This paper has been accepted for publication in the proceedings of the *27th ACM/IFIP International Middleware Conference (Middleware '26)*, December 14–18, 2026, Tarragona, Spain.

Please cite this work as:

Roberto Morabito and Guanghan Wu. *Autopilots Need Parachutes: Lessons Learned from LLM-Automated Embedded ML Pipelines.* In *Proceedings of the 27th ACM/IFIP International Middleware Conference (Middleware '26)*, Tarragona, Spain, 2026. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3801927.3810472

ACM ISBN 979-8-4007-2621-7/26/12

https://doi.org/10.1145/3801927.3810472

Autopilots Need Parachutes: Lessons Learned from LLM-Automated Embedded ML Pipelines

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

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

Abstract

We set out to build an *Autopilot* that automatically executes key stages of the embedded machine learning lifecycle, using large language models (LLMs). Rather than demonstrating seamless automation, our study exposes the trade-offs between feasibility and dependability in LLM-driven embedded ML pipelines. We present an empirical, system-level evaluation based on a fully implemented middleware framework that orchestrates the complete lifecycle, integrates validation and recovery mechanisms, and interacts with multiple LLMs. Using this framework, we execute hundreds of end-to-end runs across a wide spectrum of deployment targets, from highly constrained microcontroller-class devices to less constrained edge boards. Our results show that the primary challenge is not code generation itself, but managing silent failures, repeated retries, and unstable convergence across heterogeneous targets. Small variations in prompt structure and system constraints lead to drastically different outcomes, ranging from immediate success to costly iterative failures. While success rates improve as device constraints are relaxed, reliable automation consistently requires explicit structure, validation, and fallback mechanisms. We contribute *(i)* a systematic feasibility and dependability analysis of LLM-automated embedded ML pipelines, *(ii)* empirical insights into convergence behavior, cost, and semantic alignment across devices and models, and *(iii)* a fully released framework with execution traces and artifacts to support reproducibility. Lesson learned: automation is possible, but only when the system assumes failure from the start. *Autopilots need parachutes.*

Keywords

Embedded Machine Learning, Large Language Models, Middleware Systems, Automated Software Pipelines, Code Generation

1 Introduction

Any sufficiently advanced technology is indistinguishable from magic. – Arthur C. Clarke

This observation has gained renewed relevance with the emergence of Large Language Models (LLMs). Their ability to generate code, reason over text, and maintain context across iterations has triggered growing interest in AI-assisted software development [12, 43]. These capabilities suggest that LLMs could act not only as coding assistants, but as automation engines for complex development workflows. However, embedded machine learning (ML) exposes a markedly different set of challenges. In this domain, hallucinations, inconsistent outputs, and brittle reasoning collide with requirements for determinism, strict interfaces, and execution under severe resource constraints [9]. Success depends not on "write once, run anywhere", but on careful management of device-specific details, prompt structure, and orchestration logic. In these settings, LLMs remain powerful, but realizing their potential requires engineering rather than one-shot prompting [42].

Embedded ML sits at the intersection of ML frameworks, embedded software development, and hardware/software constraints imposed by resource-limited devices [1, 21, 39]. Beyond the classical ML workflow, embedded targets introduce additional stages such as optimization, quantization, and device-specific code generation to meet memory, latency, and portability requirements [5].

This paper examines whether orchestrated LLMs can act as an *Autopilot* for the embedded ML lifecycle. In this context, we use the term *Autopilot* to denote fully autonomous pipeline execution after initial task specification, including iterative error recovery and regeneration without human intervention, distinguishing it from copilot-style assistance. We focus on **feasibility** (*can the pipeline be automated end-to-end?*) and **dependability** (*how often does it work, and under which conditions?*). We implement a middleware framework that orchestrates the full pipeline, interacts with multiple LLMs, and records all intermediate artifacts and execution traces. Using this framework, we execute thousands of complete runs across a spectrum of deployment targets, from highly constrained microcontroller-class devices to less constrained edge boards, to study how device constraints interact with LLM behavior. Two observations emerge. First, the primary challenge is not generating code, but managing silent failures between artifacts that appear correct and those that function correctly at runtime [47]. Small variations in prompt structure can lead to drastically different outcomes, ranging from rapid convergence to long sequences of failed iterations [10, 54]. Second, constraints matter. Success rates are consistently lower on highly constrained devices and improve as portability requirements and runtime flexibility increase.

Instead of proving automation outright, we uncovered trade-offs between potential and limits. This paper reports that journey as an empirical, system-level study of feasibility and dependability, and distills the lessons required to design with feasibility awareness in mind rather than hoping that failures won't occur. Specifically:

- We design and implement an automation framework that orchestrates key stages of the embedded ML lifecycle using LLMs, including dataset handling, model conversion, and on-device code generation. The framework is the result of multiple refinement cycles and iterative prompt and workflow engineering.
- We use this framework to evaluate whether LLMs can autonomously generate all required artifacts across a wide

*Both authors contributed equally to this paper.

Device Class	CPU	Memory (RAM)	Storage	AI Acceleration	Power Budget	Code Portability / Cost
Microprocessor (MPU)	GHz-class	512 MB–several GB	GB–TB (filesystem)	Optional (GPU/TPU/NPU)	Watts	High portability, higher cost
Microcontroller (MCU)	MHz-class	2 KB–512 KB	32 KB–2 MB (no FS)	None	µW–mW (battery)	Low portability, low cost

Table 1: Comparison of microprocessors and microcontrollers and their impact on ML/LLM deployment pipelines.

spectrum of devices, from microcontroller-class targets to edge boards, quantifying success rate, deployment latency, retry behavior, and convergence stability across heterogeneous execution environments.
- We release the full automation framework, logged execution traces, and all artifacts produced during our evaluation, including successful and failed runs. These artifacts allow researchers to replicate our findings and extend them to future LLMs and device platforms.

Together, these contributions aim to provide the first evidence-based characterization of what it takes to treat LLMs as automation engines for embedded ML, and where they fall short.

The remainder of this paper is organized as follows: Section 2 introduces the problem space, while Section 3 describes the design of our automation framework. Section 4 presents our experimental methodology, and a comprehensive empirical evaluation across different LLMs and device constraints. Section 5 reviews related work, and Section 6 concludes the paper.

Note: To support reproducibility and future extensions, we release our framework, including execution traces, prompt templates, and artifacts generated during our evaluation. The implementation consists of approximately 2,610 lines of Python code and 966 comment lines documenting orchestration logic, failure-recovery policies, and device-specific constraints. The modular prompt templates constitute a substantial part of the system, with 677 comment lines and 32 executable lines. These templates encode the structural and semantic rules that guide LLMs in producing valid artifacts. All sources are available at: **https://github.com/beaver-edge/middleware26**.

2 Background and Problem Space

This section outlines why embedded ML automation differs from traditional pipelines and why LLMs are promising yet fragile in this context. Embedded targets introduce additional stages (e.g., optimization, conversion, device-specific code generation) and strict constraints (memory, timing, portability), altering both what must be automated and how automation succeeds. While LLMs offer flexibility across tools and languages, dependable automation requires explicit structure.

Resource Constraints and Device Heterogeneity. We use embedded or edge AI to denote ML inference workloads executed on-device rather than in the cloud, spanning a spectrum from microcontroller-class devices to microprocessor-class edge boards, optionally equipped with AI accelerators. Table 1 contrasts microcontrollers (MCUs) and microprocessors (MPUs) characteristics relevant to automation.

MCUs operate at MHz-scale frequencies with kilobytes to a few hundred kilobytes of RAM, limited storage, and tight energy budgets [9]. Tooling is minimal, memory is statically allocated, debugging feedback is scarce, and generated code must compile, link, fit within memory constraints, and execute deterministically on bare-metal or RTOS environments [1, 49]. In contrast, MPUs provide OS support, package managers, and richer diagnostics, enabling more flexible iteration, albeit still within embedded constraints [2].

Throughout the paper, we use this distinction to explain why automation behaves differently across device tiers. Pipelines that succeed on MPUs using dynamic languages and rich runtimes do not trivially transfer to MCUs, where correctness depends on strict interfaces, static resource allocation, and limited observability.

Traditional ML vs. Embedded ML Lifecycle. Traditional ML workflows follow a relatively stable pattern: data processing, model design and training, evaluation, and deployment [5]. Deployment typically consists of exporting a trained model in a standard format and invoking it through high-level APIs, with tooling abstracting away most hardware differences and failures remaining visible and debuggable.

Embedded ML fundamentally alters this workflow. When targeting constrained devices such as microcontrollers or single-board computers, the lifecycle introduces additional stages, including model optimization, quantization, format conversion, and device-specific code generation (Fig. 1). These stages are tightly coupled, as design choices at one step propagate downstream. For example, ❶ quantization must respect numeric support and memory constraints; ❷ format conversion may modify tensor layouts and metadata expected by the runtime; and ❸ device-level code must statically allocate buffers, interface with peripherals, and meet strict timing and energy budgets [25].

As a result, embedded ML pipelines evolve into iterative co-design loops, where validation must span optimization, conversion, and deployment stages. Embedded ML is therefore not simply "smaller" ML, but a pipeline with cascading dependencies, in which early lifecycle decisions directly constrain what can be deployed downstream [1, 49].

Software Fragmentation and Toolchain Dependency. In cloud-based ML workflows, models are typically packaged behind uniform abstractions (e.g., containers, Python environments, ONNX runtimes). In contrast, embedded ML requires direct interaction with device-specific toolchains. Each hardware class introduces its own compilers, runtimes, and build artifacts, including cross-compilers for microprocessors, vendor SDKs or firmware toolchains for microcontrollers, and AI accelerator-specific compilers [1].

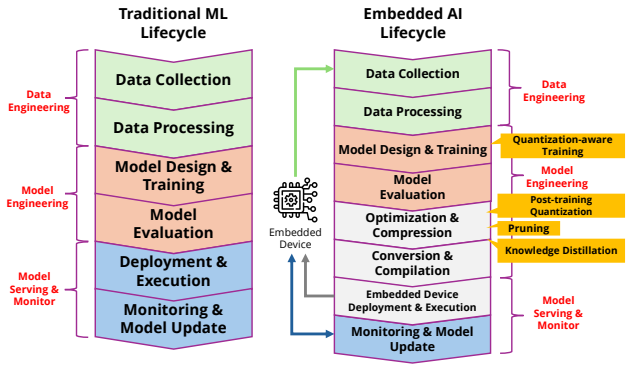


Figure 1: Comparison of traditional and embedded ML lifecycles. Embedded ML introduces additional stages such as optimization, compression, and device-specific deployment to meet constrained hardware requirements.

This fragmentation directly shapes the embedded ML lifecycle. Decisions made during optimization and conversion (e.g., quantization or pruning) must align with toolchain capabilities and runtime constraints. Developers must reason about compression limits, memory availability for intermediate activations, support for hardware acceleration, and how to integrate models, firmware, and buffers within flash and RAM budgets [28]. As a result, the workflow evolves from a simple *train–convert–deploy* sequence into a tightly coupled set of constraints imposed by the target software ecosystem.

Managing these dependencies requires expertise spanning ML, embedded software, and toolchains, which is often unavailable within a single developer profile. Consequently, a significant portion of development effort is spent debugging build systems, compiler configurations, and runtime incompatibilities rather than improving model quality [32].

Why LLMs Are Appealing (Yet Insufficient Alone). The combination of resource constraints, device heterogeneity, and fragmented toolchains turns the embedded ML lifecycle into a tightly coupled sequence of decisions, where early choices propagate downstream. Even when individual stages are well understood, assembling them into a reliable pipeline requires repeated translation across frameworks, programming languages, compiler targets, and device-specific software environments.

LLMs are appealing in this context because they can operate across abstraction levels and languages, from Python-based data processing and orchestration to C/C++ device-level inference code. Prior work has shown that LLMs can assist with code generation, debugging, and reasoning about APIs [27, 31]. Their ability to synthesize executable artifacts from natural language descriptions makes them promising candidates for reducing the complexity of heterogeneous workflows.

However, embedded ML exposes limitations that are less pronounced in cloud or server-side settings. Correctness is not determined solely by whether generated code compiles, but by strict adherence to device-specific constraints, including memory budgets, peripheral initialization, and runtime expectations. Pipelines that succeed on less constrained edge boards may fail on microcontrollers under identical prompts and logic. In IoT deployments spanning diverse devices, this heterogeneity further undermines portability and reproducibility [45].

In short, LLMs can generate individual artifacts but cannot ensure that these artifacts compose correctly across stages and devices [30]. While LLMs provide flexibility, dependable automation requires explicit structure, orchestration, and validation rather than *one-shot* prompting [48].

3 Autopilot Framework Design

3.1 Key Requirements for LLM-Driven Embedded ML Automation

From our system exploration and LLM-assisted software development related literature analysis [27, 30, 31, 37, 45, 47, 48], three key requirements emerge among others.

Stage-aware prompting and structured context injection. LLMs are sensitive to prompt structure, and the same high-level request can yield different artifacts, formats, or APIs. In the embedded ML lifecycle, each stage has different semantics (model quantization versus code generation versus compiler toolchain setup), and therefore requires different prompts. To produce deployable artifacts, prompts must inject structured context: *(i)* device constraints (flash/RAM budgets, accelerators), *(ii)* requirements on model format or quantization strategy, *(iii)* expected output schema (e.g., *"return only compilable C/C++ code"*). Without stage-aware prompting, the LLM must improvise around missing assumptions, which can potentially lead to a common source of hallucinated APIs and incorrect library calls.

Validation-driven refinement (compile / execute / feedback / re-prompt). Many artifacts generated by LLMs look correct but fail only when executed. Thus, correctness must be verified concretely. A dependable automation pipeline must: ❶ generate an artifact, ❷ compile or execute it (on real hardware or emulation), ❸ capture logs / process errors, ❹ feed structured feedback into a follow-up prompt. This creates a closed-loop refinement cycle, resonating with recent self-refinement and reflective prompting strategies [33]. Execution becomes the ground truth, not the LLM.

Auxiliary tooling with minimal human intervention. LLMs cannot orchestrate toolchains, manage intermediate files, or decide what to retry. Therefore, automation requires supporting infrastructure such as prompt management and traceability, artifact bookkeeping, and compilation/execution wrappers around toolchains and device interfaces. Our idea is also that human involvement should exist, but only when escalation is needed and not to shepherd every failed compile.

3.2 System Architecture

Fig. 2 shows the Embedded ML Autopilot architecture, organized around two complementary core elements *(i)* the *Autopilot Core Engine*, which orchestrates LLM-driven automation, and *(ii)* the *Automated Lifecycle Stages*, where deployable *artifacts* are produced.

Autopilot Core Engine. The left half of Fig. 2 illustrates the components that drives the automation orchestration. The engine transforms user inputs into deployment-ready artifacts via LLM orchestration.

This is an API-agnostic component that is cable to interact with multiple LLM providers through compatible HTTP APIs, allowing us to evaluate different model families under a common interface

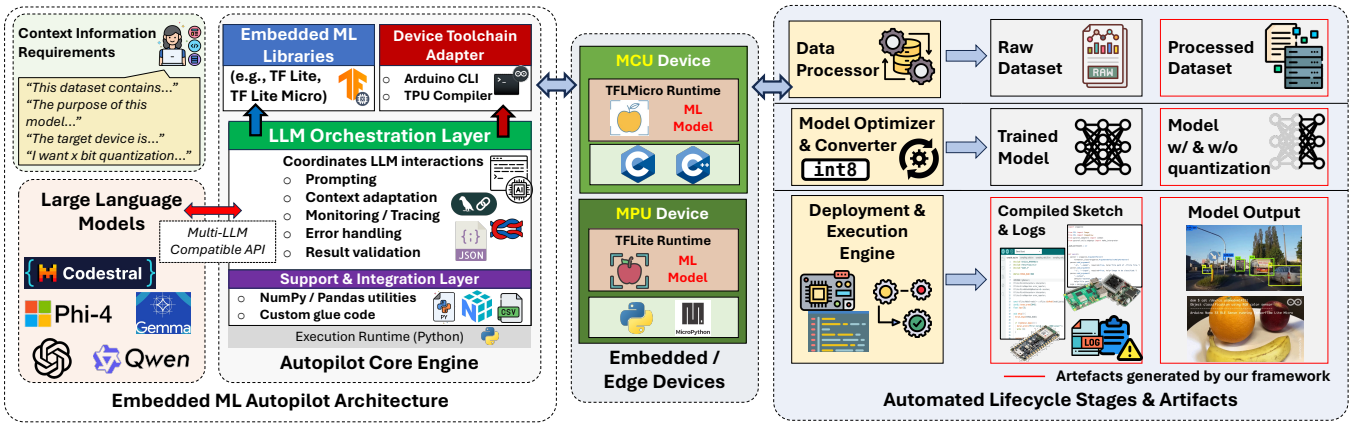


Figure 2: Left: System architecture of the Embedded ML Autopilot, showcasing its internal modules, LLM integration, and runtime environment for managing ML model preparation and deployment. Right: Operational workflow enabled by the Autopilot, outlining the ML lifecycle stages from dataset ingestion to device-ready sketch generation.

[29]. Prompts are stage-aware, as the system injects structured context (e.g., dataset characteristics, memory and power budgets, supported numeric formats, expected I/O behavior), and constrains the output schema to the artifact required at that stage—preprocessing code, conversion script, or deployment sketch. The orchestration logic also implements the validation-driven refinement loop. After an artifact is generated, it is compiled or executed, logs are collected, and failures are summarized into compact feedback that seeds a follow-up prompt. This layer also provides monitoring and tracing of every LLM interaction (prompt, response, tool output, and decision) using LangChain for orchestration and Langfuse for tracing and analytics [8, 34]. These traces serve both operational purposes (error recovery, reproducibility) and later analysis. The LLM Orchestration Layer is extended by a thin Integration Layer, which provides a minimal execution substrate to support the Autopilot Core Engine components. The Integration Layer does not embed heuristics or decision-making logic. Its responsibilities are intentionally narrow: *(i)* verify that framework requirements are satisfied (environment, dependencies, device access), *(ii)* glue together the autopilot architectural components, and *(iii)* sequentially execute the artifacts generated by the LLM.

Automated Lifecycle Stages & Artifacts. The right half of Fig. 2 shows the automated stages of the lifecycle. We present Data Processor, Model Optimizer & Converter, and Deployment & Execution Engine as abstractions of the Embedded ML Autopilot's functionality: each stage is an instance of the same control logic applied to a different part of the pipeline. Concretely, every stage *(i)* receives a stage-specific prompt and structured context from the LLM Orchestration Layer, *(ii)* produces a concrete artifact (code or configuration), and *(iii)* is validated on-device via the Device Runtime & Toolchain Layer. Failures trigger the same validation-driven refinement loop (compile / execute / capture evidence / re-prompt), while successes advance the pipeline with persisted, versioned artifacts. In this sense, stages differ in *what* they generate and *which* constraints they enforce, but share the *how*: stage-aware prompting, and on-device verification.

The first stage, the *Data Processor*, takes the raw dataset and prepares it for model training. The LLM receives contextual information automatically extracted from the input data—such as the number of samples, feature dimensionality, class labels, or sampling rate—and generates executable Python code to preprocess it. This includes common operations such as cleaning, normalization, and filtering, allowing the dataset to be transformed into a format suitable for model development with minimal manual intervention. Once generated, the code is executed locally, producing the processed dataset and execution logs. If execution fails, the Autopilot captures the failure context and returns it to the LLM through a refined prompt, enabling automatic regeneration without human intervention.

In the *Model Optimizer & Converter* stage, the Autopilot generates Python logic to optimize and quantize the model using embedded ML libraries (e.g., TensorFlow Lite / TensorFlow Lite Micro). In our experiments, this stage applies post-training INT8 quantization, while training-dependent techniques such as quantization-aware training, pruning, or knowledge distillation are not exercised, as our focus is on deployment-phase automation. The Orchestration Layer injects device constraints into the prompt, such as RAM and flash budgets, supported numeric precision (`int8` vs. `float32`), and runtime compatibility. The generated code is executed and the resulting artifact is verified; if not viable, the LLM is invoked again with richer context (e.g., error logs and validation outcomes). The final stage, the *Deployment & Execution Engine*, takes the converted model and generates device-ready software artifacts for inference on the target hardware. For MCUs, the LLM produces complete firmware sketches, while for MPU-class devices it generates runtime inference scripts and, when available, enables hardware acceleration. The Autopilot then invokes the Device Toolchain Adapter to deploy the generated code on device. Execution logs and runtime outputs are collected, and compilation or execution failures trigger the same refinement loop.

As observed, the framework operates on the automation of the *volatile* stages of the Embedded ML lifecycle. These stages are highly

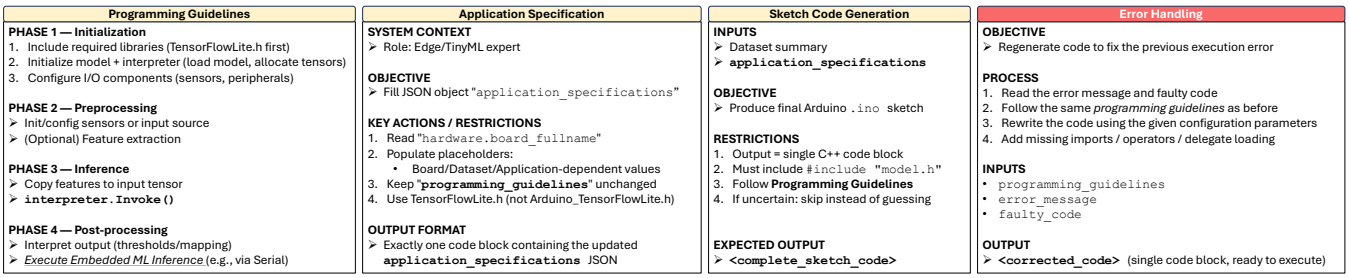


Figure 3: Prompting workflow for Embedded ML code generation: *(i)* the *programming guidelines* define the runtime pattern; *(ii)* the *application specification* captures board/dataset constraints; *(iii) sketch code generation* produces the `.ino` file; *(iv) error handling* regenerates code using the error and faulty snippet.

iterative and error-prone, making them ideal candidates for automation. Model training is intentionally excluded from the framework's scope: training typically requires human supervision (e.g., dataset curation, model design decisions, hyperparameter tuning) and does not change as frequently as the downstream deployment stages. Moreover, many practical workflows reuse pretrained models available from public "model lakes" or repositories (e.g., TensorFlow Model Garden, ONNX collections), making training unnecessary in many deployment scenarios. In these cases, the Autopilot accepts the model as input and proceeds directly with optimization and deployment. This separation of concerns aligns with the standard Embedded ML workflow, where training is performed offline and only the optimized artifact is deployed to the device. In short, we automate what changes frequently, and we do not automate what remains stable.

3.3 Prompting and Artifact Generation Workflow

Prompt Hierarchy and Modular Structure. A key element of the autopilot concerns how prompts are constructed, adapted, and used to generate executable artifacts across the lifecycle. Prompts are organized into a **hierarchical set of modules**, each responsible for a specific dimension of the generation process (Fig. 3). Specifically:

- *Programming Guidelines* define the canonical execution flow for embedded inference. These guidelines are fixed, non-negotiable, and encode the expected structure of generated code: initialization, preprocessing, inference invocation, and post-processing. Their aim is to prevent the model from hallucinating alternative execution paths and anchor generation to a deterministic pattern.
- The *Application Specification* module is populated dynamically from user inputs and dataset or model metadata. It includes fields such as the device name, quantization requirements, sensor configuration, and any restrictions specific to the target platform. It also serves as the binding layer between abstract user goals and concrete hardware constraints.
- The *Sketch Code Generation* module uses the previous two components as context to generate the device-ready code. For MCU-based targets, it produces an Arduino-compatible C++ sketch with the correct library imports and memory allocation strategy. For MPU-based targets, it generates Python inference scripts and conditionally inserts code for AI accelerators (e.g., Edge TPU interpreter bindings) when requested in the Application Specification.
- The *Error Handling* module activates when execution or compilation fails. It receives three inputs: the faulty code block, the error message, and the original Programming Guidelines. Rather than restarting from scratch, it instructs the LLM to repair the artifact while preserving the correct structure. This module is crucial for enabling the system's iterative debugging capabilities and for preventing the LLM from drifting into inconsistent or unrelated code.

To sum up, guidelines constrain generation, specifications contextualize it, the sketch generator produces the artifact, and the error handler refines it. The idea behind this modular approach is to make sure that prompts can adapt to different devices and toolchains, but always within a predictable envelope.

From Prompts to Executable Artifacts. Fig. 4 illustrates the deployment-stage workflow as an example, but the same principles hold for data processing and conversion.

At a high level, the Autopilot receives a task specification, for example, *"deploy this model to an Arduino-class MCU with int8 quantization"*, and a device profile describing ML runtimes, memory layout, peripheral availability, and optional AI accelerators for MPU class of devices. These two inputs are passed through the Prompt Adapter, which selects the appropriate prompt module (data processing, model conversion, or sketch generation) and injects device-specific context. The resulting prompt is sent to the LLM, which returns an initial artifact such as preprocessing code or a C++ inference sketch.

This generated artifact is then handed back to the Integration Layer for local execution. For deployment, this means attempting to compile the sketch through the Device Toolchain Adapter (e.g., Arduino CLI, TPU compiler). If compilation succeeds, the artifact is deployed to the device. If it fails, the system enters an iterative refinement loop, with the Autopilot that extracts the minimal error delta (not full logs), appends it to the prompt, and requests a corrected version. This loop continues until the artifact compiles or the maximum retry budget is reached. The figure shows this process as a message-passing flow between the user, the Autopilot, the LLM, the toolchain, and the device.

For clarity, Fig. 3 shows a simplified abstraction of the prompting logic. In practice, the prompts used by the Autopilot are the result

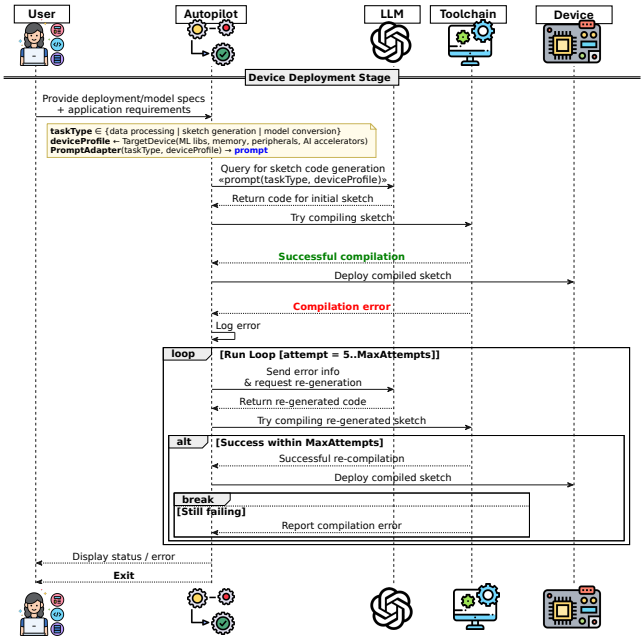


Figure 4: Device Inference Code Generation Sequence. Based on task type and target device, prompts are adapted and sent to the LLM. Generated code is compiled and executed; on failure, execution traces are fed back to the LLM to regenerate improved code until success or max retries.

of extensive iteration, trial-and-error refinement, and progressive constraint tightening. Additionally, while we often use the term *sketch*, a convention inherited from Arduino workflows, the term is used generically throughout this paper to denote the final code or script deployed to the device for executing ML inference.

4 Experimental Methodology

4.1 Experimental Setup.

The evaluation of the Embedded ML Autopilot aims to assess its capabilities across heterogeneous devices, multiple lifecycle stages, and different classes of LLMs. All LLM reasoning and artifact generation occur off-device, and deployed binaries run natively on the target hardware without additional overhead.

To evaluate the framework end-to-end, we selected a set of representative embedded ML applications covering different sensing modalities, hardware classes, and levels of software complexity. For MCU-based pipelines, we used the widely adopted computer vision task of fruit classification for the *Arduino Nano 33 BLE*. For MPU and AI accelerator-equipped devices, we selected a popular open-source object detection project built with TensorFlow Lite, runnable on different Single-Board Computer. These workloads were intentionally selected within the same application domain to isolate lifecycle volatility and orchestration behavior across heterogeneous devices, LLMs, and execution setups, rather than introduce variability from task diversity.

Across these projects, we evaluated five target tasks: *(i)* Data Processing (DP), *(ii)* Model Conversion (MC with INT8 quantization), *(iii)* Arduino Sketch Generation (ArdSG), *(iv)* Python Sketch Generation (Py-CPU), and *(v)* Python Sketch Generation targeting TPU-accelerated execution (Py-TPU).

For each task, we performed 30 independent runs per LLM. The study includes a selection of closed-source models alongside medium-sized, code-oriented open-source models (Table 2). This mix allows us to compare both frontier and open models under realistic resource budgets. Besides the default settings for each LLM, we introduce an additional experimental condition (denoted with the suffix "-p") using `temperature = 0.1, top_p = 0.3`. This configuration restricts generative randomness and focuses the model on high-probability continuations. It allows us to probe whether reduced stochasticity improves determinism and correctness in code-heavy tasks.

Abbreviation	Model Name
G4o	GPT-4o
G4o-m	GPT-4o-mini
Q32B	Qwen2.5-Coder-32B
Q14B	Qwen2.5-Coder-14B
P4-14B	Phi-4-14B
C22B	Codestral-22B
G27B	Gemma-3-27B
DS14B	Deepseek-r1:14B

Table 2: Models evaluated and their abbreviations.

We measure performance using two primary metrics: *(i) Token consumption*, which serves as a proxy for LLM inference cost and resource usage, and *(ii) Success Rate*, defined as the proportion of runs that produce artifacts satisfying stage-specific validation criteria. Importantly, success goes beyond compilation or execution alone. Depending on the stage, validation includes: *(i)* successful compilation or execution, *(ii)* runtime checks (e.g., tensor compatibility, inference invocation), *(ii)* artifact inspection, and *(iv)* structural similarity analysis (Section 4.4) to detect deviations from expected implementation patterns. These checks do not guarantee full semantic equivalence, but they provide a practical notion of deployment-level correctness aligned with real-world embedded ML workflows. The maximum number of regeneration attempts per run is fixed to five, balancing recovery capability with computational and cost overhead.

All back-end experiments were executed on a shared HPC infrastructure with an NVIDIA A100 GPU and 35 GB of dedicated VRAM. We intentionally exclude wall-clock time as a comparative metric across LLMs, since load fluctuations on the shared node may introduce uncontrolled variance. However, runtime remains meaningful within each LLM's own test batch, where tasks are executed sequentially under consistent conditions, allowing stage-to-stage comparisons.

4.2 Evaluation on MCU Device

End-to-End Results on Microcontroller Pipelines. To evaluate the Autopilot on MCU-class of devices, we first consider the fruit classification use case targeting the Arduino Nano 33 BLE. The pipeline builds and deploys a compact convolutional neural network (CNN) that performs on-device inference using an embedded RGB color sensor. This application exercises all volatile stages of the

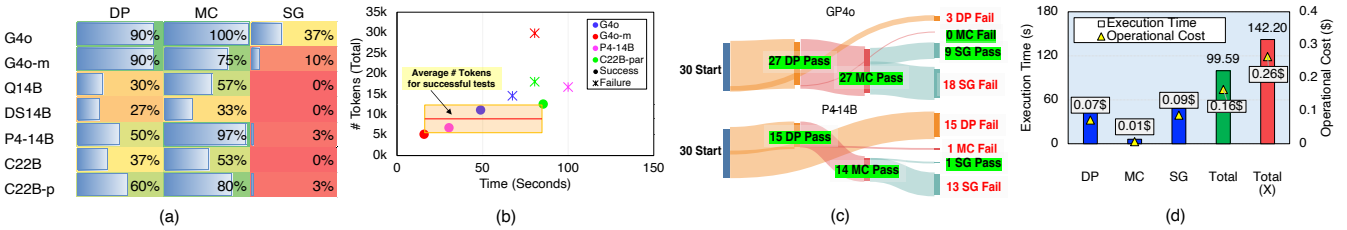


Figure 5: MCU-level performance. Panels show stage-wise success rates across LLMs (a), execution time vs. token consumption for successful pipelines (b), Sankey diagram illustrating where GPT-4o runs fail (c), and GPT-4o's execution time and monetary cost, including overhead from failed attempts.

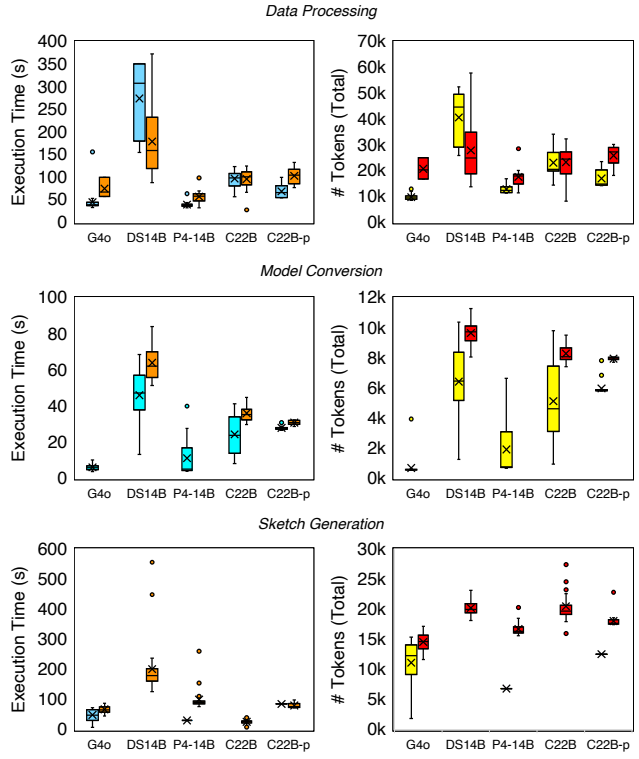


Figure 6: Execution time and token consumption across the three stages of the pipeline, shown for successful (cyan/yellow) and failed (orange/red) runs.

embedded ML lifecycle: preprocessing the dataset, converting and quantizing the model for TensorFlow Lite Micro, and generating a device-ready sketch that configures the sensor, loads the model, and performs inference on streaming data. Fig. 5 summarizes the behavior of the Autopilot across LLMs and stages.

Success Rates Across Stages (Fig. 5a). Three observations stand out clearly. First, sketch generation (SG) is the most challenging stage by a wide margin. Only the most capable model (i.e., G4o) manages to reach a non-trivial success rate, completing SG in roughly 37% of runs. All other models struggle and even G4o-m exhibits a sharp drop, confirming that SG requires capabilities beyond standard code synthesis. The fragility of this stage stems from the fact that the LLM must satisfy multiple device-level constraints simultaneously in a C++ environment. For example, correct header includes, static buffer sizing, I/O and peripheral configuration, memory-safe tensorallocation, and strict adherence to TensorFlow Lite Micro runtime conventions. Unlike Python-based stages, C++ sketch generation leaves little room for ambiguity or dynamic correction, and small inconsistencies propagate into compilation errors. Small deviations easily produce sketches that compile incorrectly. Second, MC is consistently more stable across models. Most LLMs achieve above 50% success in MC, with the exception of DS14B, which fails more frequently. This suggests that MC, while complex, is structurally easier for LLMs because it relies more on generating Python scripts that call well-defined APIs rather than producing device-specific C++ code. Third, among open-source models, P4-14B stands out. It achieves the strongest overall balance between DP, MC, and SG among non-frontier models, showing that mid-sized, code-specialized open models can perform competitively when prompts and orchestration are well structured. Finally, adjusting sampling parameters can yield modest improvements. The constrained-sampling version of Codestral-22B (C22B-p) shows slightly better stability in MC and SG compared to its default configuration. While the gains are modest, they illustrate that reducing generative stochasticity can help LLMs adhere more closely to the required code patterns.

Execution Time vs. Token Cost (Fig. 5b). The scatter plot shows execution time and total token consumption for instances in which an LLM succeeded at least once in completing the entire MCU pipeline. A consistent pattern is that successful runs cluster within a narrow band of token usage, while more unstable configurations exhibit a long-tail dispersion, suggesting that excessive verbosity or drift is correlated with failed attempts. Models that succeed tend to produce shorter, more focused interactions rather than long, diffuse generations. While execution time is influenced by toolchain latency, the spread reflects the cumulative retries and re-prompting cycles that occur during failure cascades. This behavior indicates that excessive token consumption can serve as a signal of instability, motivating cost-aware strategies such as retry budgeting, LLM re-routing or early human intervention.

End-to-End Pipeline Flow (Fig. 5c). The Sankey diagrams illustrate how the 30 independent runs flow through the entire pipeline. The diagrams make the "leak points" explicit. For G4o, most runs pass DP and MC, but the SG becomes the main point of attrition. For smaller models (e.g., P4-14B), divergence occurs much earlier, with

SKETCH GENERATION PROMPT TEMPLATE 1

```
### OBJECTIVE ###
"task":{
 ...
    "application_specification":{
        ...
        "guideline":{
            ...
        }
    }
}
```

SKETCH GENERATION PROMPT TEMPLATE 2

```
### OBJECTIVE ###
- **TASK**:...
...

### APPLICATION SPECIFICATIONS
###
"application_specification":{
        ...
        "guideline":{
            ...
        }
    }
```

SKETCH GENERATION PROMPT TEMPLATE 3

```
### OBJECTIVE ###
- **TASK**:...
...

### APPLICATION SPECIFICATIONS
###
"application_specification":{
        ...
}
### GUIDELINE FOR SKETCH ###
# Steps...
...
```

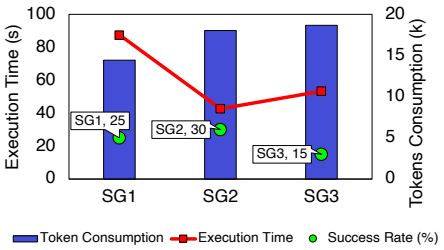


Figure 7: Comparison of three prompt templates (SG1, SG2, SG3) for sketch generation, highlighting trade-offs between token usage, latency, and success rate. SG2 achieves the highest success rate, while SG1 minimizes token consumption.

DP and MC failing more often and leaving only a small fraction of runs reaching SG. This visualization underscores the central message that even models that are strong at code generation may struggle to maintain consistency across a multi-stage embedded pipeline with cumulative constraints.

Operational Cost of Failures (GPT-4o Case Study, Fig. 5d). While open-source models incur no per-token cost, GPT-4o provides a useful reference for understanding the monetary implications of full-pipeline automation. Successful GPT-4o runs complete the entire MCU pipeline in roughly 100 seconds, demonstrating that automation can be practical in latency-constrained workflows. However, the system also pays for every failed attempt—both in wall-clock time and API cost. Toolchain failures trigger iterative refinement loops that accumulate additional tokens and delays. At hyperscale, this tail of unsuccessful generations becomes a non-trivial operational expense.

Fig. 6 shows the distribution of execution time and token consumption across lifecycle stages, separating successful and unsuccessful runs. Across all models, successful executions cluster within narrower token and latency ranges, while failures exhibit broader dispersion and more extreme outliers.

The DP stage operates in a stable regime: successful and failed runs display similar distributions, with only modest efficiency differences. This suggests that DP failures are typically caused by minor formatting or API issues rather than fundamental reasoning instability. In contrast, the MC stage exhibits a clear separation between outcomes. Successful MC runs maintain low token usage, whereas failed attempts show a systematic upward shift in both median and mean token cost, reflecting longer, off-pattern outputs that violate the rigid conversion structure required by the toolchain. Parameter-sensitive configurations (e.g., C22B-p) reduce variance and correlate with slightly higher success rates, indicating that constrained decoding can improve stability in conversion tasks.

The SG stage remains the most challenging and model-sensitive step. Success rates are extremely low for most open-source models, and the few successful runs exhibit substantially lower token and latency values than failures. Failed SG attempts tend to cluster around incomplete or structurally invalid sketches, suggesting partial recognition of the task without satisfying its strict constraints. We observe diverse failure modes in this stage, including missing dependencies, tensor shape mismatches, incorrect peripheral initialization, and API misuse. A dedicated taxonomy and cross-model analysis of these failures is provided in [35].

Across stages, DS-14B—the only model with explicit reasoning mechanisms—exhibits pronounced long-tail behavior in failed runs, producing extended reasoning traces that inflate token usage and execution time. These artifacts disrupt the rigid control flow required for MCU deployment, making the model both less predictable and more error-prone.

The results highlight a strong structural asymmetry in the MCU pipeline. DP tolerates moderate variation, MC is stable when outputs remain syntactically bounded, and SG dominates variability and failure. For MCU-class targets, LLM-driven automation succeeds primarily when generations remain short, focused, and tightly constrained.

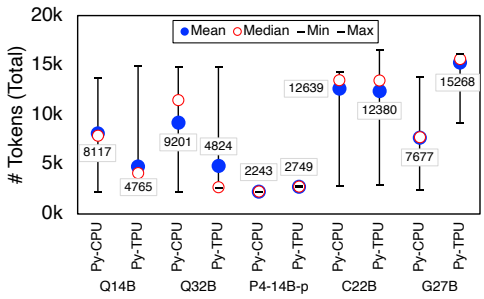
Prompt Structure Sensitivity. Sketch generation emerged as the most fragile stage of the MCU pipeline, but a closer examination showed that failures were not driven solely by code complexity or device constraints. Instead, the structure of the prompt itself played a decisive role [24]. To study this effect, we compared three variants of the sketch-generation prompt (SG1, SG2, SG3), each containing the same semantic content but arranged with different levels of nesting and formatting. SG1 followed a deeply structured layout in which task information, application specifications, and guidelines were all embedded in nested JSON-like blocks. SG2 flattened the structure by extracting the task into Markdown while keeping the specifications in structured form. SG3 flattened the prompt even further, placing both the task and the guideline outside the JSON block and presenting them as separate Markdown sections.

The results, shown in Fig. 7, reveal a surprisingly strong dependency on structural layout rather than content alone. SG2 achieved the highest success rate (30%), using a moderate number of tokens and producing the lowest execution time among the three variants. SG1 performed slightly worse (25%) and was notably slower despite its efficient token usage. SG3 delivered the poorest performance (15%) while consuming the largest number of tokens and exhibiting relatively high latency. These differences cannot be explained by the content of the prompts as each variant conveys the same instructions. Instead, they suggest that LLMs respond unevenly to format-induced constraints, consistent with observations in recent studies on structural sensitivity in prompting [16].

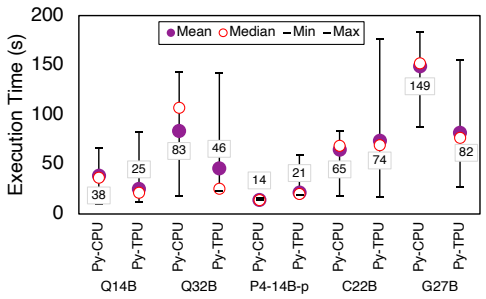
We interpret this behavior as a form of format-induced misalignment. Deeply nested schemas, overloaded blocks, or unconventional layouts appear to trigger internal parsing heuristics that degrade the model's ability to plan or to produce consistent C/C++ code for deployment. The results also show that resource usage (i.e.,tokens

Model	Success Rate (%) Py-CPU	Py-TPU
Q14B	76.67	100.00
Q14B-p	6.67	100.00
Q32B	60.00	100.00
Q32B-p	43.33	60.00
P4-14B	60.00	100.00
P4-14B-p	100.00	100.00
C22B	16.67	33.33
C22B-p	20.00	26.67
G27B	80.00	6.67
G27B-p	56.67	0.00

(a)



(b)



(c)

Figure 8: Sketch generation on MPU-class devices. Success rate (a), token consumption (b), and execution time (c) for Py-CPU and Py-TPU deployments across open-source LLMs. The center and right panels report only the most successful configurations to highlight convergence behavior.

or execution time) is not a reliable proxy for correctness. More detailed prompts, or more verbose interactions, do not necessarily yield better outcomes and may even exacerbate the problem.

4.3 Evaluation on MPU devices

4.3.1 Sketch Generation on MPU-type of Devices. We next evaluate the Autopilot on MPU-class edge devices, focusing on Python-based and accelerator-enabled SG The experiments target an object detection use case executed on two representative platforms: a Raspberry Pi 4 Model B and a Google Coral Dev Board, both equipped with 4 GB of RAM. In this setting, the generated sketch is required to read an MP4 video as input and produce an output video where detected objects are marked with bounding boxes and labels. The ML workload relies on a quantized SSDLite MobileNet V2 model pre-trained by Google, along with its Edge TPU–compiled variant for accelerator execution on the Coral board.

Unlike the MCU evaluation, here we focus exclusively on sketch generation (Py-CPU and Py-TPU). Earlier results showed that DP and MC are comparatively stable, even under stricter MCU constraints, while SG remains the dominant source of instability. To ensure a fair comparison, we restrict this analysis to open-source LLMs and exclude DS14B due to consistently poor performance in prior stages. We also omit GPT-based models, as the goal of this section is to assess the feasibility and reliability of Autopilot-style automation using locally deployable models.

Fig. 8 reports the results of sketch generation on MPU-class devices, comparing Py-CPU on Raspberry Pi and TPU-enabled SG (Py-TPU) on the Coral Dev Board. Fig. 8a shows success rates across models and configurations, while the center and right panels report token consumption and execution time for successful runs only. Overall, SG on MPUs is significantly more reliable than on MCUs, even though the workload (i.e., video-based object detection) is more complex. This improvement is largely attributable to the use of Python-based runtimes, which relax constraints on memory management, static allocation, and compilation compared to C++ MCU sketches. As a result, the Autopilot can converge more often without triggering deep refinement loops.

A striking result is that TPU-based sketch generation consistently outperforms CPU-based execution for most models. With the exception of G27B, Py-TPU achieves near-perfect success rates across LLMs, while Py-SG on Raspberry Pi exhibits noticeably lower and more variable performance.

This behavior is counterintuitive at first glance. TPU deployments are more hardware-specific and require accelerator-aware code. However, our analysis suggests that workflow ambiguity, rather than hardware complexity, is the dominant factor. The Coral TPU ecosystem enforces a highly standardized and dedicated pipeline. From INT8 TFLite models, through the Edge TPU compiler, to a small set of canonical Python runtime APIs. Public examples and documentation converge on a narrow set of templates (e.g., `pycoral.adapters`, `tflite_runtime.Interpreter`), leaving little room for variation. In contrast, Raspberry Pi–based vision pipelines are inherently flexible. The LLM must choose among multiple valid software stacks (TensorFlow Lite vs. full TensorFlow, OpenCV, different camera interfaces, varying dependency versions, etc.). This broader decision space increases the likelihood of hallucinations [26], such as importing unavailable modules, mixing APIs, or assuming unsupported libraries on ARM platforms. We conjecture that LLMs implicitly rely on pattern completion over their training distribution [6, 7]. The Coral task corresponds to a low-entropy, repetitive pattern space, while Raspberry Pi sketch generation maps to a high-entropy space with many locally valid but globally incompatible solutions. The reduced ambiguity of the TPU workflow leads to higher execution success. In short, specialization reduces ambiguity, and ambiguity is adversary of LLM-driven automation.

Applying constrained sampling parameters (`temperature`=0.1, `top_p`=0.3) yields mixed results. For most models, parameter tuning provides limited or no benefit and can even degrade performance. An exception is P4-14B, which achieves near-perfect success across both Py-CPU and Py-TPU configurations. This suggests that parameter tuning can help models that already exhibit strong structural alignment, but it is not a general remedy for MPU-type SG failures.

Figs. 8b and 8c report token consumption and execution time, respectively, for successful executions only. Two trends emerge. First, substantial variance across runs, with some executions converge in a small number of attempts, while others require multiple refinement cycles before success. Second, TPU-based deployments require fewer tokens and shorter execution times than CPU-based ones, reflecting faster convergence and fewer retries. G27B again appears as an outlier, exhibiting higher variability in both metrics.

	Q14B		Q32B		P4-14B		C22B		G27B	
	default	p	default	p	default	p	default	p	default	p
Py-CPU	2.48	3.00	2.17	4.54	2.94	1.00	3.00	3.67	2.38	4.65
Py-TPU	1.70	1.00	1.70	3.83	1.50	1.00	2.90	3.63	4.00	0.00
Py-CPU	17%	–	56%	57%	33%	100%	20%	–	25%	–
Py-TPU	50%	100%	–	–	63%	100%	10%	–	–	–

Table 3: Convergence behavior of sketch generation on MPU-class devices. Top: average number of attempts required to reach a successful deployment. Bottom: first-attempt success rate among successful runs, for CPU- and TPU-based workflows. "default" denotes standard sampling parameters, while "-p" denotes constrained sampling (temperature = 0.1, top_p = 0.3).

Table 3 characterizes how quickly sketch generation converges once success is achievable. The top half reports the average number of attempts required to obtain a successful deployment, while the bottom half shows the *first-attempt success rate*, i.e., the fraction of successful runs that converge without any refinement.

Two trends stand out. First, larger models do not converge faster. Despite their higher parameter counts, 32B-class models often require more refinement iterations than smaller ones. This suggests that additional capacity does not necessarily translate into better alignment with tightly constrained system tasks, and may even introduce more variability in early outputs. Second, TPU-based workflows consistently converge in fewer attempts than CPU-based ones. Across models, Py-TPU typically reaches success in one to two iterations, while Py-CPU requires more retries. This reinforces the earlier observation that standardized, accelerator-oriented pipelines reduce ambiguity and help LLMs stabilize faster. From a systems perspective, this also implies that specialized hardware pipelines may be cheaper to automate, even when using paid APIs, because fewer retries directly translate into lower token usage and shorter wall-clock time. The first-attempt success rates further emphasize model-dependent behavior. Several models never succeed on the first attempt under certain configurations. This indicates that refinement loops, as shown in Fig. 4, are not an optimization but a necessity. Once again, P4-14B stands out, as it consistently achieves high first-attempt success rates across both CPU and TPU deployments, confirming its strong structural alignment for code generation tasks in "less constrained" embedded workflows.

4.3.2 Alternative Success Metrics. From the results in the previous section, we observe that Python-based SG exhibits heterogeneous retry behavior. In particular, some LLMs converge immediately, while others produce a valid artifact only after multiple regeneration attempts. A traditional success rate, which merely captures whether a run eventually succeeds, reflects functional correctness but overlooks how efficiently that success is achieved [13]. To provide a more expressive characterization of model performance, we define three complementary success metrics. Each metric incorporates the number of generations required before a valid output is produced, denoted as g_i for the ith successful run. These metrics can result useful in practical deployments, where retries translate directly into latency, cost, and possibly energy consumption.
The standard pass/fail measure (*Base Success Rate – BSR*):

$$R_{\text{trad}} = \frac{N_{\text{success}}}{N_{\text{total}}} \times 100,$$

indicates the fraction of runs that eventually produce a valid artifact. Although widely used, this metric ignores differences in convergence effort. To apply a smooth, gradually increasing penalty, we use an *Exponential-Weighted Success Rate* (ExpSR):

$$R_{\text{exp}} = \frac{1}{N_{\text{total}}} \sum_{i=1}^{N_{\text{success}}} e^{-0.5(g_i-1)} \times 100.$$

This formulation acts as a "forgiveness curve", where additional retries reduce credit, but not as abruptly as in efficiency-based weighting [56]. We also consider a simple proportional penalty (*Linear-Weighted Success Rate – LwST*):

$$R_{\text{lin}} = \frac{1}{N_{\text{total}}} \sum_{i=1}^{N_{\text{success}}} \max(1 - 0.2(g_i - 1),\ 0.1) \times 100.$$

Each retry reduces the score by approximately 20%, with a minimum floor of 10% [53]. Finally, we define a discretized scoring scheme (*Robust Rank-Based Success Rate – RobSR*):

$$s(g_i) = \begin{cases} 1.0, & g_i \le 2, \\ 0.6, & 3 \le g_i \le 4, \\ 0.3, & 5 \le g_i \le 6, \\ 0.1, & g_i > 6, \end{cases}$$

$$R_{\text{robust}} = \frac{1}{N_{\text{total}}} \sum_{i=1}^{N_{\text{success}}} s(g_i) \times 100.$$

This bucketed approach is resilient to noise and well-suited for large-scale comparisons where retry variability is significant [13].

These different metrics aim to reveal two dimensions of behavior. First, **capacity** (*can the model eventually succeed?*) and, second, **discipline** (*how reliably and quickly does it converge?*). For example, a target processor (e.g., CPU or TPU) may exhibit a high traditional success rate yet a low efficiency-weighted score. This indicates that it frequently succeeds but only after many retries. Differently, a high efficiency-weighted score reflects strong internal stability and deterministic behavior, properties that are particularly desirable in LLM-driven automation pipelines where cumulative retries amplify latency and operational cost.

Table 4 reports how success rates drop when progressively penalizing late convergence, moving beyond the traditional binary notion of success. While BSR captures whether a run eventually succeeds, the efficiency-weighted metrics (ExpSR, LwSR, RobSR) expose how disciplined that success is in terms of retries.

A first observation is that models with similar BSR values can exhibit markedly different efficiency profiles. For instance, several

		BSR	ExpSR	LwSR	RobSR
Q14B	Py-CPU	76.67	-35.10%	-22.67%	-14.33%
	Py-TPU	100.00	-23.63%	-14.00%	-6.33%
Q32B	Py-CPU	60.00	-18.58%	-14.00%	-11.33%
	Py-TPU	100.00	-22.29%	-14.00%	-7.67%
P4-14B	Py-CPU	100.00	-29.52%	-23.33%	-18.67%
	Py-TPU	100.00	-17.30%	-10.00%	-4.00%
C22B	Py-CPU	16.67	-9.39%	-6.67%	-5.33%
	Py-TPU	33.33	-17.81%	-12.67%	-8.67%
G27B	Py-CPU	80.00	-34.29%	-22.00%	-14.67%
	Py-TPU	6.67	-4.99%	-4.00%	-3.67%

Table 4: SG success rate variation under alternative efficiency-aware metrics, highlighting penalties for late convergence across models and execution targets.

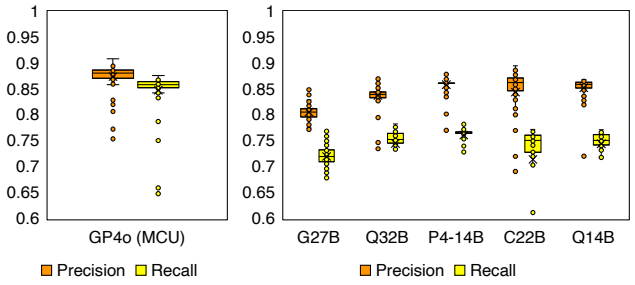


Figure 9: Semantic similarity between generated code and reference implementations, measured using CodeBERT, for successful executions on MCU and MPU targets.

configurations reach high or even perfect BSR, yet suffer substantial drops under ExpSR and LwSR. In these cases, success is often achieved only after multiple regeneration attempts, masking instability when using a pass/fail metric alone. The color gradients make this contrast immediately visible. P4-14B consistently outperforms because, although its BSR is comparable to other strong models, it experiences significantly smaller penalties under all efficiency-aware metrics, especially in the TPU setting. This confirms earlier observations that P4-14B converges quickly and reliably, not merely eventually. Models such as Q14B and G27B show sharp degradations under ExpSR and LwSR, particularly in CPU-based execution. Despite decent BSR values, their efficiency-weighted scores reveal a tendency to require repeated retries before convergence. This pattern highlights why raw success rate alone is insufficient for evaluating LLM-driven automation pipelines. Similarly, the RobSR metric emphasizes this distinction by discretizing convergence quality. Here, TPU-based executions again appear more robust, with smaller penalties across models.

4.4 Semantic Fidelity of Generated Code

So far, our evaluation has focused on whether the Autopilot can produce deployable artifacts and how efficiently it converges. However, successful execution alone does not guarantee that the generated code faithfully implements the intended algorithm [3]. To complement execution-based validation, we explicitly analyze the semantic fidelity of generated artifacts. Specifically, earlier sections assess whether the pipeline converges to executable outputs, while the following analysis examines whether those outputs remain aligned with the intended algorithmic behavior. In embedded settings, it is possible for code to compile, run, and produce outputs, while still deviating from the reference logic. To probe this dimension, we analyze the semantic similarity between generated code and trusted reference implementations.

We restrict this analysis to successful executions only, ensuring that all compared artifacts compile and run correctly. For microcontroller targets, generated Arduino sketches are compared against the official Arduino fruit classification reference implementation. For MPU-class devices, Python sketches are compared against a widely used MobileNet-based inference repository. These references represent canonical, hand-engineered implementations of the same functionality targeted by the Autopilot.

To quantify similarity, we compute CodeBERT-based similarity scores between each generated program and its corresponding reference. It is worth mentioning that, while such metrics do not capture functional equivalence exhaustively, they provide a useful proxy for assessing whether LLM-generated code follows the expected algorithmic structure or diverges into alternative, but potentially fragile, implementations [17, 23].

For both MCU and MPU targets, the results of Fig. 9 show that successful executions tend to exhibit high semantic similarity to the corresponding reference implementations. Across models, CodeBERT scores frequently cluster around values (e.g., $\approx 0.8 - 0.9$) that are typically associated with strong structural alignment, indicating that the generated artifacts often follow the intended algorithmic design rather than relying on ad hoc or spurious behaviors.At the same time, we observe non-trivial variance in semantic similarity, even among executions that compile and run correctly.In both device classes, some generated sketches deviate noticeably from the reference logic while still producing valid outputs [40, 44]. This variance highlights an important limitation of execution-based validation alone, as functional success does not guarantee semantic fidelity. Looking at this in practice, such cases would benefit from lightweight human inspection or additional validation mechanisms to ensure that deviations do not introduce latent errors or reduce long-term robustness.

Looking at the models' performance, P4-14B consistently achieves the highest and most stable similarity scores. This result is fully aligned with the earlier findings that it combines reliable convergence with semantically-aligned code generation. Other models perform comparably on MPU targets, though with broader dispersion. Gemma exhibits higher variability, aligning with its less stable behavior observed in previous analyses. The main outcome of this analysis is that our framework can, overall, produce code that is both executable and semantically aligned with expert-written implementations. However, we also expose a residual gap between working code and assured code, underscoring the need for additional validation and fallback mechanisms.

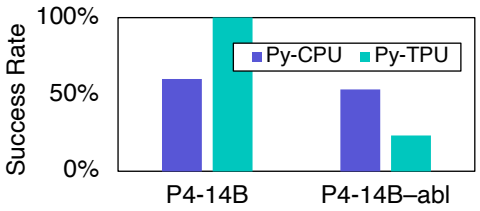


Figure 10: Impact of reduced prompt constraints on Py-CPU success rate for P4-14B across CPU and TPU targets.

4.5 Ablation Study: Effect of Reduced Prompt Constraints

We now want to assess the contribution of engineering choices in the Autopilot design by conducting a targeted ablation study. We evaluate the impact of simplifying the prompt structure by removing several explicit constraints, validation cues, and formatting guidelines used in the full prompt pipeline (Fig. 3) All other components remain unchanged, including the model (P4-14B), hardware targets, execution environment, and success criteria. We select P4-14B for this study as it exhibited the most stable and highest success rates in prior experiments, allowing us to isolate the effect of prompt constraints under otherwise favorable conditions [3]. This ablated configuration approximates a *relaxed* LLM generation setting, with reduced prompt structure, compressed *context*, and simplified validation cues, serving as a natural baseline for assessing the contribution of structured orchestration.

Fig. 10 compares success rates for the full Autopilot configuration and the ablated variant across CPU and TPU execution. The evaluation shows a clear degradation in performance when prompt constraints are relaxed [20]. While the full configuration achieves consistently high success rates, reaching full success on TPU targets, the ablated variant exhibits a substantial drop, particularly on TPU, where success decreases sharply.

This represents a key property, indicating that reliable convergence is not solely a property of the underlying LLM, but emerges from the interaction between the model and the entire middleware logic [50], highlighting the limitations of unconstrained generation. Explicit constraints encoded in the prompt serve as guardrails that reduce ambiguity, limit invalid generations, and improve determinism across heterogeneous execution targets [4].

5 Related Work

Middleware and Toolchains for Embedded ML. Middleware systems for edge environments traditionally abstract hardware heterogeneity [38, 41]. In the context of ML, embedded ML inference engines, such as TensorFlow Lite Micro [11], provide the runtime environment for model execution but treat deployment artifacts as static external inputs. To streamline development, popular frameworks like TensorFlow Lite [18] and PyTorch [36] provide comprehensive tools for training, conversion, and inference; commercial platforms like Edge Impulse [5] and vendor-specific tools (e.g., STM32Cube.AI [46]) offer end-to-end pipelines. While these platforms streamline workflows, they provide orchestration rather than synthesis, where developers manually produce and integrate artifacts. Our framework addresses this brittleness by leveraging LLM-based code generation as middleware that dynamically synthesizes artifacts in response to changing requirements.

LLM-Driven Embedded Code Generation. Recent research has applied LLMs to embedded software engineering, progressing from simple snippet generation [15] to more robust agentic workflows. Tools like LLMind 2.0 [14], *spec2code* [37], and IoTPilot [22] address specific code generation challenges through LLM fine-tuning, knowledge integration, or formal verification. However, these approaches often require device-specific fine-tuning, address isolated issues, or demand substantial manual overhead, limiting scalability.

LLMs for Embedded and IoT Workflow Automation. Beyond system development, LLMs have been applied to workflow automation of embedded and IoT workflows. AutoIOT [45] translates natural language descriptions into executable Artificial Intelligence of Things (AIoT) applications using iterative prompt refinement, but validates solutions within virtualized environments. Other platforms, including EmbedGenius [52] and EmbedAgent [51], focus on improving dependency resolution in generated code or benchmarking LLM performance through multi-agent development paradigms. ChatIoT [19] and LLM4TAP [55] achieve workflow automation through LLM-generated trigger-action programs (TAP) in IoT and smart home systems.

6 Conclusion

This paper presented an extensive experimental study of LLM-driven automation for embedded ML pipelines, grounded in the design and deployment of a middleware Autopilot that orchestrates heterogeneous tools, devices, and execution targets. Through large-scale experimentation across MCU- and MPU-class platforms, we showed that end-to-end automation is feasible in practice, but remains inherently fragile when driven by unconstrained model generation. These findings suggest that LLM-based automation for embedded ML should be approached as a middleware problem, where guardrails, validation, and fallback mechanisms are first-class design elements, as execution-based success alone does not guarantee semantic correctness and deployable artifacts may still deviate from intended logic. Autopilots can reduce human effort and enable rapid deployment, but they require "parachutes", in the form of structured control and recovery mechanisms (e.g., stage-aware prompting, validation loops, constrained regeneration), as well as the need of additional safeguards highlighted by our empirical findings. We believe these lessons will inform the design of future middleware systems that integrate generative models into dependable, resource-constrained computing environments.

References

[1] Youssef Abadade, Anas Temouden, Hatim Bamoumen, Nabil Benamar, Yousra Chtouki, and Abdelhakim Senhaji Hafid. 2023. A Comprehensive Survey on TinyML. *IEEE Access* 11 (2023), 96892–96922. doi:10.1109/ACCESS.2023.3294111

[2] Muhammad Arif and Muhammad Rashid. 2025. A Literature Review on Model Conversion, Inference, and Learning Strategies in EdgeML with TinyML Deployment. *Computers, Materials & Continua* 83, 1 (2025).

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[4] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073* (2022).

[5] Colby Banbury, Vijay Janapa Reddi, Alexander Elium, Shawn Hymel, David Tischler, Daniel Situnayake, Carl Ward, Louis Moreau, Jenny Plunkett, Matthew Kelcey, Mathijs Baaijens, Alessandro Grande, Dmitry Maslov, Arthur Beavis, Jan Jongboom, and Jessica Quaye. 2023. Edge Impulse: An MLOps Platform for Tiny Machine Learning. In *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5. Curan, 254–268. https://proceedings.mlsys.org/paper_files/paper/2023/file/49fe55f5e9574714dda575bfb2177662-Paper-mlsys2023.pdf

[6] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the dangers of stochastic parrots: Can language models be too big?. In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*. 610–623.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[8] Langfuse by ClickHouse. 2022. Langfuse. Retrieved April 27, 2026 from https://langfuse.com/

[9] Luigi Capogrosso, Federico Cunico, Dong Seon Cheng, Franco Fummi, and Marco Cristani. 2024. A Machine Learning-Oriented Survey on Tiny Machine Learning. *IEEE Access* 12 (2024), 23406–23426. doi:10.1109/ACCESS.2024.3365349

[10] QiHong Chen, Jiachen Yu, Jiawei Li, Jiecheng Deng, Justin Tian Jin Chen, and Iftekhar Ahmed. 2025. A Deep Dive Into Large Language Model Code Generation Mistakes: What and Why? arXiv:2411.01414 doi:10.48550/arXiv.2411.01414

[11] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. 2021. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. 3 (2021), 800–811. https://proceedings.mlsys.org/paper_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf

[12] Hao Ding, Ziwei Fan, Ingo Guehring, Gaurav Gupta, Wooseok Ha, Jun Huan, Linbo Liu, Behrooz Omidvar-Tehrani, Shiqi Wang, and Hao Zhou. 2024. Reasoning and Planning with Large Language Models in Code Development. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Barcelona Spain, 2024-08-25). ACM, 6480–6490. doi:10.1145/3637528.3671452

[13] Elizabeth D. Dolan and Jorge J. Moré. 2004. Benchmarking Optimization Software with Performance Profiles. arXiv:cs/0102001 doi:10.48550/arXiv.cs/0102001

[14] Yuyang Du, Qun Yang, Liujianfu Wang, Jingqi Lin, Hongwei Cui, and Soung Chang Liew. 2025. LLMind 2.0: Distributed IoT Automation with Natural Language M2M Communication and Lightweight LLM Agents. arXiv:2508.13920 [eess] doi:10.48550/arXiv.2508.13920

[15] Zachary Englhardt, Richard Li, Dilini Nissanka, Zhihan Zhang, Girish Narayanswamy, Joseph Breda, Xin Liu, Shwetak Patel, and Vikram Iyer. 2024. Exploring and Characterizing Large Language Models for Embedded System Development and Debugging. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems* (Honolulu HI USA, 2024-05-11). ACM, 1–9. doi:10.1145/3613905.3650764

[16] Federico Errica, Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. 2025. What Did I Do Wrong? Quantifying LLMs' Sensitivity and Consistency to Prompt Engineering. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 1543–1558. doi:10.18653/v1/2025.naacl-long.73

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[18] Google AI for Developers. 2026. LiteRT – High-Performance On-Device Machine Learning Framework. Retrieved April 27, 2026 from https://ai.google.dev/edge/litert

[19] Yi Gao, Kaijie Xiao, Fu Li, Weifeng Xu, Jiaming Huang, and Wei Dong. 2024. ChatIoT: Zero-code Generation of Trigger-action Based IoT Programs. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 3 (Aug. 2024), 1–29. doi:10.1145/3678585

[20] Gaël Gendron, Qiming Bao, Michael Witbrock, and Gillian Dobbie. 2023. Large language models are not strong abstract reasoners. *arXiv preprint arXiv:2305.19555* (2023).

[21] Sukhpal Singh Gill, Muhammed Golec, Jianmin Hu, Minxian Xu, Junhui Du, Huaming Wu, Guneet Kaur Walia, Subramaniam Subramanian Murugesan, Babar Ali, Mohit Kumar, Kejiang Ye, Prabal Verma, Surendra Kumar, Felix Cuadrado, and Steve Uhlig. 2025. Edge AI: A Taxonomy, Systematic Review and Future Directions. *Cluster Computing* 28, 1 (Feb. 2025), 18. doi:10.1007/s10586-024-04686-y

[22] Kaijie Gong, Wei Dong, Hao Wang, Yingqi Peng, and Yi Gao. 2025. Programming Embedded IoT Applications in Natural Language with IoTPilot. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services* (Hilton Anaheim Anaheim CA USA, 2025-06-23). ACM, 70–82. doi:10.1145/3711875.3729136

[23] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[24] Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X. Wang, and Sadid Hasan. 2024. Does Prompt Formatting Have Any Impact on LLM Performance? arXiv:2411.10541 doi:10.48550/arXiv.2411.10541

[25] Riku Immonen and Timo Hämäläinen. 2022. Tiny Machine Learning for Resource-Constrained Microcontrollers. 2022 (2022), 1–11. doi:10.1155/2022/7437023

[26] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM computing surveys* 55, 12 (2023), 1–38.

[27] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 1430–1442. doi:10.1109/ICSE48619.2023.00125

[28] Sam Leroux, Pieter Simoens, Meelis Lootus, Kartik Thakore, and Akshay Sharma. 2022. TinyMLOps: Operational Challenges for Widespread Edge AI Adoption. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (Lyon, France, 2022-05). IEEE, 1003–1010. doi:10.1109/IPDPSW55747.2022.00160

[29] LiteLLM. 2023. LiteLLM. Retrieved April 27, 2026 from https://www.litellm.ai/

[30] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 943, 15 pages.

[31] Mingxing Liu, Junfeng Wang, Tao Lin, Quan Ma, Zhiyang Fang, and Yanqun Wu. 2024. An Empirical Study of the Code Generation of Safety-Critical Software Using LLMs. 14, 3 (2024), 1046. doi:10.3390/app14031046

[32] Minh Tri Lê and Julyan Arbel. 2023. TinyMLOps for real-time ultra-low power MCUs applied to frame-based event classification. In *Proceedings of the 3rd Workshop on Machine Learning and Systems* (Rome Italy, 2023-05-08). ACM, 148–153. doi:10.1145/3578356.3592586

[33] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems* 36 (2023), 46534–46594.

[34] Vasilios Mavroudis. 2025. LangChain. (2025).

[35] Roberto Morabito and Guanghan Wu. 2025. When the Code Autopilot Breaks: Why Large Language Models Falter in Embedded Machine Learning. *Computer* 58, 11 (2025), 32–41.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems* (2019), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[37] Minal Suresh Patil, Gustav Ung, and Mattias Nyberg. 2025. Towards Specification-Driven LLM-Based Generation of Embedded Automotive Software. In *Bridging the Gap Between AI and Reality* (Cham, 2025), Bernhard Steffen (Ed.). Springer Nature Switzerland, 125–144. doi:10.1007/978-3-031-75434-0_9

[38] Tobias Pfandzelter, Aditya Dhakal, Eitan Frachtenberg, Sai Rahul Chalamalasetti, Darel Emmot, Ninad Hogade, Rolando Pablo Hong Enriquez, Gourav Rattihalli, David Bermbach, and Dejan Milojicic. 2023. Kernel-as-a-Service: A Serverless Programming Model for Heterogeneous Hardware Accelerators. In *Proceedings of the 24th International Middleware Conference* (Bologna, Italy) *(Middleware '23)*. Association for Computing Machinery, New York, NY, USA, 192–206. doi:10.

1145/3590140.3629115

[39] Partha Pratim Ray. 2022. A review on TinyML: State-of-the-art and prospects. *J. King Saud Univ. Comput. Inf. Sci.* 34, 4 (April 2022), 1595–1623. doi:10.1016/j.jksuci.2021.11.019

[40] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond accuracy: Behavioral testing of NLP models with CheckList. *arXiv preprint arXiv:2005.04118* (2020).

[41] Isabelly Rocha, Pascal Felber, Valerio Schiavoni, and Lydia Chen. 2022. EdgeTune: Inference-Aware Multi-Parameter Tuning. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference* (Quebec, QC, Canada) *(Middleware '22)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3528535.3533273

[42] Fatemeh Sarhaddi, Ngoc Thi Nguyen, Agustin Zuniga, Pan Hui, Sasu Tarkoma, Huber Flores, and Petteri Nurmi. 2025. LLMs and IoT: A Comprehensive Survey on Large Language Models and the Internet of Things. doi:10.36227/techrxiv.174063060.01215875/v1

[43] Jaakko Sauvola, Sasu Tarkoma, Mika Klemettinen, Jukka Riekki, and David Doermann. 2024. Future of software development with generative AI. 31, 1 (2024), 26. doi:10.1007/s10515-024-00426-z

[44] Richard Schumi and Jun Sun. 2023. Semantic-based neural network repair. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 150–162.

[45] Leming Shen, Qiang Yang, Yuanqing Zheng, and Mo Li. 2025. AutoIOT: LLM-Driven Automated Natural Language Programming for AIoT Applications. (2025).

[46] STMicroelectronics. 2022. STM32Cube.AI. Retrieved April 27, 2026 from https://stm32ai.st.com/stm32-cube-ai/

[47] Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. 2025. Bugs in large language models generated code: an empirical study. *Empirical Software Engineering* 30, 3 (Feb. 2025). doi:10.1007/s10664-025-10614-4

[48] Jianxun Wang and Yixiang Chen. 2023. A Review on Code Generation with LLMs: Application and Evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. 284–289. doi:10.1109/MedAI59581.2023.00044

[49] Pete Warden and Daniel Situnayake. 2019. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media.

[50] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.

[51] Ruiyang Xu, Jialun Cao, Mingyuan Wu, Wenliang Zhong, Yaojie Lu, Ben He, Xianpei Han, Shing-Chi Cheung, and Le Sun. 2025. EmbedAgent: Benchmarking Large Language Models in Embedded System Development. arXiv:2506.11003 [cs] doi:10.48550/arXiv.2506.11003

[52] Huanqi Yang, Mingzhe Li, Mingda Han, Zhenjiang Li, and Weitao Xu. 2024. EmbedGenius: Towards Automated Software Development for Generic Embedded IoT Systems. arXiv:2412.09058 [cs] doi:10.48550/arXiv.2412.09058

[53] Naoki Yokoyama, Sehoon Ha, and Dhruv Batra. 2021. Success Weighted by Completion Time: A Dynamics-Aware Evaluation Criteria for Embodied Navigation. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 1562–1569. doi:10.1109/IROS51168.2021.9636743

[54] Xiang Zhang, Juntai Cao, Chenyu You, and Dujian Ding. 2025. Why Prompt Design Matters and Works: A Complexity Analysis of Prompt Search Space in LLMs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 32525–32555. doi:10.18653/v1/2025.acl-long.1562

[55] Valerie Zhao, Lefan Zhang, Bo Wang, Michael L. Littman, Shan Lu, and Blase Ur. 2021. Understanding Trigger-Action Programs Through Novel Visualizations of Program Differences. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–17. doi:10.1145/3411764.3445567

[56] Shlomo Zilberstein. 1996. Using anytime algorithms in intelligent systems. *AI magazine* 17, 3 (1996), 73–73.