When the Code Autopilot Breaks: Why LLMs Falter in Embedded Machine Learning

Roberto Morabito
EURECOM
Biot, France
roberto.morabito@eurecom.fr

Guanghan Wu
University of Helsinki
Helsinki, Finland
guanghan.wu@helsinki.fi

***Abstract*—Large Language Models (LLMs) are increasingly used to automate software generation in embedded ML workflows, yet their outputs often fail silently or behave unpredictably. This article presents an empirical investigation of failure modes in LLM-powered pipelines, based on an autopilot framework that orchestrates data preprocessing, model conversion, and on-device deployment code generation. We show how prompt format, model behavior, and structural assumptions influence both success rates and failure characteristics, often in ways that standard validation pipelines fail to detect. Our analysis reveals a diverse set of error-prone behaviors, including format-induced misinterpretations and runtime-disruptive code that compiles but breaks downstream. We derive a taxonomy of failure categories and analyze errors across multiple LLM families, highlighting common root causes and systemic fragilities. We conclude with practical mitigation strategies and emphasize the need for failure-aware orchestration mechanisms to improve the reliability and traceability of LLM-powered embedded ML systems.**

***Index Terms*—Large Language Models (LLMs), Embedded Machine Learning, Code Generation Failures, Prompt Engineering, Failure-Aware Automation**

I. Introduction

Automating embedded machine learning (ML) workflows, especially for resource-constrained IoT devices, remains a highly complex challenge [1]. From data processing to model deployment, each stage in the embedded ML lifecycle requires domain-specific expertise, toolchain orchestration, and careful hardware alignment. These challenges are amplified in large-scale deployments, where manual coordination becomes impractical. While traditional automation frameworks exist [2], they often address isolated tasks, such as model quantization or deployment, but leave the coordination between stages to human developers. In particular, there is no end-to-end tooling that automates the entire pipeline from raw dataset ingestion to deployable microcontroller code. This lack of integration increases development time and limits scalability across hardware platforms and application domains.

In parallel, recent advances in Large Language Models (LLMs) have opened new opportunities for integrating natural language understanding and code generation into ML workflows [3], [4]. Yet, using LLMs as "plug-and-play" components within embedded pipelines is far from trivial. For example, the automation of *deployment sketches*—i.e., the executable code responsible for running the final ML inference on-device—is not an isolated task. It depends critically on the correctness and alignment of earlier pipeline stages (e.g., model conversion – MC), seamless interaction with device I/O, and tight coupling with the hardware-specific constraints of the target platform. These limitations raise a fundamental question: *Can LLMs be relied upon to automate critical stages of embedded ML workflows, and if not, why do they fail?*

In this article, we report on a comprehensive system we built to explore this question. Rather than conducting isolated prompt tests, we developed and evaluated an end-to-end middleware framework that orchestrates LLM interactions across key embedded ML lifecycle stages [5]. Our system integrates structured prompt design, iterative feedback loops, local validation steps, and tooling integration with embedded ML libraries (e.g., TensorFlow Lite) and constrained IoT devices such as Arduino. This system, which we refer to as the *Embedded ML Autopilot*, was not only designed to reduce human effort, but it also served as a practical lens through which we encountered, firsthand, the limitations and failure points of LLM-powered automation pipelines.

Through a detailed case study and cross-model analysis, we uncover a range of error-prone behaviors, including format-induced misalignment, semantic errors that pass compilation, and unexpected behavioral variations across models. Our findings are synthesized into a taxonomy of failure types derived from an analysis of over a thousand log-traced errors collected across multiple LLMs. We show how prompt structure, decoding strategy, and model family influence both success rates and failure patterns, and we identify a specific lifecycle stage, sketch generation (SG), as disproportionately fragile.

While combining LLM reasoning with embedded computing systems constraints, this work aims to contribute with practical insights, actionable diagnostics, and forward-looking strategies for building failure-aware AI systems. We argue that reliable automation will require not just better models, but also architectures that can anticipate, detect, and respond to LLM-induced faults [6], which is an essential step toward scalable, intelligent automation in the embedded AI domain.

This paper is currently under review for publication in an IEEE magazine. If accepted, the copyright will be transferred to IEEE.

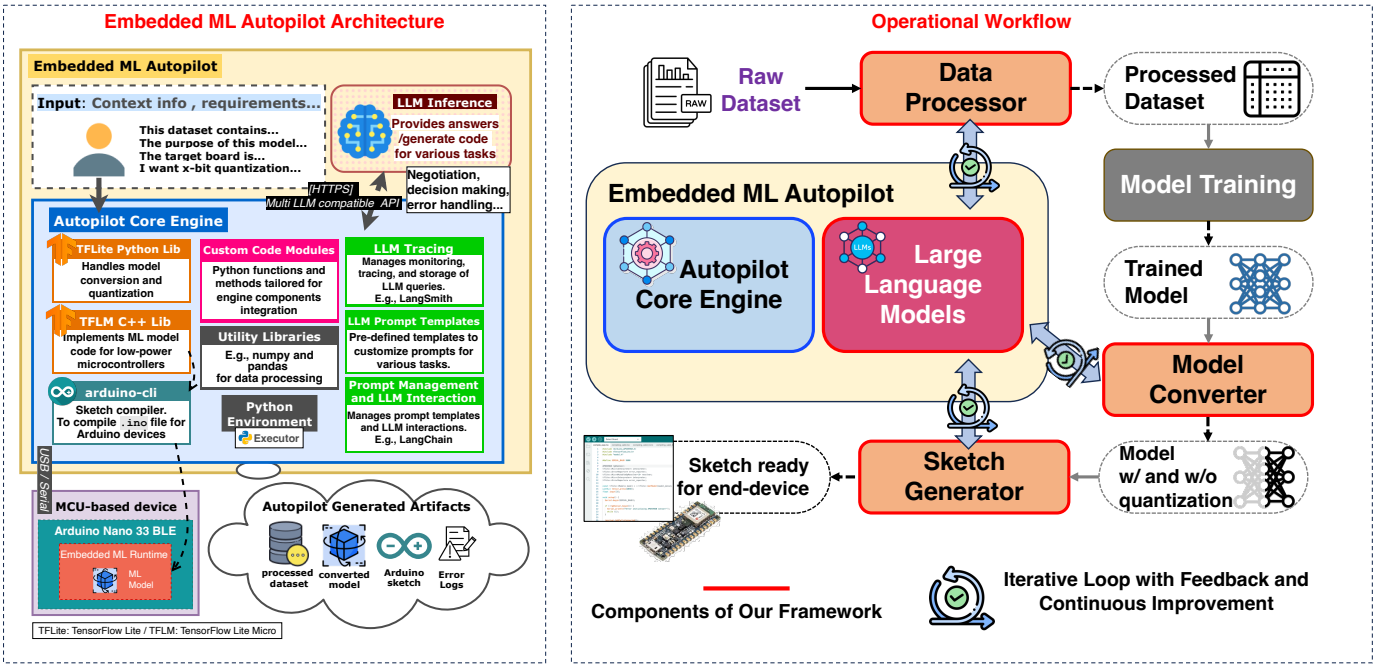

Fig. 1: Left: System architecture of the Embedded ML Autopilot, showcasing its internal modules, LLM integration, and runtime environment for managing ML model preparation and deployment. Right: Operational workflow enabled by the Autopilot, outlining the ML lifecycle stages from dataset ingestion to device-ready sketch generation.

II. Background and Context

Automating embedded ML pipelines requires bridging high-level model reasoning with the realities of low-level hardware constraints. In this context, our work builds on a comprehensive system architecture, the *Embedded ML Autopilot*, designed to orchestrate interactions between LLMs and the stages of the embedded ML workflow. The Autopilot framework enables not only generation of deployment artifacts (such as microcontroller-ready code), but also serves as an experimental infrastructure for observing and diagnosing model behavior across stages.

A. System Overview: The Embedded ML Autopilot

Figure 1 outlines the two central components of our system: the **Autopilot Core Engine** and the **Operational Workflow**.

The left panel presents the architectural components of the Embedded ML Autopilot. At the heart of the system is the Core Engine, which integrates a range of task-specific modules: prompt templates, utility libraries, API handling logic, and dedicated components for LLM interaction and validation. These modules coordinate actions such as MC, quantization, and SG. Notably, the engine supports different LLM families and backends and is designed to operate with limited or no internet connectivity, which is highly important for private or local deployments.

The right panel illustrates the end-to-end data flow. The workflow begins with raw dataset ingestion and proceeds through a sequence of transformation steps, including DP, model training and conversion, and deployment SG. At each step, the Autopilot invokes the LLM to generate intermediate outputs, which are validated and refined through feedback loops. This design allows the framework to simulate real-world embedded ML development conditions, where generated code must align with hardware constraints, available libraries, and execution requirements.

B. Scope and Focus of This Article

A full architectural and implementation-level description of the Embedded ML Autopilot is provided in our earlier work [5]. Instead, the focus of this article is to investigate the behavioral patterns and failure mechanisms that emerge during its operation. Despite our initial goal of building a fully automated framework for embedded ML development, our hands-on experience revealed fundamental limitations that persist even with sophisticated prompt engineering, carefully tailored system integration, and extensive iteration. These findings are not just theoretical: they stem from months of engineering effort, testing, and system refinement, ultimately exposing a gap between the promise of LLM-driven automation and its current real-world reliability.

To study these gaps systematically, the Autopilot system captures detailed logs of each interaction, including prompt structure, decoding outcomes, LLM responses, and downstream errors. These logs form the foundation of the failure analysis discussed in later sections of this paper. Unlike prompt-centric benchmarking setups, our framework observes the complete lifecycle behavior of the models, from data ingestion to on-device code readiness, revealing rich patterns of structural fragility, behavioral inconsistencies, and task-stage failures.

Among the various use cases supported by our framework, this article focuses specifically on one of the most demanding scenarios we tested: the automated generation of executable code (*"sketch"*) for a microcontroller-based vision application. This setup involves an embedded ML model running on a resource-constrained device (e.g., Arduino), integration with a color sensor for I/O operations, and execution of model inference through deployment sketches. We chose this case as a *stress test* for the system, as it exercises all key stages of the pipeline—data handling, MC, on-device inference setup, and hardware-specific code generation—and thus offers a rich space for observing how and where LLM-based automation breaks down. While data preprocessing (DP) and MC stages generally exhibited high success rates with only minor recoverable issues, we observed that the SG stage consistently suffered from the lowest reliability, falling below 40% success across models and settings. This stage revealed disproportionately fragile behavior across prompt formats and model outputs, making it the primary focus of our analysis.

III. Characterizing Failures In Sketch Generation

We now focus on the SG stage, where we observed the most persistent reliability issues. Rather than concentrating solely on explicit failure states such as syntax errors or missing includes, we take a broader view: examining subtle behavioral inconsistencies, formatting fragility, semantic gaps, and runtime-disrupting edge cases. We highlight how seemingly minor prompt variations can dramatically affect outcomes, how different model families exhibit divergent generation tendencies, and how even well-formed code can silently fail during deployment. These findings lay the foundation for the formal taxonomy introduced in the next section and inform our recommendations for model selection, prompt design, and failure-aware orchestration.

A. Prompt Structure Sensitivity: A Source of Hidden Fragilities

While SG consistently emerged as the most failure-prone stage in our framework, a deeper investigation revealed that the structure of the prompt itself, particularly the use of nested JSON-like objects, significantly influenced LLM behavior. To evaluate this effect, we designed an experiment comparing three prompt variants (SG0, SG1, and SG2), each differing in how task, specification, and guideline components were embedded. As can be observed on the left side of Figure 2, SG0 places the entire prompt in a deeply nested JSON format, including the task and guideline sections. SG1 flattens the structure slightly by moving the task outside the JSON but retains the application specification and embedded guideline as JSON-like objects. SG2 flattens the structure further by moving the guideline out into Markdown format as a separate prompt block.

The right side of Figure 2 shows the trade-offs across the three prompt templates in terms of token consumption, execution latency, and success rate. Despite SG2 consuming the highest number of tokens (18.6K avg) and requiring the longest inference time among successful runs, it had the lowest success rate (15%). SG1, by contrast, achieved the highest success rate (30%) with moderately lower token usage and significantly shorter inference time. SG0 had an intermediate success rate (25%) but was the least efficient, with the longest execution time (87.4s avg) and highest resource variability across all runs. Finally, SG3 is the most efficient in terms of token and time consumption, but this comes at the cost of reduced reliability.

This behavior suggests that LLM performance can degrade not only due to ambiguous content, but also due to the structure in which content is presented [7], even if semantically equivalent. It also highlights a non-monotonic correlation between resource use and success: more detailed prompts do not necessarily yield better outcomes.

We interpret this as a form of format-induced misalignment, where the LLM's internal parsing strategies or token interpretation heuristics may fail to resolve deeper structural contexts, especially when involving nested schemas or overloaded input formats.

These findings have several practical implications:

- **(i)** Prompt structures must be carefully engineered, not only for correctness but for LLM readability under constrained conditions.
- **(ii)** Token budget alone is an unreliable predictor of task success, suggesting that iterative retries may only exhaust cost without gain
- **(iii)** Failure modes linked to structure are often silent, meaning they do not manifest in error messages, making them harder to detect without systematic testing.

This prompt structure sensitivity introduces a new class of AI failures—those rooted in format misinterpretation rather than knowledge gaps or logic errors. In the broader context of deploying LLMs in IoT or embedded workflows, this insight calls for integrating prompt testing frameworks [8], adopting explicit interface contracts, and enforcing consistency checks between stages.

B. Emergent Patterns from Open Models: What They Do (and Don't) Get Right

To assess whether the SG failures observed in GPT-4o are unique to closed-source commercial LLMs, we replicated key lifecycle steps using a set of open-source LLMs integrated into our framework. These included models optimized for instruction-following or code generation, run locally without *over-the-Internet* API calls. Our goal was not to benchmark performance, but to evaluate whether structural and format-related failures generalize across model types.

The results confirmed our hypothesis: open-source models frequently exhibited the same classes of failure, often with additional parsing-related breakdowns.

To support this qualitative breakdown, we extended our evaluation to include several state-of-the-art open-source LLMs, including Phi-4, Llama3.1, Qwen2.5-Coder, Deepseek-R1, and Codestral. These models were tested under the same

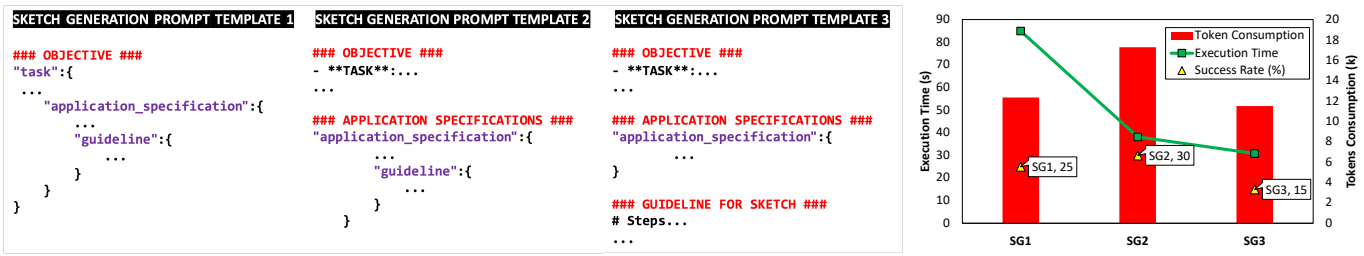

Fig. 2: Comparison of three prompt templates (SG1, SG2, SG3) used for SG. The results highlight trade-offs between token usage, execution latency, and success rate. SG2 demonstrates the highest success rate, while SG3 minimizes token consumption.

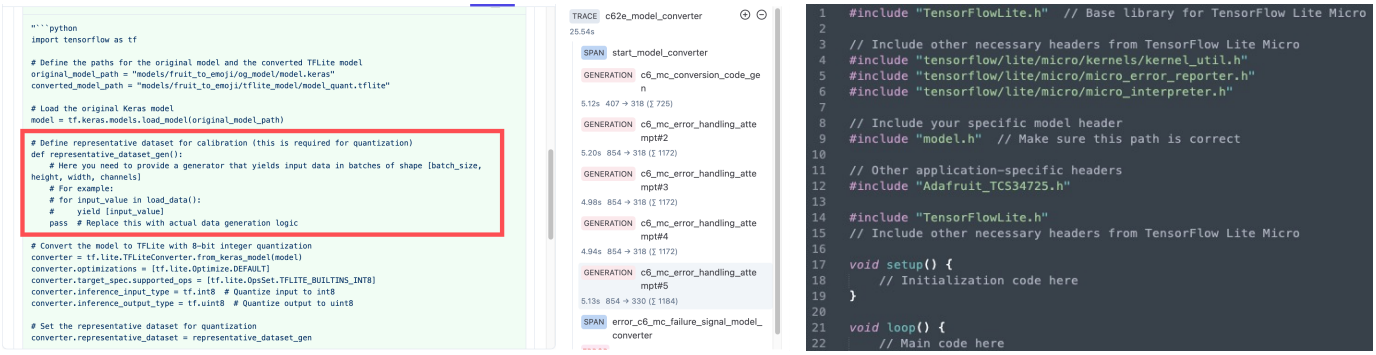

Fig. 3: Left: Example of multiple candidate completions, where the LLM embeds alternative code paths inside the explanation, introducing ambiguity and requiring manual disambiguation. Right: Example of a compilable but incomplete sketch, where structurally valid code omits the core inference logic, resulting in silent functional failure.

three-stage lifecycle pipeline, i.e., DP, MC, and SG, used for the GPT-based models.

Across these models, we observed a notable decline in reliability, most prominently in SG, where success rates dropped to 0% in all but one case. Only Codestral 22B reached ≈11% success in SG, while others remained at or near zero. Even in DP and MC, the performance was significantly lower than with GPT-4o, where success rates routinely exceeded 90%. Phi-4 14B, for instance, achieved a 50% success rate in DP and 97% in MC, but only 3% in SG.

These observations underscore that model capability significantly affects success, particularly in tasks requiring multi-step reasoning and structurally valid output [9]. However, the persistence of similar error types across model families suggests that these failures stem not only from model limitations, but also from deeper vulnerabilities in the current LLM-based development paradigm, such as weak enforcement of format consistency, lack of semantic validation, and prompt interpretation fragility.

We highlight two failure patterns that were especially common:

Case 1 **Unparsable or mixed-format output:** Some models produced outputs with interleaved explanation and code, or alternated between multiple code blocks without presenting a complete executable sketch. This caused our automated parsing logic to fail.

Case 2 **Multiple candidate completions without structure:** In several runs, the model proposed alternative snippets or variants, asking the user to *"choose the appropriate one"*. While this may be helpful in interactive settings, it introduces ambiguity in automation pipelines. Figure 3-left illustrates such a scenario, where multiple solutions are embedded inside the explanation text, requiring human disambiguation.

These examples reinforce that prompt sensitivity [7] and validation blind spots are not exclusive to proprietary LLMs. Even with full model transparency and local deployment, the lack of enforced output structure and the absence of end-to-end behavioral validation can lead to silent or delayed failures. Moreover, format inconsistencies—such as Markdown misalignment, missing triple backticks, or ambiguous nesting—proved equally disruptive in open-source settings.

Mitigation Strategy

From an engineering standpoint, these findings suggest the need for model-agnostic validation layers, such as: *(i)* output format checkers (e.g., complete sketch in a single block), *(ii)* structural linters for code completeness, and *(iii)* post-generation semantic validators (e.g., checking if inference function is called).

C. When Code Compiles but Breaks: Semantic Gaps and Pipeline Disruptions

In addition to structural issues in prompt formatting and code block generation, our evaluation uncovered a set of more subtle and insidious failures. These are not directly tied to LLM misunderstanding of prompts, but rather arise from semantic mismatches or pipeline-breaking behaviors, even when the output passes syntactic or structural checks.

Group	Meta-Category	Description	Example
Specification & Prompt Errors	Specification Parsing Failure	*The model failed to interpret or generate valid application specifications.*	`Failed to generate valid application specifications after 5 attempts.`
	Prompt Handling / Format Error	*Errors due to improper or unrecognized input format to the model.*	`Unsupported output format. No code blocks found.`
LLM Execution & Backend	LLM Runtime / Backend Error	*Failures in reaching or using the LLM backend or runtime environment.*	`OllamaException: Connection refused`
	Prompt Execution Error	*Internal exceptions during LLM prompt execution cycles.*	`Exception caught in prompting attempt 3, loop continues.`
Code Generation	Code Generation Failure	*The code generation process failed or produced non-compilable output.*	`code generation process failed`
	Syntax Error in Generated Code	*The generated code has syntax-level issues, e.g., stray characters or malformed expressions.*	`error: stray '`' in program`
	Include / Import Error	*Compiler errors triggered by incorrect or missing include/import statements.*	`In file included from ...`
	Missing Utility Function*	*Functions expected by the compiler were not declared in the generated code.*	`error: 'loadBinaryFile' was not declared in this scope`
Libraries & APIs	Missing / Incorrect Libraries	*Missing library files or incorrect library configurations in the build.*	`No such file or directory`
	Symbol / Class API Issues	*Code references unknown or incorrectly declared classes or methods.*	`'SomeClass' has no member named 'run'`
Embedded ML / TensorFlow Lite	Invalid TensorFlow Lite Usage	*Incorrect use of TensorFlow Lite types, objects, or APIs.*	`'tflite' has not been declared`
	Invalid Custom TensorFlow Usage*	*Attempts to use full TensorFlow functionality on constrained platforms like MCUs.*	`tensorflow::MicroTensorArena does not name a type`
	Sensor / Hardware Constant Not Declared	*References to device-specific constants or macros not defined in the scope.*	`'TCS34725_INTEGRATIONTIME_700MS' was not declared in this scope`
	Missing Main Functions*	*The generated code lacks the required Arduino main functions like `setup()` or `loop()`.*	`undefined reference to `setup'`

TABLE I: Grouped taxonomy of LLM-driven code generation errors, with descriptions and real-world examples.

These failures are especially concerning in automation pipelines, where generated code is validated through compilation, simple I/O checks, or success logs. The following three real-world examples illustrate such issues and highlight why engineering workflows must go beyond output correctness and introduce semantic and contextual validation.

Case 1 **False Positives from Compilable but Incomplete Sketches:** We observed that some SGs produced compilable `.ino` files that passed our structural and compilation checks, yet failed to perform the intended operations when deployed to the target microcontroller. In one instance, the sketch invoked a color sensor routine instead of running the actual TinyML inference. The resulting code was structurally valid, included the necessary headers, and returned no errors, but did not contain any logic related to the trained model or data input pipeline. This failure highlights a dangerous case: compilation $\neq$ functional correctness [10]. While automated workflows typically validate compilation and function calls, the LLM-generated code here skipped the core application logic, leading to silent deployment of a non-functional model. Figure Figure 3-right provides a visual example of this issue, where downstream sensor logic is missing or replaced with irrelevant placeholder code.

Mitigation Strategy

LLM-based SG tools require behavioral validation layers, including: *(i)* minimal I/O tracing or signal path verification during deployment, *(ii)* targeted unit testing of generated functions, or *(iii)* testbed-based validation (e.g., observing serial outputs in response to simulated inputs).

Case 2 **Silent Breakage from Incomplete Data Processing:** In the DP stage, we identified a similar pattern: the generated Python script would load and analyze a dataset but fail to save the transformed output, even though a success message was printed and the next step was triggered. In several instances, the script reported a new file path (via JSON), but that file is never actually written to disk. This results in false positives in step-wise validation: the system believes the dataset transformation succeeded, while the next component fails when attempting to load the nonexistent file. This type of failure arises from semantic hallucination in output logging or response formatting: the LLM generates correct-looking

I/O structures but does not enforce file I/O consistency. In long pipeline workflows, these inconsistencies can propagate silently and are hard to trace back without audit logging or output path enforcement.

Mitigation Strategy

We emphasize the need for: *(i)* runtime existence checks of output artifacts, *(ii)* semantic consistency verification across stages, and *(iii)* potentially adopting output schema templates to align model-generated paths with expected file structure.

IV. Taxonomy Of Failure Categories

The previous section described key behaviors and system interactions observed during SG. We now broaden the view with a categorized taxonomy of over a thousand observed failure cases.

The errors observed across both proprietary and open-source LLMs reflect a diverse set of underlying causes that span beyond prompt engineering or token length limitations [11]. We analyzed over 200 of failed SG attempts per model and categorized the resulting errors into a multi-layered taxonomy based on where and why the generation failed.

The grouped failure categories, shown in Table I, cover a range of issues, from shallow formatting errors and missing libraries to deeper semantic and API-level faults. While some failures (e.g., missing `setup()` functions or malformed headers) are recoverable through retries or static validation, others (e.g., incorrect use of embedded TensorFlow APIs or symbol mismatches) point to more fundamental limitations in LLM reasoning and code alignment.

Table I presents the grouped taxonomy of SG failures, organized into five major groups: *(i)* Specification & Prompt Errors, *(ii)* LLM Execution & Backend, *(iii)* Code Generation, *(iv)* Libraries & APIs, *(v)* Embedded ML / TensorFlow Lite. This breakdown allows us to identify both frequent error types (e.g., missing includes or syntax errors) and critical bottlenecks (e.g., incorrect TensorFlow Lite usage on MCUs), and forms the basis for the mitigation strategies discussed in the following section.

Furthermore, several of these categories represent failure types that pass conventional compilation checks but are only caught at deployment or runtime, such as semantic misalignments, placeholder logic, or undeclared sensor constants. These are especially dangerous in automated pipelines, as they evade early validation layers and silently propagate incorrect behavior. Through the formalization of this taxonomy, we aim to shift the discussion from isolated errors to systemic design limitations in prompt-driven code generation workflows. Each category reveals not just a symptom, but also a class of assumptions about what the model understands and what it doesn't.

A. Error Profiling Across Language Models

To quantify how different LLMs behave in real-world SG tasks, we performed a fine-grained log-based analysis across multiple model families. The resulting distribution of errors was categorized using the taxonomy shown in Table I, and aggregated to highlight model-specific weaknesses and systemic failure trends.

Figure 4 summarizes the normalized error rate per failure category for four representative LLMs: GPT-4o (`gpt-4o-2024-08-06`), Qwen2.5-Coder 32B, Codestral 22B, and Gemma 3 27B. Each green bar indicates the frequency of a particular error type across test samples for that model, normalized per sample. The cumulative error curve (in red) captures how quickly the most frequent error types accumulate across all failures.

From this analysis, several key insights emerge. *Code Generation Failure* and *Syntax Errors* dominate across all models, especially GPT-4o and Qwen2.5-Coder. This suggests a recurring inability to produce fully compilable code when following structured multi-step prompts. *TensorFlow Lite misuse*, such as incorrect type references or attempts to invoke functions unsupported on microcontrollers, was especially prominent in open-source models like Codestral and Gemma. This reflects a lack of model grounding in hardware-specific constraints. *Missing or incorrect library usage*, including absent includes or undeclared utility functions, accounted for a large share of failures, particularly in the open-source models. *Prompt execution and runtime exceptions* were far more common in open-source models, possibly due to weaker instruction following or model misalignment with the JSON-like prompt structure used in the system. Some models (notably GPT-4o) exhibited a wider distribution of low-frequency errors (e.g., undeclared sensor constants, API symbol mismatches), pointing to higher variability in code generation attempts.

Main Takeaway

This empirical profiling reinforces the notion that certain failure types are deeply systemic, while others are amplified by specific model weaknesses, such as lack of embedded ML domain grounding or inconsistent formatting behavior.

Mitigation Strategy

Being able to characterize the relative prominence of failure categories per model, we can prioritize mitigation strategies that are both model-agnostic (e.g., checking for syntax or missing includes) and model-specific (e.g., preventing TensorFlow misuse on MCUs). These insights also inform the need for failure-aware orchestration systems, where the choice of model is influenced by its known error tendencies for a given task type.

B. Recommendations for Failure-Aware System Design

Our empirical study reveals that many failure types stem not only from model limitations but from broader mismatches between LLM behavior and pipeline requirements. Based on the patterns observed, we highlight four mitigation priorities for practitioners building similar systems:

1) **Failure-aware orchestration**. Pipeline components should incorporate guardrails, retries, and structural validators that anticipate silent or partial failures, especially in later stages like SG. This includes embedding explicit I/O tracing, output structure enforcement, and semantic verifiers into the code execution layer.

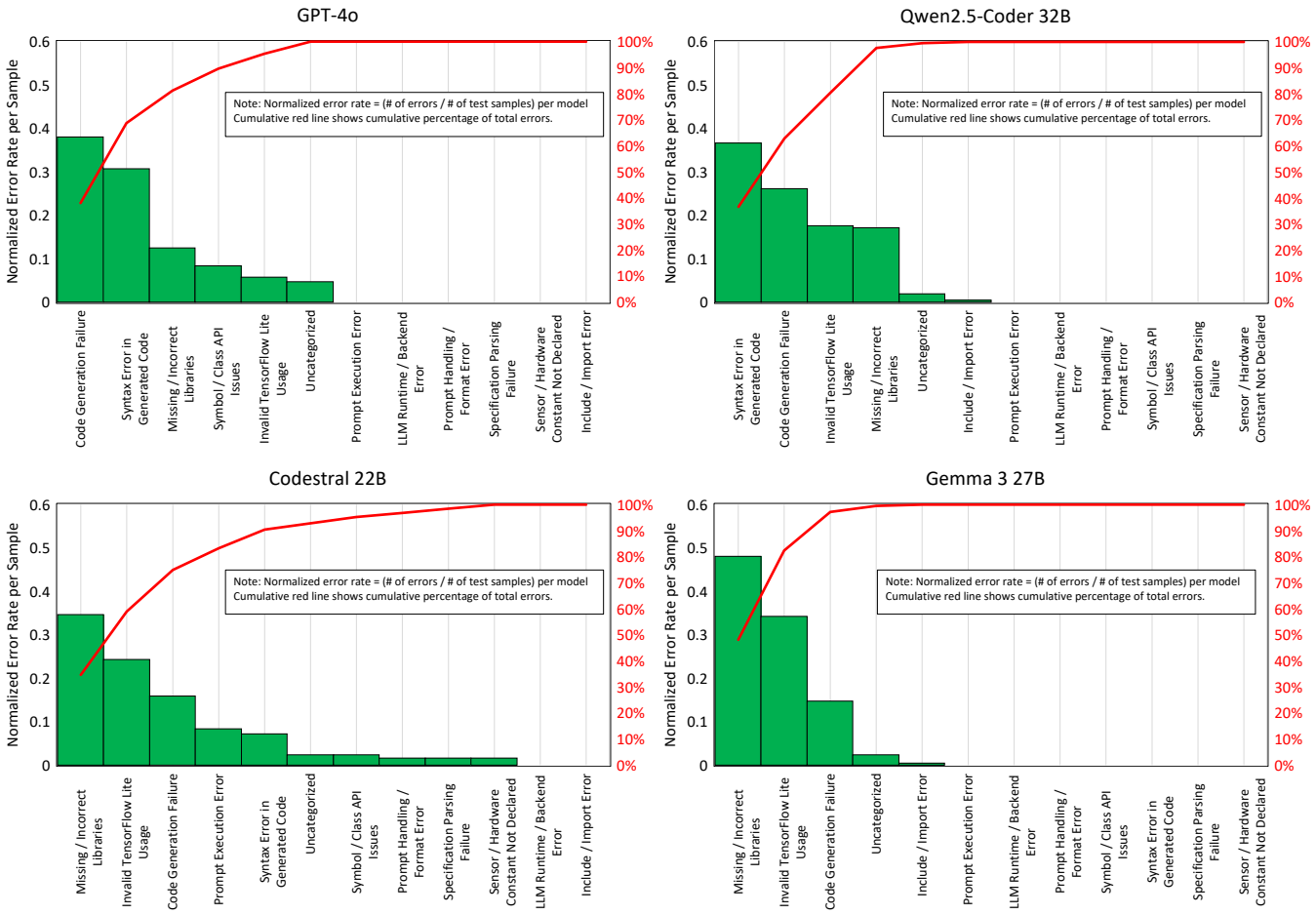

Fig. 4: Distribution of error types across four LLMs evaluated in our system. The bars represent the normalized error rate per test sample for each error category, while the red line shows the cumulative percentage of total errors. This analysis highlights systematic trends and model-specific weaknesses in code generation workflows for embedded systems.

2) **Prompt and decoding standardization.** Consistent prompt formatting and fixed decoding parameters reduce variance and improve reliability. While open-source models allow direct control, developers using closed APIs should push for transparency and interface stability to ensure replicable outcomes.
3) **Design for introspection.** LLM integration layers should not only generate code but log, annotate, and monitor key decisions. Our use of structured logs, runtime metadata, and stage-wise validation proved essential in surfacing otherwise undetected issues.
4) **Towards symbolic guidance and smart prompt templates.** Future systems may benefit from symbolic or constraint-based instruments to guide generation and enforce structure explicitly. As model providers increasingly adopt templated interaction formats (e.g., system prompts in Ollama or HuggingFace), a unified, context-aware orchestration layer will be crucial to adapt these templates intelligently based on task, hardware, or pipeline stage.

V. Conclusion

In this work, we presented an empirical study of failure modes emerging from LLM-driven automation of embedded ML pipelines, with a particular focus on the SG stage. By operating a real-world orchestration system and analyzing thousands of generation logs, we uncovered recurring patterns of structural, semantic, and behavioral fragility—many of which are silent or hard to trace. Our findings highlight the need for failure-aware orchestration, structured prompt design, and symbolic or template-guided control. As LLMs are increasingly integrated into development workflows, robustness and explainability must become first-class design goals.

VI. Note on Reproducibility

To support transparency and enable further exploration, we have released the generation logs and analysis scripts used in this study via a public GitHub repository[0]. The provided resources allow reproduction of key results and offer flexibility

[0]https://github.com/robertmora/embeddedML-autopilot-failures

for extending the failure analysis across decoding settings, models, or error types.

REFERENCES

[1] M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, "Tinyml: Current progress, research challenges, and future roadmap," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1303–1306.

[2] S. Hymel, C. Banbury, D. Situnayake, A. Elium, C. Ward, M. Kelcey, M. Baaijens, M. Majchrzycki, J. Plunkett, D. Tischler, A. Grande, L. Moreau, D. Maslov, A. Beavis, J. Jongboom, and V. J. Reddi, "Edge Impulse: An MLOps Platform for Tiny Machine Learning," Apr. 2023. [Online]. Available: http://arxiv.org/abs/2212.03332

[3] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "Llm-based test-driven interactive code generation: User study and empirical evaluation," *IEEE Transactions on Software Engineering*, 2024.

[4] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 2023, pp. 31–53.

[5] G. Wu, S. Tarkoma, and R. Morabito, "Consolidating tinyml lifecycle with large language models: Reality, illusion, or opportunity?" *IEEE Internet of Things Magazine*, 2025, accepted for publication; preprint available at arXiv:2501.12420.

[6] X. Zhang, F. T. Chan, C. Yan, and I. Bose, "Towards risk-aware artificial intelligence and machine learning systems: An overview," *Decision Support Systems*, vol. 159, p. 113800, 2022.

[7] F. Errica, G. Siracusano, D. Sanvito, and R. Bifulco, "What did i do wrong? quantifying llms' sensitivity and consistency to prompt engineering," *arXiv preprint arXiv:2406.12334*, 2024.

[8] L. Wang, X. Chen, X. Deng, H. Wen, M. You, W. Liu, Q. Li, and J. Li, "Prompt engineering in consistency and reliability with the evidence-based guideline for llms," *NPJ digital medicine*, vol. 7, no. 1, p. 41, 2024.

[9] Y. Wu, X. Han, W. Song, M. Cheng, and F. Li, "Mindmap: constructing evidence chains for multi-step reasoning in large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 17, 2024, pp. 19 270–19 278.

[10] T. Chen, "Challenges and opportunities in integrating llms into continuous integration/continuous deployment (ci/cd) pipelines," in *2024 5th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*. IEEE, 2024, pp. 364–367.

[11] F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code: An empirical study," *Empirical Software Engineering*, vol. 30, no. 3, pp. 1–48, 2025.