

Article

Benchmarking Large Language Models for Embedded Systems Programming in Microcontroller-Driven IoT Applications

Marek Babiuch *  and Pavel Smutný * 

Department of Control Systems and Instrumentation, VSB—Technical University of Ostrava,
70800 Ostrava, Czech Republic

* Correspondence: marek.babiuch@vsb.cz (M.B.); pavel.smutny@vsb.cz (P.S.)

Abstract

Large language models (LLMs) have shown strong potential for automated code generation in software development, yet their effectiveness in embedded systems programming—requiring understanding of software logic and hardware constraints—has not been well studied. Existing evaluation frameworks do not comprehensively cover practical microcontroller development scenarios in real-world Internet of Things (IoT) projects. This study systematically evaluates 27 state-of-the-art LLMs across eight embedded systems scenarios of increasing complexity, from basic sensor reading to complete cloud database integration with visualization dashboards. Using ESP32 microcontrollers with environmental and motion sensors, we employed the Analytic Hierarchy Process with four weighted criteria: functional, instructions, output and creativity, evaluated independently by two expert reviewers. Top-performing models were Claude Sonnet 4.5, Claude Opus 4.1, and Gemini 2.5 Pro, with scores from 0.984 to 0.910. Performance degraded with complexity: 19–23 models generated compilable code for simple applications, but only 3–5 produced functional solutions for complex scenarios involving Grafana and cloud databases. The most frequent failure was hallucinated non-existent libraries or incorrect API usage, with functional capability as the primary barrier and instruction-following quality the key differentiator among competent models. These findings provide empirical guidance for embedded developers on LLM selection and identify limitations of zero-shot prompting for hardware-dependent IoT development.

Keywords: embedded system; ESP32; IoT; LLM; microcontroller



Academic Editors: Yunwei Dong,
Guanjun Liu and Xiaomin Wei

Received: 16 January 2026

Revised: 9 February 2026

Accepted: 9 February 2026

Published: 11 February 2026

Copyright: © 2026 by the authors.
Licensee MDPI, Basel, Switzerland.
This article is an open access article
distributed under the terms and
conditions of the [Creative Commons
Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Large language models (LLMs) and AI-powered chatbots have become powerful tools in software development, enabling automated code generation from natural-language descriptions [1]. These technologies use deep learning architectures trained on large repositories of source code to assist developers in writing, debugging, and optimizing software. The core ability of LLMs to understand programming requirements expressed in natural language and convert them into executable code has greatly reduced development time and lowered barriers to software creation [2]. AI-assisted code generation tools can automate routine programming tasks, offer code suggestions, and generate boilerplate code, boosting developer productivity across various fields, including web development, data science, and enterprise software [3].

The use of LLMs in code generation covers various areas of software engineering, such as code completion, test creation, bug fixing, and documentation. However, the

trustworthiness of AI-generated code remains a significant concern because LLMs can produce code that is syntactically correct but functionally wrong or does not meet the specified requirements [4]. Beyond general functional errors, research has highlighted that LLMs frequently generate problematic outputs in specialized domains, ranging from security vulnerabilities [5] and SQL syntax inconsistencies [6] to poor quality in multilingual code commenting [7]. To tackle these issues, researchers have created standardized metrics and evaluation methods to assess code accuracy and performance. In traditional software engineering, code correctness is usually checked through automated testing frameworks that run unit tests, integration tests, and regression tests repeatedly without human input [8]. These automated methods help developers verify code functionality efficiently by running extensive test suites that compare expected results with actual results, measure code coverage, and identify edge cases. One example for evaluating LLM-generated code is Pass@k, which measures the chance that at least one of k generated code samples passes all the predefined test cases for a specific programming task [9].

Microcontroller and embedded systems development present unique challenges that distinguish them from general software development [10]. A key difference lies in the testing methods: while general software engineering allows for fully automated, repeatable code integrity testing cycles run entirely in software, embedded system development often involves physical hardware interactions that require human involvement for validation and final integration testing. Automated tests alone are not enough for developing microcontroller applications because the code interacts with physical components, sensors, actuators, and communication peripherals whose behavior depends on real-world conditions. Verification includes observing program output via serial or debug interfaces, assessing peripheral and sensor responses, simulating inputs, and ensuring correct timing and control during real-time operation. This hardware-dependent validation adds significant complexity, as developers must physically connect components, configure hardware, and interpret results that can vary due to environmental factors, component tolerances, and signal quality. Additionally, embedded systems often operate under strict constraints, including limited computational resources, real-time processing needs, and direct hardware register manipulation or precise timing control. The unique challenges of embedded development require that the generated code not only be syntactically correct but also deterministic, memory-efficient, and compatible with real-time operating system requirements.

Leaderboards and benchmarks for large language models offer a structured way to compare how well different models perform on the same tasks. They utilize standardized tests to ensure results are measured consistently, helping researchers track progress and identify strengths and weaknesses across models [11]. These evaluations are important because they highlight whether a model can reason, follow instructions, or produce reliable technical or domain-specific content. However, benchmarks also have limitations: they often oversimplify real-world situations, can reward models for memorizing patterns instead of understanding them, and may become outdated as new types of tasks emerge. Therefore, leaderboard rankings should be viewed as useful indicators rather than comprehensive measures of a model's overall ability or practical quality.

Current evaluation frameworks do not fully cover the range of practical microcontroller development scenarios faced by developers in real-world projects. To fill this gap, our research systematically assesses and compares a wide variety of LLMs for microcontroller code generation across eight development scenarios, using expert-based evaluations and the Analytic Hierarchy Process (AHP) scoring method. By centering the evaluation on the ESP32 ecosystem, this study leverages a platform that transitions seamlessly from hobbyist prototyping to industrial IoT applications, providing a representative yet specific hardware context for the analysis. These scenarios represent common tasks in modern

embedded systems development, such as environmental multi-output sensor reading with multiple displays, distance measurement with multiple displays, cloud-based Internet of Things (IoT) data platforms, cloud databases and data management systems, and IoT environmental monitoring stacks.

The assessment includes models across the performance spectrum, from advanced reasoning systems with extended thinking and general-purpose high-performance architectures to specialized regional variants, efficiency-optimized models, and emerging research implementations.

This comprehensive evaluation aims to give embedded systems developers empirical data on which LLMs are most effective for microcontroller code generation, helping guide tool selection and highlight areas for further model improvement.

The remainder of this paper is organized as follows: Section 2 details the experimental setup, the definition of the evaluation scenarios, LLMs and the AHP used for scoring. Section 3 presents the experimental results and performance benchmarks of the models. Section 4 provides a comprehensive discussion of the findings. Finally, Section 5 offers concluding remarks and outlines directions for future research.

2. Related Works

LLMs are primarily trained on vast collections of web-scraped text—including extensive repositories of open-source software projects—they develop a rich, semantic understanding of programming languages and syntactic structures. As a result, AI assistants and integrated development environments (IDEs) equipped with these models have demonstrated the ability to generate software that matches the quality of skilled human developers. While base models offer strong general-purpose coding skills, fine-tuning LLMs on domain-specific languages significantly improves their accuracy and usefulness compared to generic models. In the realm of hardware-related development, industry tools are beginning to emerge. For instance, a blog post by the Arduino team [12] compares a specialized “Arduino AI Assistant” to general-purpose models like ChatGPT, suggesting that tools integrated directly into the cloud environment with access to specific documentation might reduce library errors. However, such industrial comparisons often lack the thorough, systematic assessments needed to fully understand these tools’ limitations in complex engineering settings.

The landscape of LLM code generation evaluation has expanded significantly, with several specialized benchmarks addressing limitations of earlier frameworks. Big-CodeBench [13] introduced rigorous evaluation of practical task automation, challenging LLMs to invoke multiple function calls from extensive library collections across diverse domains and achieving high branch coverage with comprehensive test cases. Evaluation of numerous LLMs revealed that even top-performing models substantially lag behind human performance, indicating considerable room for improvement. LiveCodeBench [14] addressed data contamination concerns by continuously collecting problems from competitive programming platforms (LeetCode, AtCoder, CodeForces), expanding evaluation beyond code generation to include self-repair, test output prediction, and code execution scenarios. Notably, the benchmark showed only moderate correlation with HumanEval+ performance [15], with significantly larger performance variations across models. For real-world software engineering tasks, SWE-bench [16] established a framework comprising thousands of authentic GitHub issues and pull requests from multiple Python repositories, requiring models to generate multi-file patches validated through execution-based testing. EvoCodeBench [17] introduced an evolving, repository-level benchmark aligned with real-world code and dependency distributions, releasing periodic updates to prevent data

leakage. Evaluations showed that even state-of-the-art models achieve relatively low pass rates on these repository-level tasks.

Beyond generation, CodeEditorBench [18] specifically evaluates code editing capabilities, including debugging, translating, polishing, and requirement switching, emphasizing practical software development scenarios. Domain-specific benchmarks have also emerged, with SciCode [19] curating hundreds of subproblems from research-level scientific coding problems across numerous natural science subfields to evaluate numerical methods, system simulation, and scientific calculations, and McEval [20] providing the first massively multilingual evaluation across 40 programming languages to assess cross-linguistic capabilities, revealing that code LLMs perform better in object-oriented, high-resource languages while struggling with functional and procedural, low-resource languages. However, comprehensive benchmarks specifically targeting microcontroller code generation across various real-world scenarios are still scarce in the literature.

While software code generation is well-studied, applying LLMs to embedded systems—which require understanding both software logic and hardware constraints—presents unique challenges. Englhardt et al. [21] investigated how LLMs perform on embedded programming tasks using an automated testbench with 450 trials, revealing that although these models may fail to produce immediately working code, they generate valuable reasoning about embedded design tasks and provide specific debugging suggestions beneficial to both novice and expert developers. Quan et al. [22] introduced SensorBench, a comprehensive benchmark for evaluating LLMs on coding-based sensor processing tasks with diverse real-world sensor datasets. Their findings showed that while LLMs are quite proficient in simpler tasks, they face fundamental challenges in compositional tasks involving parameter selection when compared to domain experts, with self-verification prompting strategies outperforming other methods in 48% of the evaluated tasks. Beyond assessing functional correctness, researchers have also attempted to measure the “creativity” of LLM-generated hardware solutions. The CreativEval framework [23] evaluated models on fluency, flexibility, originality, and elaboration in Register Transfer Level (RTL) code generation. Their results suggest that models like GPT-3.5 can demonstrate measurable creativity in hardware design, surpassing other models in producing novel solutions.

Recent non-peer-reviewed pre-prints have introduced more autonomous frameworks for embedded development, indicating possible future directions for the field. The Embed-Genius platform [24] offers a fully automated approach for general-purpose embedded IoT systems. By using a component-aware library resolution method, this work claims to outperform human-in-the-loop benchmarks in task completion rates, aiming to address the complex hardware dependencies that often delay manual development [5]. Similarly, the EmbedAgent pre-print [25] presents a benchmark called “Embedbench” to simulate professional roles such as System Architect and Integrator. This work highlights a performance gap in cross-platform migration; while LLMs performed fairly well migrating code to MicroPython, they struggled with more complex environments like ESP-IDF. The authors suggest that general-purpose models often fail to retrieve relevant pre-trained domain knowledge effectively without Retrieval-Augmented Generation (RAG) strategies.

In summary, although current research emphasizes the increasing abilities of LLMs in general software tasks, there is a significant lack of systematic, peer-reviewed benchmarking specifically for microcontroller-related development. This area is often hampered by complex hardware-software dependencies, which can cause library hallucinations and implementation failures.

3. Materials and Methods

The experimental setup (Figure 1) used an ESP32 development board (Espressif Systems, Shanghai, China) as the main processing unit because of its built-in dual-core processor and Wi-Fi connectivity, both crucial for simultaneous sensing and network tasks. Environmental parameters, specifically barometric pressure (hPa), ambient temperature (°C), relative humidity (%), and calculated altitude (m), were measured with a BME280 sensor (Bosch Sensortec GmbH, Reutlingen, Germany). Non-contact distance measurements were made using an HC-SR04 ultrasonic sensor (ElecFreaks, Shenzhen, China). For local feedback and diagnostics, two display technologies were integrated: a pixel OLED display (Waveshare Electronics, Shenzhen, China) and a character LCD based on HD44780 controller (Hitachi Ltd., Tokyo, Japan). The OLED displayed sensor readings and system status at high resolution, while the character LCD provided simple, low-power textual output. Standard Dupont jumper wires (generic OEM, Shenzhen, China) were used to interconnect all components during prototyping, as no custom PCB was available at this stage. During development and calibration, real-time data logging and debugging were performed by streaming data through the serial port. All testing and deployment required the ESP32 to connect to a standard local Wi-Fi network (with an average speed of 20 Mbit/s) with a predefined SSID. The network functionality had two parts: first, the ESP32 was set up as a local HTTP web server to stream real-time sensor data within HTML pages, making the data immediately available to any device on the network. Second, the system was configured to connect to multiple IoT platforms and databases, including ThingSpeak (<https://thingspeak.com>), Thinger.io (<https://thinger.io>), Beebotte (<https://beebotte.com>), Firebase Realtime Database (firebase.google.com) and InfluxDB (<https://www.influxdata.com>, all platforms accessed on 10 February 2026). Authentication data, such as API keys, tokens, and credentials, were prepared and set up before testing the LLM scenarios. The same credentials were used throughout all testing phases and all IoT platforms to keep data posting consistent and comparable.

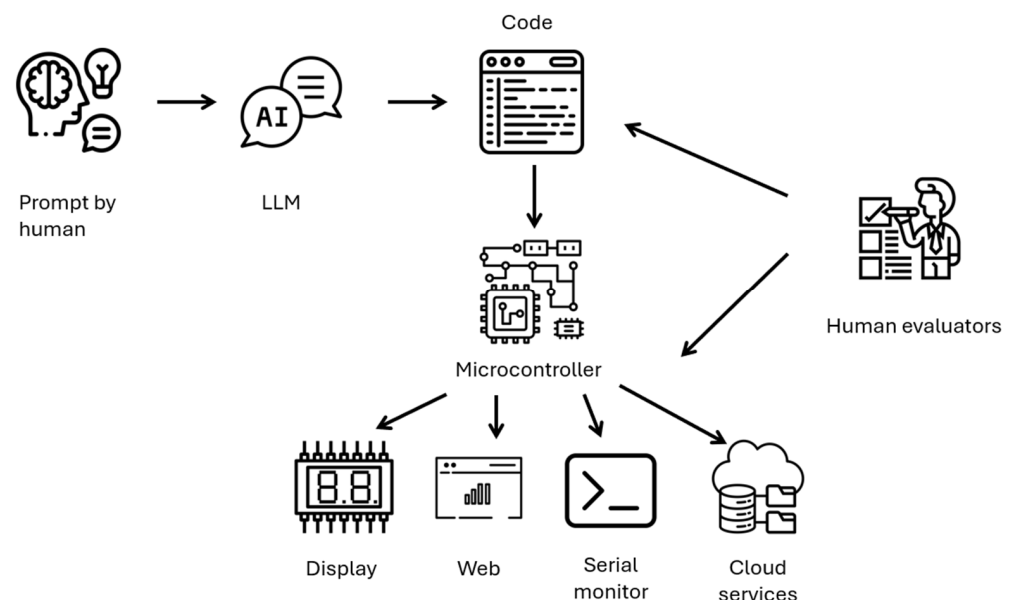


Figure 1. Framework for benchmarking LLMs in embedded contexts.

To systematically assess the performance of LLMs in generating embedded IoT code for ESP32 microcontrollers, eight distinct scenarios were created with increasing complexity. These scenarios evaluate LLM capabilities across three main aspects: local multi-output

implementations, cloud platform integrations, and complete end-to-end data visualization workflows.

1. Basic Multi-Output Sensor Reading: LLMs generate code for periodic BME280 readings with simultaneous output to the Serial Monitor, an OLED display, and a local web server.
2. Distance Measurement with Multi-Display: LLMs integrate the HC-SR04 sensor, measure obstacle distance, and show results on the Serial Monitor, an I2C LCD display, and a locally hosted webpage while managing Wi-Fi credentials correctly.
3. ThingSpeak Cloud Integration: LLMs create ESP32 code that reads BME280 values, displays them locally (OLED and Serial Monitor), and successfully uploads data to a ThingSpeak channel using provided API keys.
4. Thingier.io Platform Setup: LLMs prepare complete code and user instructions for connecting an ESP32 with a BME280 sensor to Thingier.io, covering device provisioning and dashboard configuration.
5. Beebotte Data Publishing: LLMs build ESP32 applications that send BME280 data to Beebotte via API or MQTT and include step-by-step platform setup instructions for users.
6. Firebase Realtime Database: LLMs implement secure, authenticated logging of BME280 data to Firebase, including timestamp synchronization and email/password authentication.
7. InfluxDB Time-Series Storage: LLMs generate valid code for periodically sending BME280 measurements to InfluxDB Cloud using the correct credentials, measurement names, and tags, along with instructions for dashboard creation.
8. ESP32—Grafana End-to-End Visualization: LLMs produce a full workflow for ingesting BME280 data, storing it in InfluxDB, and visualizing it in Grafana, including dashboard steps and valid JSON import configurations.

We focus on zero-shot prompting to establish a baseline of the LLMs' inherent "initial" reasoning and domain-specific knowledge regarding the ESP32 ecosystem. While techniques such as RAG and few-shot prompting can enhance performance, they introduce external variables. The quality of the retrieved documentation or the bias of the provided examples can mask the model's actual generative capabilities. By using zero-shot prompts, we simulate the most common and accessible developer workflow and evaluate the model's ability to interpret complex hardware requirements without extensive prompt engineering or external data infrastructure. This approach ensures that the resulting benchmarks reflect the model's fundamental understanding of embedded systems rather than the effectiveness of a specific retrieval system.

The code generation performance was tested on LMArena.ai, which offers direct chat access to a wide range of state-of-the-art LLMs through a unified interface (Table 1). This setup ensures a consistent and comparable testing environment. In all scenarios, the LLMs were used in direct chat mode, simulating a typical developer interaction where context and instructions are given conversationally.

We employed the AHP to systematically evaluate LLMs for generating microcontroller code, enabling us to structure evaluation criteria hierarchically. Through pairwise comparisons (Table 2), we determined the relative importance of main categories—Functional, Instructions, Output, and Creativity—using Saaty's 1–9 scale [26], where 1 indicates equal importance, 3 moderate, 5 strong, 7 very strong, and 9 extreme importance of one criterion over another. We derived criterion weights from the resulting pairwise comparison matrix, which is shown below, with Functional assigned the highest priority (1 over Instructions, 5 over Output, 9 over Creativity). The matrix yielded normalized priority weights of 0.544 for Functional, 0.292 for Instructions, 0.107 for Output, and 0.057 for Creativity, calculated via the principal eigenvector.

Table 1. Classification of Large Language Models evaluated for microcontroller-driven IoT applications based on technical reports and documentation provided by the model developers.

Category	LLMs
Reasoning Models	claude-sonnet-4-5-20250929-thinking-32k deepseek-v3.2-exp-thinking o4-mini-2025-04-16 hunyuan-vision-1.5-thinking step-3
General-Purpose High-Performance Models	gemini-2.5-pro gpt-5-high grok-4-0709 claude-opus-4-1-20250805 claude-opus-4-20250514 gpt-4.1-2025-04-14 glm-4.5 mai-1-preview qwen-vl-max-2025-08-13 command-a-03-2025 mistral-medium-2508 magistral-medium-2506 gemma-3-27b-it
Mixture of Experts Models	qwen3-coder-480b-a35b-instruct llama-4-maverick-17b-128e-instruct llama-4-scout-17b-16e-instruct minimax-m1
Fast/Efficient Models	gemini-2.5-flash grok-4-fast longcat-flash-chat glm4-6 kimi-k2-web

Table 2. Pairwise comparison matrix of evaluation criteria using Saaty’s 1–9 scale.

	Functional	Instructions	Output	Creativity
Functional	1	2	5	9
Instructions	1/2	1	3	5
Output	1/5	1/3	1	2
Creativity	1/9	1/5	1/2	1

Matrix consistency was verified with $\lambda_{\max} = 4.006$, Consistency Index (CI) = $(\lambda_{\max} - n)/(n - 1) = 0.002$, and Consistency Ratio (CR) = $CI/RI = 0.002$ ($RI = 0.90$ for $n = 4$), confirming acceptable consistency ($CR < 0.10$).

Detailed sub-criteria (Table 3) for each category included: Functional (complete code provision, correct libraries, error-free compilation); Instructions (step-by-step setup guides with hardware connections and code comments explaining structure); Output (serial monitor clarity, display information on OLED/LED, reliable Web/IoT/database connections); and Creativity (additional features and novel graphical outputs beyond requirements).

Two independent evaluators—a senior microcontroller expert (25 years of experience) and an advanced practitioner (15 years of experience)—conducted assessments separately before reaching consensus on final scores, ensuring reliability and minimizing individual bias in LLM performance rankings.

Table 3. Hierarchical sub-criteria and descriptions for AHP evaluation.

Category	Subcategory	Description
Functional	Code generated in full	The complete program code is provided and ready to use.
	Correct libraries	All required libraries are included and correctly referenced.
	Compilation without errors	The code compiles and runs without errors.
Instructions	Setup Instructions	A clear, step-by-step guide, including hardware connections and alternative connection options
	Extra information in code comments	Code comments that explain the structure, purpose, and different setup or configuration options.
Output	Serial monitor	The amount of information and the clarity and structure of the data shown in the serial monitor.
	Display	The amount of information and the clarity and structure of the data shown on OLED or LED displays.
	Web/IoT Platforms/Databases	The reliability of the local web server, successful connections to IoT platforms or databases, and the clarity and consistency of the stored or displayed data.
Creativity	New features	Programming paradigm, additional functionality and creative graphical or visual output beyond the basic requirements.

4. Results

This analysis uses a comprehensive mixed-methods approach that combines both qualitative and quantitative techniques to evaluate LLMs across various IoT-related tasks. We analyse performance metrics such as functionality, instruction clarity, output quality, and creativity of LLM-generated code and documentation in detail. Our findings highlight the strengths and challenges LLMs face across different platforms and integration complexities, demonstrating their growing ability to automate embedded IoT development and cloud visualization processes.

4.1. Environmental and Motion Tracking Sensors

The AHP results for both environmental sensors and distance measurement show several key similarities and performance factors for deployment in sensor integration applications. In both cases, models from the gemini-2.5 family showed optimal performance, with gemini-2.5-flash winning in the environmental sensor reading task (Final score ≈ 1.00) and tying with gemini-2.5-pro in the distance measurement task (Final score ≈ 0.9857). Most candidates achieved the highest Functional score (≈ 0.5435). Still, a few models consistently failed this essential baseline (8 models in sensor reading, 4 models in distance measurement), highlighting a sharp functional threshold needed for entry into the high-competency group. For models that met the functional requirements, the main factor influencing ranking was Instruction score compliance, which created the most significant gap between the top performers (≈ 0.292) and mid-tier models (≈ 0.2435). This consistent result across both tasks underscores that while LLM functional ability is generally accessible, success in high-stakes sensor and motion processing relies not just on output quality but

on the model's capacity to follow complex operational constraints and output formatting rules strictly (Figure 2).

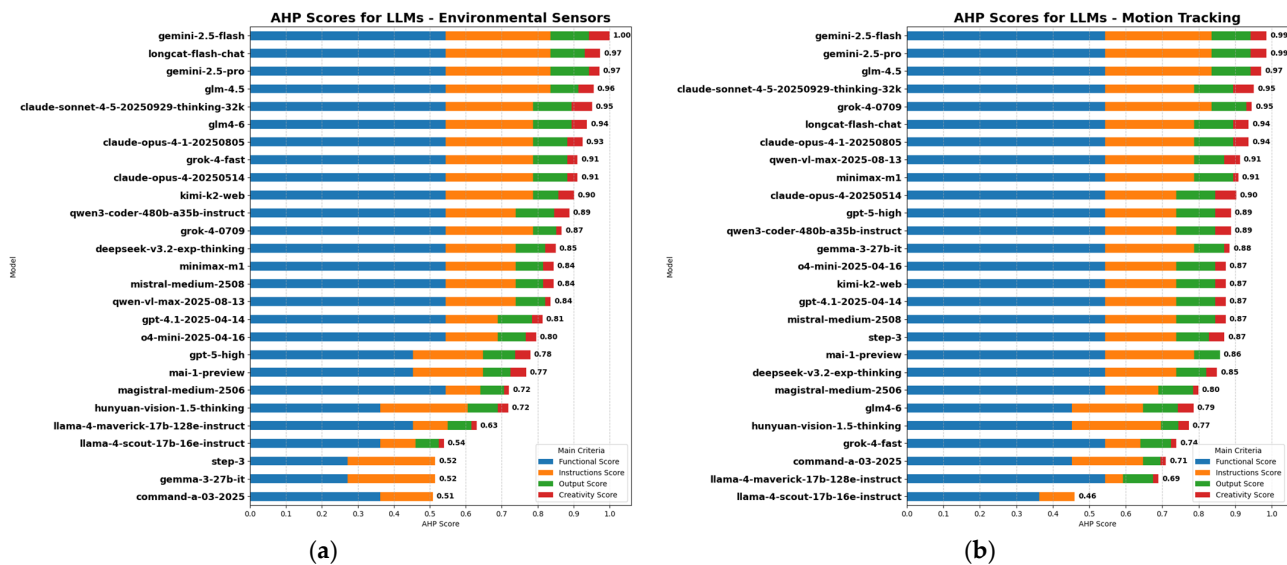


Figure 2. Distribution of aggregate and categorical AHP scores for each evaluated LLM across two experimental tasks: (a) environmental multi-output sensor reading and (b) distance measurement.

In the environmental sensor task, 19 of 27 models produced compilable code on the first attempt, correctly configuring the I2C bus and including explanatory comments for exceptional cases. The remaining 8 models failed at this initial hurdle because they targeted incompatible OLED driver libraries or referenced obscure third-party display libraries, even though all models selected appropriate BME280 and Wi-Fi libraries. For the distance measurement task, simplifying the display subsystem led to 23 models compiling successfully on the first attempt, with the remaining models again hindered by unrelated or missing third-party display libraries rather than by sensor or connectivity misconfiguration.

In the environmental sensor scenario, models were required to deploy an ESP32-based web server that exposed measurements, but they received no guidance on layout, styling, or visualization strategy. As a result, the graphical interfaces emerged entirely from the models' own design choices, with variations in layout structure, data grouping, and presentation style (Figure 3). Because the visual form was neither specified nor constrained by the task definition, these outcomes were assessed under the creativity dimension.

During the implementation of both the environmental sensor and distance measurement tasks, an additional qualitative distinction emerged in how models designed the web server component. Without explicit prompting, five models adopted an asynchronous web server implementation on ESP32, typically relying on specialized asynchronous libraries to support concurrent client connections and non-blocking request handling. This design reduced main-loop blocking, improved responsiveness under load, and aligned more closely with contemporary embedded networking practices. Models that chose this approach were awarded higher creativity scores in the evaluation. In contrast, most other models generated simpler polling-based solutions that updated sensor readings via periodic page refreshes (for example, using a 5 s meta refresh in the HTML header). These are straightforward to implement but generate unnecessary network traffic and limit interaction flexibility. The contrast between these two patterns—static, refresh-driven pages versus modern, event-driven asynchronous servers—illustrates that, under identical prompts and functional requirements, models can differ not only in correctness but also in architectural sophistication, with direct implications for robustness and scalability in real deployments.

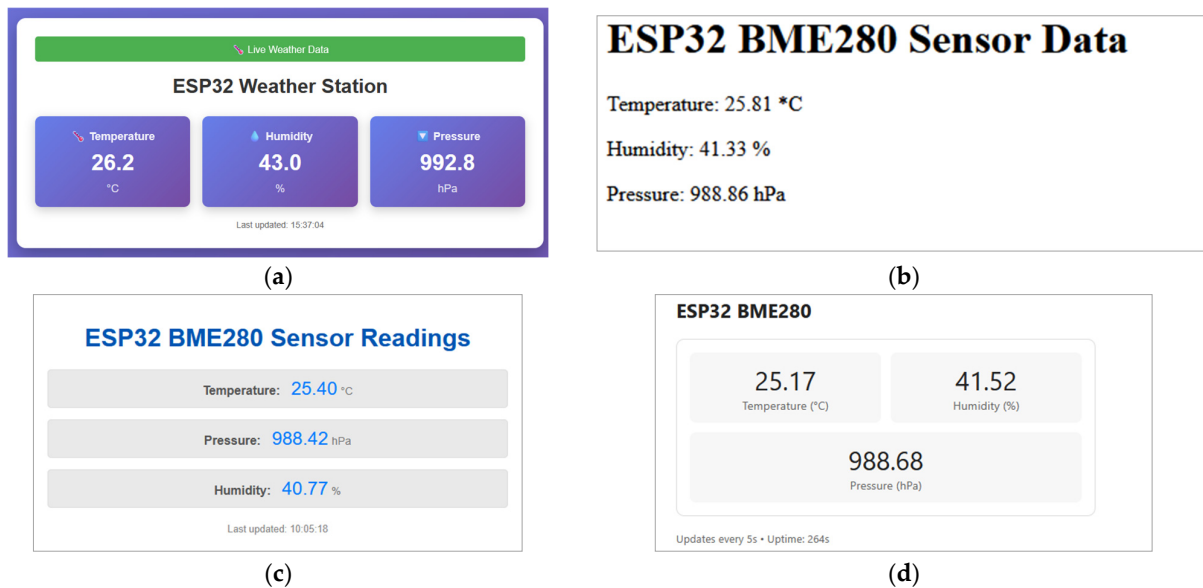


Figure 3. Examples of web-based graphical interfaces generated by selected models for the environmental sensor scenario: (a) Claude Sonnet 4.5 with a card-based dashboard, (b) Magistral Medium showing a minimal text-oriented layout (c) Gemini 2.5 Flash presenting a structured list and (d) GPT-5 High with a clean tile-based layout with separated metrics and update status information.

4.2. IoT Platforms

The analysis of the three IoT platform integration scenarios (ThingSpeak, Thingier.io, and Beebotte) revealed significant shifts in top-tier performance compared to typical sensor tasks, with the Claude architecture and gpt-5-high model consistently ranking highest (Figure 4). Notably, BeeBotte integration led to a three-way tie for the top model (Final score ≈ 1.00) among claude-opus-4-1, claude-sonnet-4-5, and gpt-5-high. Later, claude-sonnet-4-5 secured the top position in both the ThingSpeak (Score ≈ 1.00) and Thingier.io (Score ≈ 0.9713) scenarios, demonstrating its robustness across different integration protocols. Analyzing the functional threshold showed varying difficulty levels across platforms: only 16 models achieved the high Functional score (≈ 0.5435) needed for Thingier.io integration, while 20 models reached this benchmark for ThingSpeak. This indicates differing barriers to entry, likely due to platform complexity or API documentation. Regardless of the platform, the key factor that distinguished high-performing models was the Instructions score (Max ≈ 0.292), which remained the most important performance metric after meeting the core functional requirement. This supports the conclusion that, for complex, platform-specific integration tasks, a large language model's ability to carefully follow predefined operational constraints is the ultimate factor in determining suitability.

For the set of IoT service scenarios, using the correct libraries for each service was essential. When testing with ThingSpeak, 20 models compiled successfully initially, while 7 failed, mainly due to missing the ThingSpeak communication library. In the fourth scenario, 23 models correctly selected Thingier.io service libraries, compared with 3 that chose incorrectly. However, only 18 source codes compiled on the first try, versus 11, indicating that choosing the right library did not ensure that the LLMs would use the functions properly. In the Beebotte test, 24 models had suitable libraries, with 5 opting for MQTT communication and the remaining 19 choosing API access. Three models selected incorrect libraries, and overall, 8 models failed to compile initially. The communication method was determined by the LLM, as it was not specified in the input prompt. For example, glm 4.5 chose MQTT, while its successor, glm 4.6, chose API access.

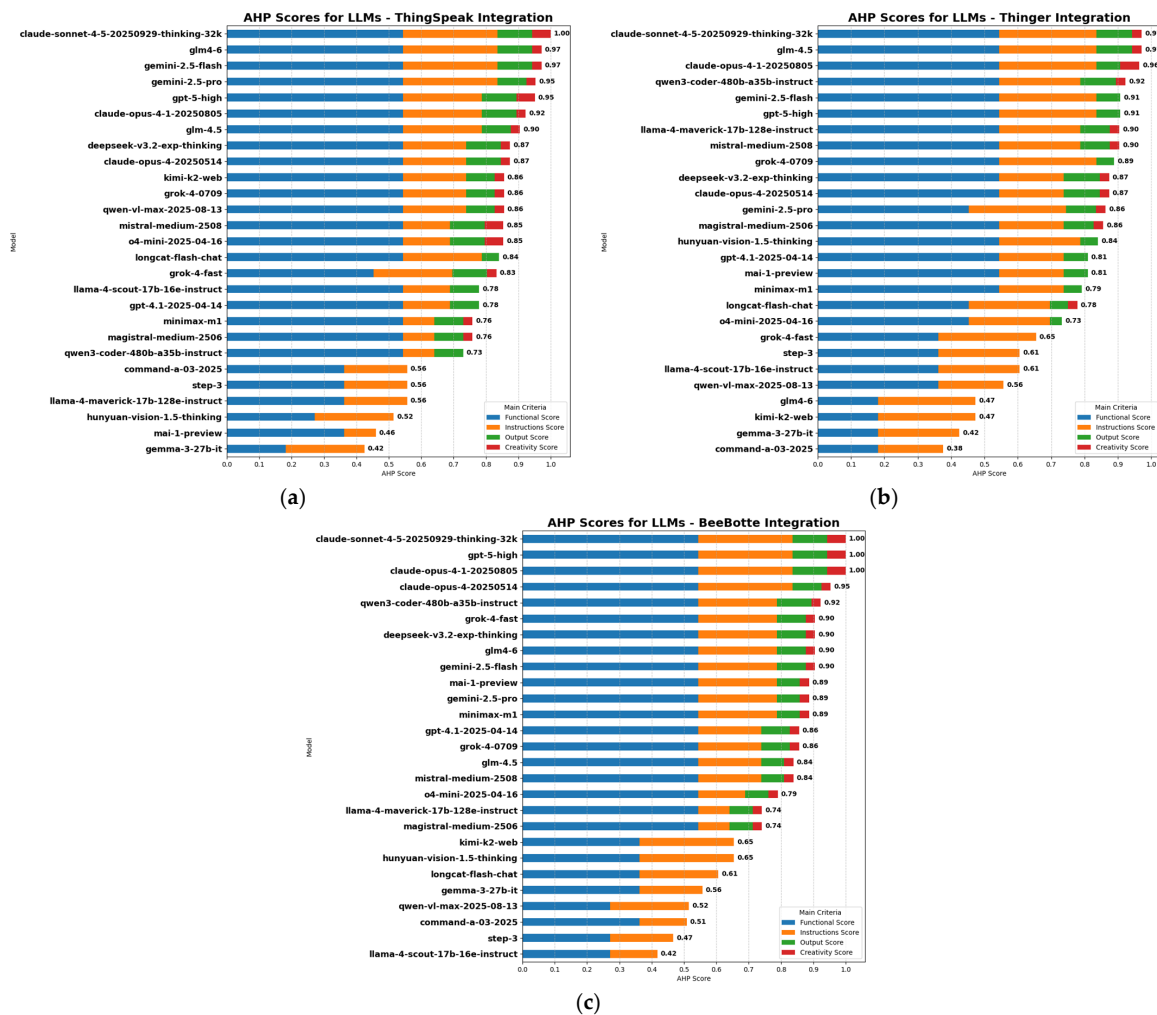


Figure 4. Distribution of aggregate and categorical AHP scores for each evaluated LLM across three experimental tasks: Setup, integration and publishing in ThingSpeak (a), Thinger.io (b), Beebotte (c) IoT cloud platforms.

4.3. Data Storage in Cloud Databases

The two cloud database integration scenarios—Firebase Realtime Database and InfluxDB Time-Series Storage—were evaluated to assess LLM performance in generating secure, authenticated ESP32 code for logging BME280 sensor data and providing clear instructions for database setup and dashboard creation. Firebase is a Google cloud platform that enables microcontrollers to store and retrieve data in real time over the Internet, while InfluxDB is an open-source database optimized for time-series data, making it ideal for storing large volumes of sensor data from IoT devices.

The evaluation revealed a consistently high barrier to entry for these complex, security-sensitive cloud integration tasks, significantly limiting the pool of viable models (Figure 5). The Functional score threshold, which assessed the security and correctness of the generated authenticated code, proved the most restrictive metric: only nine models achieved the highest Functional score (≈ 0.5435) for Firebase, and only 11 did so for InfluxDB. Models falling below the minimum required functional quality (≈ 0.4529) failed to produce secure, platform-specific authentication logic—the primary challenge for current LLMs. Among functionally capable models, the Instructions score (≈ 0.292) differentiated those that could generate correct code from those that could also provide clear, precise guidance on database setup and dashboard creation. Ultimately, claude-sonnet-4-5 emerged as the sole optimal performer in both scenarios, achieving a maximum Final score of approximately 1.00.

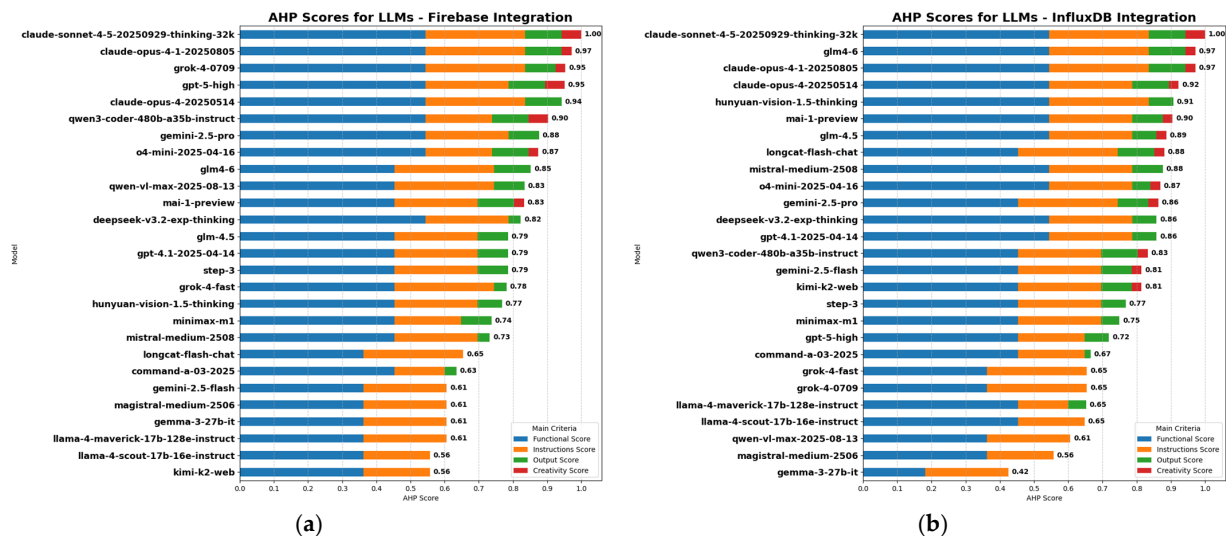


Figure 5. Distribution of aggregate and categorical AHP scores for each evaluated LLM across two experimental tasks: InfluxDB Time-Series Storage (a), Firebase Realtime Database (b).

In scenario 6 (Firebase), nine models generated executable code on the first attempt, and eight produced fully functional applications. Seven additional models achieved functionality after minor adjustments to formal errors or with supplementary prompts, yielding 15 functional applications in total. In scenario 7 (InfluxDB), 11 compilations succeeded initially, but only six resulted in functional applications, with five failing in various areas. Five additional models achieved functionality after minor code adjustments and prompting, resulting in 11 functional applications overall. Notably, errors in both scenarios stemmed from flawed source code functionality, improper communication function calls, and inadequate cloud configuration rules—not from inappropriate library selection, as all libraries were suitable.

4.4. Cloud Database and Visualization Integration

The most complex scenario required a complete integration chain: compilable, functionally correct ESP32 code; successful data transmission to InfluxDB; a valid query usable in the InfluxDB/Grafana dashboard; and a syntactically correct Grafana JSON dashboard description. This setup extends the earlier InfluxDB-only task by adding the full visualization layer, and the AHP results show that this added complexity sharply amplifies differences between models. Only three models produced a completely error-free solution that satisfied all four requirements in a single pass: claude-opus-4-1-20250805, claude-sonnet-4-5-20250929-thinking-32k, and gpt-5-high. All three achieved the maximum Functional score (≈ 0.5435), the maximum Instructions score (≈ 0.2923), and the highest Output score tier (≈ 0.1069), indicating correct ESP32—InfluxDB communication, a valid dashboard query, and a valid Grafana JSON file. Their final scores clustered at or near 1.0, marking them as the clear top performers (Figure 6).

One additional model, command-a-03-2025, initially failed to compile due to a minor, easily correctable code issue. After this fix, it successfully sent data to InfluxDB, produced a correct dashboard query, and generated a valid Grafana JSON configuration. In the AHP scoring, this model exhibits a slightly reduced Functional score (≈ 0.4529) and a mid-range Instructions score (≈ 0.1948), but a solid Output score (≈ 0.0713) and a non-zero Creativity contribution (≈ 0.0287), yielding a respectable Final score of ≈ 0.748 . This demonstrates that modest weaknesses in code quality and documentation can coexist with a fully usable end-to-end solution.

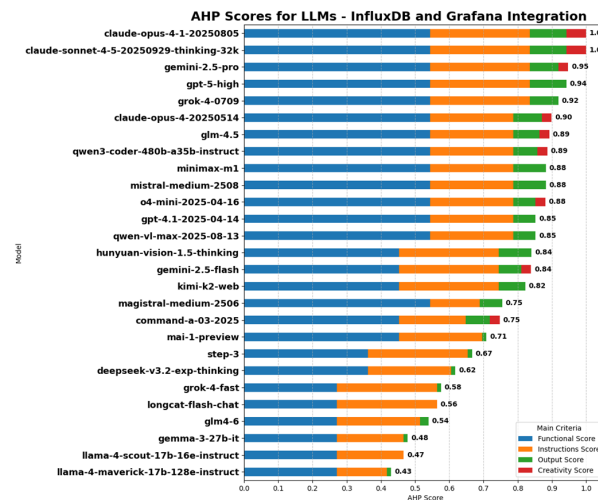


Figure 6. Distribution of aggregate and categorical AHP scores for each evaluated LLM for the complex IoT data-to-dashboard scenario.

Across the full set of 27 models, the Output score quantifies how many of the three integration endpoints—InfluxDB ingestion, a correct dashboard query, and a valid Grafana JSON dashboard—were achieved. Models that failed to produce any working component have Output scores at or near 0.0, whereas models such as gemini-2.5-pro, grok-4-0709, claude-opus-4-20250514, glm-4.5, and others with Output scores around 0.077–0.095 indicate that two of the three elements worked ($\approx 66\%$ success). Only the top trio plus command-a-03-2025 reached the highest Output tier (≈ 0.071 – 0.107), corresponding to full or near-full correctness across all integration steps.

Creativity scores reflect the qualitative design sophistication of the Grafana dashboards. Models with zero Creativity scores (gpt-5-high, grok-4-0709, mistral-medium-2508, and several mid- to low-ranked systems) produced minimal or purely utilitarian dashboards, whereas models with non-zero Creativity scores (such as claude-opus-4-1-20250805, claude-sonnet-4-5-20250929-thinking-32k, gemini-2.5-pro, claude-opus-4-20250514, glm-4.5, qwen3-coder-480b-a35b-instruct, and command-a-03-2025) proposed richer layouts with multiple panels, clearer structure, or additional informative elements, thereby improving interpretability without sacrificing correctness (Figure 7).



Figure 7. Example of the final appearance of a visual dashboard produced by a successful claude-opus-4-1 model in a scenario focused on measured data visualization, with InfluxDB serving as the data source and Grafana as the visualization layer.

4.5. The Overall Ranking of Models

The comparative evaluation of 27 large language models revealed substantial performance heterogeneity, with normalized total scores ranging from 0.984 to 0.539 (Table 4). At the top of the ranking, claude-sonnet-4-5-20250929-thinking-32k achieves the highest overall AHP score (0.984), followed by claude-opus-4-1-20250805 (0.961) and gemini-2.5-pro (0.918), with several other high-capacity general-purpose or advanced reasoning models (claude-opus-4-20250514, glm-4.5, gpt-5-high) also clustered above 0.89. These models include both explicitly reasoning-oriented systems (e.g., claude-sonnet “thinking”) and general-purpose flagships, suggesting that strong chain-of-thought capabilities and broad instruction-following competence translate into superior functional code quality, better-structured setup instructions, and more mature output handling for ESP32 scenarios. The presence of o4-mini-2025-04-16, deepseek-v3.2-exp-thinking, and gpt-4.1-2025-04-14 in the upper segment (scores around 0.83–0.83) further indicates that compact or experimental “thinking” variants can reach near-flagship performance when judged on this practically oriented, multi-criteria coding task.

Table 4. Overall ranking of LLMs in embedded contexts.

Rank	Model	Total Score	Scenarios							
			1	2	3	4	5	6	7	8
1	claude-sonnet-4-5-20250929-thinking-32k	0.984	0.951	0.951	1.000	1.000	0.971	1.000	1.000	1.000
2	claude-opus-4-1-20250805	0.961	0.925	0.937	0.923	1.000	0.964	0.971	0.971	1.000
3	gemini-2.5-pro	0.918	0.971	0.986	0.954	0.887	0.863	0.876	0.863	0.948
4	claude-opus-4-20250514	0.910	0.911	0.903	0.874	0.954	0.874	0.943	0.923	0.899
5	glm-4.5	0.901	0.956	0.971	0.905	0.838	0.971	0.786	0.887	0.893
6	gpt-5-high	0.892	0.780	0.888	0.951	1.000	0.907	0.951	0.719	0.943
7	gemini-2.5-flash	0.879	1.000	0.986	0.971	0.905	0.907	0.606	0.814	0.839
8	qwen3-coder-480b-a35b-instruct	0.872	0.888	0.888	0.730	0.923	0.923	0.903	0.832	0.887
9	grok-4-0709	0.868	0.867	0.945	0.856	0.856	0.889	0.954	0.655	0.919
10	mistral-medium-2508	0.851	0.844	0.874	0.854	0.838	0.905	0.732	0.876	0.882
11	o4-mini-2025-04-16	0.834	0.795	0.874	0.854	0.790	0.732	0.874	0.869	0.881
12	deepseek-v3.2-exp-thinking	0.831	0.850	0.850	0.874	0.905	0.874	0.823	0.858	0.618
13	gpt-4.1-2025-04-14	0.828	0.813	0.874	0.779	0.856	0.810	0.786	0.858	0.852
14	minimax-m1	0.820	0.844	0.908	0.759	0.887	0.792	0.737	0.750	0.882
15	glm4-6	0.804	0.937	0.786	0.971	0.905	0.473	0.852	0.971	0.539
16	longcat-flash-chat	0.779	0.974	0.937	0.840	0.606	0.779	0.655	0.881	0.564
17	mai-1-preview	0.778	0.768	0.858	0.460	0.887	0.810	0.832	0.905	0.708
18	grok-4-fast	0.756	0.911	0.738	0.832	0.905	0.655	0.781	0.655	0.576
19	hunyuan-vision-1.5-thinking	0.752	0.718	0.773	0.515	0.655	0.840	0.768	0.907	0.840
20	qwen-vl-max-2025-08-13	0.746	0.836	0.913	0.856	0.515	0.557	0.834	0.606	0.852
21	kimi-k2-web	0.744	0.901	0.874	0.856	0.655	0.473	0.557	0.814	0.822
22	magistral-medium-2506	0.724	0.721	0.799	0.759	0.741	0.856	0.606	0.557	0.755
23	step-3	0.654	0.515	0.870	0.557	0.467	0.606	0.786	0.768	0.666
24	llama-4-maverick-17b-128e-instruct	0.651	0.630	0.690	0.557	0.741	0.905	0.606	0.652	0.430

Table 4. Cont.

Rank	Model	Total Score	Scenarios							
			1	2	3	4	5	6	7	8
25	command-a-03-2025	0.588	0.508	0.710	0.557	0.508	0.376	0.635	0.666	0.748
26	llama-4-scout-17b-16e-instruct	0.559	0.539	0.460	0.779	0.418	0.606	0.557	0.648	0.467
27	gemma-3-27b-it	0.539	0.515	0.885	0.425	0.557	0.425	0.606	0.425	0.478

The middle of the distribution is populated by a mixture of high-performance general models, mixture-of-experts (MoE) systems, and fast/efficient variants, with total scores roughly between 0.75 and 0.83. qwen3-coder-480b-a35b-instruct (0.872) and minimax-m1 (0.820) illustrate that MoE architectures can achieve competitive overall quality, but in this evaluation, they do not systematically outperform monolithic flagship models. Fast- and efficiency-oriented systems such as gemini-2.5-flash (0.879), mistral-medium-2508 (0.851), glm4-6 (0.804), longcat-flash-chat (0.779), grok-4-fast (0.756), and kimi-k2-web (0.744) generally occupy the upper-middle band, indicating that designs optimized for latency and throughput can still deliver functionally adequate and reasonably well-documented ESP32 solutions, albeit with modest deficits relative to the very best systems in at least one of the heavily weighted criteria (typically completeness of code or richness of instructional material).

5. Discussion

This study shows that a zero-shot prompt strategy can produce fully compilable ESP32/IoT applications for a substantial subset of LLMs, but performance degrades sharply as application complexity increases. While 19–23 models successfully built simple sensor applications, only 5–8 produced usable outputs for complex scenarios involving Grafana dashboards and cloud databases. This performance gap aligns with recent benchmarking on embedded systems, such as the EmbedBench study [25], which reported significantly lower pass rates for complex ESP-IDF migration tasks (29.4%) than for simpler MicroPython implementations (73.8%). Similarly, the EmbedGenius framework [24] found that while LLMs can handle modular IoT tasks, their efficacy drops when coordinating multiple hardware-software interfaces without RAG or compiler feedback loops. Our results confirm that for intricate integrations, zero-shot prompting alone—despite its labor-saving appeal—often lacks the necessary context to resolve multi-dependency constraints.

The most frequent cause of compilation failure was hallucinated non-existent libraries or incorrect API usage. This reflects a broader trend identified in software engineering research: a study analyzing [27] over 2 million code samples found that hallucinated package references are pervasive, with open-source models exhibiting hallucination rates up to four times higher (21.7%) than closed-source counterparts like GPT-4 (5.2%). In our specific context, these hallucinations were most prevalent in the integration of IoT services (e.g., ThingSpeak, Thingier.io, and Beebotte), display drivers, and environmental sensor configurations. Rather than utilizing official or currently supported libraries, the LLMs frequently referenced individual, deprecated repositories that are no longer available or functional. These technical shortcomings are directly quantified in our AHP framework under the Functionality score; a lower score in this category typically indicates that the model failed to identify official library standards, resulting in non-compilable code.

Our observation that models often generate valid logic but invalid syntax for specific sensor libraries is consistent with findings that general-purpose LLMs struggle to use domain-specific pre-trained knowledge effectively in embedded contexts. Furthermore,

the prevalence of invalid JSON configurations for Grafana dashboards suggests that while models excel at imperative code (C++/Python), they are less robust at generating strictly schema-compliant configuration files without few-shot examples.

LLMs generally produced instructions that were practical for implementation, but they varied in how they distributed this guidance between the conversational response and the source code. Three recurring strategies were observed: (i) a text-centric approach, in which models provide detailed, step-by-step setup guidance in the conversational response while leaving the source code with minimal or incomplete explanatory comments; (ii) a code-centric approach, in which models give only a brief conversational summary but embed substantial, context-aware usage guidance directly in the code through headers and inline comments; and (iii) a combination approach, in which models provide both a structured setup narrative and complementary code comments that clarify configuration choices and reduce ambiguity during deployment. The combination strategy was the most common outcome, suggesting that many models implicitly recognize that embedded workflows require both an external procedural description (to support initial assembly and configuration) and in-code annotation (to support later modification, troubleshooting, and reuse). However, performance across these two documentation channels was not always consistent: in some scenarios, models tended to deliver stronger step-by-step setup explanations than code comments, while in other scenarios the code comments were relatively richer than the accompanying conversational setup narrative. This pattern indicates that conversational setup guidance and in-code guidance capture different aspects of instructional quality and should be evaluated as complementary, rather than assuming that strength in one channel implies strength in the other.

Our findings suggest that while LLMs can serve as effective “teachable agents” for novices by generating initial boilerplate and wiring instructions, reliance on them for complex system integration requires caution. Literature in computing education, including the HypoCompass study [28], indicates that involving novices in debugging LLM-generated code can be educationally beneficial, but only if the initial generation is close enough to be “repairable”. The high variability in documentation quality we observed—where some models provided excellent step-by-step wiring guides while others offered none—mirrors the inconsistency noted in studies evaluating LLMs for introductory programming assistance. For less-experienced users, the “human-in-the-loop” strategy remains essential: our results show that even when the initial code fails, the provided comments and hardware instructions often provide enough scaffolding for a user to finalize the project, provided they can interpret compiler errors.

6. Conclusions

This comprehensive evaluation of 27 large language models across eight embedded systems scenarios reveals substantial performance heterogeneity in microcontroller code generation capabilities. The study demonstrates that while a majority of contemporary LLMs can successfully generate basic sensor integration code for ESP32 platforms, performance degrades precipitously as task complexity increases—from 19 to 23 models succeeding in simple applications to only 3–5 achieving full functionality in complex cloud database and visualization integrations. The claude-sonnet-4-5-20250929-thinking-32k model emerged as the highest-performing system overall (0.984), followed closely by claude-opus-4-1-20250805 (0.961) and gemini-2.5-pro (0.918), indicating that both advanced reasoning architectures and general-purpose flagship models demonstrate superior capabilities for embedded systems code generation. Critically, functional correctness constituted the primary barrier to entry, with instruction-following quality serving as the principal differentiator among models that achieved baseline functional competence.

6.1. Theoretical and Practical Implications

The findings have significant implications for both embedded systems development practice and LLM evaluation methodology. For practitioners, the results indicate that zero-shot prompting strategies, while labor-efficient for simple modular tasks, prove insufficient for complex multi-dependency integrations requiring coordinated hardware-software interfaces. The high prevalence of hallucinated libraries and incorrect API usage—consistent with broader software engineering research showing hallucination rates up to 21.7% in open-source models—underscores the necessity of human-in-the-loop verification for production embedded systems. Furthermore, the observation that models employ heterogeneous documentation strategies (text-centric, code-centric, or combination approaches) suggests that conversational setup guidance and in-code annotation capture complementary aspects of instructional quality and should be evaluated independently. For novice developers, LLMs can serve as effective teachable agents by generating initial boilerplate and wiring instructions, provided the generated code is sufficiently close to functional to enable educational debugging.

6.2. Limitations

Several constraints delimit the scope and generalizability of these findings. The evaluation focused exclusively on ESP32 microcontrollers with specific sensor configurations, and results may differ across alternative embedded platforms, architectures, or peripheral ecosystems. The zero-shot prompting approach, while representative of typical developer interactions, does not capture the potential performance improvements achievable through RAG, few-shot examples, or iterative compiler feedback loops—techniques shown to enhance LLM efficacy in related embedded systems research. Furthermore, the qualitative assessment of creativity and instruction following was conducted by a single domain expert and a single practitioner, which introduces the possibility of subjective bias; however, this was mitigated by assigning these criteria lower priority weights within the AHP framework compared to objective functional metrics.

6.3. Future Research Directions

Several promising avenues emerge for extending this research. Comparative studies incorporating RAG frameworks, multi-turn conversational refinement, and automated compiler feedback could quantify the performance gains achievable beyond zero-shot prompting and identify optimal human-AI collaboration patterns for embedded development. Expansion of the evaluation framework to encompass additional microcontroller families (STM32, Arduino, Raspberry Pi Pico), real-time operating systems (FreeRTOS, Zephyr), and communication protocols (LoRaWAN, BLE, Zigbee) would enhance generalizability across the embedded systems landscape. Investigation of fine-tuning approaches using domain-specific embedded systems codebases could establish whether specialized training substantially improves hardware-aware code generation compared to general-purpose models. Finally, longitudinal studies examining how rapidly evolving LLM architectures close the performance gap on complex integration tasks would inform adoption timelines and investment strategies for organizations considering AI-assisted embedded development workflows.

This study establishes that contemporary LLMs have reached a threshold of practical utility for embedded IoT development, capable of substantially accelerating routine programming tasks while requiring domain expertise for verification, integration, and troubleshooting of complex multi-component systems.

Author Contributions: Conceptualization, M.B. and P.S.; methodology, M.B. and P.S.; validation, M.B. and P.S.; formal analysis, M.B. and P.S.; investigation, M.B. and P.S.; resources, M.B. and P.S.;

data curation, M.B. and P.S.; writing—original draft preparation, M.B. and P.S.; writing—review and editing, M.B. and P.S.; visualization, M.B. and P.S.; supervision, M.B. and P.S.; project administration, M.B. and P.S.; funding acquisition, M.B. and P.S. All authors have read and agreed to the published version of the manuscript.

Funding: This article was co-funded by the European Union under the REFRESH—Research Excellence for Region Sustainability and High-Tech Industries Project—CZ.10.03.01/00/22_003/0000048 via the Operational Programme Just Transition. It was also completed in connection with the project “Innovative and additive technologies for sustainable energy industry”, registration no. CZ.02.01.01/00/23_021/0010117, financed by the Structural Funds of the European Union project and the project SP2026/020 “Research in the Field of Intelligent Control of Machines and Processes”, supported by the Ministry of Education, Youth and Sports, Czech Republic.

Data Availability Statement: The original data presented in the study are openly available in OSF at https://osf.io/zpvrc/overview?view_only=268c727e44b9416faaeab44058ad8191 (accessed on 8 February 2026).

Acknowledgments: During the preparation of this work, the authors used Grammarly Premium and Perplexity Pro in order to improve language and readability. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AHP	Analytic Hierarchy Process
AI	Artificial Intelligence
API	Application Programming Interface
BLE	Bluetooth Low Energy
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	JavaScript Object Notation
LCD	Liquid Crystal Display
LLM	Large Language Model
LoRaWAN	Long Range Wide Area Network
MQTT	Message Queuing Telemetry Transport
OLED	Organic Light-Emitting Diode
RAG	Retrieval-Augmented Generation
SSID	Service Set Identifier
Wi-Fi	Wireless Fidelity

References

1. Fan, A.; Gokkaya, B.; Harman, M.; Lyubarskiy, M.; Sengupta, S.; Yoo, S.; Zhang, J.M. Large Language Models for Software Engineering: Survey and Open Problems. In *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering, ICSE-FoSE 2023, Melbourne, Australia, 14–20, May 2023*; Institute of Electrical and Electronics Engineers Inc.: New York, NY, USA, 2023; pp. 31–53.
2. Paradis, E.; Grey, K.; Madison, Q.; Nam, D.; Macvean, A.; Meimand, V.; Zhang, N.; Ferrari-Church, B.; Chandra, S. How Much Does AI Impact Development Speed? An Enterprise-Based Randomized Controlled Trial. In *Proceedings of the IEEE/ACM International Conference on Software Engineering-Software Engineering in Practice, Ottawa, ON, Canada, 27 April–3 May 2025*; Institute of Electrical and Electronics Engineers Inc.: New York, NY, USA, 2025; pp. 618–629.

3. Ross, S.I.; Martinez, F.; Houde, S.; Muller, M.; Weisz, J.D. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces, Sydney, Australia, 27–31 March 2023*; ACM: New York, NY, USA; pp. 491–514.
4. Clark, A.; Igbokwe, D.; Ross, S.; Zibran, M.F. A Quantitative Analysis of Quality and Consistency in AI-Generated Code. In *Proceedings of the 2024 7th International Conference on Software and System Engineering, ICoSSE 2024, Paris, France, 19–21 April 2024*; Institute of Electrical and Electronics Engineers Inc.: New York, NY, USA, 2024; pp. 37–41.
5. Peng, J.; Cui, L.; Huang, K.; Yang, J.; Ray, B. CWEval: Outcome-Driven Evaluation on Functionality and Security of LLM Code Generation. In *Proceedings of the 2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, Ottawa, ON, Canada, 3 May 2025; pp. 33–40.
6. Kim, E.; Lee, S. SQL Injection in LLM-Generated Queries: Systematic Analysis of Detection Gaps and Security Risks. *IEEE Access* **2026**, *14*, 12797–12815. [\[CrossRef\]](#)
7. Katzy, J.; Huang, Y.; Panchu, G.R.; Ziemlewski, M.; Loizides, P.; Vermeulen, S.; van Deursen, A.; Izadi, M. A Qualitative Investigation into LLM-Generated Multilingual Code Comments and Automatic Evaluation Metrics. In *Proceedings of the PROMISE'25: 21st International Conference on Predictive Models and Data Analytics in Software Engineering, Trondheim, Norway, 26 June 2025*; Association for Computing Machinery, Inc.: New York, NY, USA, 2025; pp. 31–40.
8. Saxena, A. Rethinking Software Testing for Modern Development. *Computer* **2025**, *58*, 49–58. [\[CrossRef\]](#)
9. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.d.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374. [\[CrossRef\]](#)
10. Cheour, R.; Khriji, S.; Abid, M.; Kanoun, O. Microcontrollers for IoT: Optimizations, Computing Paradigms, and Future Directions. In *Proceedings of the IEEE World Forum on Internet of Things, WF-IoT 2020-Symposium Proceedings, New Orleans, LA, USA, 2–16 June 2020*; Institute of Electrical and Electronics Engineers Inc.: New York, NY, USA, 2020.
11. Chang, Y.; Wang, X.; Wang, J.; Wu, Y.; Yang, L.; Zhu, K.; Chen, H.; Yi, X.; Wang, C.; Wang, Y.; et al. A Survey on Evaluation of Large Language Models. *ACM Trans. Intell. Syst. Technol.* **2024**, *15*, 39. [\[CrossRef\]](#)
12. Cavagnis, L. Arduino AI Assistant vs. ChatGPT: Which One to Use for Your Projects? | Arduino Blog. Available online: <https://blog.arduino.cc/2025/10/16/arduino-ai-assistant-vs-chatgpt-which-one-should-you-use-for-your-projects/> (accessed on 29 November 2025).
13. Zhuo, T.Y.; Vu, M.C.; Chim, J.; Hu, H.; Yu, W.; Widyasari, R.; Yusuf, I.N.B.; Zhan, H.; He, J.; Paul, I.; et al. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *Proceedings of the 13th International Conference on Learning Representations, ICLR 2025; International Conference on Learning Representations, ICLR, Singapore, 24–28 April 2025*; pp. 99488–99542.
14. Jain, N.; Han, K.; Gu, A.; Li, W.D.; Yan, F.; Zhang, T.; Wang, S.I.; Solar-Lezama, A.; Sen, K.; Stoica, I. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *Proceedings of the 13th International Conference on Learning Representations, ICLR 2025; International Conference on Learning Representations, ICLR, Singapore, 24–28 April 2025*; pp. 25479–25519.
15. Liu, J.; Xia, C.S.; Wang, Y.; ZHANG, L. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Proceedings of the Thirty-Seventh Conference on Neural Information Processing Systems, New Orleans, LA, USA, 10–16 December 2023*.
16. Yang, J.; Jimenez, C.E.; Zhang, A.L.; Lieret, K.; Yang, J.; Wu, X.; Press, O.; Muennighoff, N.; Synnaeve, G.; Narasimhan, K.R.; et al. SWE-Bench Multimodal: Do AI Systems Generalize to Visual Software Domains? In *Proceedings of the The Thirteenth International Conference on Learning Representations, Singapore, 24 April 2025*.
17. Li, J.; Li, G.; Zhang, X.; Zhao, Y.; Dong, Y.; Jin, Z.; Li, B.; Huang, F.; Li, Y. EvoCodeBench: An Evolving Code Generation Benchmark with Domain-Specific Evaluations. In *Proceedings of the Thirty-Eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track, Vancouver, BC, Canada, 10–15 December 2024*.
18. Guo, J.; Li, Z.; Liu, X.; Ma, K.; Zheng, T.; Yu, Z.; Pan, D.; LI, Y.; Liu, R.; Wang, Y.; et al. CodeEditorBench: Evaluating Code Editing Capability of Large Language Models. *arXiv* **2025**, arXiv:2404.03543.
19. Callan, J.; Chen, X.; Du, Y.; Fan, C.; Gao, L.; Guo, X.; Haas, R.; Huerta, E.; Ji, P.; Krongchon, K.; et al. SciCode: A Research Coding Benchmark Curated by Scientists. In *Proceedings of the Advances in Neural Information Processing Systems 37, Vancouver, BC, Canada, 10–15 December 2024*; Neural Information Processing Systems Foundation, Inc. (NeurIPS): San Diego, CA, USA, 2024; pp. 30624–30650.
20. Chai, L.; Liu, S.; Yang, J.; Yin, Y.; Jin, K.; Liu, J.; Sun, T.; Zhang, G.; Ren, C.; Guo, H.; et al. McEval: Massively Multilingual Code Evaluation. *arXiv* **2024**, arXiv:2406.07436. [\[CrossRef\]](#)
21. Englhardt, Z.; Li, R.; Nissanka, D.; Zhang, Z.; Narayanswamy, G.; Breda, J.; Liu, X.; Patel, S.; Iyer, V. Exploring and Characterizing Large Language Models for Embedded System Development and Debugging. In *Proceedings of the Conference on Human Factors in Computing Systems-Proceedings, Yokohama, Japan, 26 April–1 May 2025*; Association for Computing Machinery: Honolulu, HI, USA, 2024.

22. Quan, P.; Ouyang, X.; Jeyakumar, J.V.; Wang, Z.; Xing, Y.; Srivastava, M. SensorBench: Benchmarking LLMs in Coding-Based Sensor Processing. In *Proceedings of the 26th International Workshop on Mobile Computing Systems and Applications, La Quinta, CA, USA, 26–27 February 2025*; ACM: New York, NY, USA, 2025; pp. 25–30.
23. Delorenzo, M.; Gohil, V.; Rajendran, J. CreativEval: Evaluating Creativity of LLM-Based Hardware Code Generation. In *Proceedings of the 2024 IEEE LLM Aided Design Workshop, LAD 2024, Almaden, CA, USA, 28 June 2024*; Institute of Electrical and Electronics Engineers Inc.: New York, NY, USA, 2024.
24. Yang, H.; Li, M.; Han, M.; Li, Z.; Xu, W. EmbedGenius: Towards Automated Software Development for Generic Embedded IoT Systems. In *Proceedings of the ACM Conference (Conference’17), Virtual, 12 December 2024*; Volume 1.
25. Xu, R.; Cao, J.; Wu, M.; Zhong, W.; Lu, Y.; He, B.; Han, X.; Cheung, S.-C.; Sun, L. EmbedAgent: Benchmarking Large Language Models in Embedded System Development. In *Proceedings of the Symposium on Operating Systems Principles, Seoul, Republic of Korea, 13–16 October 2025*; Volume 1.
26. Saaty, R.W. The Analytic Hierarchy Process—What It Is and How It Is Used. *Math. Model.* **1987**, *9*, 161–176. [[CrossRef](#)]
27. Spracklen, J.; Wijewickrama, R.; Sakib, A.H.M.N.; Maiti, A.; Viswanath, B.; Jadliwala, M. We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs. In *Proceedings of the 34th USENIX Conference on Security Symposium, Seattle, WA, USA, 13–15 August 2025*; USENIX Association: Berkeley, CA, USA, 2025.
28. Ma, Q.; Shen, H.; Koedinger, K.; Wu, S.T. How to Teach Programming in the AI Era? Using LLMs as a Teachable Agent for Debugging. In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer Nature: Cham, Switzerland, 2024; Volume 14829 LNAI, pp. 265–279. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.