA-7B with custom adapters+MRP layers	GSM8K, MMLU, ScienceQA, Alpaca-52K, HotpotQA, LVLM-eHub	Meta-Reasoning Prompting (MRP).	Accuracy, Params, Training time, Strategy Switch Rate, BLEU@4, CIDEr
(Kok, Demirci and Ozdemir, 2024)	N/A	GPT, LLaMA, Claude, Gemini, Mistral, BERT (reviewed).	N/A (Literature Review)	Survey of prompting strategies.	Accuracy, latency, memory, power (reviewed).
(Su, et al., 2023)	Python	ChatGPT, Claude, Spark, Bing AI	LLMC Dataset (45 tasks).	Manual Standard/ Template-Based Prompting.	Validity, Correctness (pass rate), Complexity, Reliability/Security, Readability
(Shuvo, et al., 2025)	C++, Python	ChatGPT (GPT-4).	LeetCode (102 tasks), Codeforces (150 tasks)	Few-Shot+Iterative/ Multi-Round Prompting.	Accepted, Wrong Answer, Time Limit Exceeded, Runtime Error, Memory Limit Exceeded, Compile Error.
(Palla and Slaby, 2025)	Python	GPT series, Gemini series, LLaMA 3, Claude 3 series	10 custom coding tasks of varying complexity.	Standard/Template-based Prompting.	Syntax, Completeness, Response Time, Accuracy, Reliability, Exception-handling, Cost, Efficiency
(Bucaioni, et al., 2024)	C++, Java	ChatGPT (GPT-4).	LeetCode (240 tasks)	Efficiency/Conciseness Prompts.	Correctness, Runtime Efficiency, Memory Usage
(Liu, et al., 2024)	Java, C#	ChatGPT (GPT-3.5-Turbo).	CodeXGlue	Structured CoT+Template-Based Prompting	BLEU, CodeBLEU
(Petrovic, Konicanin and Suljovic, 2023)	Arduino	ChatGPT (GPT-4).	Two Arduino case studies.	In-Context/Zero-Shot Prompting.	Prediction time, Context size, Accuracy, Code gen time, Compile errors
(Paul, Zhu and Bayley, 2024)	Java	ChatGPT.	ScenEval dataset	Zero-Shot Prompting.	Pass@1, Avg Pass, Complexity (cyclomatic, cognitive, LOC)
(Miah and Zhu, 2024)	R	ChatGPT.	R programming tasks (351).	Iterative/Multi-Round User-Centric Prompting.	Usability attributes (Acc, Completeness, Conciseness, Readability, etc.), Attempts, Completion time
(DeLorenzo, Gohil and Rajendran, 2024)	Verilog	GPT-3.5, GPT-4, CodeLlama, VeriGen	HDLBits	Structured/ Creativity-Oriented Prompting	Fluency, Flexibility, Originality, Elaboration, Functionality, GNN4IP similarity
This Work	Arduino	ChatGPT-4o, Gemini 2.0 Flash, DeepSeek-V3, Claude 3.5 Sonnet, GitHub Copilot, LLaMA-3	31 performance-focused Arduino programs.	Zero-Shot/Role-Based Prompting.	Syntactic and Functional correctness, Execution time, SRAM, Flash Memory, Multi-round Error Correction, Cyclomatic Complexity, NCLOC, CodeBLEU.

Language) code, focusing on novelty and functionality. They evaluated solutions across diverse hardware design tasks and benchmarked responses from multiple LLMs, including ChatGPT and Code Llama. Their work demonstrates potential but is limited to only two LLMs, GPT-3 and GPT-4. Similarly, authors (Paul, Zhu and Bayley, 2024) introduced ScenEval, a benchmark tailored for scenario-based evaluation of the generated code. ChatGPT was evaluated on the benchmark using a range of metrics, including functional correctness, cyclomatic complexity, cognitive complexity, and the average number of attempts. However, their evaluation is also limited to a single LLM and focuses on general-purpose Java programming. Consequently, it lacks systematic, multi-model comparative analysis and does not address critical performance constraints – such as execution time. Recently, authors (Shuvo, et al., 2025) conducted an empirical study evaluating ChatGPT's code generation performance using several datasets. Their analysis revealed that ChatGPT achieved high accuracy on concise problem descriptions, whereas performance dropped significantly on narrative-driven problems, highlighting challenges in problem recognition and strategic planning under extended contexts. They further explored multiple programming languages (Python and C++), iterative prompting, and targeted feedback loops, demonstrating limited improvements in accuracy and efficiency through error-driven refinement. Despite these contributions, their work remains limited to a single LLM (ChatGPT-4) and focused primarily on accuracy and runtime performance. Broader aspects such as memory consumption and systematic multi-model comparisons remain unaddressed. Beyond code generation, recent research has explored enhancing LLMs for specialized tasks through improved prompting. For instance, authors (Liu, et al., 2024) conducted an empirical study on guiding ChatGPT for improved code generation using prompt engineering techniques. They evaluated ChatGPT's performance on two tasks – text-to-code and code-to-code generation – using the widely adopted CodeXGlue benchmark. Their methodology applied chain-of-thought (CoT) prompting and multi-step optimizations, exploring factors such as prompt specificity,

http://dx.doi.org/10.14500/aro.12344

conciseness, session settings, and generation randomness. Similarly, authors (Mirjalili, et al., 2025) integrated meta-reasoning prompting with adapter methods to boost the efficiency and task-adaptive reasoning of models such as LLaMA in multi-modal contexts. Their work underscores the importance of sophisticated prompt design and model fine-tuning for specialized domains, a consideration that aligns with the challenges of generating efficient code for resource-constrained environments.

While previous studies have investigated LLMs for general-purpose programming or domain-specific code generation (e.g., Python), and some have explored basic integrations with Arduino, none have systematically evaluated LLMs for Arduino code generation. Unlike prior research, which focused on single models, limited task types, or a narrow set of metrics, our work provides a comparative analysis of six state-of-the-art LLMs across 31 performance-focused Arduino coding tasks. Beyond functional correctness, we measure execution time, memory usage, code complexity, and code similarity to reference code. Our findings reveal each model's trade-offs between speed, efficiency, and maintainability, contributing to improvements in the reliability and efficiency of LLMs for code generation in resource-constrained applications.

III. Methodology

The methodology employed in this study consists of seven steps: identifying research questions, preparing the subject programs, selecting LLMs, defining evaluation metrics, prompting the models to generate codes, testing codes, and analyzing the results to evaluate LLMs' capabilities in generating correct, efficient, and high-quality Arduino codes, with each step described in detail in the following sections.

A. RQs

Several RQs have been identified and addressed in this study, each focusing on a specific aspect of the topic as follows.

- RQ1 (Overall Correctness): Is the code generated by LLMs syntactically and functionally correct?
- RQ2 (Code Performance): How efficient is the generated code in terms of runtime and memory usage?
- RQ3 (Multi-round Error Correction): Can iterative prompting improve the correctness of initially incorrect code?
- RQ4 (Code Complexity): Does the generated code reflect the same level of complexity as code written by human developers?
- RQ5 (Code Similarity): How similar is the generated code to the original developer-implemented code?

B. Subject Program Preparation

The subject programs, consisting of 31 optimized Arduino programs, were constructed using coding examples from the official Arduino reference website[2] to cover all programming constructions. The subject programs cover a wide range of Arduino fundamentals, instructions, and tasks, where for each program task, the authors prepared two reference solutions to reflect real-world developer variation and to ensure we explored the best possible approaches, as it was not always clear which coding strategy would perform better. Some programs were more efficient in terms of memory usage, whereas others offered lower execution times. The reference solutions were developed with a strong emphasis on performance, and manual optimization was applied to ensure they were efficient, reliable, and of high quality, with each code task accompanied by a specific zero-shot prompt (Li, et al., 2024) (directly ask the model to perform a task without providing any examples) each clearly defines the role and expertise level, specifies the board, states the task (code question), and requests optimized code for best performance. The prompt is structured as follows:

> *"You are an experienced software developer. Write an optimized Arduino Uno Rev3 code that [code question]. Only write the code and ensure it's optimized for best performance."*

An example prompt and the code generated by ChatGPT are shown in Fig. 1.

C. LLMs Selection

LLMs are trained on existing datasets, and in the simplest terms, they are a black box that solves the problem of predicting the next word in a sequence. To evaluate their capabilities and limitations in code generation, we selected six of the most well-known models available: ChatGPT-4o[3] by OpenAI, Gemini 2.0 Flash[4] by Google, DeepSeek-V3[5] by DeepSeek, Claude 3.5 Sonnet[6] by Anthropic, GitHub Copilot[7] (powered by ChatGPT-4o) by OpenAI in collaboration with GitHub, and LLaMA-3[8] by Meta. Each of these models is developed and maintained by a leading tech company, ensuring a diverse representation of current advancements in LLMs.

D. Evaluation Metrics

Our research follows a quantitative approach, and to ensure the accuracy and reliability of our results, we selected a set of well-known evaluation metrics, including Cyclomatic Complexity (Ebert, et al., 2016) and CodeBLEU (Ren, et al., 2020). In addition, we introduce new metrics used for the first time in this context, including SRAM (Static Random Access Memory) usage and Flash memory usage. These metrics are chosen carefully to match each RQ and to provide a clear understanding of the performance and effectiveness of our methods.

To answer RQ1 (Overall Correctness), we evaluate both syntactic and functional correctness (Moradi Dakhel,

2 https://www.arduino.cc/reference/en/

3 https://chatgpt.com/
4 https://gemini.google.com/
5 https://www.deepseek.com/
6 https://claude.ai/
7 https://github.com/copilot
8 https://www.meta.ai/

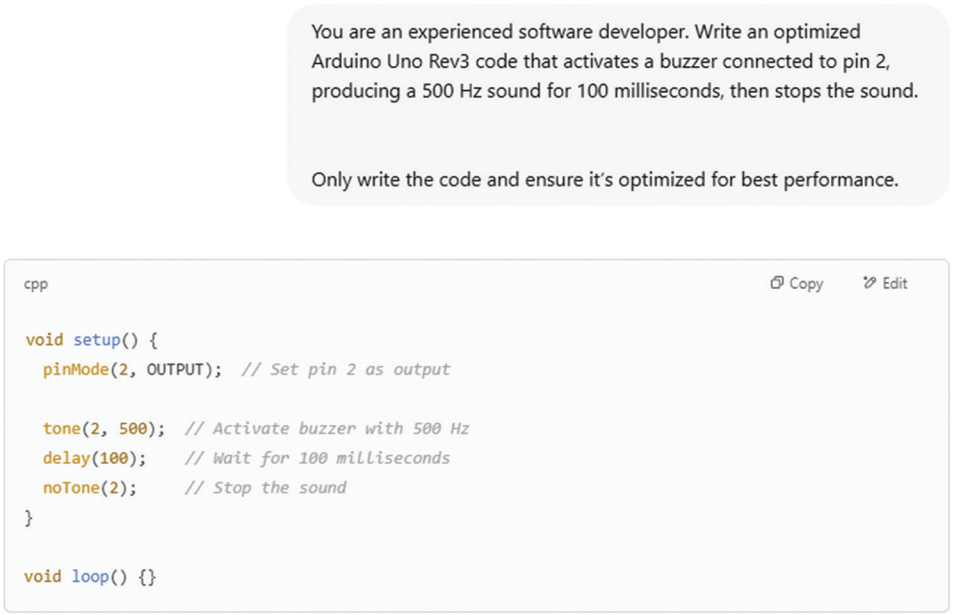

Fig. 1. Example of prompt and code solution generated by ChatGPT.

et al., 2023) of the generated codes. Syntactic correctness is verified using the Arduino compiler to detect syntax mistakes, undeclared variables, missing semicolons, etc., while functional correctness is evaluated by executing the generated code and comparing its outputs with expected results. If the initial attempt generated correct code, it was marked successful; otherwise, it was marked failed.

To answer RQ2 (Code Performance), we compare the performance of the generated code with our reference code. The performance of each code was measured in terms of execution time/runtime (Niu, et al., 2024) (the amount of time required to execute the code), flash memory usage (the space required to store the sketch that executes the code), and SRAM (Static Random Access Memory) usage (the space required for the sketch that executes the code to create and manipulate variables when it runs). The Arduino IDE was used to measure the amount of SRAM and flash memory usage, whereas execution time was measured in microseconds. Furthermore, each experiment was repeated for two runs: the first run (automatically triggered through the Arduino IDE upload process) was treated as a warm-up and discarded to reduce measurement noise, whereas the second run was obtained by manually pressing the board's reset button and used as the recorded measurement. Furthermore, input/output operations (e.g., Serial.print()) were isolated from timing to ensure that only the core computation was measured. The execution time measurement was determined using the pseudo-code shown below. The pseudo-code was added to all codes before execution to obtain execution time, Flash memory, and SRAM. Any code that executes faster, uses less flash memory, and consumes less SRAM is considered the best and outperforms the others. However, not all codes perform well in all metrics. Therefore, we have compared each metric individually.

Execution time measurement pseudo-code:

- Begin
- Get start time
- Execute the code
- Get end time
- Elapsed time = (end time - start time)
- End

To answer RQ3 (Multi-round Error Correction), we instruct the models to regenerate code that did not pass the overall correctness in RQ1. We aimed to complete each task in as few attempts as possible. If the code generated by LLM is incorrect, the second attempt is made using compiler error feedback or author feedback. In case the output is incorrect, the process continues until either the number of attempts reaches the maximum of five or a correct solution is obtained. Therefore, it leads to the #attempt metric (Miah and Zhu, 2024), which represents the number of times the user prompted LLMs to generate a correct solution.

To answer RQ4 (Code Complexity), we compare the complexity of the generated code to the reference code. Code complexity is a critical factor that significantly affects readability, maintainability, and overall code quality (Tashtoush, et al., 2023). Highly complex code can be harder to understand (readability) and maintain. Hence, code quality cannot be assessed solely on correctness (RQ1) or performance (RQ2); it requires an understanding of the structural complexity and adherence to established standards (Clark, et al., 2024). For this RQ, we used the following metrics:

- Cyclomatic Complexity (CC) (Ebert, et al., 2016): It measures the number of linearly independent paths in a program, which directly correlates with the number of decision points in the code. In addition, lower cyclomatic

complexity is generally considered a sign of higher code quality, which is calculated using Equation 1.

$$M = E - N + 2P \quad (1)$$

Where:
- M is the CC of a program or function.
- E is the number of edges.
- N is the number of nodes.
- P is the number of connected components or exit points.

- Non-Comment Lines of Code (NCLOC) (Beurer-Kellner, Vechev and Fischer, 2023): It measures the number of executable lines in a code, excluding comments and empty lines, which is often used to estimate development effort, cost, and productivity. Therefore, it supports defect analysis and maintainability assessment (Nuñez-Varela, et al., 2017). Furthermore, functions with higher NCLOC are typically more complex and harder to maintain, whereas shorter functions improve readability and ease of testing.

It is worth mentioning that both CC and NCLOC metrics were computed using the Lizard tool provided by (Yin, 2024).

To answer RQ5 (Code Similarity), we compare the generated code to our reference code to derive additional information about the difference between the two codes by employing the CodeBLEU (Bilingual Evaluation Understudy) score (Ren, et al., 2020). CodeBLEU is a composite metric with the scores being the weighted average of 4 different sub-metrics treating code differently: n-gram matching (BLEU), syntax match, data flow match, and semantic match (Evtikhiev, et al., 2023). In addition, it is a widely adopted similarity metric, and it can be calculated using Equation 2.

$$\text{CodeBLEU} = \alpha\cdot\text{BLEU} + \beta\cdot\text{Syntax Match} + \gamma\cdot\text{Data Flow Match} + \delta\cdot\text{Semantic Match} \quad (2)$$

Where:
- BLEU: measures n-gram overlap between generated and reference code.
- Syntax Match: compares abstract syntax trees (AST).
- Data Flow Match: evaluates the consistency of variable usage and dependencies.
- Semantic Match: assesses code structure and functionality.

The weights α, β, γ, and δ are tunable hyperparameters, allowing flexibility based on the importance of each component in a specific context.

A value of 0 in CodeBLEU indicates no similarity between the generated and reference code, whereas a value closer to 1 indicates a high degree of similarity. We compute this metric using the tool provided by CodeXGLUE (Lu, et al., 2021).

E. Prompting, Testing, and Analyzing

Fig. 2 shows an overview of our evaluation procedure. Our experiments begin by retrieving a prompt (code question) from our subject programs and passing it to the models to generate an initial code solution. Using the same Arduino hardware and Arduino IDE software with its default configurations and settings, the solution is assessed for overall correctness, including syntactic and functional correctness. If the solution fails the correctness check, the model is re-prompted using feedback—compiler feedback in the case of syntactic errors or author feedback in the case of functional errors. This process is repeated up to five times if the model continues to generate incorrect solutions. If a solution passes the overall correctness check, then analyze it in terms of code performance and complexity using the Arduino device. Otherwise, it is only evaluated for code similarity and multi-round error correction. The same Arduino device, which is the original Arduino Uno Rev3 and the most commonly used device, has been employed in all tests to ensure consistent hardware specifications[9], with the Arduino IDE configured as specified in Table II. All code generation experiments were conducted during April 2025 using the default configuration settings (e.g., temperature) of each model.

IV. Experimental Results

This section presents and discusses the experimental results of this study, where each RQ is summarized with a short title and discussed in its respective subsection based on the study's findings.

A. Overall Correctness (RQ1)

We evaluate the syntactic and functional correctness success rate of all generated codes in the first iteration (zero-shot ability). As shown in Fig. 3, we found an impressive performance by ChatGPT-4o, achieving a remarkable 96.8% correctness rate on first-iteration solutions. Similarly, DeepSeek-V3 follows with a strong performance of 90.3%, whereas GitHub Copilot attains 87.1%. Furthermore, Gemini 2.0 Flash and LLaMA-3 both achieve 80.6%. On the other hand, Claude 3.5 Sonnet shows the lowest performance among the evaluated models, with a correctness rate of 67.7%. When looking at the errors in the generated codes, most were functional errors (logical errors), either incorrect outputs or included unnecessary infinite loops, indicating that models are very effective in generating syntactically correct code and rarely make syntax errors.

B. Code Performance (RQ2)

In this RQ, we aim to investigate how the performance of code generated by LLMs compares to our reference code. Performance is a crucial factor in systems with limited resources, such as Arduino devices, which are constrained by limited resources and computing capabilities. Hence, we evaluate and compare the generated and reference code based on three key performance metrics: execution time, flash memory usage, and SRAM usage. Fig. 4 compares execution time for all correct code generated by LLMs in the first iteration. The execution time was categorized into three outcomes:
- Equal (blue bars): The generated code had the same execution time as the reference code

9 https://store.arduino.cc/products/arduino-uno-rev3

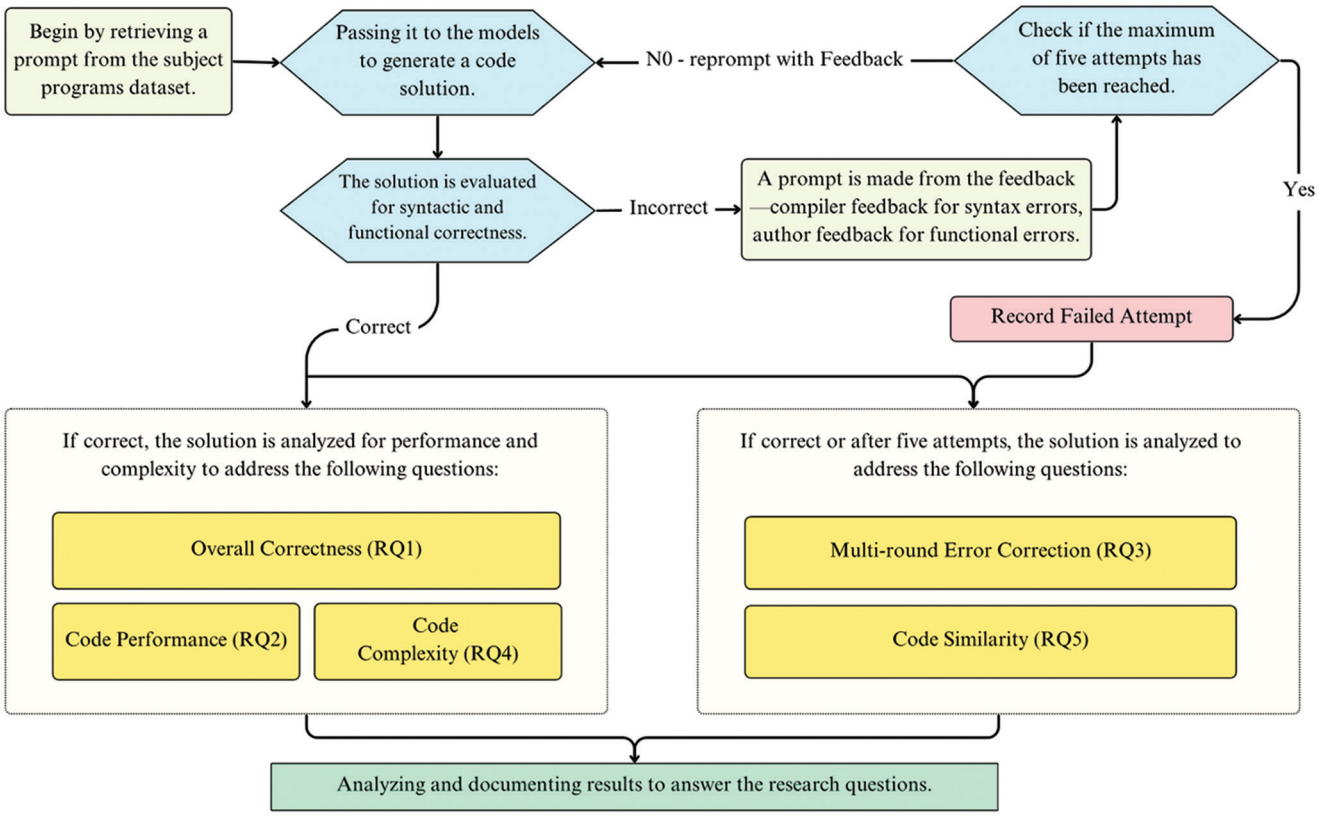

Fig. 2. The evaluation procedure used.

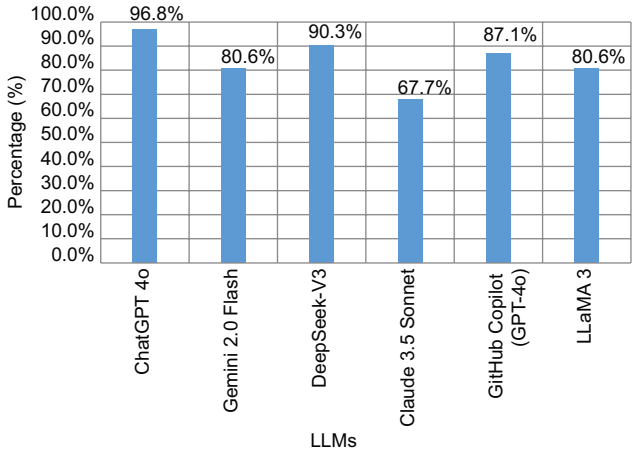

Fig. 3. Zero-shot overall correctness (RQ1) rate (%) of a large language model.

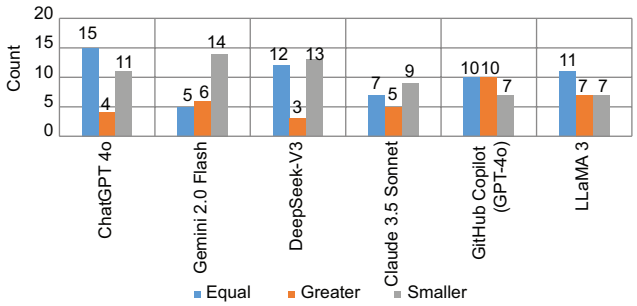

Fig. 4. Comparison of execution time outcomes: Equal, greater, or smaller than reference code.

- Greater (orange bars): generated code had a longer execution time
- Smaller (gray bars): generated code had a shorter execution time.

From Fig. 4, it is observed that ChatGPT-4o shows the highest number of codes matching the execution time of our reference codes, whereas Gemini 2.0 Flash performed strongly, which shows a higher number of codes where execution time is shorter. In addition, DeepSeek-V3 performed well with equal and smaller execution times. Interestingly, Claude 3.5 Sonnet shows a higher number of codes where execution time is shorter, but overall correctness (RQ1) is the poorest among the models. GitHub Copilot had an even distribution between equal and greater execution times, whereas LLaMA-3 presented a balanced number of greater and smaller codes.

TABLE II
ARDUINO IDE SPECIFICATIONS

Platform	Specifications	Detail
Arduino IDE	IDE version	2.3.5
	GCC compiler version	7.3.0
	GCC compiler optimization levels	Os (default)

Fig. 5 shows the comparison of flash memory usage for all generated codes in the first iteration, which are categorized into three outcomes: Equal, greater, and smaller, similar to what we did for execution time.

http://dx.doi.org/10.14500/aro.12344

The results show that DeepSeek-V3 has strong optimization capability with the highest number of codes with smaller flash memory usage than reference codes. In comparison, models such as Gemini 2 Flash and GitHub Copilot exhibited more code with higher flash memory usage. ChatGPT-4o presents a relatively balanced distribution, whereas LLaMA-3 maintains an even spread across equal, greater, and smaller memory usage categories.

Fig. 6 compares SRAM usage across evaluated LLMs, categorizing outcomes into equal, greater, or smaller usage.

Based on the results shown, ChatGPT-4o demonstrates the highest number of codes where the generated code exhibits equal SRAM usage compared to the reference code. followed closely by DeepSeek-V3 and LLaMA-3. Furthermore, GitHub Copilot performed well. In contrast, Gemini 2.0 Flash exhibits a more varied distribution with a higher number of greater SRAM usages, whereas Claude 3.5 Sonnet shows fewer codes of reduced SRAM usage compared to other models.

C. Multi-attempt Code Correction (RQ3)

We examine the effectiveness of the multi-attempt correction process in enhancing code generation for functional correctness across all models. Since all models support multi-attempt conversations, we instruct them to regenerate code snippets up to five times if they fail during the functional correctness test in RQ1.

As shown in Fig. 7, all models reduced incorrect outputs over attempts. Claude 3.5 Sonnet started with the highest number of incorrect codes (10) but managed to reduce them to just one by the second attempt, maintaining that level through the fifth. Similarly, Gemini 2.0 Flash and LLaMA-3 showed consistent improvement, reaching a minimum of one incorrect output by the fifth attempt. DeepSeek-V3 and ChatGPT-4o achieved full correction by the second attempt, whereas GitHub Copilot quickly dropped from four to one by the second attempt. These results indicate that multi-attempt code correction is effective in improving code correctness, particularly within the first few iterations.

D. Code Complexity (RQ4)

Code complexity is a critical factor that significantly affects readability, maintainability, and overall code quality (Tashtoush, et al., 2023). Hence, highly complex code can be harder to understand (readability) and maintain. The results for Cyclomatic Complexity are shown in Fig. 8, where most LLMs, including ChatGPT-4o, Claude 3.5 Sonnet, and LLaMA-3, tend to generate code with low

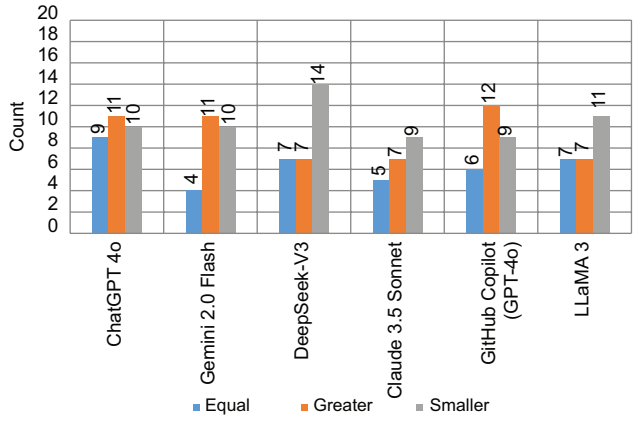

Fig. 5. Comparison of flash memory usage: Equal, greater, or smaller than reference.

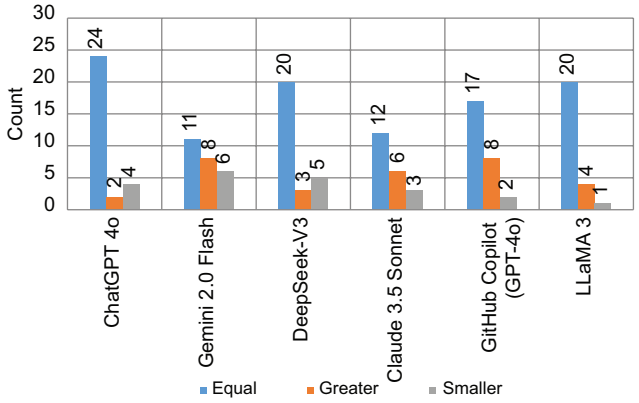

Fig. 6. Comparison of SRAM usage: Equal, greater, or smaller than reference.

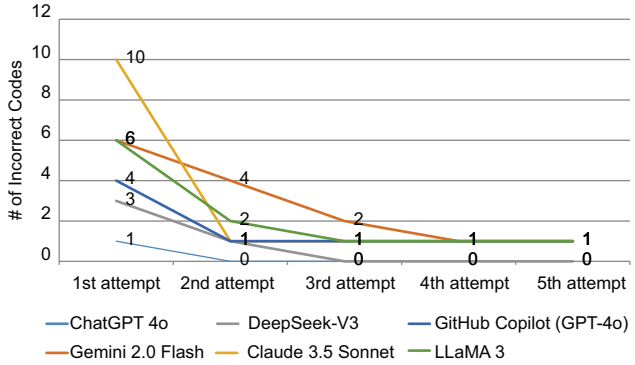

Fig. 7. Multi-attempt code fixing progress across a large language model (RQ3).

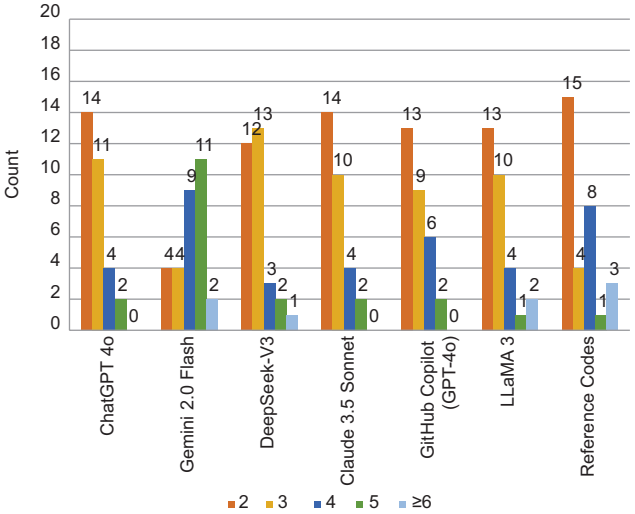

Fig. 8. Distribution of cyclomatic complexity values.

cyclomatic complexity (values of 2 and 3), closely mirroring our reference code, indicating that these models generate relatively simple and maintainable logic structures. In contrast, Gemini 2.0 Flash stands out by generating a higher number of codes with complexity levels of 4 and 5, which reduces code readability and maintainability. In addition, a high cyclomatic complexity can potentially lead to a high probability of errors and bugs in the code.

Fig. 9 categorizes the NCLOC into five ranges. The results show that most models have generated short codes in the 6–10 line range, including ChatGPT-4o, DeepSeek-V3, and LLaMA-3. Claude 3.5 Sonnet and GitHub Copilot generated longer codes, with most in the 6–10 and 11–15 line range. However, Gemini 2.0 GPT-4 not only has more lines but also more complex codes compared to other models and the reference code. Overall, these results highlight that while some LLMs align closely with the reference in terms of code length and complexity (ChatGPT-4o, DeepSeek-V3, and LLaMA-3), others vary significantly (Gemini 2.0 Flash, Claude 3.5 Sonnet, and GitHub Copilot).

E. Code Similarity (RQ5)

To derive additional information about the difference between the generated and reference code, we computed the CodeBLEU score. Fig. 10 shows the CodeBLEU scores between the reference and generated codes by all models. The CodeBLEU scores vary across models, with ChatGPT-4o and LLaMA-3 achieving the highest median values, indicating that their outputs are most similar to reference codes. In contrast, Gemini 2.0 Flash shows the lowest median and a narrower interquartile range, reflecting greater deviations from the reference. Moreover, other models such as DeepSeek-V3, Claude 3.5 Sonnet, and GitHub Copilot show moderate performance with varying levels of consistency. These results suggest that while some models (ChatGPT-4o, LLaMA-3) are adept at mimicking human-like code patterns, others (Gemini 2.0 Flash) generate code with significant structural or semantic deviations, potentially impacting software quality – notably through reduced readability and maintainability.

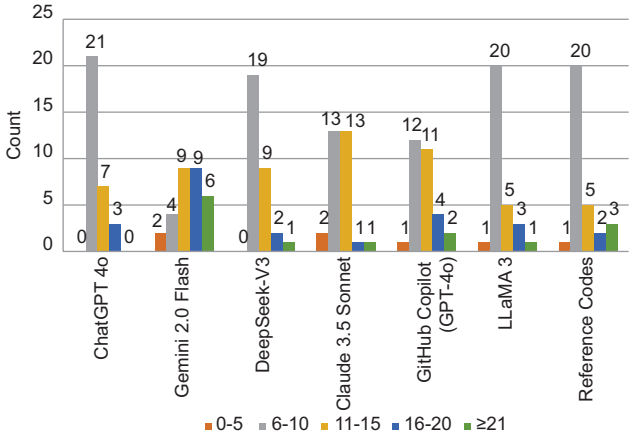

Fig. 9. Distribution of non-comment lines of code.

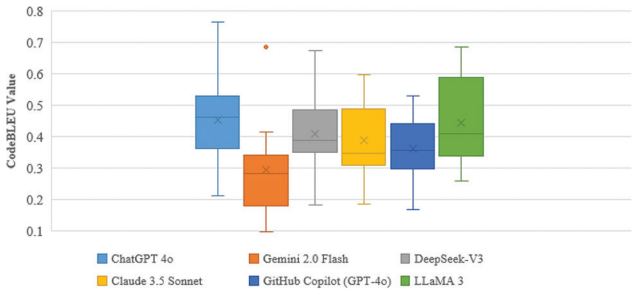

Fig. 10. Distribution of CodeBLEU scores (0–1 scale) between reference and generated codes.

V. Discussion

Our evaluation of state-of-the-art LLMs on Arduino code generation reveals nuanced trade-offs between correctness, performance, and code quality. Although ChatGPT-4o achieved the highest zero-shot functional correctness rate (96.8%), its generated code consistently has longer execution times than Gemini 2.0 Flash, indicating that there is no positive correlation between functional correctness and code efficiency. In contrast, DeepSeek-V3 has a zero-shot functional correctness rate of 90.3%, less than ChatGPT-4o, which delivered significantly lower execution times and reduced Flash memory consumption compared to ChatGPT-4o. Claude 3.5 Sonnet has the lowest zero-shot functional correctness rate of 67.7%. Error analysis revealed that many of its generated codes appended unnecessary infinite loops or the code was inside Arduino's continuously running loop(), even when the prompt did not require it. In addition, despite using the same zero-shot prompts across all models, Claude 3.5 Sonnet showed the least ability to interpret and adhere to the prompt constraints. GitHub Copilot achieved a zero-shot functional correctness rate of 87.1%, ranking third behind ChatGPT-4o and DeepSeek-V3. Finally, although Copilot was trained on a large corpus of code from GitHub and explicitly designed for programming tasks (Jiang, et al., 2024), it generated less efficient code compared to other models, which are trained on a broader range of text data and have a better understanding of the overall prompt context.

The effectiveness of multi-attempt error correction across all models highlights the potential of iterative refinement in practical development workflows. Both ChatGPT-4o and DeepSeek-V3 were able to resolve 100% of their initial failures by the second attempt (Miah and Zhu, 2024), and all models corrected the majority of their errors within five rounds. These results suggest that integrating compiler or developer feedback into prompts can be a powerful strategy to overcome initial model limitations.

Code complexity metrics further highlight the trade-off between maintainability and performance. Models such as ChatGPT-4o, LLaMA-3, and Claude 3.5 Sonnet consistently generated code with cyclomatic complexity values in the 2–3 range, comparable to our code. This lower complexity enhances readability and reduces the cognitive load required for developers to understand and maintain the code. In contrast, performance-focused models such as Gemini 2.0 Flash tended to generate code with higher complexity (≥4) and

 http://dx.doi.org/10.14500/aro.12344

TABLE III
PERFORMANCE OF LARGE LANGUAGE MODELS (LLMS) IN ARDUINO CODE GENERATION (RQ1–RQ5)

Model	Overall Correctness (RQ1)	Execution Time (RQ2)	Flash Usage (RQ2)	SRAM Usage (RQ2)	Multi-Attempt Fixing (RQ3)	Cyclomatic Complexity (RQ4)	Code Length (NCLOC) (RQ4)	Code Similarity (RQ5)
ChatGPT-4o	96.8% (Best)	Mostly Equal	Balanced (Equal/ Greater/Smaller)	Mostly Equal (Best)	Fixed All by 2nd Attempt	Low	Short	Most similar (Best)
DeepSeek-V3	90.3%	Equal/Smaller	Most Memory Efficient (Best)	Mostly Equal	Fixed All by 2nd Attempt	Low	Short	Mostly Similar
GitHub Copilot	87.1%	Balanced (Equal/ Greater/Smaller)	Mostly Greater	Mostly Equal	Needed up to 2nd Attempt	Low/High	Medium	Less Similar
Gemini 2.0 Flash	80.6%	Mostly Smaller (Best)	Mostly Greater	Equal/Greater	Fixed Most by 5th Attempt	High	Longer	Least Similar
LLaMA-3	80.6%	Mostly Equal	Mostly Smaller	Mostly Equal	Fixed Most by 5th Attempt	Low	Short	Mostly Similar
Claude 3.5 Sonnet	67.7%	Equal/Smaller	Mostly Smaller	Mostly Equal	Fixed by 2nd Attempt	Low	Short/Medium	Less Similar

longer lines, often due to optimizations such as loop unrolling or generating custom functions instead of using built-in functions or libraries. Although these strategies can improve execution speed, they also increase code complexity, raising concerns about code understandability and maintainability.

Our CodeBLEU similarity analysis reinforces these findings by clearly highlighting differences in how closely the models align with human coding practices. ChatGPT-4o and LLaMA-3 achieved the highest CodeBLEU scores, indicating strong structural and semantic similarity to the reference code. This alignment translated not only into readable and maintainable code but also into comparable performance and code complexity. In contrast, although Gemini 2.0 Flash produced longer and more complex code with faster execution times, its substantially lower CodeBLEU scores reveal a significant departure from the reference code.

Overall, our comprehensive evaluation underscores that none of the evaluated LLMs excel across all dimensions of Arduino code generation. Table III clearly compares models across correctness, performance, error correction, complexity, and similarity to reference code. These findings emphasize the importance of selecting models based on a project's priorities – correctness, performance, maintainability, or alignment with human coding practices.

VI. Threats to Validity

There are some potential threats affecting the validity of our experimental results and conclusions.

- External Validity. In this study, only six LLMs were selected. Therefore, the results cannot be generalized to all LLMs, nor to future model versions. However, these models were carefully chosen based on their popularity and recent advancements in the field. Another threat is that commercial LLMs are closed and may be updated or fine-tuned without notice, leading to hosted-model drift and potential differences in output over time. Both the model versions and the access data have been clearly reported to support transparency and reproducibility.
- Internal Validity. One potential threat is that using only Arduino tasks and a specific set of 31 programs may introduce a single-board bias, potentially affecting the causal interpretation of results. This threat was mitigated by ensuring consistent prompts, using the same Arduino board, and using identical IDE settings across all experiments.
- Construct Validity. The study employed multiple metrics, including functional correctness, execution time, cyclomatic complexity, and CodeBLEU, using established tools such as Lizard and CodeXGLUE to ensure reliability. However, some limitations remain: CodeBLEU, originally designed for general-purpose languages, may not fully capture semantic or logical equivalence in Arduino code, and cyclomatic complexity may not reflect all aspects of code complexity. These considerations were carefully acknowledged, and the use of well-defined and diverse metrics ensures that the evaluation meaningfully captures Arduino code generation performance.

VII. Conclusion

This study provides a comprehensive evaluation of LLMs for Arduino code generation, highlighting each model's strengths and limitations in producing efficient, maintainable, and human-like code. Our analysis of six state-of-the-art models shows that none of the evaluated LLMs excel across all dimensions: ChatGPT-4o leads in zero-shot functional correctness and alignment with human coding practices, whereas Gemini 2.0 Flash excels in runtime efficiency at the expense of readability, and DeepSeek-V3 demonstrates strong potential for memory-optimized applications through efficient flash memory usage. The study also identifies limitations in current LLMs, including prompt adherence problems in Claude 3.5 Sonnet and unexpected inefficiencies in GitHub Copilot despite its domain-specific training.

For future work, this research can be extended to other resource-constrained devices, such as Raspberry Pi, ESP32, or ARM-based microcontrollers, and to programming languages that target hardware devices, such as MicroPython. Additional studies could investigate optimization techniques to further improve code correctness, efficiency, and maintainability. By addressing these directions, future research can advance the potential of LLMs as tools for efficient and automated microcontroller programming used in IoT and embedded systems, covering many aspects of daily life.

References

Abdullah, A.A., Mohammed, N.S., Khanzadi, M., Asaad, S.M., Abdul, Z.K., and Maghdid, H.S., 2025. In-depth analysis on machine learning approaches: Techniques, applications, and trends. *The Scientific Journal of Koya University*, 13(1), pp.190-202.

Beurer-Kellner, L., Vechev, M., and Fischer, M., 2023. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7, pp.1946-1969.

Bucaioni, A., Ekedahl, H., Helander, V., and Nguyen, P.T., 2024. Programming with ChatGPT: How far can we go? *Machine Learning with Applications*, 15, p.100526.

Clark, A., Igbokwe, D., Ross, S., and Zibran, M.F., 2024. A Quantitative Analysis of Quality and Consistency in AI-Generated Code. In: *Proceedings - 2024 7th International Conference on Software and System Engineering, ICoSSE 2024*. Institute of Electrical and Electronics Engineers Inc., pp.37-41.

Coello, C.E.A., Alimam, M.N., and Kouatly, R., 2024. Effectiveness of ChatGPT in coding: A comparative analysis of popular large language models. *Digital*, 4(1), pp.114-125.

DeLorenzo, M., Gohil, V., and Rajendran, J., 2024. CreativEval: Evaluating Creativity of LLM-Based Hardware Code Generation. In: *Conference: 2024 IEEE LLM Aided Design Workshop (LAD)*. pp.1-5.

Ebert, C., Cain, J., Antoniol, G., Counsell, S., and Laplante, P., 2016. Cyclomatic complexity. *IEEE Software*, 33(6), pp.27-29.

Evtikhiev, M., Bogomolov, E., Sokolov, Y., and Bryksin, T., 2023. Out of the BLEU: How should we assess quality of the Code Generation models? *Journal of Systems and Software*, 203, p.111741.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H., 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), pp.1-79.

Jiang, J., Wang, F., Shen, J., Kim, S. and Kim, S., 2024. A survey on large language models for code generation. *arXiv*, arXiv:2406.00515. [Last accessed on 2025 Apr 25].

Kim, S.M., Choi, Y., and Suh, J., 2020. Applications of the open-source hardware arduino platform in the mining industry: A review. *Applied Sciences*, 10, 5018.

Kok, I., Demirci, O., and Ozdemir, S., 2024. When IoT Meet LLMs: Applications and Challenges. In: *2024 IEEE International Conference on Big Data (BigData)*. Los Alamitos, CA, USA: IEEE Computer Society. pp.7075-7084.

Koubaa, A., Qureshi, B., Ammar, A., Khan, Z., Boulila, W., and Ghouti, L., 2023. Humans are still better than ChatGPT: Case of the IEEEXtreme competition. *Heliyon*, 9(11), p.e21624.

Li, J., Li, G., Li, Y., and Jin, Z., 2024. Structured Chain-of-Thought Prompting for Code Generation. *ACM Transactions on Software Engineering and Methodology*, 34, pp.1-23.

Liu, C., Bao, X., Zhang, H., Zhang, N., Hu, H., Zhang, X., and Yan, M., 2024. Guiding ChatGPT for Better Code Generation: An Empirical Study. In: *Proceedings - 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024*. Institute of Electrical and Electronics Engineers Inc. pp.102-113.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D. and Li, G., 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv*, arXiv:2102.04664. [Last accessed on 2025 Apr 25].

Miah, T., and Zhu, H., 2024. User Centric Evaluation of Code Generation Tools (Invited Paper). In: *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. Los Alamitos, CA, USA: IEEE Computer Society. pp.109-119.

Mirjalili, S., Abdulla, A.A., Hassan, B.A., and Rashid, T.A., 2025. LLaMA-Adapter + MRP: Integrating Meta-Reasoning Prompting with LLaMA-Adapter for Efficient Multi-Modal and Task-Adaptive Reasoning. TechRxiv, June 18.

Moradi Dakhel, A., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M.C., and Jiang, Z.M. (Jack), 2023. GitHub Copilot AI pair programmer: Asset or Liability? *The Journal of Systems and Software*, 203(C), p.111734.

Nayyar, A., and Puri, V., 2016. A review of Arduino board's, Lilypad's & Arduino shields. In: *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*. pp.1485-1492.

Nazir, A., and Wang, Z., 2023. A comprehensive survey of ChatGPT: Advancements, applications, prospects, and challenges. *Meta-Radiology*, 1, p.100022.

Niu, C., Zhang, T., Li, C., Luo, B., and Ng, V., 2024. On Evaluating the Efficiency of Source Code Generated by LLMs. In: *Proceedings - 2024 IEEE/ACM 1st International Conference on AI Foundation Models and Software Engineering, FORGE 2024*. Association for Computing Machinery, Inc. pp.103-107.

Nuñez-Varela, A.S., Pérez-Gonzalez, H.G., Martínez-Perez, F.E., and Soubervielle-Montalvo, C., 2017. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128, pp.164-197.

Palla, D., and Slaby, A., 2025. Evaluation of generative AI models in python code generation: A comparative study. *IEEE Access*, 13, pp.65334-65347.

Paul, D.G., Zhu, H., and Bayley, I., 2024. ScenEval: A Benchmark for Scenario-Based Evaluation of Code Generation. In: *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE. pp.55-63.

Petrovic, N., Konicanin, S., and Suljovic, S., 2023. ChatGPT in IoT Systems: Arduino Case Studies. In: *2023 IEEE 33rd International Conference on Microelectronics, MIEL 2023*. Institute of Electrical and Electronics Engineers Inc., pp.1-4.

Rai, L., Khatiwada, S., Deng, C., and Liu, F., 2024. Cross-Language Code Development with Generative AI: A Source-to-Source Translation Perspective. In: *2024 IEEE 7th International Conference on Electronic Information and Communication Technology, ICEICT 2024*. Institute of Electrical and Electronics Engineers Inc., pp.562-565.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A. and Ma, S., 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv*, arXiv:2009.10297. [Last accessed on 2025 Apr 25].

Sharma, T., 2024. LLMs for Code: The Potential, Prospects, and Problems. In: *Proceedings - IEEE 21st International Conference on Software Architecture Companion, ICSA-C 2024*. Institute of Electrical and Electronics Engineers Inc. pp.373-374.

Shuvo, U.A., Dip, S.A., Vaskar, N.R., and Al Islam, A.B.M.A., 2025. Assessing ChatGPT's Code Generation Capabilities with Short vs Long Context Programming Problems. In: *Proceedings of the 2024 11th International Conference on Networking, Systems and Security, NSysS 2024*. Association for Computing Machinery, Inc. pp.32-40.

Su, H., Ai, J., Yu, D., and Zhang, H., 2023. An Evaluation Method for Large Language Models' Code Generation Capability. In: *Proceedings - 2023 10th International Conference on Dependable Systems and Their Applications, DSA 2023*. Institute of Electrical and Electronics Engineers Inc. pp.831-838.

Tashtoush, Y., Abu-El-Rub, N., Darwish, O., Al-Eidi, S., Darweesh, D., and Karajeh, O., 2023. A notional understanding of the relationship between code readability and software complexity. *Information (Switzerland)*, 14(2), 81.

Yin, T., 2024. *Lizard: A Simple Code Complexity Analyser without Caring about the c/c++ Header Files or Java Imports, Supports Most of the Popular Languages*. pp. 21-27. Available from: https://github.com/terryyin/lizard [Last accessed on 2025 Apr 25].

Yusro, M., Guntoro, N., and Rikawarastuti, R., 2021. Utilization of microcontroller technology using Arduino board for Internet of Things (a systematic review). *AIP Conference Proceedings*, 2331, p.060004.