POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

From IDE to Edge: A Cloud-Native Multi-Agent Framework for Automated Edge AI Deployment on STM32

Advisors
Prof. Gianvito Urgese
Andrea Pignata
Giuseppe Fanuli

Candidate
Michele Russo

March 2026

Abstract

The deployment of Artificial Intelligence (AI) on resource-constrained embedded systems, commonly known as Edge AI, is often hindered by stringent memory, computational power, and energy constraints. While the STM32 ecosystem offers specialized tools such as STM32CubeMX and X-CUBE-AI, orchestrating the full Machine Learning Operations (MLOps) lifecycle, from firmware configuration to model optimization and deployment, remains a complex and primarily manual process.

This thesis introduces a scalable Agentic MLOps Orchestration Framework designed to automate and optimize the Edge AI development workflow on STM32 microcontrollers. To move beyond the limitations of local execution, the proposed architecture adopts a cloud-native approach that maximizes resource utilization by decoupling the multi-agent orchestration logic (CPU-bound) from the Large Language Model (LLM) inference engine (GPU-bound).

A central contribution of this work is the adoption of a scalable inference architecture that transitions from monolithic, resource-monopolizing environments to a distributed model where high-performance LLMs are served as shared network resources. This design facilitates dynamic resource management, which allows multiple concurrent agentic workflows to utilize centralized computational power efficiently. This approach addresses the limitations of local execution by providing a more flexible and hardware-aware infrastructure for multi-agent MLOps.

The framework manages seven specialized workflows, including automated firmware project generation, hardware-aware AI model analysis, and "neural network surgery," all orchestrated via LangGraph and integrated directly into the Visual Studio Code IDE. The effectiveness of this framework is evaluated through experimental validation across various system configurations and its impact on the development lifecycle. The findings indicate that the proposed agentic orchestration substantially enhances developer productivity and reduces manual errors, establishing a reliable path toward optimized and reliable Edge AI deployment on STM32 microcontrollers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The deployment of Artificial Intelligence (AI) on embedded systems, commonly referred to as Edge AI, is one of the most active research areas in computing today. Unlike cloud-based AI, which relies on virtually unlimited computational resources, Edge AI requires the execution of complex models on hardware with severe constraints in terms of memory, processing power, and energy consumption. Among the most prevalent platforms for these applications is the STM32 family of microcontrollers, which offers a broad range of solutions for industrial, automotive, and consumer electronics.

However, the path from a high-level AI model developed in Python to an efficient firmware implementation in C remains fraught with complexity. Developers must navigate a fragmented ecosystem of tools, including STM32CubeMX for hardware configuration, STEdgeAI for model optimization, and various Python libraries for training, creating a significant "plumbing" problem. This disconnect between the AI development lifecycle and embedded systems engineering leads to slow iteration cycles, high error rates, and a steep barrier to entry for practitioners who lack deep expertise in both domains.

This research addresses these challenges by proposing an **Agentic MLOps orchestration framework** specifically designed for the STM32 ecosystem. Rather than treating the deployment as a linear sequence of tasks, the proposed approach introduces a **hardware-aware** assistant capable of automating the entire development lifecycle, from the initial data preparation to the final on-device verification. The framework utilizes the reasoning capabilities of Large Language Models (LLMs) within a multi-agent architecture built on LangGraph to provide a unified interface connecting AI research workflows to embedded implementation.

The unique contribution of this work lies in the integration of **TinyMLOps** principles through an agent-driven paradigm. The assistant does not merely act as a code generator; it performs specialized operations such as "Neural Network Customization," where model architectures are automatically modified to comply

with hardware-specific constraints. This included replacing unsupported layers, optimizing memory footprints, and orchestrating hyperparameter tuning via the Neural Network Intelligence (NNI) toolkit, all while maintaining functional correctness across the MLOps pipeline.

The proposed architecture is organized into several specialized workflows. The system manages modular subgraphs for firmware project generation, AI-driven model analysis and quantization, and hardware-software integration. Each node in the graph represents a distinct phase of the MLOps lifecycle, ensuring that design decisions, such as the selection of quantization levels or the insertion of specific integration headers, are made with a deep understanding of the performance requirements of the target microcontroller.

Experimental results demonstrate that this structured, agent-based orchestration significantly reduces the cognitive load on developers and accelerates the prototyping phase of Edge AI applications. By automating the integration between heterogeneous tools and providing an interactive assistant that navigates the STEdgeAI suite, the system allows practitioners to iterate on their designs without requiring deep familiarity with every underlying CLI tool.

The primary contributions of this work are threefold. First, a detailed **multi-agent orchestration framework** for Edge AI development is presented, showing how agentic workflows can manage the end-to-end MLOps lifecycle. Second, a **hardware-aware customization pipeline** is developed, enabling automated model adaptation and optimization specifically tailored for STM32 resource profiles. Third, a rigorous evaluation methodology is established, combining qualitative assessments of development efficiency with quantitative validations using hardware-specific metrics (latency, memory usage) and state-of-the-art LLM-as-Judge frameworks (*DeepEval*) to ensure the reliability and faithfulness of the automated configurations.

Chapter 2

Background

This background section reviews the foundational concepts and methodologies that support the integrated workflow presented in this thesis. It begins by covering the basics of Artificial Neural Networks and the constraints of Edge AI systems, particularly within the STM32 ecosystem. It then explores the evolution of Large Language Models and the emerging field of Agentic MLOps (Machine Learning Operations). Finally, it surveys tools for workflow orchestration, information retrieval using RAG (Retrieval-Augmented Generation), and automated validation.

All these components serve as the basic building blocks for the final goal of this work, which consists in orchestrating heterogeneous processes, involving large language models, hardware-specific optimization, and automated firmware generation, into a single, reliable framework [1, 2]. Achieving this requires integrating diverse components, from Edge AI conversion tools and micro-controller configuration environments to agentic reasoning engines and code verification tools.

2.1 Artificial Neural Networks

The scientific quest to create machines capable of emulating human intelligence, formally recognized as Artificial Intelligence (AI), began in the mid-20th century. While early efforts focused on symbolic AI, where expert knowledge was translated into rigid, rule-based logic, these systems often struggled with the ambiguity and complexity of real-world data environments [3].

To overcome these limitations, the field shifted toward Machine Learning (ML) in the 1980s. This paradigm emphasized data-driven learning, where algorithms improve their performance through experience rather than explicit programming [4]. By the 2010s, the advent of Deep Learning (DL) revolutionized the industry. By utilizing multi-layered neural networks, deep learning enabled the automatic extraction of hierarchical features from raw data, leading to unprecedented breakthroughs in computer vision and natural language processing [5].

2.1.1 The Perceptron: The Fundamental Building Block

The structural foundation of modern neural networks is the perceptron, a computational model inspired by the biological neuron. A perceptron processes input signals by calculating a weighted sum and applying a step function to produce a binary classification [6]:

$$y = \text{step}(\mathbf{w}^\top \mathbf{x} + b) \tag{2.1}$$

where $\mathbf{w}$ represents the weight vector, $\mathbf{x}$ the input, and b the bias. While a single perceptron is limited to linearly separable tasks, stacking multiple neurons into layers, forming a Multilayer Perceptron (MLP), allows for the approximation of highly non-linear functions, as illustrated in Fig. 2.1 [7, 8].

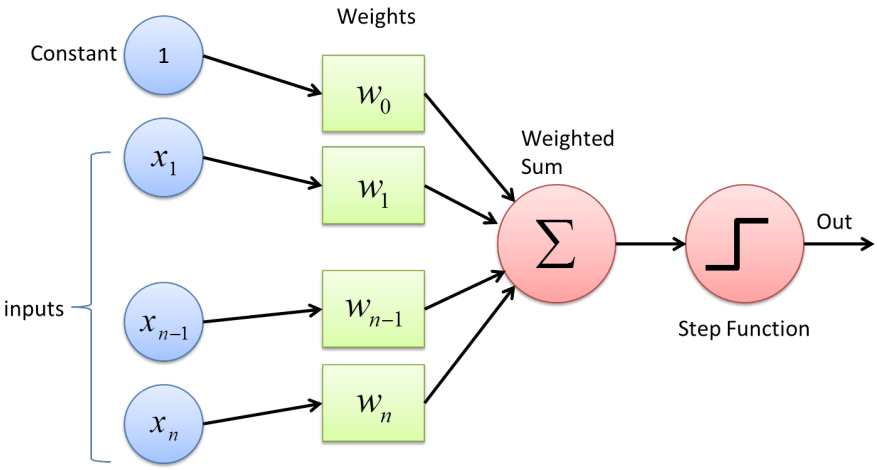

Figure 2.1: Visual representation of a perceptron and its logical components [9].

2.1.2 Optimization and Training

Training a neural network is an optimization problem: finding the parameter set $\boldsymbol{\theta}$ that minimizes a cost function $J(\boldsymbol{\theta})$. For regression, a standard metric is the Mean Squared Error (MSE):

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.2}$$

where y_i is the target and $\hat{y}_i$ is the model's prediction [6].

Minimization is typically achieved via Gradient Descent, where weights are updated iteratively in the direction of the steepest descent:

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t) \tag{2.3}$$

The learning rate α is a critical hyperparameter that determines the step size. To handle large-scale datasets efficiently, researchers employ Stochastic Gradient

Descent (SGD) and its variants, which estimate gradients using small batches of data. While SGD is effective, it can struggle with local minima and saddle points. To address this, **Momentum** was introduced to accelerate gradients in the right direction, thus leading to faster converging.

However, the current standard for training deep neural networks is **Adam (Adaptive Moment Estimation)** [10]. Adam combines the advantages of two other extensions of stochastic gradient descent: Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). It computes adaptive learning rates for each parameter by maintaining exponentially decaying averages of past gradients (m_t, first moment) and squared gradients (v_t, second moment). This allows the algorithm to handle sparse gradients on noisy problems, making it highly suitable for large-scale data and parameter-heavy models like LLMs.

The efficiency of this process relies on the backpropagation algorithm (Fig. 2.2), which uses the chain rule of calculus to propagate errors from the output layer back to the input, enabling automated weight updates in complex computational graphs [8].

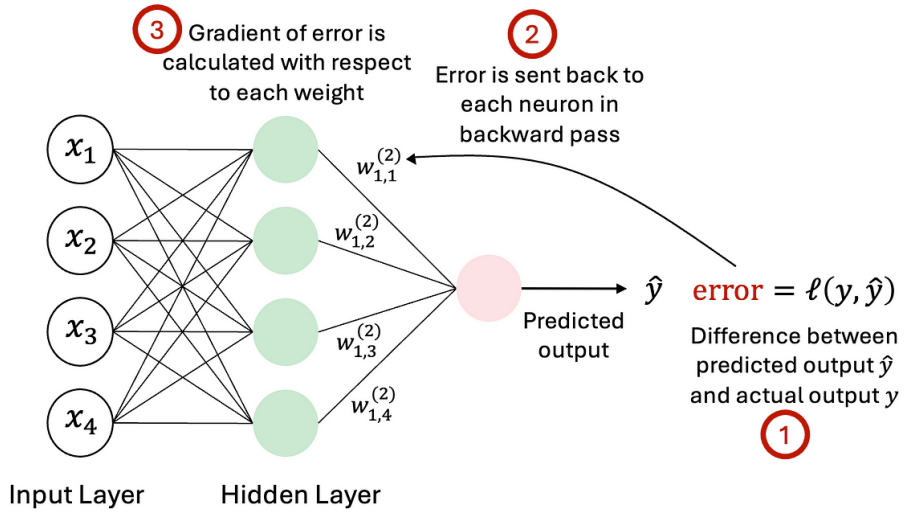

Figure 2.2: Mechanism of the backpropagation algorithm [11].

2.1.3 Specialized Architectures: CNNs and RNNs

Different data types require specialized neural architectures to capture relevant patterns efficiently.

Convolutional Neural Networks (CNNs): Optimized for spatial data (e.g., images), CNNs use convolutional kernels to detect local features like edges and textures [12]. By sharing weights across the spatial domain, CNNs achieve translation invariance and significantly reduce the total number of parameters compared to fully connected layers.

Mathematically, for a two-dimensional input image I and a filter kernel K of dimensions $k_1 \times k_2$, the discrete convolution operation at position (i, j) is defined as:

$$(I * K)(i, j) = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i - m, j - n)K(m, n) \tag{2.4}$$

This sliding window operation computes the dot product between the kernel weights and local input regions, allowing the aforementioned weight sharing across all spatial positions and consequently building feature maps that highlight spatial hierarchies.

This is often followed by activation functions like ReLU and pooling layers to downsample feature maps (Fig. 2.3) [13].

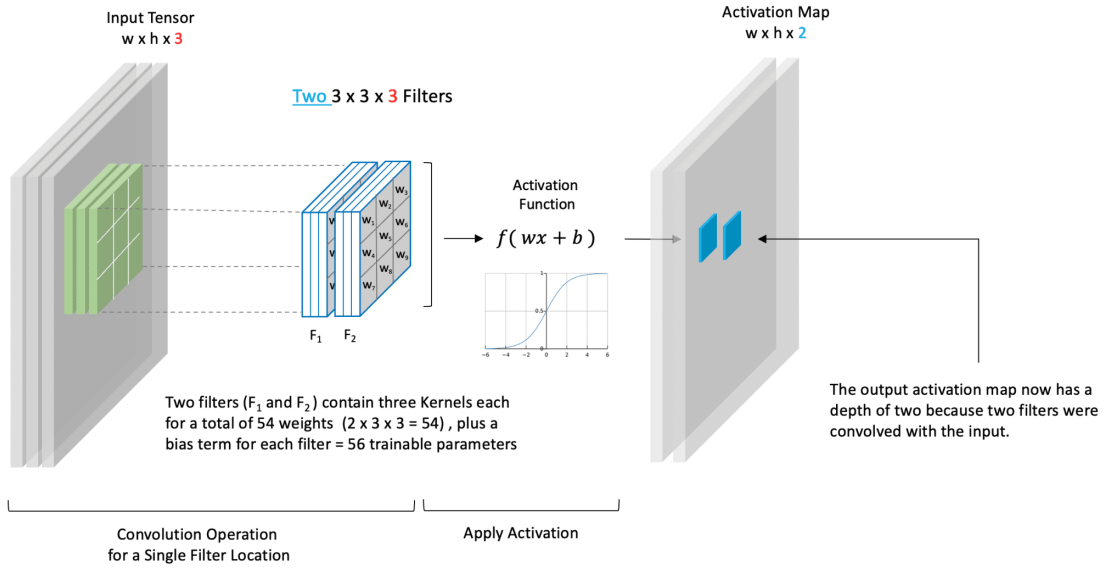

Figure 2.3: Convolution operation within a neural network layer [14].

Recurrent Neural Networks (RNNs): Tailored for sequential data where the current output depends on previous inputs, RNNs maintain a "hidden state" that acts as a memory trace [15]. The hidden state h_t at time step t is updated based on the previous hidden state h_{t-1} and the current input x_t via a non-linear activation function (typically tanh or ReLU):

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b) \tag{2.5}$$

Despite their utility, vanilla RNNs struggle with long sequences due to the **vanishing gradient problem**, where gradients diminish exponentially during Backpropagation Through Time (BPTT).

To mitigate this, **Long Short-Term Memory (LSTM)** networks introduce a more complex memory cell to regulate information flow [16]. An LSTM unit

maintains a cell state C_t controlled by three gating mechanisms: the forget gate (f_t), input gate (i_t), and output gate (o_t). The formal transition equations are:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.6}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.7}$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.8}$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{2.9}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.10}$$
$$h_t = o_t \odot \tanh(C_t) \tag{2.11}$$

where $\odot$ denotes element-wise multiplication. This architecture allows the network to effectively learn long-term dependencies (Fig. 2.4) [17].

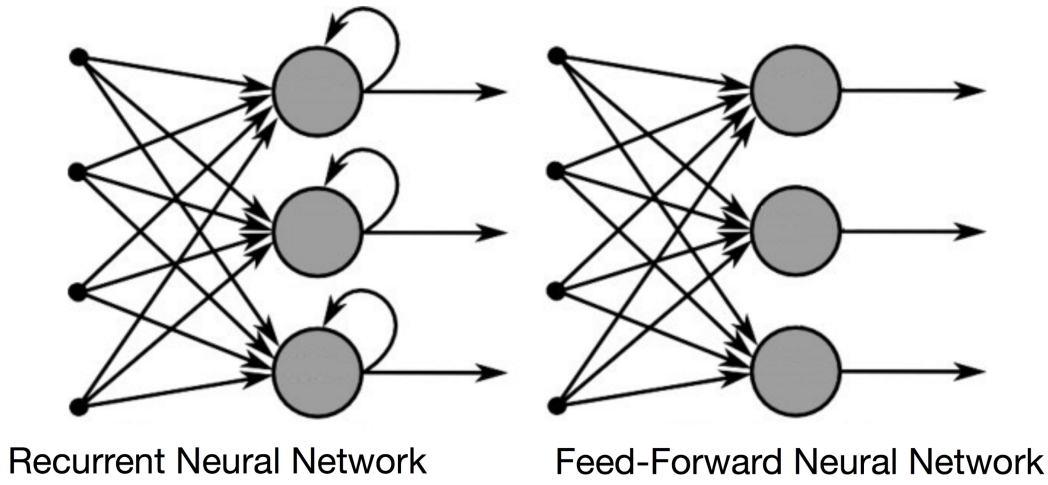

Figure 2.4: Architecture comparison: Feed-forward vs. Recurrent structures [18].

For Edge AI deployment, pre-trained models rarely satisfy the memory and latency constraints of a target microcontroller out-of-the-box. Adapting a model's architecture, by freezing backbone layers, replacing the classification head, or injecting regularization, is a standard practice for domain adaptation and hardware-driven compression. These techniques, collectively referred to as *neural network customization*, are a prerequisite for any production Edge AI workflow. Chapter 3 describes how the framework proposed in this thesis automates this process programmatically.

2.2 Edge AI and TinyML Systems

Edge Artificial Intelligence (Edge AI) refers to the deployment of machine learning models directly on embedded devices, such as microcontrollers (MCUs) and digital signal processors (DSPs), rather than on centralized cloud servers. This shift originates from the demand for low latency, reduced bandwidth consumption, data privacy, and improved energy efficiency [19]. However, deploying sophisticated AI on resource-constrained hardware like the STM32 family presents precise technical challenges.

2.2.1 Challenges of TinyML

The field of Tiny Machine Learning (TinyML) focuses on enabling inference on devices with power consumption typically in the milliwatt range [19]. The primary constraints include:

- **Memory Limitations**: MCUs generally have restricted RAM (hundreds of KB) and Flash memory (a few MB). This structural limit requires hardware-aware model compression techniques like quantization and pruning [20].
- **Computational Power**: Without high-end GPUs, models must be explicitly optimized for execution on ARM Cortex-M cores or dedicated hardware accelerators, such as the STM32 NPU.
- **Energy Efficiency**: Many embedded applications are battery-powered, demanding architectures that minimize active clock cycles and maximize sleep states.

2.2.2 TinyMLOps: MLOps for Resource-Constrained Devices

The rapid expansion of Artificial Intelligence on embedded systems forced the adaptation of traditional Machine Learning Operations (MLOps) principles to the specific constraints of micro-controllers. This domain, known as **TinyMLOps**, addresses the lifecycle management of AI models in environments where memory, compute, and power are severely limited [19]. TinyMLOps provides the methods and the tooling necessary to face the challenges outlined above.

Distinction from Cloud-Scale MLOps

While traditional MLOps manages model deployment across cloud clusters, massive datasets, and microservices, TinyMLOps prioritizes strict hardware awareness. The main differences are summarized in Table 2.1.

Table 2.1: Comparison between traditional Cloud MLOps and TinyMLOps [21].

Feature	**Cloud MLOps**	**TinyMLOps**
Target Hardware	GPU Clusters / Data Centers	MCUs (KB-scale RAM)
Optimization	Throughput / Latency	Memory / Energy Efficiency
Deployment Unit	Docker Containers / APIs	Firmware Binary Images
Connectivity	Persistent / High-bandwidth	Intermittent or Offline
Metric Priority	Prediction Accuracy	Accuracy vs. Resource Footprint

Core Pillars of TinyMLOps

A systematic TinyMLOps framework typically implements three fundamental pillars:

- **Hardware-Aware Optimization**: Unlike cloud models that scale easily with more VRAM, embedded models undergo structural transformations (e.g., quantization, pruning, and "neural surgery") to fit into specific MCU memory profiles.
- **Integrated Automation ("Plumbing")**: TinyMLOps requires the direct integration of traditionally disjoint toolchains. It connects Python-based training environments, C-based firmware generators (like STM32Cube), and binary flashing utilities.
- **Reproducibility**: Defining the end-to-end workflow as a declarative graph (e.g., via LangGraph) ensures that engineers can reconstruct and verify an optimized model consistently across different hardware revisions [22].

Adopting a TinyMLOps approach transitions the development of embedded AI from an expert-intensive manual process to an automated engineering discipline.

2.2.3 STM32 Microcontrollers Overview

To fully appreciate the scope of TinyMLOps, it is important to define the target hardware ecosystem. The STM32 portfolio, built upon ARM Cortex-M processors (such as Cortex-M4, Cortex-M33, and Cortex-M7), is divided into specific families designed for different application domains. More information on all families is given in appendix A.

The main categories relevant to Edge AI deployment include:

- **Foundation Family (e.g., STM32F4)**: These general-purpose microcontrollers feature RAM between 96 and 128 KB and Flash memory of approximately 500 to 512 KB. They power applications involving basic audio processing or simple sensor data analysis.

- **Low Power Family (e.g., STM32L4)**: Destined for applications where energy consumption is critical, this family offers extended resources, with RAM scaling up to 640 KB and Flash reaching 2 MB. They are highly suitable for low-power vision or continuous monitoring tasks.
- **High Performance Family (e.g., STM32H7)**: When the application relies strictly on the core's high performance, the H7 family delivers. With RAM capacities up to 1.4 MB, it hosts heavier models, including complex object detection computer vision architectures.

2.2.4 Model-Hardware Compatibility Matrix

To provide concrete guidance on model selection for these resource-constrained devices, Table 2.2 presents a condensed compatibility analysis of vision models from the STM32 Model Zoo across common MCU families. These models target **image classification** and **pattern recognition** tasks, ranging from simple digit recognition (ST MNIST) to object categorization (MobileNet and ResNet variants). The analysis considers both uncompressed (FP32) and INT8-quantized versions, demonstrating the effect of compression in enabling computer vision applications on lower-tier devices.

Table 2.2: Representative Model-Board Compatibility (INT8 Quantized)

Model	**Input Size**	**RAM (INT8)**	**Flash (INT8)**	**F401 64KB/256KB**	**L476 128KB/1MB**	**H743 1MB/2MB**	**N6570 5MB/4MB**
ST MNIST	28×28	5 KB	25 KB	✓	✓	✓	✓
FD-MobileNet 0.25	96×96	80 KB	180 KB	×	✓	✓	✓
MobileNet V2 0.35	128×128	150 KB	280 KB	×	~	✓	✓
ResNet V1 32 (CIFAR)	32×32	200 KB	450 KB	×	×	✓	✓
MobileNet V2 1.0	224×224	600 KB	1.2 MB	×	×	~	✓

Legend: ✓= Fits comfortably (<80% resources); ~= Tight fit (80-100%); × = Does not fit

I derived key observations from this analysis:

- **INT8 Quantization Impact**: Compression reduces the model footprint by approximately 4× compared to FP32. It enables the deployment of models like MobileNet V2 0.35 on mid-tier MCUs (L476) that would otherwise exceed hardware limits [20].
- **Vision Models on F401**: Standard image classification architectures remain incompatible with the STM32F401 (64KB RAM) even with extreme quantization, limiting this platform to audio-based inference or ultra-lightweight custom Neural Networks.

- **H7 Family as Sweet Spot**: The STM32H743/H747 (1MB RAM) emerges as the optimal target for production-grade vision AI, executing MobileNet variants at 224×224 resolution with adequate latency.

The complete compatibility matrix, including FP32 resource requirements and additional model architectures (EfficientNet, SqueezeNet, ResNet50), is available in Appendix A.

2.2.5 The STM32 Developer Toolchain

STMicroelectronics maintains an ecosystem for migrating high-level AI models to optimized C code. The core tools integrated within our workflow include:

- **STM32CubeMX**: A graphical tool for configuring MCU peripherals and generating initialization code. In an automated MLOps workflow, this tool provides the "hardware context" (e.g., pinout, clock settings) necessary for firmware integration.
- **STEdgeAI / X-CUBE-AI**: This specialized utility converts pre-trained models (from TensorFlow Lite, Keras, or ONNX) into optimized C code for STM32. It yields critical feedback on memory footprint (RAM/Flash) and execution cycles, acting as the interface between the AI and the embedded domains [20].

The orchestration of these tools typically forces engineers into iterative manual cycles. Developers repeatedly adjust model parameters in Python, execute the conversion tool, and supervise hardware compatibility. The "GUI-centric" operation of tools like STM32CubeMX acts as a bottleneck for industrial MLOps pipelines because it prevents version control and automation within continuous integration systems. To resolve this, the framework introduced in this thesis automates the "plumbing" of the pipeline via headless agentic workflows.

2.3 Large Language Models

Large Language Models (LLMs) represent a significant shift in artificial intelligence, moving from specialized pattern matchers to flexible reasoning engines capable of understanding and generating human-like language across diverse domains. These models are designed to capture the structural and semantic nuances of text, allowing them to perform complex tasks such as reasoning, summarization, and code synthesis [23]. In the context of MLOps, LLMs serve as intelligent orchestrators, capable of interpreting high-level user requirements and translating them into executable workflows or hardware-specific configurations [24].

2.3.1 Evolution of Language Modeling

The development of modern LLMs scales from early statistical approaches to the massive neural architectures of today [25]. Traditional language models relied on N-grams and the Markov assumption to predict the next word based on a limited history, often suffering from sparse data and a lack of long-range coherence. The introduction of Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) units allowed for better temporal modeling, but it was the *Transformer* architecture [26] that truly unlocked the current era of LLMs.

The self-attention mechanism enables Transformers to process data in parallel and capture global dependencies. Following the success of GPT-2 [27] and GPT-3 [28], the industry observed "scaling laws": increasing model parameters, data volume, and compute budget leads to predictable improvements. This trend was further refined by compute-optimal strategies, such as those in the Chinchilla study [29], and by alignment techniques like Reinforcement Learning from Human Feedback (RLHF), which help ensure models are helpful, honest, and harmless (HHH) [30, 31, 32]. The exponential growth of these models is visualized in Fig. 2.5.

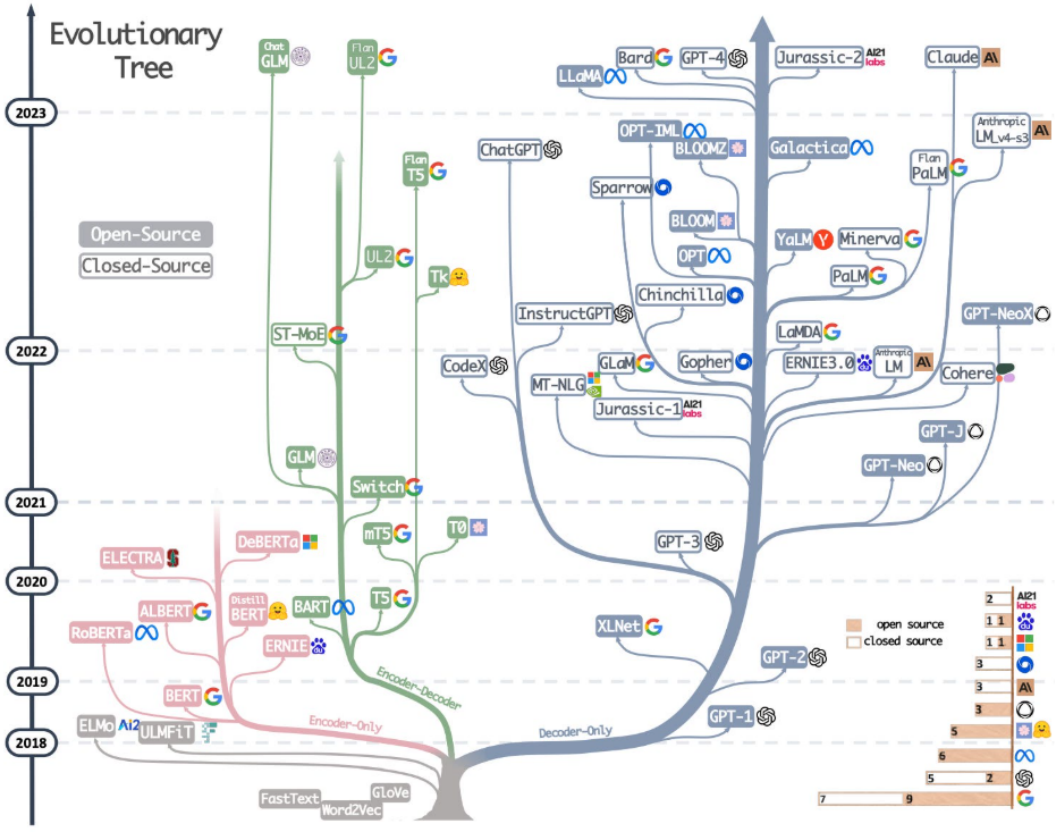

Figure 2.5: The rapid expansion of LLM parameters and capabilities over time [33].

2.3.2 The Attention Mechanism

The core innovation of the Transformer is the self-attention mechanism. It allows the model to dynamically weight the importance of different tokens in a sequence relative to one another. Unlike sequential models, self-attention computes pairwise interactions between all tokens simultaneously, effectively bypassing the limits of fixed context windows.

Mathematically, the mechanism uses three linear projections of the input embeddings: Queries (Q), Keys (K), and Values (V). The attention score is calculated as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2.12}$$

where d_k is the dimensionality of the keys, used as a scaling factor to maintain gradient stability [26]. This process, which helps context-heavy tasks like translation, is visualized in Fig. 2.6. Multi-head attention extends this by running several attention layers in parallel, allowing the model to focus on different aspects of the input representation, such as syntax and semantics, concurrently [34].

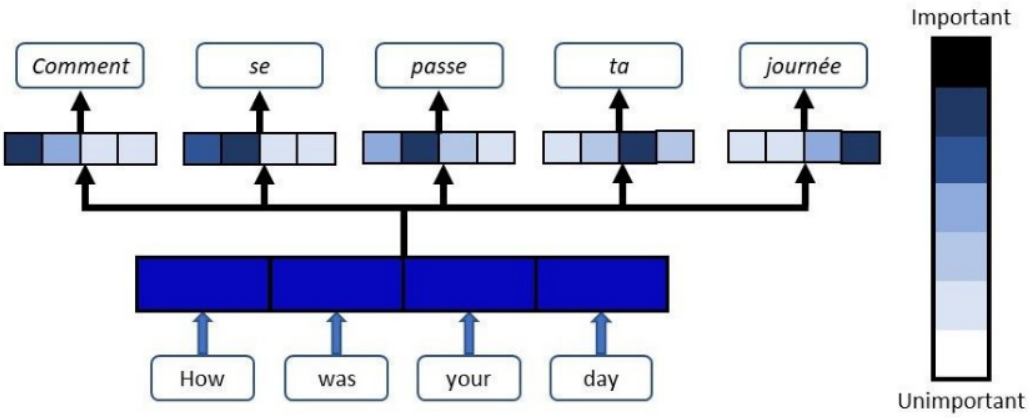

Figure 2.6: Conceptual visualization of the attention mechanism in sequence-to-sequence translation [35].

2.3.3 Training and Fine-Tuning Paradigms

The development of a production-ready LLM typically follows a three-stage pipeline:

1. **Pre-training:** The model is trained on massive, unlabeled text corpora using self-supervised objectives (e.g., Causal Language Modeling or Masked

Language Modeling) to learn general linguistic patterns and factual knowledge [36, 37].

2. **Fine-tuning:** The pre-trained model is adapted to domain-specific tasks using smaller, high-quality labeled datasets. This improves accuracy in specialized areas like medical diagnosis or firmware development [38].

3. **Instruction Tuning:** The model is trained to follow specific natural language instructions or "prompts." This stage enables zero-shot generalization, where the model can perform new tasks simply by following a well-defined prompt [39].

2.3.4 State-of-the-Art and Industry Landscape

As of 2025, a few key players dominate the LLM ecosystem, each focusing on different optimization goals. OpenAI, with the GPT-o series (including o1 and o3), has emphasized advanced reasoning and chain-of-thought capabilities. Anthropic's Claude 3.5 and 3.7 models are known for their coding precision and "Claude Artifacts" UI [40]. Google's Gemini 3 Pro [41] has recently emerged as a leader, surpassing many competitors in technical benchmarks and achieving a record 91.9% on GPQA Diamond while scaling massive 1M token context windows. However, the concept of collective intelligence challenges the prevailing paradigm of monolithic models. Orchestrated routing and aggregation of multiple open-source models can now surpass proprietary ones while cutting inference costs [42]. This shift highlights the importance of agentic orchestration frameworks, which treat models as modular components. Meanwhile, open-source initiatives like Mistral and Meta's Llama 3/4 families provide accessible models that users can deploy locally via frameworks like Ollama, supporting data privacy in research [43, 44, 45].

2.3.5 Limitations and Deployment Challenges

Despite their power, LLMs possess inherent flaws. Hallucination, the generation of plausible-sounding but factually incorrect information, remains a significant hurdle for high-stakes MLOps [46]. Models also lack true causal reasoning, relying instead on statistical patterns in their training data. In addition, the high computational cost of running large-scale models often necessitates a hybrid approach, where lightweight local models (e.g., Llama-3-8B) handle routine tasks while larger proprietary APIs are reserved for complex reasoning. This thesis addresses these limitations through Agentic Orchestration and RAG, ensuring that the LLM's outputs are grounded in hardware-specific documentation and validated through execution loops.

2.4 Agentic MLOps

The term *MLOps* (Machine Learning Operations) describes the set of practices that aims to deploy and maintain machine learning models in production reliably and efficiently. While traditional MLOps focus on cloud-centric pipelines, the emergence of LLM-based agents has introduced a new paradigm: *Agentic MLOps*.

2.4.1 Agents as Orchestrators

In an agentic MLOps framework, the traditional linear pipeline is replaced by a multi-agent system capable of reasoning about the state of the project [1, 24]. Using frameworks like LangGraph, the system can:

- **Decompose Complex Tasks**: Break down a high-level request (e.g., "Optimize this model for an STM32F4 with 128KB RAM") into sub-tasks like quantization, memory analysis, and code generation.
- **Iterate Based on Feedback**: If the memory analysis indicates a model is too large, the agent can autonomously trigger a "Neural Network Customization" process to adapt the architecture.

2.4.2 Neural Network Customization

A key component of automated MLOps for Edge AI is the ability to programmatically modify model architectures. "Neural Network Customization" refers to the automated replacement of layers, adjustment of input shapes, or modification of operations that are incompatible with specific hardware kernels. The orchestrator performs these transformations by utilizing the code-generation and reasoning capabilities of LLMs while maintaining the functional intent of the original model.

This automation addresses the "plumbing" problem, the friction between heterogeneous tools and languages, treating hardware constraints as first-class citizens in the AI optimization loop rather than late-stage integration concerns [47].

2.5 Agent Orchestration Frameworks

In modern artificial intelligence, an *agent* goes beyond a static model; it is defined as an autonomous computational entity capable of sensing its environment, reasoning through complex decision trees, and executing actions to achieve specific objectives. While Large Language Models (LLMs) serve as powerful reasoning engines, their standalone utility is constrained by their inability to interact with the external world or maintain long-term state. To bridge this gap, specific orchestration frameworks have emerged to coordinate the interactions between models, external tools (such as

compilers and hardware optimization engines), and persistent databases, effectively transforming a text-generator into a functional system [23, 1].

2.5.1 LangChain: Modular Composition of LLM Applications

LangChain has established itself as a foundational framework for simplifying the construction of LLM-powered applications [48]. Its architecture is predicated on the concept of composability, where complex applications are broken down into smaller, modular building blocks. The framework provides high-level abstractions for key components:

1. **Chains**: Sequences of operations where the output of one step serves as the input for the next, allowing for the construction of linear pipelines.

2. **Prompts**: Templates that structure the input to the model, ensuring consistent and context-aware interactions.

3. **Agents**: Entities that use an LLM as a reasoning engine to determine which actions to take and in what order.

4. **Tools**: Interfaces that allow the agent to interact with the outside world, from web searching to code execution.

This modularity allows developers to move beyond simple "prompt-response" paradigms, facilitating the creation of systems that can retrieve external data (RAG) and interact with APIs. However, typical LangChain implementations act as Directed Acyclic Graphs (DAGs), which can limit the development of highly iterative workflows where an agent needs to loop back, retry, or refine its approach based on dynamic feedback [2].

2.5.2 LangGraph: Cyclic and Stateful Orchestration

To address the limitations of linear execution, LangGraph extends the ecosystem by introducing a graph-theoretic approach to orchestration. Unlike the rigid sequences of its predecessor, LangGraph models agentic processes as stateful graphs, where **nodes** represent computational units (functions or model calls) and **edges** define the control flow logic [22, 2].

The core innovation of this framework lies in its support for **cycles**, enabling the definition of iterative loops for complex tasks. This architectural difference between linear and cyclic flows is visualized in Fig. 2.7.

For example, in the "Neural Network Customization" workflow, the system orchestrates an iterative refinement process. The assistant uses **NNI** (Neural

Network Intelligence) [49], an open-source AutoML toolkit by Microsoft, to perform internal search loops and hyperparameter optimization. The agent validates the resulting models against hardware constraints until it selects a viable configuration. This "test-and-revise" capability, which may also involve Human-in-the-Loop (HITL) feedback, is difficult to implement in linear chains but natural in a graph structure. LangGraph treats the agent's execution as a state machine, where decisions govern transitions between states, allowing for dynamic branching and error recovery.

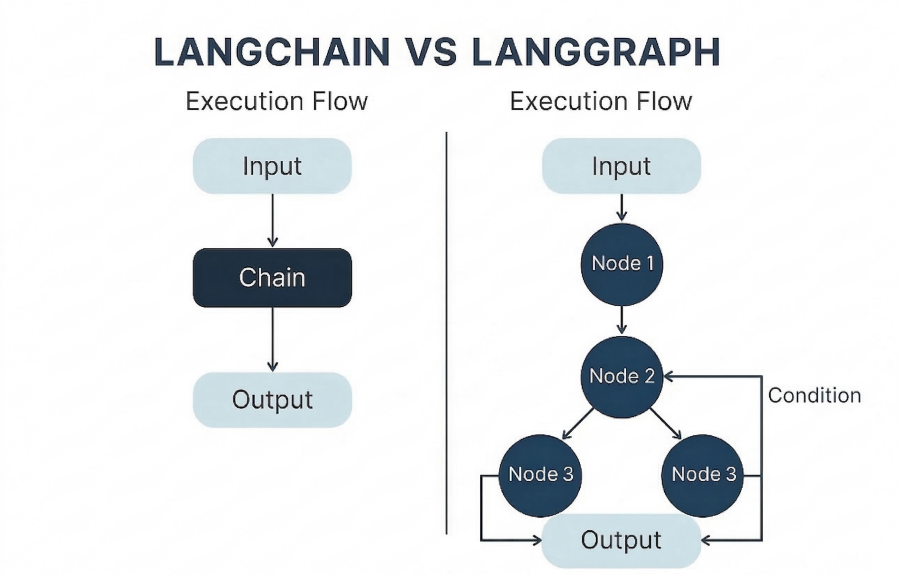

Figure 2.7: Conceptual difference between a linear chain (left) and a cyclic graph (right) in agentic workflows. The cycle allows for iterative refinement and self-correction.

2.5.3 State Management and Persistence

A defining characteristic of reliable agentic systems is the ability to maintain context over extended horizons. Modern orchestration frameworks distinguish themselves through dedicated state management mechanisms, moving away from stateless architectures where every interaction is independent [50].

In the context of this work, state persistence is achieved through a "checkpointing" system that saves a detailed snapshot of the graph's state, including variable values, conversation history, and execution progress, at every step. This architectural choice unlocks critical capabilities for MLOps:

1. **Fault Tolerance and Recovery**: If a long-running process (like a firmware compilation) fails due to transient errors, the system can resume execution from the last valid checkpoint rather than restarting from scratch.

2. **Human-in-the-loop (HITL)**: The execution can be paused to allow human experts to inspect the state, approve sensitive actions, or manually correct the agent's trajectory before resuming.

3. **Time-Travel Debugging**: Developers can "rewind" the agent's actions to virtually any previous state to diagnose logic errors or test alternative execution paths [51].

2.6 Information Retrieval and Retrieval-Augmented Generation

Information Retrieval (IR) is essential for systems needing domain-specific knowledge outside their training data. Traditional IR relies on lexical techniques like BM25, ranking documents by keyword frequency. However, these methods often miss the semantic intent of technical queries [52]. Specs for STM32 Edge AI are scattered across datasheets, forums, and technical manuals, making retrieval by meaning a fundamental requirement. Modern automation efforts increasingly rely on semantic search to navigate these complex API constraints and produce context-aware code [53, 54].

2.6.1 Semantic Search and Vector Embeddings

Keyword-based search systems look for exact string matches between a query and a document. For instance, if a developer searches for "memory optimization", a purely lexical engine misses a paragraph about "reducing RAM footprint" because the words do not match. *Semantic* search addresses this limit by processing the underlying meaning of the text.

Embeddings drive this semantic understanding. An embedding is a mathematical vector (a list of hundreds or thousands of floating-point numbers) mapping textual concepts into a high-dimensional space. Texts with similar meanings are placed closer together, regardless of their specific vocabulary. Specialized *Embedding Models*, like Sentence Transformers, generate these vectors by encoding text into fixed-length arrays that preserve semantic relationships [55].

Systems use *Vector Databases* (such as Chroma or FAISS) to make this space searchable. Unlike relational databases, vector databases index and store embeddings natively. When a user submits a query, the system first converts it into an embedding vector. The database then performs a "Nearest Neighbor" search, calculating spatial distances (e.g., cosine similarity) to retrieve the relevant document chunks. This permits the agent to access hardware constraints in milliseconds and synthesize an answer [56].

2.6.2 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) addresses two practical limitations of static model weights: the knowledge cutoff, which prevents the model from reasoning about documentation released after training, and the tendency to hallucinate plausible but incorrect facts. Augmenting each inference call with retrieved text gives the LLM access to current, domain-specific evidence without the need for periodic retraining [57].

The workflow of a standard RAG system, spanning from document indexing to the final response, is depicted in Fig. 2.8. This pipeline typically follows three primary stages:

1. **Indexing:** Technical documents are ingested, cleaned (e.g., removing HTML tags), and segmented into manageable "chunks." These chunks are then embedded and indexed. The granularity of chunking is critical to balance context preservation with retrieval precision.

2. **Retrieval:** Upon receiving a user query (e.g., "How to map float32 to INT8 on STM32H7?"), the system performs a similarity search against the index to retrieve the top-k most relevant document segments.

3. **Generation:** The retrieved context is injected into the LLM's prompt via a structured template. The model then synthesizes a final answer, conditioned on both its internal parametric knowledge and the non-parametric external evidence [58].

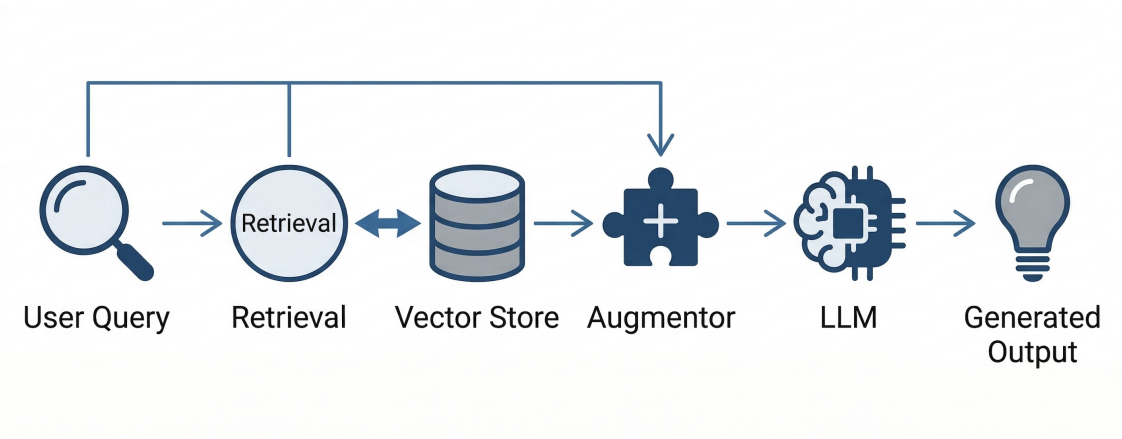

Figure 2.8: Operation cycle of a Retrieval-Augmented Generation system: from Indexing to Generation [59].

2.6.3 Web Research and Hardware Knowledge Grounding

Beyond static documents, modern retrieval frameworks integrate an active "Web Research" component. The agent can perform live lookups for real-time data, such as software patches, by utilizing search APIs (e.g., Google Search). This dynamic grounding is necessary for making operational decisions that remain valid over time.

2.6.4 Challenges and Bias Mitigation

Despite its benefits, RAG introduces specific failure modes. "Retriever failure" occurs if the embeddings fail to align the query with the correct document, usually due to a domain mismatch, like using general English models on technical datasheets. "Generator failure" happens when the LLM prioritizes its training data over the context, leading to *hallucination despite retrieval* [54]. To fix this, systems often use "Self-Correction" loops, where the agent checks its output against the source documentation before finalizing the result.

2.7 Knowledge Validation and Code Verification

The deployment of Large Language Models in critical engineering workflows, such as generating firmware or optimizing neural networks, introduces non-trivial risks. LLMs are probabilistic by nature, and while they can generate syntactically correct code, they do not inherently guarantee functional correctness or adherence to strict hardware constraints. To mitigate these risks, a reliable agentic framework must move beyond simple generation and incorporate layers of rigorous validation [53].

2.7.1 Evaluation of LLM-Generated Code

Assessing the quality of AI-generated code, particularly for embedded systems, requires evaluating logic and structural stability:

- **Functional Correctness:** Is the code doing what was asked? This is measured through execution-based validation, where the generated script is run against a test case. If it crashes or produces the wrong output, it is marked as incorrect, triggering a retry loop.
- **Execution Safety:** Unlike standard software, embedded AI code interacts with hardware resources. Ensuring that the generated code does not request invalid memory allocations or incompatible library versions is a key part of the verification process.

For the "knowledge" aspect, specifically the answers retrieved via RAG, verification protocols like the "LLM-as-a-Judge" paradigm are employed. *DeepEval* serves

as an automated judge that scores the system's reasoning on quantitative scales [60]. The interaction between these metrics, known as the "RAG Triad," is illustrated in Fig. 2.9:

- **Faithfulness (or Groundedness):** Ensures the generated answer is strictly derived from the retrieved documentation, minimizing hallucinations. This metric checks if every claim in the response can be inferred from the source context.
- **Contextual Relevancy:** Measures whether the retrieval step actually fetched useful documents or just noise.
- **Answer Relevancy:** Penalizes verbose or evasive answers that do not directly address the user's query.

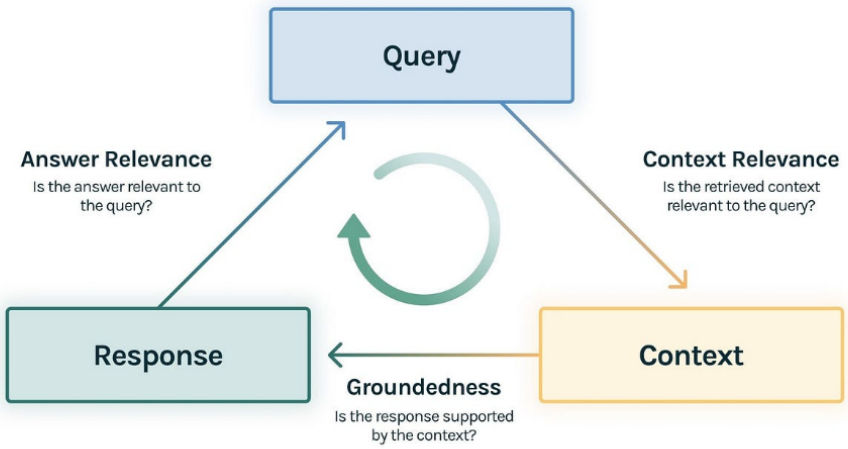

Figure 2.9: The "RAG Triad" evaluation metrics used by DeepEval: Faithfulness, Answer Relevance, and Contextual Relevance [60].

2.7.2 Sandbox Execution and Safety

Given the potential for generating unsafe or buggy code, execution must occur in a controlled environment. **Sandboxing** provides an isolated runtime where code can be executed without checking the host system's integrity [61]. In advanced agentic systems, this is often implemented as a "Safe Execution Loop". When the agent generates a script (e.g., for data preprocessing or model inspection), it is executed within a supervised subprocess. If the script fails due to syntax errors or runtime

exceptions, the sandbox traps the failure and returns the standard error (stderr) trace to the agent. This facilitates a self-healing loop where the agent reads the error, reasons about the fix, and re-generates the code, mimicking the iterative debugging process of a human developer [62].

2.7.3 Syntax and Logic Verification

Before runtime, implicit verification occurs. Since the generated code (typically Python) is interpreted, syntax errors are caught immediately by the interpreter in the sandbox. This "Syntax Check" acts as a first line of defense. Only if the code parses correctly does the logic verification (functional correctness) proceed, ensuring that resources are only consumed by valid scripts.

2.8 Human-in-the-Loop: Collaboration and Governance

The concept of Human In The Loop (HITL) defines a design paradigm where human intelligence and machine learning are integrated to optimize performance, interpretability, and safety [63]. In an agentic MLOps framework, HITL is not merely an optional feature but a core safeguard. It recognizes that while LLMs can automate the majority of the development lifecycle, human experts possess the nuanced domain knowledge required to navigate edge cases, ethical dilemmas, and complex hardware trade-offs that purely statistical models might overlook [64].

2.8.1 Patterns of Human-Agent Interaction

Iterative refinement within HITL workflows typically follows a generate-evaluate-update cycle. In agentic orchestration frameworks, this interaction is structured through several specialized patterns:

1. **Human-as-Gate:** The most common pattern for risky operations (e.g., rewriting MCU flash memory). The agent pauses execution and presents its plan for explicit approval before proceeding.

2. **Human-as-Critic:** The user provides qualitative feedback on an agent's output (e.g., "the generated C code is too verbose"). The agent then uses this critique to re-generate the artifact in the next iteration [65, 66].

3. **Human-as-Instructor:** The user resets or re-scopes the agent's task by providing new constraints or updated project requirements, effectively steering the orchestration logic.

These patterns facilitate a "collaborative intelligence" where the agent handles the heavy lifting of code generation and retrieval, while the human retains sovereign control over architectural choices, preventing the system from pursuing suboptimal or unsafe paths [47].

2.8.2 Advantages in Embedded Systems Development

Integrating humans into the MLOps loop provides significant benefits for Edge AI deployment. First, it ensures **Factual Accuracy and Safety**: humans can intercept hallucinations or unsafe compiler flags before they lead to hardware failure or invalid model weights. Second, it improves **Interpretability**: by requiring agents to explain their reasoning before a human-in-the-loop gate, the opaque decision-making of the LLM becomes transparent and auditable. Finally, it enables **Efficient Resource Allocation**: agents can automate 90% of the tedious boilerplate generation, allowing engineers to focus their attention on the 10% of complex optimization challenges that require deep expertise [67].

2.8.3 Implementation Challenges

Despite its benefits, HITL introduces increased latency and a potential reduction in overall system throughput. Designing effective "interruption points" is critical: too many interruptions lead to user fatigue and slow development cycles, while too few increase the risk of undetected errors. In addition, ensuring that human feedback is correctly interpreted and consistently applied by the agent across multiple turns remains an active area of research in agentic state management [24]. This thesis addresses these challenges by using LangGraph's checkpointing system to provide an integrated "interrupt-and-resume" experience tailored for STM32 developers.

2.9 Open-Source LLMs and Local Deployment

The democratization of large language models has been accelerated by the rise of open-source initiatives, allowing researchers and organizations to deploy high-performance models on their own infrastructure. Local deployment is particularly critical for industrial MLOps, where data privacy, latency, and operational cost are paramount concerns.

2.9.1 The Rise of Open-Source Model Families

Models such as Meta's **Llama 3** and **Mistral** have demonstrated that smaller, well-trained models can approach the performance of massive proprietary APIs on many reasoning and coding tasks [45, 32]. These models are often distributed in quantized

formats (e.g., GGUF), which compress the model weights from 16-bit floating point to 4-bit or 8-bit integers. This reduction in memory footprint allows state-of-the-art models (even those with 8B to 70B parameters) to run on consumer-grade hardware or local workstations without specialized GPU clusters.

2.9.2 Ollama: Streamlining Local Inference

Ollama is a lightweight, open-source framework designed to simplify the local deployment and management of LLMs [68]. It packages model weights, configurations, and inference engines into a single, easy-to-use interface. By utilizing the `llama.cpp` backend, Ollama provides high-performance inference through hardware acceleration (Metal on macOS, CUDA on NVIDIA, and ROCm on AMD).

Key features of Ollama integrated into this work include:

1. **Model Versioning:** Simple commands (e.g., `ollama pull mistral`) allow for quick swapping between different model variants during testing.

2. **Custom Modelfiles:** Developers can define specialized system prompts and parameters (e.g., temperature, context window) directly in a declarative Modelfile.

3. **REST API Integration:** Ollama exposes a standard local API, allowing LangChain and LangGraph to consume local models as if they were remote cloud endpoints, facilitating a "hybrid" architecture where sensitive code generation happens locally while non-sensitive research tasks can use larger cloud models.

In local deployment scenarios, tools like Ollama serve as the primary inference backend, ensuring that all model "surgery" and firmware configuration logic remains within the user's local network, thereby preserving intellectual property and reducing external dependencies.

2.10 LLM Inference Infrastructure

Running a single large language model on a developer's workstation is straightforward, while serving multiple models to concurrent users from shared GPU hardware is not. As LLM-based workflows grow from single-user experiments to team-scale deployments, the naive approach of launching one inference process per model quickly hits the limits of available GPU memory: a 7B-parameter model in 16-bit precision occupies roughly 14 GB of VRAM, leaving no room for a second model on a typical research GPU.

Inference servers address this by acting as a centralized manager: client applications send HTTP requests specifying the model they need, and the server handles

loading, eviction, batching, and GPU allocation transparently. A single GPU can then serve a portfolio of models without each client managing hardware resources directly.

2.10.1 NVIDIA Triton Inference Server

NVIDIA Triton Inference Server is an open-source serving platform designed to host multiple AI models simultaneously across CPU and GPU backends [69]. Unlike single-model deployment scripts, Triton defines a structured *model repository*: a directory tree where each subdirectory represents one model, versioned and described by a Protocol Buffer configuration file (`config.pbtxt`). This file declares the model's I/O tensor contract, the backend used for execution, and the GPU instance allocation policy, decoupling all deployment configuration from the application code.

Triton exposes three network endpoints: HTTP (port 8000), gRPC (port 8001), and metrics (port 8002). The HTTP endpoint follows the KServe v2 inference protocol and, when a compatible backend is used, also accepts OpenAI-style chat completions, making Triton a drop-in replacement for cloud LLM APIs within existing LangChain or LangGraph pipelines.

A key operational mode is `model_control_mode: EXPLICIT`, which disables automatic model loading at startup. The server boots with no models resident in VRAM; loading is triggered on demand by issuing `POST /v2/repository/models/{name}/load`, and the symmetric unload endpoint releases VRAM when the model is no longer needed. For a server hosting several large LLMs on a single GPU, this on-demand lifecycle is essential: it ensures that only the model currently needed occupies VRAM, while the others remain on disk or in system RAM.

2.10.2 vLLM and PagedAttention

vLLM [70] is a high-throughput inference engine that addresses a specific bottleneck in transformer serving: the KV cache. During autoregressive generation, each newly produced token requires access to the key and value projections of all previous tokens. Naive implementations pre-allocate a contiguous memory block for the maximum expected sequence length, which wastes a significant fraction of VRAM when actual sequences are shorter and prevents concurrent batches from sharing fragmented free memory.

vLLM replaces contiguous KV allocation with **PagedAttention**, a mechanism borrowed from virtual memory management in operating systems. The cache is divided into fixed-size pages that are allocated dynamically as the sequence grows, and different requests can share physical pages when their prefix tokens are identical (a technique called *prefix caching*). The practical outcome is higher GPU utilization and support for larger effective batch sizes within the same VRAM envelope.

When integrated as a Triton Python backend, vLLM provides these memory efficiency gains while keeping the standard Triton request interface intact. Application code sends a prompt to the Triton HTTP endpoint; vLLM handles batching, page allocation, and generation internally before returning the response.

Chapter 3

Materials and methods

3.1 Introduction and System Architecture Overview

The framework is organized as a multi-layered, cloud-native stack that connects the developer's local IDE to a centralized GPU cluster on the other end of the chain. As shown in Figure 3.1, five distinct layers handle different concerns: user interaction, request routing, agentic orchestration, state persistence, and language model inference.

Each layer has a well-defined responsibility and communicates with the adjacent ones through standardized interfaces:

1. **IDE Layer.** The developer interacts with the system through a custom **VS Code** extension that hosts the chat interface directly inside the editor. Responses are streamed asynchronously over an NDJSON connection, so the editor stays responsive even during long-running tasks such as model training or firmware compilation.

2. **API Gateway Layer.** A **FastAPI** server, running in the `langgraph-app` container, exposes a single `/stream` endpoint. It receives user messages from the extension, retrieves the user's long-term profile from **Redis**, constructs the initial graph state, and streams back structured JSON events as the graph executes. Although it shares the same container as the orchestration engine, it encapsulates a distinct concern: protocol translation between the HTTP world and the graph execution model.

3. **Orchestration Layer.** A **LangGraph** stateful graph coordinates the entire agentic pipeline. An LLM-based router classifies each incoming request and dispatches it to one of three primary subgraphs (firmware, AI analysis, or web research), while the remaining workflows are reached through inter-workflow transitions. Domain-specific tools, namely **STM32CubeMX**, **STEdgeAI**, and **Microsoft NNI**, run as managed subprocesses inside the same container,

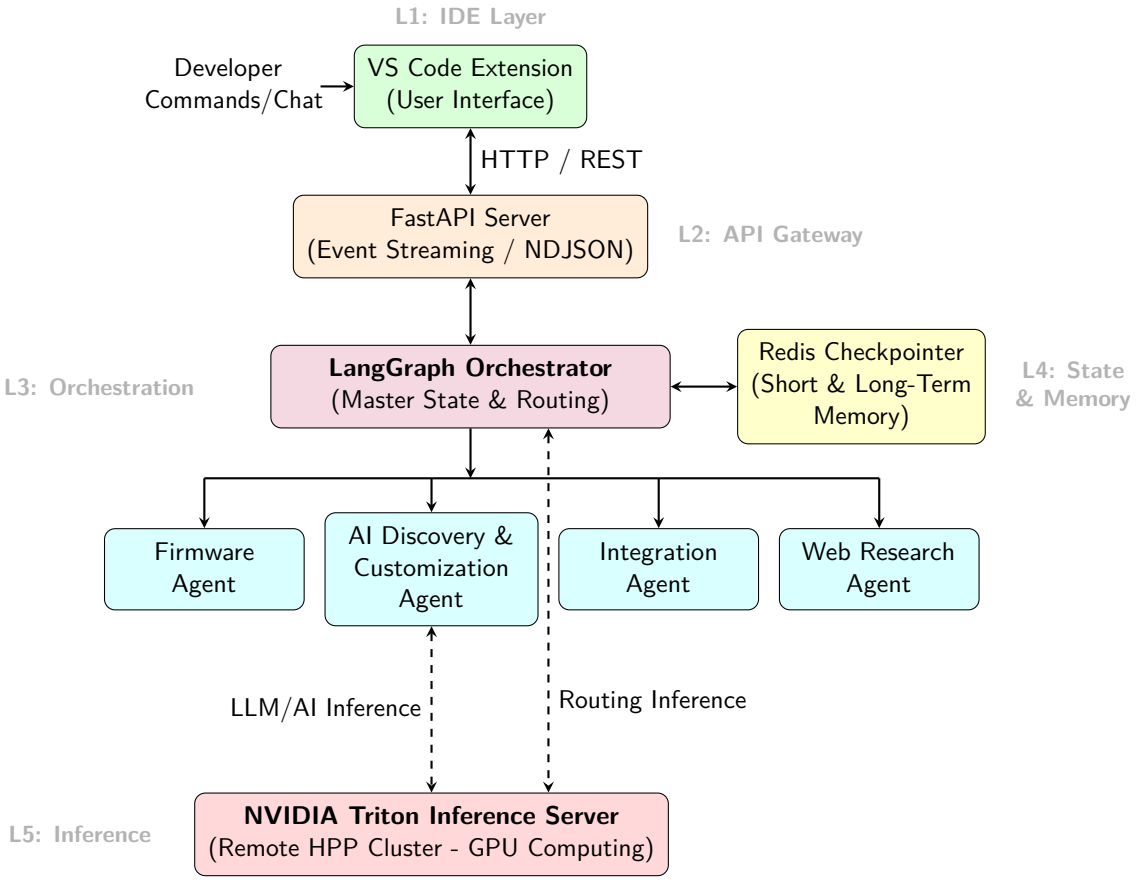

Figure 3.1: Top-Down System Architecture. The framework is organized into five operational layers, spanning from the VS Code user interface down to the centralized GPU inference backend.

invoked by the agent nodes and wrapped to normalize their CLI output into structured state updates.

4. **State & Memory Layer.** A dedicated **Redis** container serves two distinct roles. First, it acts as the LangGraph checkpointer, persisting the full `MasterState` at every node transition to support fault-tolerant resumption and Human-in-the-Loop (HITL) interrupts. Second, it stores per-user long-term profiles (preferred board, last model, project paths) that survive across sessions.

5. **Inference Layer.** All LLM calls are forwarded over the network to a centralized **NVIDIA Triton Inference Server**. Triton handles dynamic model loading and GPU memory sharing across concurrent users, keeping the local orchestration containers free of any GPU dependency.

Two auxiliary sub-workflows support the customization stage: *Synthetic Data Generation*, which produces controlled time-series and audio signals for pipeline validation, and *Dataset Selection*, which handles the download and format conversion of public datasets. Both are described in Section 3.2.6.

The **Orchestration Layer** (L3) coordinates five agentic workflows, each mapping to a specific phase of the Edge AI development lifecycle:

1. **Firmware Generation** (`firmware_flow`): Automates STM32CubeMX project initialization, package installation, and code generation via CLI scripting.

2. **AI Model Discovery and Optimization** (`ai_flow`): Implements a two-tier model discovery system (curated task-based models plus iterative web search), followed by STEdgeAI analysis, validation, and C code generation.

3. **Model Customization** (`customization_flow`): Enables layer-level neural network modification, dataset handling, and automated hyperparameter optimization with NNI.

4. **Firmware-AI Integration** (`integration_flow`): Merges generated C inference code into the STM32 project structure and verifies build compatibility.

5. **Web Research** (`search_flow`): Provides technical guidance through dynamic web search with DeepEval quality evaluation metrics.

When a new message arrives, the orchestrator routes it through an LLM-based classifier. The classifier analyzes the user's natural language input, assigns a confidence score, and selects the target workflow. If confidence falls below 0.6, the system asks the user to choose explicitly from a structured menu. Figure 3.2 illustrates this decision process.

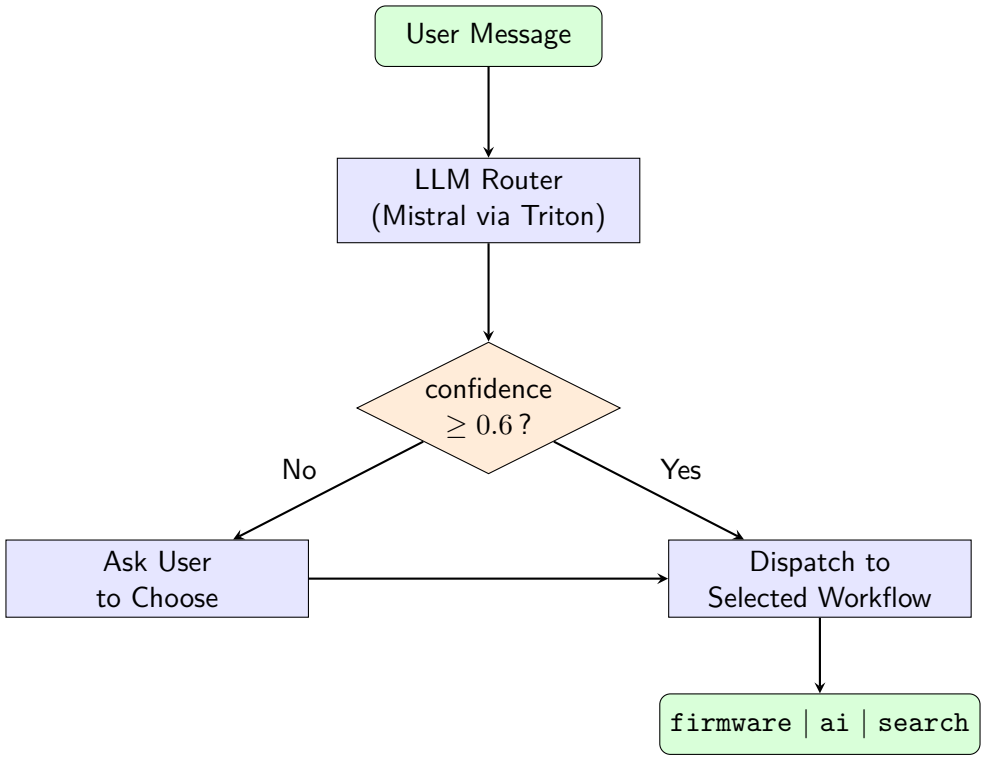

Figure 3.2: LLM-based routing logic. The classifier dispatches user requests to one of three primary workflows. Customization and Integration are reached through inter-workflow transitions.

3.2 The Core Orchestrator (LangGraph & Agentic Workflows)

The orchestration logic splits into several interrelated agentic workflows. They handle firmware generation, AI model discovery, customization, and deployment. We map the complete execution path across three detailed flowcharts: Figure 3.3 covers system entry, initial routing, the firmware generation workflow and the first part of the AI model workflow, Figure 3.4 details model customization, and Figure 3.5 breaks down STEdgeAI Analysis, integration, and web research.

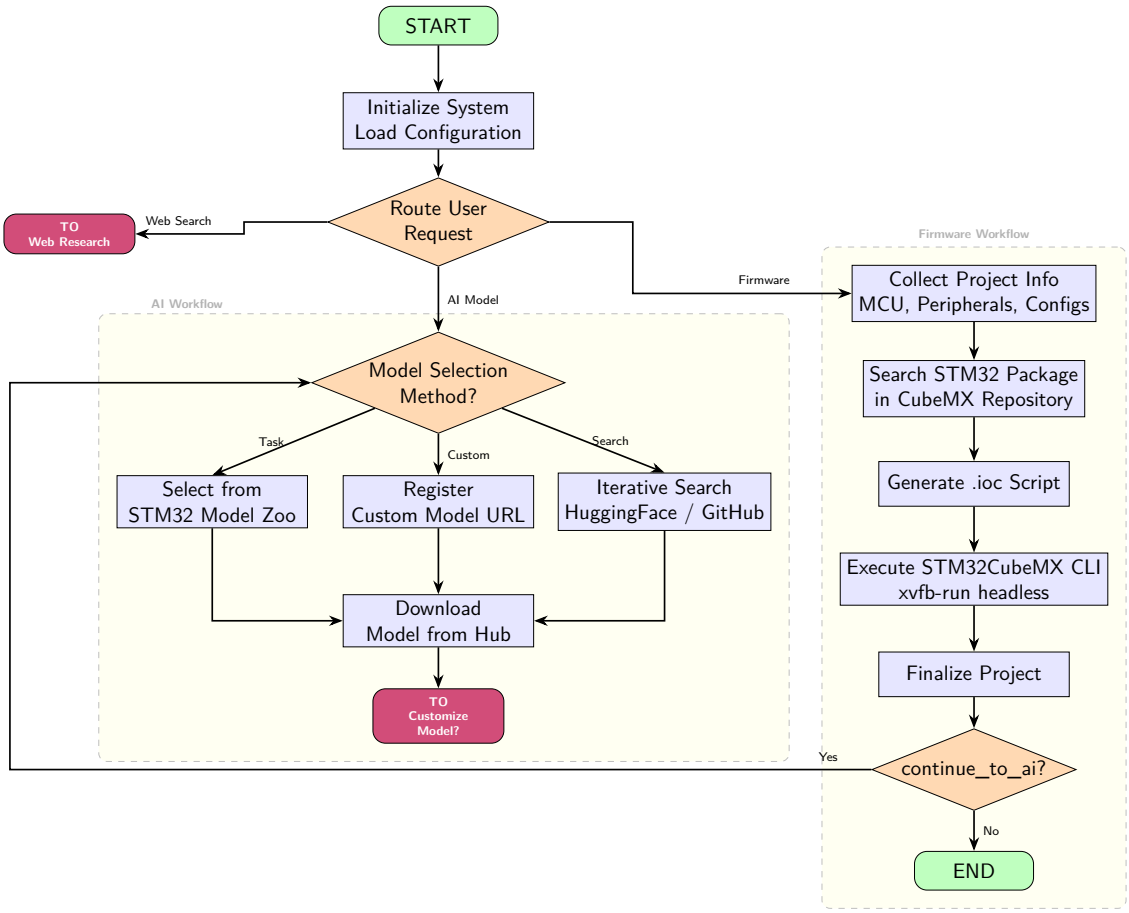

Figure 3.3: Detailed Workflow Diagram - System initialization, request routing, and entry workflows (Firmware Generation and AI Model Discovery). Web Search requests exit directly to Figure 3.5.

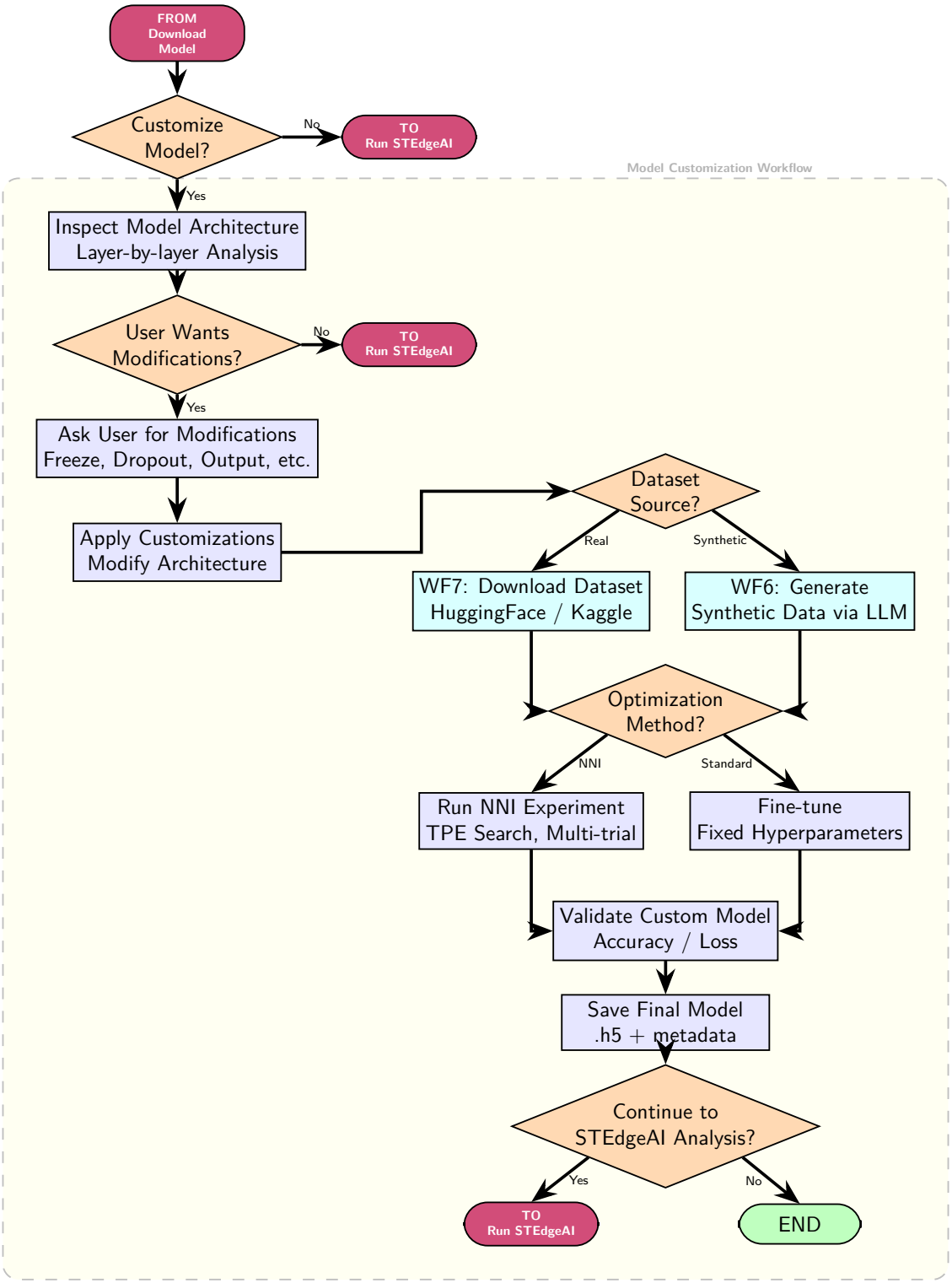

Figure 3.4: Detailed Workflow Diagram - *Model Customization* workflow. Left column: architecture inspection and modification. Right column: dataset acquisition, hyperparameter optimization via NNI, and model validation.

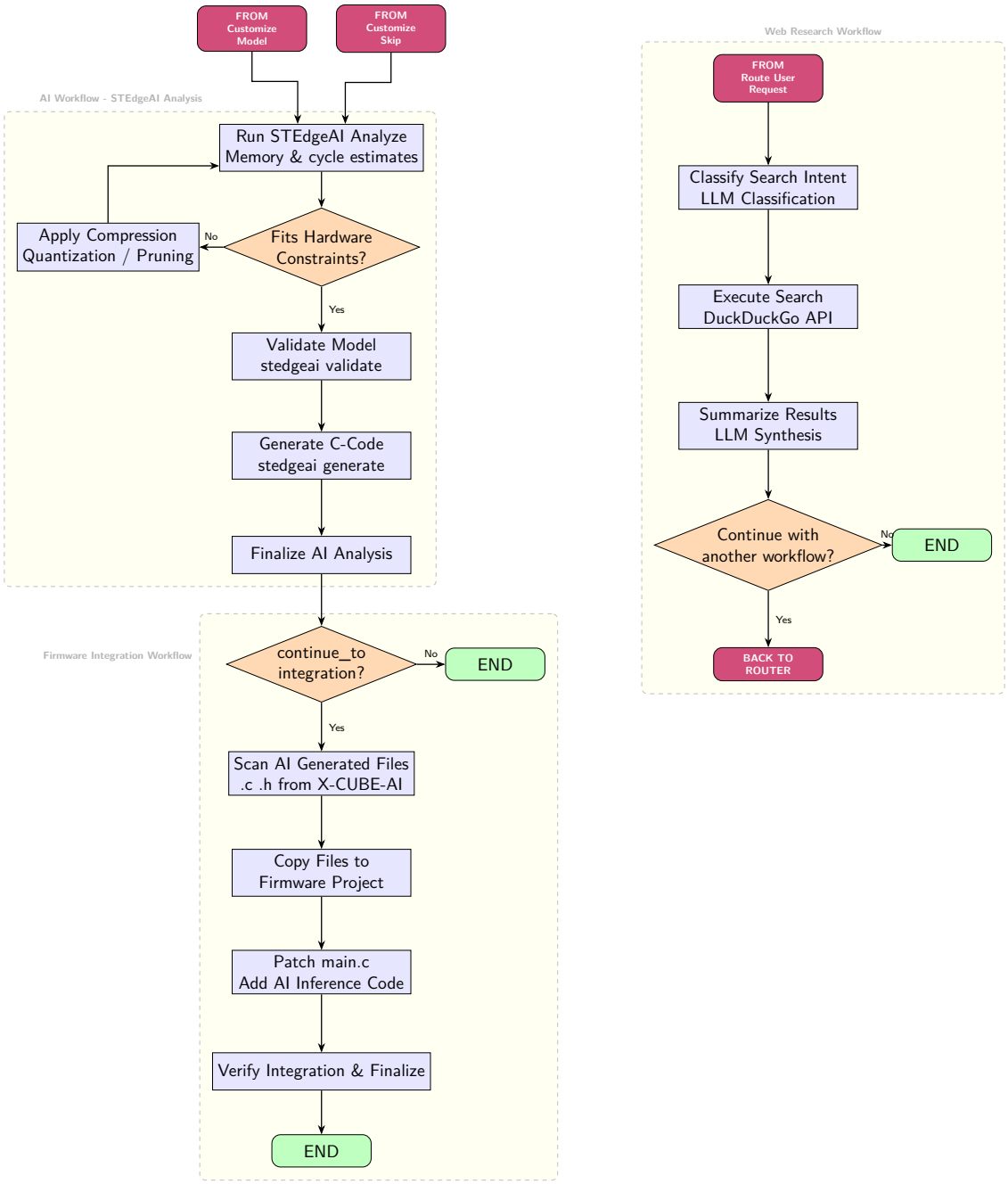

Figure 3.5: Detailed Workflow Diagram - *AI Model Discovery* (STEdgeAI profiling and C-code generation), *Firmware-AI Integration*, and on-demand *Web Research*.

3.2.1 Firmware Generation Workflow

The *Firmware Generation* workflow automates the initialization of STM32CubeMX projects through CLI-driven scripting, removing the need for manual GUI interaction and producing a reproducible firmware project structure.

Project Information Collection

The `collect_project_info()` node extracts hardware specifications from the user's natural language input. Parsing is delegated to `get_llm()`, a centralized utility in `utils.py` that routes inference requests to the NVIDIA Triton Inference Server when the `USE_TRITON_BACKEND` environment variable is set, and falls back to a local Ollama instance otherwise. The utility is configured with the `ProjectInfoExtraction` Pydantic schema for structured output:

Listing 3.1: Firmware project info extraction

```
# User: "Create a project for STM32F401 with STM32CubeIDE, name MyApp"

llm = get_llm(config)
llm_extractor = llm.with_structured_output(ProjectInfoExtraction)
result = llm_extractor.invoke([
    SystemMessage(content=project_info_extraction_instructions),
    HumanMessage(content=f"Messaggio: {state.message}")
])

# Parsed: {ioc_file_path: null, board_name: "STM32F401", mcu_series: "F4
    ",
#         project_name: "MyApp", toolchain: "STM32CubeIDE",
#         peripheral_config: []}
```

The schema includes a `peripheral_config` field containing CubeMX commands for pins and peripherals. When the user mentions specific hardware in their prompt, for example "activate pin PA5 as output" or "use timer 1", the LLM converts these into CubeMX script directives (`set_pin PA5 GPIO_Output`, `set_peripheral TIM1`) that are injected directly into the generation script. Alternatively, users can specify an existing `.ioc` file path, bypassing project creation entirely.

The node also handles two edge cases before issuing the interrupt. If `board_name` is already set in the graph state from a previous session, the node skips the interrupt and reuses the existing value. If the user responds with a phrase such as "use the one from yesterday" or "same as before", the LLM maps the input to the token `USE_PROFILE`, which triggers a lookup of the last used board and MCU series from the Redis persistent context.

Automated MCU Package Management

The `search_and_install_stm32_package()` function handles the download of STM32Cube firmware packages from GitHub. Without this step, the CubeMX CLI fails silently with a "Project loading error" when a required MCU package is absent from the local repository. This is a practical difference from the GUI, which offers an interactive download prompt.

The node first checks whether a package folder for the detected MCU series already exists inside `~/STM32Cube/Repository/`. If present, the step is skipped. Otherwise, the function uses `git ls-remote -tags` to read available version tags without cloning the full repository, selects the latest semantic version, and performs a shallow recursive clone:

```
git clone --recursive --branch v1.28.1 --depth 1 \
    https://github.com/STMicroelectronics/STM32CubeF4.git \
    /tmp/STM32CubeF4_<timestamp>
```

After cloning, it verifies that `Drivers/`, `Middlewares/`, and `Projects/` are present, reads the version from `Release_Notes.html`, and moves the repository to the final path under `STM32Cube/Repository/`. If the clone fails or times out, the error is stored in `state.package_error_message` and the routing node short-circuits to `finalize_project`, skipping script generation.

Headless Generation and CLI Scripting

STM32CubeMX supports headless operation via the `-q` flag. The `generate_cubemx_script()` node builds a `.scr` file in three steps.

First, it searches for a usable IOC file in priority order: a user-supplied path, a pre-validated template matching the exact board name, and finally a series-level template (e.g., `STM32H7A3ZITx.ioc` for any H7 device). A matched template is loaded with `config load`, which avoids the interactive recommendation dialogs. Only when no template is available does the script fall back to `load <MCU>`, risking GUI pop-ups.

Second, the project name, toolchain, and output path are appended. Third, any `peripheral_config` entries are added, after filtering out directives that do not start with a recognized prefix (`set_pin`, `set_peripheral`, `config`). The complete script is written to `/tmp/script_<timestamp>.scr`.

The `execute_generation()` node then invokes CubeMX with a command adapted to the host OS. On macOS, CubeMX is called directly. On Linux, the node checks at runtime for `xvfb-run` and uses it if found to provide a virtual framebuffer, dropping back to a direct call otherwise:

```
# macOS
```

```
<cubemx_path> -q <script_file.scr>

# Linux (with xvfb-run if available)
xvfb-run -a <cubemx_path> -q <script_file.scr>
```

Note on headless licensing: CubeMX CLI requires a one-time GUI acceptance of the license agreement before it can run headlessly. If this step is skipped, the process hangs indefinitely. The workflow assumes the license has already been accepted on the host.

Two-Stage Generation with Temporary Directory

CubeMX writes its output to a local temporary directory at `/tmp/stm32_<timestamp>/` rather than to the final project destination. The `project path` directive in the script is patched dynamically just before execution to point to this interim path. This avoids filesystem locking on NFS-mounted output directories.

After the process exits with return code 0, the node polls the temporary directory up to 10 times at one-second intervals, waiting for `Src/` and `Inc/` to appear. Once confirmed, the project tree is moved to the final path with `shutil.move()`. If the process is interrupted mid-generation, the output folder on the target filesystem remains clean.

Overcoming GUI Dependencies: The Pre-seeded IOC Strategy

For known chip families, the firmware workflow uses a **pre-seeded IOC strategy** to avoid the interactive pop-ups described above. A set of pre-validated `.ioc` templates, one per supported MCU family (e.g., `STM32H7A3ZITx.ioc`, `STM32U585AIIxQ.ioc`), is maintained in the `templates/ioc_files/` directory. Each file is configured manually once in the GUI to answer all the recommendation wizards, including MPU regions and TrustZone context.

At runtime, `get_template_ioc_path()` tries an exact board-name match first, then a series-level match via the `SERIES_MAP` dictionary. The chosen template is loaded with `config load "template.ioc"`, and the `project name` and `project path` directives re-target it to the user's directory, producing a fresh project that inherits the silent configuration.

Error Recovery and Workflow Handoff

When the first CubeMX attempt returns a non-zero exit code or times out, `recover_with_ioc_fallback()` takes over. It selects the best matching IOC template for the detected board or series, rewrites the `.scr` script to use `config load` instead of the generic board directive, and relaunches CubeMX with a 600-second timeout. The two-stage temporary directory strategy described in Section 3.2.1 applies unchanged

to this retry. On completion, the output tree contains `main.c`, `stm32xxxx_hal_msp.c`, the linker script, and the IDE project files.

Once generation succeeds, `finalize_project()` stores the firmware project path in the graph state and yields control back to the orchestrator. The user is asked whether to proceed to AI model selection: confirming routes to the *AI Model Discovery* workflow (Section 3.2.2), while declining closes the session.

3.2.2 AI Model Discovery and Validation Workflow

The *AI Model Discovery* workflow handles the selection, profiling, and C code generation of pre-trained neural networks for STM32 microcontrollers (see Figure 3.3 for the entry routing logic). Embedded AI differs from conventional MLOps in one fundamental respect: raw accuracy is only one metric among several. RAM, Flash, and MACC budgets set hard physical limits that constrain every design choice. The workflow addresses this by combining a curated local registry with iterative web search and delegating quantization decisions to the STEdgeAI toolchain.

Analysis Parameters Collection

`collect_analysis_info()` extracts the target MCU and the desired compression level from the user's message. The node scans the initial prompt for known MCU series tokens (e.g., `h7`, `u5`, `f4`). If nothing is found, it issues an interrupt and surfaces the last used series from the persistent context as a suggestion. When the *Firmware Generation* workflow (Section 3.2.1) has already set `state.target` from the selected board, the interrupt is skipped entirely. Inference is delegated to `get_llm()`, which routes requests to the NVIDIA Triton Inference Server when `USE_TRITON_BACKEND` is set, falling back to a local Ollama instance otherwise:

Listing 3.2: Parameter extraction from user input

```
# User: "I need a model for image classification on STM32H7 with high
    compression"

llm_extractor = get_llm(
    config=config,
    structured_schema=AnalysisInfoExtraction,
    temperature=0
)
result = llm_extractor.invoke([
    SystemMessage(content=analysis_info_extraction_instructions),
    HumanMessage(content=f"Risposta utente: {user_text}")
])

# result.target = "stm32h743"
```

```
# result.compression = "high"
```

Both values are stored in `MasterState` and reused across model selection, profiling, and code generation without being re-parsed.

Three-Tier Model Discovery Architecture

Model discovery follows a three-tier hierarchy, tried in order until the user confirms a candidate.

Tier 1 - Curated Task-Based Models. The first tier runs across two nodes. `choose_ai_task()` presents all task categories defined in `predefined_models.json`, a JSON registry decoupled from the discovery logic so it can be updated without touching any source code. The menu covers standard embedded tasks (image classification, object detection, human activity recognition) and exposes two shortcuts: "Register your own model" and "Online search". An LLM classifier maps the user's free-text response to the appropriate category key.

`choose_ai_model()` then handles selection within the chosen category. Before displaying the list, it computes a hardware compatibility estimate for each entry by comparing the stated file size against the flash limit returned by `get_mcu_limits(state.target)`. Models fitting natively are marked **Fits** (✓), models that can be compressed to fit are labeled **Compressible** with the size ratio shown, and models too large even at maximum compression are flagged **Too Large** (×). This information is surfaced before any download or conversion is attempted.

Tier 2 - Iterative Web Search with Hybrid Retrieval. When no curated model satisfies the request, the system enters `search_recommendation_model()`, a loop capped at three iterations. Each iteration tries two backends in sequence. The primary backend scans the `STMicroelectronics/stm32ai-modelzoo` GitHub repository via the PyGitHub API, recursively traversing the task-specific subfolder (depth limit: 5, early exit at 20 files) and collecting files with extensions `.h5`, `.keras`, `.onnx`, or `.tflite`. An LLM then scores each candidate on task relevance, architecture recognition, and file size; the top result is validated with a lightweight HTTP HEAD check before being shown to the user. The secondary backend is a DuckDuckGo-based web search, tried only if GitHub yields nothing. It does not consume an iteration, so a failed GitHub scan followed by a successful web result leaves the full three-iteration budget intact.

After either backend finds a candidate, the node issues a LangGraph interrupt for explicit user approval:

Listing 3.3: User confirmation interrupt after each candidate

```
user_confirmation = interrupt({
```

```
    "instruction": f"""Model found: {model_name} ({ext}, {size})
URL: {url_raw}

Accept this model? (yes/no)
- 'yes': Proceed to download
- 'no': Continue searching"""
})
```

A rejection resets `state.model_discovery_method` to `"search"` and the routing node re-enters the loop. After three failed iterations, the system selects a task-based fallback from the local registry and presents it with one final confirmation interrupt before starting the download.

Tier 3 - User Model Registration. Users can register their own local or remote models through `add_custom_model_procedure()`, providing a category, a name, and a direct download URL. The entry is written to `predefined_models.json` only after `stedgeai analyze` confirms the model is structurally valid for the target runtime.

Model Download and Inspection

`download_model_to_cache()` saves the selected model to `~/.stm32_ai_models/`. A cached copy, if present, is reused without re-downloading. For fresh downloads, data streams in 8 KB chunks with progress logged every 20%. The file size is checked after saving, and if the model entry carries an optional `sha256` field, the hash is verified before proceeding.

Architecture inspection follows a two-level subprocess strategy. The model is loaded inside an isolated subprocess in the correct conda environment (`stm32` for `.keras`, `stm32legacy` for `.h5`). The inspection script is generated inline as a Python string and executed via `execute_in_environment()`; results are returned through a `JSON_START...JSON_END` marker in stdout:

Listing 3.4: Inline subprocess script for model inspection

```
script = f"""
import tensorflow as tf, json, sys

try:
    model = tf.keras.models.load_model(r'{model_path}', compile=False)
    info = {{
        'input_shape': str(model.input_shape),
        'n_layers': len(model.layers),
        'total_params': int(model.count_params()),
        'has_batchnorm': any(['BatchNormalization' in
```

```
            l.__class__.__name__ for l in model.layers]),
    }}
    print("JSON_START" + json.dumps(info) + "JSON_END")
except Exception as e:
    print(f"ERROR:{{e}}")
    sys.exit(1)
"""
res = execute_in_environment(script, python_path, timeout=30)
```

If the subprocess fails, a second pass opens the file with `h5py` and reads the raw HDF5 metadata without loading any weights. The extracted metadata is stored in `state.model_architecture` for use by the *Model Customization* workflow (Section 3.2.3).

STEdgeAI Three-Phase Profiling and Code Generation

With a model in hand, the workflow runs a sequential three-phase cycle using the STEdgeAI toolchain. An optional fine-tuning and customization step may precede this phase; it is described separately in Section 3.2.3.

Phase 1 - Model Analysis. `run_analyze()` calls the STEdgeAI analysis engine to estimate resource consumption on the target MCU:

```
stedgeai analyze --model <model.h5> --target <stm32h743>
                 --output ./analisiAI/report_analyze
```

The resulting report covers RAM usage (tensor buffers, activation memory, working memory), Flash usage after quantization, MACC count per inference (the standard metric for neural network computational cost), and estimated latency at typical clock speeds. If RAM or Flash limits are exceeded, the node automatically increments the quantization aggressiveness from `medium` to `high` and repeats the command, continuing until the model fits or the maximum compression level is reached.

Phase 2 - On-Target Validation. `run_validate()` produces a validation report for the target architecture:

```
stedgeai validate --model <model.h5> --target <stm32h743>
                  --output ./analisiAI/network_validate_report.txt
```

The report confirms model integrity, that activation buffers fit within target RAM, and that all operators are supported on the specific MCU series.

Phase 3 - C Code Generation. `run_generate()` produces the final inference artifacts:

```
stedgeai generate --model <model.h5> --target <stm32h743>
                  --compression <high> --output ./analisiAI/code_resnet
```

Output includes `network.c/h` (fixed-point neural network implementation), `network_data.c` (quantized weights as const arrays), and `network_config.h` (MCU-specific memory layout and hardware acceleration flags). Quantization is applied here via `-compression`, not during training, so the framework can evaluate models in full-precision FP32 and defer bit-width decisions until the exact MCU constraints are known from Phase 1.

Legacy Model Handling with Dual-Environment Strategy

Modern Keras 3.x, required by LangGraph and NNI, is incompatible with the Keras 2.x models widespread in STModelZoo and academic repositories. Rather than forcing a single environment, the framework maintains two separate conda installations:

- **Primary environment** (`stm32`): hosts LLM agents, LangGraph, and NNI with the latest dependencies
- **Legacy environment** (`stm32legacy`): pinned to TensorFlow 2.15 and Keras 2.15 for inspection and fine-tuning of older model files

At download time, `download_model_to_cache()` reads the file extension and routes accordingly: `.keras` files go to `stm32`, while `.h5` files are directed to `stm32legacy` via the `ARCH_ENVIRONMENT_MAP` dictionary, which also maps specific architectures (e.g., YOLO variants) to their required environment. The chosen interpreter path is resolved from the system configuration and passed to `execute_in_environment()`, keeping the main runtime isolated from TensorFlow dependency conflicts.

3.2.3 Neural Network Customization Workflow

Once a base model has been discovered and profiled, the user may need to adapt its architecture to meet specific hardware constraints, retarget its output to a new set of classes, or improve its accuracy on a domain-specific dataset. The *Model Customization* workflow implements a customization pipeline that breaks this process into five stages: model inspection, best practice retrieval, interactive modification planning, architecture patching, and fine-tuning (detailed in Figure 3.4). The implementation is contained in `workflow5_customization.py` and runs as a LangGraph graph with conditional routing between nodes.

Model Architecture Inspection

The workflow begins with the `inspect_model_architecture()` node, which analyses the downloaded model file and populates the graph state with the architectural metadata needed by all subsequent steps. The extracted data includes layer types (Conv2D, Dense, Dropout), output shapes, the total parameter count, and whether the model contains BatchNormalization or Dropout layers. This profile feeds directly into the best practice retrieval and modification planning steps.

Model loading is not a trivial operation: the same file can fail with a standard `load_model()` call if it was saved with a different Keras version or if it contains custom objects. For this reason, the node implements a three-level fallback chain. In the first attempt, it calls `tf.keras.models.load_model()` with `compile=False` and reads the architecture directly from the live model object. If that fails on a `.h5` or `.keras` file, the node switches to a raw HDF5 inspection: it opens the file with `h5py` and parses the `model_config` attribute to recover the layer list and names without loading any weights. If both attempts fail, the node fills the state with a minimal default dictionary (all counts set to zero, format set to `unknown`) so that the pipeline can still continue and present the user with a fallback interface.

Files in `.onnx` or `.tflite` format are handled separately. Since these formats do not expose a Keras layer graph, the node sets a generic metadata block and sets the format field accordingly. The user is informed that structural customization (layer editing and fine-tuning) is not available for these formats; the workflow then routes them directly to the STEdgeAI analysis step.

After inspection, the results are also checked against the idempotency flag: if the architecture data is already present from a previous *AI Model Discovery* run (Section 3.2.2), the node skips the loading step entirely.

Best Practice Retrieval

Before asking the user to specify modifications, the system retrieves a tailored set of fine-tuning recommendations for the detected architecture. The `retrieve_best_practices_for_architecture()` function follows a fallback strategy, moving to the next step only if the previous one fails.

First, it attempts to fetch recent architectural best practices dynamically via web search (e.g., searching for "MobileNetV2 fine-tuning best practices STM32"). If the search is successful, an LLM synthesizes the retrieved documents into a concise guide. This ensures the workflow has access to the latest community knowledge without requiring a pre-populated offline vector database.

If the web search fails or returns useless results, the system invokes the local Ollama LLM (Mistral) to generate a concise best practice guide from its internal weights. The prompt is deliberately structured and constrained, requesting a bullet-point output with four specific sections (strategy, hyperparameters, quantization, and memory constraints), and keeps the response under 200 tokens:

Listing 3.5: LLM-based best practice generation fallback

```
llm = get_llm(None, temperature=0.3, keep_alive="5m")
prompt = f"""You are an expert embedded AI engineer.
Provide a concise, bullet-point checklist for fine-tuning
{model_name} ({arch_type}) on STM32.
Keep it under 200 words. Schematic only."""

response = llm.invoke(prompt)
```

If even the LLM call fails (for example, due to an Ollama timeout), the function falls back to a set of hardcoded, architecture-specific guidance texts compiled into the module. Each entry covers the typical layer-freezing ratio, recommended learning rate range, quantization considerations for STM32, and any known constraints specific to that architecture. For example, the VGG entry warns explicitly that the architecture is memory-heavy and may not fit many STM32 targets even after INT8 quantization.

Modification Intent Classification

The `ask_modification_intent()` node determines, without mandatory user input, whether the current request requires any architectural changes. It first analyses the initial message sent to start the workflow. If the text contains explicit modification keywords (e.g., “freeze”, “dropout”, “change input”) with a confidence above 90%, the node sets `wants_model_modifications = True` and skips the interrupt. Generic triggers like “analyze” or “start ai” are treated as negative signals regardless of confidence, since those phrases imply the user wants to proceed without changes.

When the intent is ambiguous or entirely absent, the node pauses with an LangGraph interrupt and presents a structured prompt:

Listing 3.6: Modification intent classification schema

```
class ModificationDecision(BaseModel):
    wants_modifications: bool
    reasoning: str
    confidence: float = Field(ge=0.0, le=1.0)

llm_classifier = get_llm(config).with_structured_output(
    ModificationDecision)
decision = llm_classifier.invoke([
    SystemMessage(content=modification_decision_instructions),
    HumanMessage(content=f"User response: {user_text}")
])
```

If the user replies negatively, the graph routes to the STEdgeAI analysis step directly, bypassing all customization nodes. The persistent context is also checked: if

the user ran a customization in the previous session, the prompt displays a reminder suggesting they might want to do so again.

Modification Parsing and Validation

When modifications are requested, the `ask_and_parse_user_modifications()` node collects the user's natural language description and converts it into a structured plan using the `ParsedModificationsPlan` Pydantic schema. The schema captures a list of individual modifications, an overall confidence score, training recommendations (learning rate, batch size, epochs, optimizer), and a validation sub-object listing any detected issues.

The node supports seven distinct modification types:

- **freeze_layers**: Freezes the first N layers of the model, preserving pre-trained features.
- **freeze_almost_all**: Leaves only the last N layers trainable, useful for fast adaptation.
- **change_output_layer**: Replaces the final Dense layer to match a new number of target classes.
- **add_dropout**: Inserts a Dropout layer before the output, with a user-specified rate.
- **change_input_shape**: Rebuilds the model with a different input resolution by patching the model config via `Model.from_config()`.
- **change_learning_rate**: Sets a custom learning rate, stored in the training recommendation field.
- **add_resizing_layer**: Wraps the existing model in a new input layer that accepts images of any resolution and resizes them automatically to the model's original input shape. This is the only valid option for detection models (YOLO, SSD) where changing the input shape directly would break the output grid structure.

The LLM extractor is called with strict instructions to parse only what the user explicitly asked for. A second validation pass then checks each modification for parameter completeness and value ranges (for example, dropout rates must be between 0.0 and 1.0, and the layer count must not exceed the total layers in the model). The `change_input_shape` type is also cross-checked against a blocklist of model families for which it is unsupported:

Listing 3.7: Input shape compatibility guard

```
INCOMPATIBLE_INPUT_SHAPE_MODELS = {
    'yolo': ['tiny_yolo_v2', 'yolov2', 'yolov3', 'yolov4'],
    'ssd': ['ssd_mobilenet', 'st_ssd_mobilenet_v1'],
    'time_series_models': ['gmp', 'har', 'activity_recognition'],
}
```

If a blocked modification is detected, it is removed from the plan, a warning is added to the validation issues list, and the user is advised to use `add_resizing_layer` instead.

Preview and Confirmation

Before any file is modified, the `collect_modification_confirmation()` node displays a structured preview to the user: a summary of all planned modifications, the overall confidence of the parsing, the training recommendations, and any validation warnings. The user can respond in one of three ways: confirm the plan, reject it entirely, or request changes. The response is interpreted by an LLM agent, with a keyword-based fallback parser in case the LLM call fails.

If the user asks to revise the plan, the graph sets `user_wants_to_edit = True` and routes back to the modification parsing node, allowing a full re-specification without restarting the workflow.

Architecture Patching

Once the plan is confirmed, the `apply_user_customization()` node patches the model architecture in a subprocess. The patching itself is divided into two phases. Historically, `.h5` formats forced the system to invoke legacy TensorFlow environments. The current implementation standardizes on Keras 3. It injects a native `import keras` fallback mechanism directly into the generated scripts. This allows the framework to smoothly parse and convert older models into TFLite format without requiring separate legacy conda environments, greatly simplifying the deployment architecture.

Non-reconstructive modifications, that is layer freezing and learning rate changes, are applied directly to the model object in memory since they do not alter the graph topology. Reconstructive modifications, those that change the layer graph (output size, dropout injection, input shape, resizing wrapper), are queued and then applied in a specific order to preserve skip connections:

1. **Output layer change**: The model config dict is patched to update the units count of the last Dense layer, and the model is rebuilt with `Model.from_config()`. Weights from all layers except the last Dense are copied to the new model.
2. **Dropout insertion**: A Dropout layer is inserted between the penultimate layer and the output layer using the functional API.
3. **Input shape change**: The `batch_input_shape` field of the first layer's config is patched, and the entire model is rebuilt. Weights are copied where shapes allow; layers with a shape mismatch are re-initialized.
4. **Resizing wrapper**: A new `Input` layer accepting `(None, None, C)` is created, a `Resizing` layer is appended to scale images to the model's original resolution, and the existing model is called on the resized output.

The entire patching script runs in a subprocess via `execute_in_environment()`, which writes the script to a temporary file, runs it with the selected Python binary, and streams output line by line. A `SUCCESS:` marker in stdout signals that the modified model was saved to `/tmp/customized_model.h5` along with a JSON-encoded info block.

Standard Fine-Tuning Path

The `fine_tune_customized_model()` node fine-tunes the patched model on the user's dataset inside another subprocess (using the `stm32` conda environment). The training configuration is taken from the `training_recommendation` field of the parsed modification plan, with two safety caps applied: the learning rate is clipped at `1e-3` to avoid destabilizing pre-trained weights, and the epoch count is capped at 10 for standard runs.

The node supports two dataset sources. When `dataset_source = "real"`, it loads the images from the configured path. Initially, dataset processing relied on loading full NumPy arrays into memory, a design that restricted batch sizes and caused Out-Of-Memory (OOM) crashes on consumer GPUs. The current iteration migrates completely to a highly optimized `tf.data.Dataset` pipeline. This architecture processes data lazily, applying dynamic resizing, grayscale-to-RGB conversion, and data augmentations (such as random flips and zooms) on the CPU in parallel with GPU training. Consequently, it eliminates the severe *Dst tensor is not initialized* memory transfer errors and easily scales to large resolutions like MobileNetV2 inputs. If a separate `x_test` split exists, it is mapped as the validation set. When `dataset_source = "synthetic"`, the node generates random noise data instead. If neither option is available, it falls back to pure dummy data so that the training step always completes and the user gets a functional model file, even without a dataset.

Before training begins, the node detects class count mismatches automatically. If the number of unique labels in the dataset differs from the model's output size (a common situation when adapting a 1000-class ImageNet model to a 5-class custom task), the final Dense layer is replaced on the fly without requiring the user to re-run the modification step:

Listing 3.8: Automatic class mismatch correction during fine-tuning

```
if dataset_num_classes != model_num_classes:
    base_output = model.layers[-2].output
    dropout = tf.keras.layers.Dropout(dropout_rate)(base_output)
    new_output = tf.keras.layers.Dense(
        dataset_num_classes, activation='softmax'
    )(dropout)
    model = tf.keras.Model(inputs=model.input, outputs=new_output)
```

Images are also resized in batches of 500 if the loaded data shape does not match the model's expected input, handling grayscale-to-RGB conversion along the way.

NNI Script Generation

A key requirement of this step is producing complete, correct NNI experiment code tailored to the specific model and dataset in use. The generated scripts must handle non-trivial edge cases: sparse versus categorical label formats, input shape mismatches that require dynamic resizing, and class count discrepancies between the model head and the target dataset. They also need to cap memory usage, resolve port conflicts for the NNI Web UI, and avoid API calls that changed across NNI versions.

An initial implementation delegated code synthesis to the hosted LLM via a structured prompt. Testing on the remote GPU deployment revealed a hard limitation: the prompt, which must embed model metadata, dataset paths, and reference code templates, exceeded the 2048-token context window configured for `gpt-oss-20b` under GPTQ quantization. When the input exceeds `max_model_len`, vLLM silently returns an empty string, making the failure non-obvious. Increasing the context window would have required reducing `gpu_memory_utilization` further, leaving insufficient headroom for other concurrent models on the shared GPU.

Since the structure of `manager.py` and `trial.py` is fully determined by the dataset path, the model path, and a fixed search space, a template-based strategy was adopted. The `generate_nni_experiment()` function in `nni_optimization/generator.py` now writes both files through Python f-strings with direct variable injection, as illustrated below:

Listing 3.9: Template-based path injection for the NNI trial script

```
data_path = dataset_info.get("path", "")
model_path = model_info.get("path", "")

trial_content = f"""
DATA  = r'{data_path}'
MODEL = r'{model_path}'

x_train = np.load(DATA + '/x_train.npy', mmap_mode='r')
model   = keras.models.load_model(MODEL, compile=False)
"""
```

This approach removes the LLM dependency from this step entirely. Generation is instantaneous and deterministic; the only inputs are the paths already present in the graph state. The experiment directory is created under `/tmp/nni_experiments/` to avoid permission errors on the read-only project volume mounted inside the Docker container.

Generated File Structure

`manager.py` This file manages the hyperparameter search. It sets up CUDA library paths for Conda environments, defines the NNI search space (learning rate, batch size, optimizer, freeze backbone flag), configures the TPE tuner, scans for a free port in the 8080–8100 range to avoid conflicts, and launches the experiment with `experiment.run(wait_completion=True)`. After all trials complete, it manually iterates `experiment.list_trial_jobs()` to find the highest-accuracy trial, then re-runs `trial.py` with `RETRAIN_MODE=true` to produce the final saved model.

`trial.py` This file executes a single training trial. It checks the `RETRAIN_MODE` environment variable to determine whether to report metrics back to NNI or save the final model. Data is loaded with `mmap_mode='r'`, meaning NumPy maps the files directly from the disk rather than loading the entire dataset into RAM. This optimization is crucial for NNI: if multiple training trials run concurrently, loading the dataset independently for each process would quickly saturate system memory, resulting in an Out-of-Memory (OOM) crash. Instead, it creates a virtual pointer directly to the file stored on the SSD. Data is then read in small blocks only at the exact moment it is needed by the neural network for training. The 'r' stands for "read-only", ensuring that the training process cannot accidentally corrupt the original dataset. Only 50% of the data is used per trial to strictly bound memory usage. If the label array has shape `(N, 1)`, it is converted to one-hot automatically. If the model's output class count does not match the dataset, the final Dense layer is replaced in-place. Images are resized dynamically if the input shape mismatches, and data augmentation (random flips, rotations, zoom) is applied during training.

Hyperparameter Optimization Execution

Once the NNI scripts are written to `/tmp/nni_experiments/`, the training is started by `optimize_hyperparameters_with_nni()`:

1. **Validation**: Checks that both `manager.py` and `trial.py` exist and are syntactically valid Python.
2. **Subprocess Launch**: Spawns `python manager.py` in a separate process to avoid blocking the LangGraph event loop.
3. **Output Streaming**: Captures stdout/stderr in real-time for user feedback.
4. **Completion Detection**: Waits for subprocess termination (blocking until experiment finishes).
5. **Model Retrieval**: Locates `best_model.h5` in the NNI output directory and stores path in state.

The subprocess execution isolates NNI's environment from the LangGraph runtime, preventing library conflicts and allowing independent GPU scheduling.

Algorithm 1 outlines the NNI workflow, including the extraction of the best trial and retraining.

Algorithm 1 NNI Experiment with Automatic Best-Trial Retraining

Require: Model path M, dataset path D, search space S, max trials N
Ensure: Retrained best model saved to disk

$experiment \leftarrow$ NNI_Create(tuner = "TPE", S, N)
$experiment$.run(wait_completion = true)
{Extract best trial manually (stable API-agnostic method)}
$trials \leftarrow experiment$.list_trial_jobs()
$valid\_trials \leftarrow \{t \in trials \mid t.\text{status} = \text{"SUCCEEDED"}\}$
if $|valid\_trials| = 0$ **then**
  error "No successful trials"
end if
$best\_trial \leftarrow \arg\max_{t \in valid\_trials} t.\text{finalMetricData[0].data}$
$best\_params \leftarrow best\_trial$.hyperParameters[−1].parameters
{Retrain with best hyperparameters}
$env \leftarrow$ {"RETRAIN_MODE" : "true", "NNI_PARAMS" : JSON($best\_params$)}
subprocess.run(["python", "trial.py"], env = env)
$best\_model \leftarrow$ Load("best_model.h5")
return $best\_model$

Validation and Model Persistence

After NNI completes, the `validate_customized_model()` node loads `best_model.h5` and evaluates it on a held-out test set. If validation accuracy meets the user's threshold (configurable, default 0.7), the model is saved alongside a JSON metadata file:

Listing 3.10: Model persistence with metadata tracking

```
metadata = {
    "base_model": state.model_path,
    "customization_date": datetime.now().isoformat(),
    "nni_search_space": state.nni_search_space,
    "best_hyperparameters": state.best_hyperparameters,
    "validation_accuracy": final_accuracy,
}

model.save(output_path)
with open(output_path.replace('.h5', '_metadata.json'), 'w') as f:
    json.dump(metadata, f, indent=2)
```

This metadata file links the final model to the hyperparameters that produced it and to the base model it was derived from, which supports reproducibility and simplifies future analysis.

3.2.4 Firmware-AI Integration Workflow

Once the neural network code is ready (*AI Model Discovery*, Section 3.2.2) and the firmware scaffolding is in place (*Firmware Generation*, Section 3.2.1), the *Firmware-AI Integration* workflow merges the two outputs into a single deployable project, as illustrated in Figure 3.5. It handles path collection, firmware layout detection, AI file transfer, `main.c` patching, and a final consistency check, all without manual intervention.

The implementation resides in `workflow3_integration.py` and runs five LangGraph nodes in sequence: `collect_integration_info`, `scan_ai_files`, `copy_ai_files`, `modify_main_c`, and `verify_integration`.

Project Path Collection and Idempotency

The `collect_integration_info()` node determines two key paths: the firmware project directory and the folder containing the AI-generated code. When the workflow is triggered directly by the user (not as part of a sequential Firmware Generation → AI Model Discovery → Firmware-AI Integration chain), these paths are extracted from a natural language message using a Pydantic schema (`IntegrationInfoExtraction`) with structured LLM output.

If the paths are already present in the graph state from a prior step, the node skips collection entirely and goes directly to validation. This avoids unnecessary LLM calls when the workflow is invoked automatically.

When the paths cannot be inferred from the message, the system reads the agent's persistent memory to suggest the last known project directory:

Listing 3.11: Persistent context suggestion during path collection

```
last_fw = state.persistent_context.get("last_project_path", "Nessuno")
dynamic_prompt = {
    "instruction": "...",
    "suggestion": f"Last project was: {last_fw}. Use the same or a new
       path?"
}
resume_value = interrupt(dynamic_prompt)
```

If the user's reply contains a word like "yes", "previous", or "profile", the path from the persistent context is applied directly, with no further LLM extraction.

Filesystem Validation and Layout Detection

Before any file operation begins, the internal helper `_validate_and_detect_structure()` expands relative and home-directory references (e.g., `~/` or `./`) using `os.path.expanduser()` and checks that both paths exist on disk. Path expansion happens before any scan so that downstream functions always work with absolute paths.

The node then detects which of the two common STM32 project layouts is in use. The standard layout places source files under `Src/` and headers under `Inc/`, while the STM32Cube layout uses `Core/Src/` and `Core/Inc/`. The correct pair of directories is selected automatically. If neither pattern matches, the workflow raises an error before touching any file.

The node also handles the case where the project root contains a single subdirectory: in that case, the subdirectory is treated as the real project root, which is common when a CubeMX project is zipped and extracted into a wrapper folder.

AI File Scanning

The `scan_ai_files()` node lists the contents of the AI code directory and splits the files into two groups: `.c` sources (stored in `state.ai_src_files`) and `.h` headers (stored in `state.ai_header_files`). If both groups are empty, the workflow raises an error rather than producing a silent, broken integration.

File Placement and X-CUBE-AI Runtime Headers

The `copy_ai_files()` node copies the scanned files into the firmware directories identified earlier: `.c` files go to `firmware_src_dir` and `.h` files go to `firmware_inc_dir`.

Beyond the network files, the node also handles a less obvious requirement: the X-CUBE-AI runtime headers. When STEdgeAI generates the network code, it also produces a subdirectory at `st_ai_ws/inspector_network/workspace/include` containing the STMicroelectronics AI platform headers (such as `ai_platform.h`). These files are needed at compile time, but are not included in a standard CubeMX project unless the X-CUBE-AI component has been explicitly added through the `.ioc` configuration.

The node checks for this directory and, if it exists, copies all `.h` files into `Middlewares/ST/AI/Inc/`, creating the folder if necessary:

Listing 3.12: X-CUBE-AI runtime header copy

```
ws_include_dir = os.path.join(output_root, "..", "st_ai_ws",
    "inspector_network", "workspace", "include")
middlewares_ai_inc = os.path.join(proj_root, "Middlewares", "ST", "AI", "
    Inc")
```

```
if os.path.exists(ws_include_dir):
    os.makedirs(middlewares_ai_inc, exist_ok=True)
    for header in os.listdir(ws_include_dir):
        if header.endswith('.h'):
            shutil.copy2(src, dst)
else:
    logger.warning("Runtime headers not found. Add X-CUBE-AI via CubeMX .
        ioc.")
```

If the directory is absent, a warning is logged and the user is advised to add the component through CubeMX. This check prevents the common compile-time error caused by a missing `ai_platform.h`.

Automated `main.c` Modification

The `modify_main_c()` node is the most involved step. It reads the existing `main.c` and injects AI initialization and inference code into the STM32CubeMX user-code comment blocks (`USER CODE BEGIN`). Since CubeMX regenerates only its own auto-generated regions, code placed inside these blocks is preserved across future regenerations.

Before writing anything, the node creates a timestamped backup:

Listing 3.13: Backup before patching

```
backup_path = f"{state.main_c_path}.backup_{state.timestamp}"
shutil.copy2(state.main_c_path, backup_path)
```

Four injections are then applied in sequence, each guarded by a regex check to skip the block if the corresponding code is already present:

1. **USER CODE BEGIN Includes**: adds `#include "network.h"` and `"network_data.h"`.
2. **USER CODE BEGIN PV**: declares the AI handle, the activations buffer, and the input/output float arrays (`in_data`, `out_data`).
3. **USER CODE BEGIN 2**: calls `ai_network_create()`, `ai_network_init()`, and binds the buffer pointers.
4. **USER CODE BEGIN WHILE**: calls `ai_network_run()` in the main loop, with commented examples for sensor data input and output classification.

The injected loop code is shown below:

Listing 3.14: Injected inference code in the main loop (USER CODE BEGIN WHILE)

```
/* Fill in_data with sensor data (ADC, I2C, SPI, etc.)
   HAL_ADC_Start(&hadc1);
```

```
   HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
   in_data[0] = (ai_float)HAL_ADC_GetValue(&hadc1) / 4095.0f;
*/

if (ai_network_run(network, &ai_input[0], &ai_output[0]) != 1) {
    /* Inference failed - blink error LED */
    HAL_GPIO_WritePin(LED_ERROR_GPIO_Port, LED_ERROR_Pin, GPIO_PIN_SET);
    HAL_Delay(100);
    HAL_GPIO_WritePin(LED_ERROR_GPIO_Port, LED_ERROR_Pin, GPIO_PIN_RESET);

}

/* Process results (classification example):
   int predicted_class = 0;
   ai_float max_prob = out_data[0];
   for (int i = 1; i < AI_NETWORK_OUT_1_SIZE; i++) {
       if (out_data[i] > max_prob) {
           max_prob = out_data[i];
           predicted_class = i;
       }
   }
*/
```

The network name comes from `state.network_name`, so the injection works across different model names. If `modify_main` is set to `False` in the state, the step is skipped cleanly.

Verification and Persistent Context Update

The `verify_integration()` node checks that every copied file is present at its destination and that the `main.c` patch was recorded as successful. If all checks pass, the workflow saves the used paths to Redis (see Section 3.2.14):

Listing 3.15: Persistent context update on success

```
state.persistent_context["last_project_path"] = state.
    firmware_project_dir
state.persistent_context["last_ai_code_dir"] = state.ai_code_dir
```

These values are then available in future sessions, so the user will be prompted with the same paths as a default suggestion the next time the workflow runs.

If verification fails, the response includes a step-by-step breakdown showing which operations succeeded and which did not, to help with manual recovery.

3.2.5 Web Research Workflow

The *Web Research* workflow provides technical research on demand. It queries the web and synthesizes the results into a structured summary. I avoided routing requests through a static knowledge base. Instead, the workflow runs live DuckDuckGo searches across four distinct domains and evaluates the summary quality using DeepEval.

Request Classification

Before any search is issued, the `classify_search()` node determines what kind of information the user needs. A structured LLM call using `get_llm()` classifies the request into one of four categories: `ai_model` (finding neural network models compatible with STM32), `board_selection` (comparing MCU families), `optimization` (quantization and compression techniques), and `documentation` (tutorials, guides, official references). Requests that fall outside these categories are labelled `none` and routed to a clarification step instead.

From the user's natural language message the node extracts both the category and a refined English search query, which is constructed to be specific enough for a web search engine. A reasoning field explains the classification decision and is logged for debugging.

Dynamic Search Execution

The routing node `search_type_decision()` confirms the category is valid, then passes control to `execute_web_search()`, a single node that handles all four domains. The choice of a unified node rather than four separate ones keeps the graph compact: the appropriate prompt template is selected at runtime from the `SEARCH_PROMPTS` dictionary by indexing on `state.search_type`.

Each prompt template instructs the agent on the specific fields to retrieve. For AI model searches, for instance, the template asks for model name, framework, size in KB, compatible STM32 targets, quantization level, and inference benchmarks. For board comparisons it requests flash, RAM, clock speed, peripheral sets, and price.

The native Python `duckduckgo-search` library executes the search. It fetches up to six webpage snippets, extracting titles, bodies, and source URLs. I deliberately removed complex function-calling wrappers to maintain compatibility with Triton. The backend now handles standard inference requests without needing native tool-calling integrations.

Listing 3.16: Direct DuckDuckGo search execution without tool-calling overhead

```
from duckduckgo_search import DDGS
```

```
with DDGS() as ddgs:
    results = list(ddgs.text(state.search_query, max_results=6))
    for r in results:
        snippet = f"**{r.get('title', '')}**\n{r.get('body', '')}\nSource:
            {r.get('href', '')}"
        raw_results.append(snippet)
```

The raw aggregated text is stored in `state.search_results`. It is then split into paragraph-level chunks (filtered to at least 20 characters) and saved to `state.search_results_list`, which feeds the evaluation stage.

Summarization

After the search, `summarize_search_results()` condenses the raw content into a structured answer. A single LLM call (via `get_llm()`, temperature 0.2) is given the original query and up to 10,000 characters of web results. The prompt instructs the model to produce a technically focused answer in English, use bullet points for readability, cite source URLs where present, and explicitly state when the retrieved content is not relevant to the query. The resulting summary is stored in `state.search_summary` and displayed to the user by the final `finalize_search()` node.

Quality Evaluation with DeepEval

Once the summary is ready, the `finalize_search()` node runs an automated quality assessment using the **DeepEval** framework. Evaluation runs synchronously in a separate thread (via `asyncio.to_thread`) to avoid blocking the event loop. All four metrics run on a local `OllamaModel` backed by `deepseek-r1:latest`, keeping evaluation fully offline. The 0.55 threshold serves exclusively as an observability and logging baseline calibrated for local model capabilities. It does not act as a blocking filter; instead, if a metric falls below this threshold, the framework simply records a "Fail" status for offline analysis without discarding any sources or interrupting the runtime:

- **Faithfulness**: checks that claims in the summary are supported by the retrieved text (threshold: 0.55)
- **Answer Relevancy**: measures how well the summary addresses the original query (threshold: 0.55)
- **Contextual Relevancy**: assesses whether the retrieved documents were topically appropriate (threshold: 0.55)
- **Hallucination**: detects content in the summary absent from the retrieval context (threshold: 0.55)

Each metric is measured independently; failures in one do not abort the others. Scores are printed to the console alongside the summary. If the evaluation framework itself raises an exception (e.g., due to a missing dependency or model timeout), the error is logged as a warning and the workflow completes normally without blocking the user.

3.2.6 Supporting Workflows

Beyond the five primary pipelines, the framework includes two auxiliary workflows that handle data needs for the fine-tuning stage.

Synthetic Data Generation Auxiliary Workflow

The *Synthetic Data Generation* sub-workflow generates time-series and audio waveforms programmatically for two distinct purposes. First, it acts as a training pipeline validator: before committing to a full fine-tuning run on an external dataset, generating a small batch of mathematically exact signals (pure sine waves, silence, impulses) lets the system verify that the training loop is numerically sound and that the model can learn basic patterns at all. Second, it provides controlled sensor simulation. For embedded applications involving accelerometers or microphones, synthetic chirps, drift signals, and noise bursts are often more reliable test cases than scarce real-world anomaly recordings.

Signal parameters are extracted from the user's natural language request through `get_llm()` with the `SyntheticDataRequest` Pydantic schema. Supported signal types are `sine`, `white_noise`, `pink_noise`, `chirp`, `impulse`, `silence`, and `mixed`. If the initial message already contains enough information, the interrupt is skipped. If not, the node issues a LangGraph interrupt to collect frequency, duration, sample rate, amplitude, noise level, and sample count.

Generation is handled by NumPy vectorized operations inside `generate_synthetic_samples()`. Each sample is saved individually as a `.npy` file in `<base_dir>/data/synthetic/`. The node then confirms generation parameters with one final interrupt before handing the dataset path to the *Model Customization* workflow (Section 3.2.3) via `state.synthetic_data_path`.

Dataset Selection Auxiliary Workflow

The *Dataset Selection* sub-workflow manages the selection, download, and formatting of publicly available datasets for use in the fine-tuning stage. It supports four task categories: vision, audio, object detection, and human activity recognition.

The workflow begins by classifying the user's intent as one of three data source types: selecting a predefined dataset from the catalog (`real`), providing a custom download URL (`register`), or falling back to the *Synthetic Data Generation* sub-workflow. The catalog is read from two JSON files, `predefined_datasets.json`

and `dataset_mapping.json`, which decouple dataset metadata from the execution logic and can be updated without modifying the source code.

Dataset selection adapts to the model chosen in the *AI Model Discovery* workflow (Section 3.2.2). The `select_predefined_dataset()` node reads `state.last_task` to determine the task type, looks up preferred datasets from the mapping file, and orders the menu accordingly. A compatibility check then compares the model's input shape from `state.model_architecture` against the dataset's expected shape; if they differ, a warning is logged with a suggested preprocessing approach.

Download supports three backends, chosen automatically per dataset entry: Keras built-ins (`tf.keras.datasets.cifar10`, `mnist`, `fashion_mnist`), generic archive downloads from direct URLs (zip or tar.gz, with automatic extraction and cleanup), and TensorFlow Datasets (`tfds`). Audio datasets receive additional preprocessing through `process_speech_commands()`, which converts raw audio into spectrograms via STFT for compatibility with vision models. After extraction, the archive and intermediate directories are removed to free disk space.

Users can also register new datasets at runtime via `register_custom_dataset()`: a structured LLM call extracts the dataset name, category, URL, and expected shape from a natural language description, the URL is validated with an HTTP HEAD check, and the entry is appended to the JSON catalog for future sessions.

The processed dataset is saved under `<base_dir>/data/real_datasets/<dataset_name>/` and the path is recorded in `state.real_dataset_path` for the *Model Customization* workflow (Section 3.2.3).

3.2.7 State Management and Persistence

All workflow data is stored in a single `MasterState` object defined in `state.py`. This Pydantic-based state class contains over 50 typed attributes covering:

- **User Input**: Original message, intent classification, routing metadata
- **Firmware Workflow**: Board name, MCU series, .ioc file path, CubeMX output directory
- **AI Workflow**: Model path, task type, target MCU, compression level, search iteration count
- **Customization**: Model layers, modification parameters, NNI experiment directory
- **Integration**: Generated code paths, build status
- **Web Research**: Query history, search results, evaluation scores

This persistent state model enables **Context Chaining**, a key architectural feature where outputs from upstream workflows (e.g., the specific MCU target selected during `firmware_flow`) automatically populate the input context for downstream workflows (`ai_flow`). This eliminates redundant user data entry and prevents configuration mismatches, such as selecting an AI model incompatible with the previously generated firmware board. The state object acts as a *single source of truth*, synchronizing project parameters across heterogeneous toolchains (CubeMX, STEdgeAI, TensorFlow).

LangGraph automatically persists this state at every node transition through a checkpointing mechanism, enabling:

1. **Fault Tolerance**: Resume execution from the last successful checkpoint after transient failures
2. **Human-in-the-Loop Gates**: Suspend execution for user approval without losing context
3. **Time-Travel Debugging**: Replay past executions to diagnose workflow errors

A detailed analysis of the underlying storage infrastructure and the dual-layer memory management logic is provided in Section 3.2.14.

3.2.8 Reliability in LLM-Driven Orchestration

Relying on Large Language Models for orchestration decisions introduces non-determinism. During stress testing with concurrent users, unconstrained LLM outputs occasionally caused validation crashes or incorrect routing. The framework solves this by enforcing three layers of input normalization before any LLM response modifies the `MasterState`.

First, I implemented type coercion for numerical parameters. When extracting data like sample rates for synthetic data generation, LLMs frequently append human-readable units (e.g., *"2 kHz"* instead of `2000`). The framework preemptively normalizes these strings into raw floating-point measurements before passing them to Pydantic, bypassing strict validation errors.

Second, I introduced regex-based fallbacks for hardware identification. If the LLM extracts an imprecisely formatted identifier (hallucinating prefixes like *"STM32 F401"* instead of `STM32F401`), a relaxed regular expression such as:

```
r'(?:STM32)?[\s-]*([A-Z])([0-9])'
```

guarantees the core MCU series letter and digit are captured correctly for the downstream ST toolchains.

Finally, I adopted an enumerated prompt anchoring strategy. During model discovery, providing the LLM with only the total count of available models resulted

in hallucinated index selections when users specified model names. Injecting the complete, enumerated list of model names directly into the extraction prompt creates a bounded decision space and enforces deterministic routing.

3.2.9 System Integrity and Transactional Registration

To maintain the stability of the model registry, the framework implements a **Transactional Registration Flow** for user-provided models. When a user submits a new model URL, the entry is first held in a *Pending* state within the `MasterState`. The system then performs an autonomous validation via the STEdgeAI analysis engine. Only upon successful technical profiling (`analyze_success = true`) is the model permanently committed to the `predefined_models.json` registry. This transactional behavior ensures that the system's knowledge base remains free from broken links, malformed architectures, or incompatible file formats.

3.2.10 Configuration Management

Runtime parameters are managed through a dedicated `Configuration` class (`configuration.py`) that dynamically loads settings from a prioritized hierarchy of sources. **Environment Variables** are used for sensitive data such as STMicroelectronics credentials (`ST_EMAIL`, `ST_PASSWORD`) and API keys, while a local **Config File** (`config.json`) defines system-specific paths including the STM32CubeMX binary location, the STM32Cube repository path, and AI output directories.

In addition, the **LangGraph Runtime Config** enables the injection of per-request settings directly into each workflow node via the `config` parameter.

The framework relies on several important parameters to govern its behavior: `local_llm` defines the language model identifier (defaulting to `mistral:latest`), while `llm_temperature` balances deterministic routing (0.0) with creative code generation (0.2). The `llm_context_window` manages token limits, typically 4096–8192, and `cubemx_path` points to the absolute location of the STM32CubeMX CLI.

For AI-specific workflows, the `ai_target` specifies the target MCU (e.g., `stm32h743`), and `ai_compression` sets the desired quantization level ranging from `low` to `very_high`. All parameters are validated at runtime using **Pydantic validators**, to ensure that required paths exist and credentials are correctly formatted before the orchestrator starts any workflow execution.

3.2.11 Graph Topology and Node Composition

Each of the five primary workflows is implemented as a **subgraph** with its own internal routing logic. The master graph connects these subgraphs through conditional edges that allow *inter-workflow transitions* based on user decisions:

- After completing firmware generation, the system offers to proceed to AI model discovery
- After AI model optimization, the system offers to integrate the generated C code into the firmware project
- At any point, users can invoke web research to gather technical documentation

This multi-stage pipeline mirrors the typical embedded AI development lifecycle:

`Hardware Setup → Model Selection → Model Customization → Code Generation → Integration → Testing`

Figure 3.6 illustrates the full graph topology, including the feedback cycles that distinguish this architecture from a simple linear pipeline.

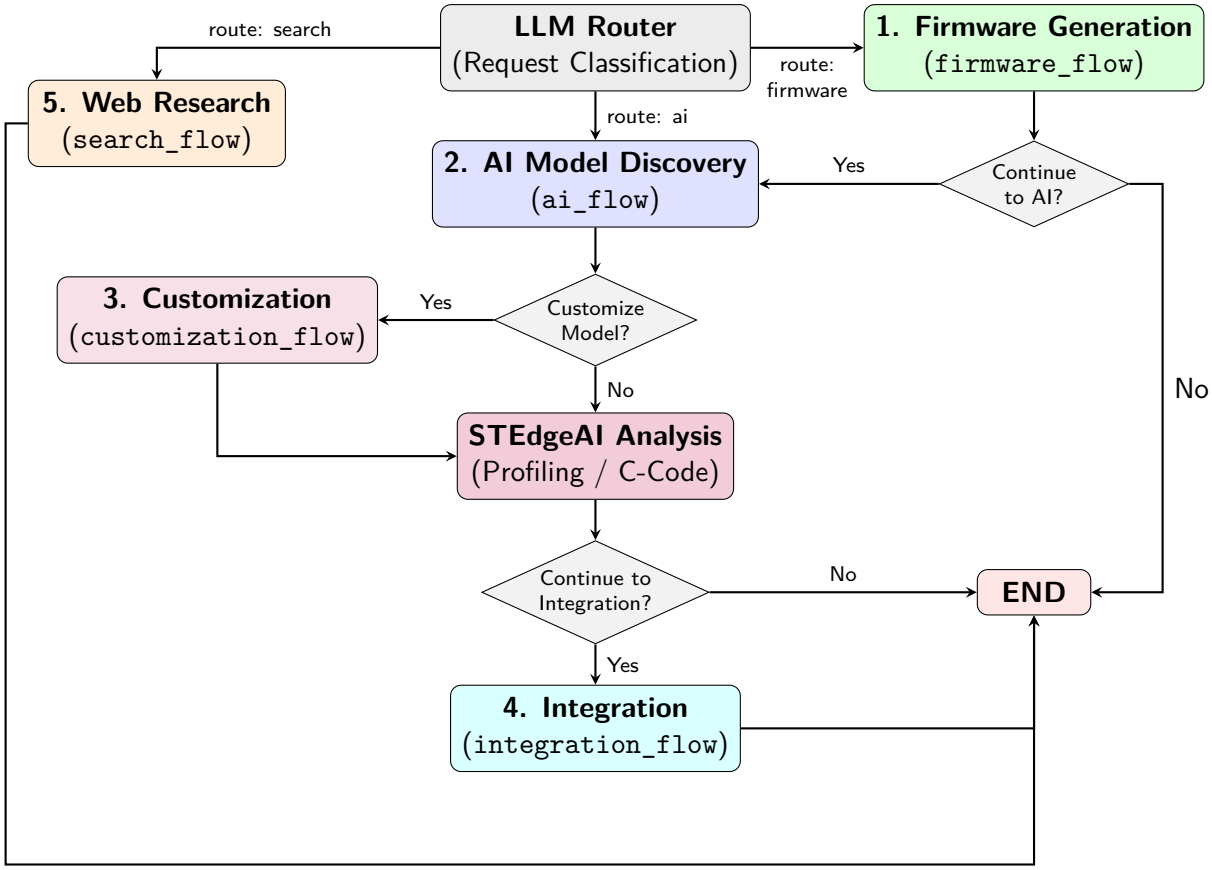

Figure 3.6: Interconnected graph of the five primary workflows. Solid arrows represent the main execution pipeline.

The graph construction logic resides in four primary "injection" functions:

- `inject_firmware_nodes()`: *Firmware Generation* workflow
- `inject_ai_analysis_nodes()`: *AI Model Discovery*, *Model Customization*, and data management workflows
- `inject_integration_nodes()`: *Firmware-AI Integration* workflow
- `inject_web_search_nodes()`: *Web Research* workflow

Each function encapsulates the node definitions and internal routing logic for its respective domain, adding them to the master `StateGraph` builder.

3.2.12 Human-in-the-Loop Interaction

The framework places humans at critical decision points using LangGraph's `interrupt()` primitive. This mechanism pauses execution and returns control to the API server until the user provides feedback.

Interrupt and Resume Mechanism

LangGraph snapshots the full graph state after every node transition. When a node calls `interrupt()`, the runtime suspends the flow and surfaces a payload to the client. The code below illustrates an approval gate between the firmware and AI workflows:

Listing 3.17: Interrupt-based approval gate

```
# From workflow1_firmware.py
def decide_continue_to_ai(state, config):
    # Suspend execution and request user feedback
    user_response = interrupt({
        "type": "decision_request",
        "message": "Firmware generated. Continue with AI selection?",
        "options": ["yes", "no", "skip"]
    })

    # Analyze response via LLM
    llm_parser = get_llm(config, structured_schema=DecisionExtraction,
        temperature=0)
    result = llm_parser.invoke([
        SystemMessage(content=instructions),
        HumanMessage(content=user_response)
    ])

    state.route = "ai_flow" if result.continue_to_ai else "END"
    return state
```

When the user replies, the graph resumes from the exact checkpoint. Since Redis handles the underlying state storage (as detailed in Section 3.2.14), even multi-hour pauses do not affect system stability.

Interaction Patterns

We categorize human interventions into two patterns. **Approval Gates** handle high-risk transitions, such as confirming hardware parameters or accepting a candidate model during iterative search. **Natural Language Steering** instead allows the user to redirect the agent mid-stream. For example, a user might state, *"I actually need a smaller model"*, triggering a transition back to the search node with updated constraints.

Actionable Reporting

To prevent information overload, the orchestrator parses raw logs from tools like CubeMX or STEdgeAI. Instead of displaying hundreds of lines, the system extracts a one-line summary, such as *"Project generated: STM32H7, 4 peripherals enabled"* or *"Memory Warning: Flash limit exceeded"*. These summaries populate the context-aware prompts at interrupt points, ensuring the user makes informed decisions without reviewing verbose technical output.

3.2.13 Logging and Observability

The system implements a multi-tier observability strategy, combining structured backend logging with real-time progress streaming. Technical logs are managed via the standard Python `logging` library with distinct severity levels.

DEBUG logs provide detailed execution traces, including LLM prompt-response pairs and internal state transitions, while **INFO** logs track high-level operational events such as workflow entry, node completion, and subprocess execution. **WARNING** logs are reserved for non-fatal anomalies, such as low LLM confidence scores or active fallback mechanisms, and **ERROR** logs capture recoverable failures, including API timeouts or malformed user inputs requiring re-interaction.

A specific architectural feature is the log interception mechanism: logs generated by critical nodes, such as training status or hardware configuration, are captured by a dedicated `QueueHandler` and streamed to the client using the **Newline Delimited JSON (NDJSON)** format. This allows frontend clients, such as the *Gradio UI* or the **VS Code extension**, to provide immediate status visualization and progress bars, transforming a complex multi-step backend process into a transparent and interactive user experience.

3.2.14 The Auxiliary Persistence Layer (Redis)

LangGraph's built-in checkpointing writes a snapshot of the graph state after every node transition, which is enough for fault tolerance within a single session. It does not, however, retain user preferences or project context across separate sessions: once a thread terminates, all accumulated knowledge, such as the last used MCU board or the preferred compression level, is lost.

To fill this gap, I introduced an auxiliary persistence layer built on Redis. It operates on two levels. The first, *short-term memory*, is the session state managed by LangGraph's `AsyncRedisSaver`. It stores the transient graph snapshot that enables HITL interrupt/resume and time-travel debugging. The second, *long-term memory*, is a custom storage layer keyed by `user_id`. It survives across sessions and retains context such as preferred hardware boards, last used models, and past project paths.

Together these two layers enable *context chaining*: outputs from a previous session (e.g., the MCU target selected during `firmware_flow`) automatically pre-populate the input context of future workflows (e.g., `ai_flow`), eliminating redundant data entry.

Short-Term Memory (LangGraph Checkpoints)

LangGraph checkpoints are stored using a composite key built from the `user_id`, the `session_id` (identifying the client instance), and a timestamp-based `checkpoint_id` managed automatically by LangGraph, as in the following example:

```
checkpoint:michele:vscode-session:1707575432123
checkpoint:michele:vscode-session:1707575455789
checkpoint:john:cli-session:1707575500000
```

This hierarchical naming enforces strict user isolation: different users can never access each other's checkpoints. It also supports session multiplexing, so a single user can run independent workflows from VS Code and CLI at the same time. Because each checkpoint ID contains a timestamp, the system preserves a clear temporal ordering that enables time-travel debugging.

Long-Term Memory (User Profiles)

User profiles follow a simpler key structure for direct access (e.g., `user:michele:profile`). Each profile is a JSON-serialized dictionary that captures the user's most recent working context, as shown in Listing 3.18.

Listing 3.18: User profile schema stored in Redis

```
1 {
2   "board_name": "NUCLEO-H743ZI",
```

```
  "mcu_series": "STM32H7",
  "last_model": "mobilenet_v1_0.25_128",
  "last_project_path": "/home/user/workspace/h7_project",
  "last_workflow": "firmware",
  "timestamp": "2024-02-10T14:35:22Z"
}
```

The schema records the most recently used STM32 board and MCU series (for pre-filling firmware generation requests), the last AI model selected for optimization, and the path of the latest firmware project directory. It also stores the last executed workflow route and a timestamp marking the most recent successful completion. When a new session starts, these fields allow the orchestrator to skip redundant questions and resume from a familiar context.

Redis Integration and Implementation

The persistence layer is initialized during FastAPI server startup. An asynchronous Redis client connects to the local `redis://localhost:6379` instance and is wrapped by LangGraph's `AsyncRedisSaver`, which handles all checkpoint I/O transparently:

Listing 3.19: Redis checkpointer initialization in server.py

```
from langgraph.checkpoint.redis.aio import AsyncRedisSaver
import redis.asyncio as redis

# 1. Create async Redis client
checkpointer_redis = redis.from_url(
    "redis://localhost:6379/0",
    encoding="utf-8",
    decode_responses=True
)

# 2. Initialize AsyncRedisSaver
memory = AsyncRedisSaver(redis_client=checkpointer_redis)

# 3. Setup internal RedisVL indexes (run once on startup)
await memory.setup()

# 4. Compile graph with checkpointer
graph = builder.compile(checkpointer=memory)
```

The `memory.setup()` call creates internal Redis indexes used by RedisVL for efficient checkpoint retrieval and time-travel queries. Each time a graph node completes, LangGraph automatically persists the `MasterState` to Redis as a MessagePack or JSON-serialized payload:

```
{
  "v": 1,
  "ts": "2024-02-10T15:30:45.123Z",
  "channel_values": {
    "messages": [...],
    "firmware_project_dir": "/tmp/stm32_project",
    "model_path": "/cache/mobilenet_v2.h5",
    "route": "ai_flow",
    "nni_search_space": {...}
  },
  "parent_config": {
    "configurable": {
      "thread_id": "michele:vscode-session",
      "checkpoint_id": "1707575432123"
    }
  }
}
```

The `channel_values` field contains the complete `MasterState` with all 50+ attributes, `parent_config` holds the reference to the previous checkpoint (for time-travel), and `ts` provides the timestamp for ordering.

The long-term memory layer is managed directly in `server.py`. Before graph execution, the server loads the user profile from Redis and injects it into the initial state:

Listing 3.20: User profile loading in server.py

```
# 1. Construct profile key
user_profile_key = f"user:{request.user_id}:profile"

# 2. Fetch from Redis
raw_profile = await redis_client.get(user_profile_key)
user_profile = json.loads(raw_profile) if raw_profile else {}

# 3. Inject into initial state
initial_state = {
    "message": last_user_message,
    "persistent_context": user_profile,
    "reset_profile": False
}
```

The `persistent_context` field is then accessible to all graph nodes for context-aware decisions. After workflow completion or critical state mutations, the profile is updated with the latest values and written back:

Listing 3.21: User profile saving in server.py

```
# 1. Extract current values from state
new_profile = {
    "board_name": state.get("board_name"),
    "mcu_series": state.get("mcu_series"),
    "last_model": state.get("selected_model"),
    "last_project_path": state.get("firmware_project_path") or state.get(
        "firmware_project_dir"),
    "last_workflow": state.get("route"),
    "timestamp": state.get("timestamp")
}

# 2. Update profile dictionary
new_profile = {k: v for k, v in new_profile.items() if v is not None and
    v != ""}
updated_profile = {**user_profile, **new_profile}

# 3. Serialize and save to Redis
await redis_client.set(user_profile_key, json.dumps(updated_profile))
```

System Interaction Scenarios

The following scenarios illustrate how the dual-memory architecture supports real-world usage patterns.

Scenario 1: Multi-Day NNI Experiment Day 1, Session 1 (VS Code):
The user requests an NNI experiment on MobileNetV2 for the Fruit-360 dataset. The framework launches 20 NNI trials, estimated to take 12 hours. After 2 hours the user closes VS Code. The short-term memory saves the checkpoint with `route = "nni_optimization"` and `nni_status = "running"`, while the long-term memory records `last_board = "STM32H7"`.

Day 2, Session 2 (CLI):
The user reconnects via CLI using the same `user_id` and `session_id`. The server calls `graph.aget_state(config)`, retrieves the Day 1 checkpoint, and detects `current_state.next = ["check_nni_status"]`. The graph resumes automatically: NNI results are retrieved from disk, the best model is saved, and integration proceeds.

Scenario 2: Board Preference Pre-Fill Session A (Firmware Generation):
The user creates a project for STM32H743. Once firmware is generated, the profile is updated with `last_board = "STM32H743ZI"`.

Session B (AI Model Selection, one week later):
The user requests "Analyze MobileNetV2" without specifying hardware. The `collect_analysis_info()` node reads `state.last_board` from the persistent context and auto-populates `target_mcu = "stm32h743"`, eliminating one clarification roundtrip.

Scenario 3: HITL Interrupt and Resume During execution, a workflow node calls `interrupt({type: "decision_request", message: "Proceed to AI?"})`. LangGraph suspends and saves a checkpoint with `next = ["decide_continue_to_ai"]`. The API server forwards a JSON event to VS Code, where the user sees: "Firmware generated. Continue with AI model selection?"

When the user replies (e.g., "Yes, use high compression"), VS Code sends a `POST /stream` request. The server retrieves the latest checkpoint, injects the user response via `graph.aupdate_state()`, and the graph resumes from the exact node that issued the interrupt, parsing the reply through an LLM and routing accordingly.

Performance Considerations

Redis I/O Overhead Checkpoint writes inherently add latency to the execution pipeline, typically in the range of 5 to 20 milliseconds per node transition. To mitigate this overhead, the framework employs several strategies, such as the use of non-blocking asynchronous I/O operations in the `AsyncRedisSaver` to prevent main event loop stalls. Network transit latency is minimized during development by defaulting to a local `redis://localhost` connection. LangGraph further optimizes this process by consolidating multiple internal state updates from a single node's execution context into a single write operation before transitioning to the next stage of the graph.

Memory Footprint and Evolution of Data Persistence During early development, the system attempted to store large artifacts (such as generated firmware zips, serialized models, and base64 images) directly within the Redis state. However, this approach presented technical bottlenecks, as Redis string size limits and synchronization overhead slowed down the orchestration loop.

To resolve this, the architectural design evolved toward a hybrid persistence model using **Container Persistent Volumes**. Instead of using an intermediate database for heavy files, the system writes artifacts directly to persistent storage volumes mapped to individual container instances. This strategy perfectly aligns with a Cloud-Native architecture: in a future Kubernetes deployment, each user will operate within a dedicated namespace with dynamically provisioned PersistentVolumeClaims (PVCs).

Redis is now strictly limited to maintaining the lightweight conversational checkpointing of the LangGraph execution thread, while the storage of project files

is handled natively by the file system. This simplifies the architecture, improves I/O performance, and guarantees environment isolation without relying on database abstractions.

Security Considerations

Given that user profiles may store sensitive information, such as STMicroelectronics developer credentials required for STEdgeAI cloud authentication, the framework must adhere to strict security practices. In production deployments, it is imperative to encrypt sensitive fields using solid algorithms like AES-256 before persisting them in the Redis data store.

Also, access control should be strictly enforced using Redis ACL (Access Control Lists) to restrict client permissions, ensuring that specific key patterns (e.g., `user:*:profile`) are only accessible to authenticated and authorized services.

Finally, to comply with GDPR data minimization principles and prevent indefinite data retention, user profiles are configured with a Time-To-Live (TTL) expiration policy, ensuring they are automatically purged after one year of complete inactivity.

3.3 The Inference Layer

Every workflow node that requires natural-language reasoning delegates its request to a centralized inference abstraction. The utility function `get_llm()` in `utils.py` serves as the single entry point: it receives a model name and optional parameters, then decides at runtime whether to route the request to a local Ollama instance or to a remote NVIDIA Triton Inference Server. This decision is based on a global `USE_TRITON_BACKEND` flag and the compatibility of the requested model with the Triton repository. The workflow code stays completely agnostic to the serving backend; changing engines requires toggling one flag and nothing else.

3.3.1 Serving with Ollama

The initial deployment relied on Ollama, an open-source tool that wraps quantized LLMs behind a local REST API. Each model is pulled once (`ollama pull mistral`) and served at `http://localhost:11434`. The LangGraph container communicated with it through the `ChatOllama` class from the `langchain-ollama` package.

Ollama worked well during early development. Its setup is minimal: one container, no custom configuration files, immediate model availability after the first download. During prototyping with limited resources, this simplicity is a clear advantage, since a developer with a single consumer GPU can run the entire pipeline locally without external dependencies.

The limitation appeared under multi-model and multi-user conditions. Ollama locks one GPU for one model at a time. Loading a second 7B model while the first is still resident causes VRAM exhaustion. In practice this forced sequential load/unload cycles (`keep_alive=0`) that added latency and occasionally triggered process crashes when the unload did not complete before the next load request arrived.

3.3.2 Serving with NVIDIA Triton Inference Server

To overcome these concurrency constraints, I migrated the inference backend to NVIDIA Triton Inference Server. Triton manages a *model repository*: a directory tree where each subfolder contains a model configuration (`config.pbtxt`) and the corresponding backend (Python, ONNX, TensorRT, or vLLM). The server exposes both HTTP and gRPC endpoints and handles model lifecycle, batching, and scheduling internally.

Custom LangChain Integration: `ChatTriton`

LangGraph expects LLM providers that implement the LangChain `BaseChatModel` interface. Triton has no native LangChain adapter, so I wrote a custom wrapper class, `ChatTriton` (in `triton_client.py`), that converts LangChain message lists into the prompt format expected by the Triton Python backend, sends the inference request via the native V2 HTTP API (`POST /v2/models/{model}/infer`), and parses the response back into a LangChain `AIMessage`.

The same file provides `TritonEmbeddings`, a lightweight embedding client that routes `embed_documents()` and `embed_query()` calls to a `nomic-embed` model configured to run on CPU, so it does not compete for GPU memory with the heavier LLMs.

Explicit Model Control and Dynamic Swapping

Triton is launched with the `-model-control-mode=explicit` flag. In this mode no model is loaded at startup. The `ChatTriton` client manages the full lifecycle through a three-stage strategy implemented in its `_ensure_model_loaded()` method.

First, it performs a *fast-path check*: a query to the Triton repository index reveals whether the target model is already in `READY` state. If so, inference proceeds immediately. Second, if the model is not loaded, the client issues an *optimistic load* via `POST /v2/repository/models/{model}/load`. When enough VRAM is available, this call succeeds without disrupting other loaded models. Third, if Triton rejects the load request (HTTP 400, typically due to GPU memory exhaustion), an *OOM fallback* activates: the client iterates over all other registered LLMs, unloads them one by one through the `/unload` endpoint, waits for their status to transition to `UNAVAILABLE`, and retries the load.

After a successful load, the client polls the V2 infer endpoint with a lightweight probe request until the backend is fully operational. This extra step is needed because the Python backend's internal CUDA context initialization can lag behind the repository status flip.

The net effect is that only one large language model occupies VRAM at any given time, while the mechanism stays invisible to the workflow nodes.

3.3.3 GPU Memory Management for Concurrent Workloads

When inference and training share the same GPU, explicit VRAM partitioning becomes necessary. TensorFlow defaults to allocating all available GPU memory at initialization. If multiple users trigger model inspection or fine-tuning concurrently, the first subprocess monopolizes the device and subsequent sessions crash with Out-of-Memory exceptions.

To support concurrent access, the framework injects strict virtual device configurations before executing any Keras or TensorFlow operation:

Listing 3.22: Strict VRAM partitioning for concurrent TensorFlow subprocesses

```
# Prevent greedy VRAM allocation by forcing a hard limit
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    for gpu in gpus:
        tf.config.experimental.set_virtual_device_configuration(
            gpu,
            [tf.config.experimental.VirtualDeviceConfiguration(
                memory_limit=4096)]
        )
```

This caps fine-tuning tasks at 4 GB and lightweight model inspection at 1 GB. On a standard research GPU the configuration supports three to four simultaneous fine-tuning sessions or up to fifteen concurrent inspection calls. The orchestrator also provides utility functions (`force_unload_triton()`, `reload_triton_models()`) that release LLM VRAM before launching a training subprocess and reload the models once training completes.

3.3.4 LLM Selection

The performance and reliability of an agentic workflow depend on the reasoning capabilities and specialized training of the underlying language models. Rather than using a single general-purpose model, the framework adopts a multi-model strategy where different LLMs are assigned to specific nodes based on their strengths.

During the initial characterization phase, five local models served through the Ollama backend were evaluated across different task types, as summarized in

Table 3.1. This benchmarking was necessary to identify the optimal ensemble for the final production deployment on Triton.

Table 3.1: Comparative analysis of local LLMs for agentic workflow tasks.

Model	**Size**	**Core Strengths**	**Workflow Role**
DeepSeek-R1	7B	Reasoning, Debugging	Search Evaluation (LLM-as-a-Judge)
Llama3-Groq	8B	Structured Formats (JSON)	Workflow Routing
Qwen 2.5	7B	Coding and Mathematics	C Selection Logic
Mistral	7B	Summarization, Multi-task	Search Summarization
GPT-OSS	20B	Code Density, Context	NNI Script Generation

Task-Specific Model Assignment

The rationale for each assignment follows from the computational demands of the corresponding node.

For NNI script generation, empirical testing showed that **GPT-OSS** (20B) consistently outperforms the 7B alternatives. Its broader parameter count translates to wider coverage of Python library APIs and fewer syntax hallucinations in the generated trial code. Compared to **DeepSeek-R1**, it also runs faster wall-clock: R1's chain-of-thought mechanism adds a significant token overhead before producing any output, while GPT-OSS generates the final script directly with high zero-shot accuracy.

For routing and structured output, **Llama3-Groq** (8B) was chosen for its strong JSON compliance under strict instruction-following conditions. Routing nodes need deterministic, well-formed outputs at low latency; Llama3-Groq proved most reliable in this role during the benchmarking phase.

DeepSeek-R1 is reserved exclusively for LLM-as-a-Judge evaluation within the DeepEval suite. The chain-of-thought overhead that makes it slower for code generation is an advantage here: the explicit reasoning trace improves the interpretability of faithfulness and relevancy scores, which is useful when diagnosing unexpected evaluation results.

This tiered assignment keeps the agentic logic reliable while distributing the GPU load across models sized to their specific tasks.

3.4 The User Interface Layer

To connect the backend LangGraph orchestrator to the developer's daily workflow, a custom Visual Studio Code extension was developed. This extension transforms the framework from a command-line tool into a real-time, context-aware assistant embedded directly in the Integrated Development Environment (IDE).

3.4.1 IDE Integration and Asynchronous Feedback

3.4.2 Architecture Overview

Before standardizing on the VS Code extension, I used **LangSmith** and **LangGraph Studio** to trace the graph execution during the initial development and debugging phases. These platforms provided a visual, node-by-node representation of the workflow. This visualization enabled direct inspection of the state object at each step and helped tune the LLM routing prompts. Once I stabilized the orchestration logic, I shifted focus entirely to the native IDE integration.

The integration follows a client-server architecture where the VS Code extension (TypeScript frontend) communicates with the FastAPI backend (Python) via HTTP streaming. This design provides several distinct advantages. Primarily, it enforces a clean separation of technologies: the complex LangGraph orchestration is executed in a dedicated Python environment, while the user interface uses VS Code's native TypeScript APIs.

Also, the use of asynchronous execution ensures that long-running workflows, such as NNI hyperparameter optimization experiments or large model downloads, do not freeze the VS Code interface, maintaining a responsive developer experience.

Finally, this decoupled approach inherently supports multi-client architecture, allowing the same backend to simultaneously serve the VS Code extension, command-line interfaces, and potential future web dashboards.

3.4.3 VS Code Extension Configuration (`package.json`)

The extension manifest registers a Chat Participant using VS Code's native chat interface API (introduced in VS Code 1.85+).

Listing 3.23: Chat Participant registration in package.json

```
{
  "name": "stm32-ai-assistant",
  "displayName": "STM32 Edge AI Chat",
  "description": "Chat with your STM32 AI Workflow Agent",
  "version": "0.0.1",
  "engines": {
    "vscode": "^1.85.0"
  },
  "contributes": {
    "chatParticipants": [
      {
        "id": "stm32-ai.assistant",
        "name": "stm32ai",
        "fullName": "STM32 Edge AI Assistant",
        "description": "Analyze models, generate firmware, integrate AI",
```

```
      "isSticky": true
    }
  ]
 }
}
```

Key Parameters:

- `id`: Unique identifier for the chat participant (used internally by VS Code).
- `name`: The `@stm32ai` handle users type to invoke the assistant.
- `isSticky: true`: Enables multi-turn conversations where context (previous messages) is automatically retained across chat interactions.

3.4.4 TypeScript Handler Implementation

The core logic resides in `extension.ts`, which implements a `ChatRequestHandler` callback.

Listing 3.24: Chat request handler in extension.ts

```
const handler: vscode.ChatRequestHandler = async (
  request: vscode.ChatRequest,
  context: vscode.ChatContext,
  stream: vscode.ChatResponseStream,
  token: vscode.CancellationToken
) => {
  stream.markdown('Contacting STM32 Brain...\n\n');

  // 1. Prepare payload for FastAPI server
  const payload = {
    messages: [{ role: 'user', content: request.prompt }],
    user_id: "michele",
    session_id: "vscode-session"
  };

  // 2. POST request to streaming endpoint
  const response = await fetch('http://127.0.0.1:8000/stream', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(payload)
  });

  // 3. NDJSON stream parsing
  const reader = response.body.getReader();
```

```
  const decoder = new TextDecoder("utf-8");
  let buffer = "";

  while (true) {
    if (token.isCancellationRequested) break;

    const { done, value } = await reader.read();
    if (done) break;

    buffer += decoder.decode(value, { stream: true });
    const lines = buffer.split("\n");
    buffer = lines.pop() || "";

    for (const line of lines) {
      if (!line.trim()) continue;
      const data = JSON.parse(line);

      if (data.type === 'markdown') {
        stream.markdown(data.content);
      } else if (data.type === 'progress') {
        stream.progress(data.content);
      } else if (data.type === 'error') {
        stream.markdown('**Error:** ' + data.content);
      }
    }
  }
};

vscode.chat.createChatParticipant('stm32-ai.assistant', handler);
```

Implementation Details. The handler implementation incorporates several critical functionalities to manage errors and provide a fluid user experience.

First, it implements graceful cancellation handling via the `CancellationToken`; this allows users to abort long-running requests simply by closing the chat window or clicking a stop button, which in turn drops the HTTP connection and signals the backend to terminate the LangGraph execution.

Then, to handle the unreliability of network streams where chunks might contain partial JSON objects, the extension utilizes NDJSON buffering. A `buffer` string accumulates incoming data until a complete, newline-delimited JSON object is successfully parsed.

Finally, the handler processes typed responses dynamically; the server emits distinct event types (such as `markdown`, `progress`, or `error`), and the frontend routing logic renders each type using the appropriate UI components within the VS

Code chat view.

3.4.5 The Backend Communication Layer: FastAPI Streaming

The backend implements the `/stream` endpoint using FastAPI's `StreamingResponse`.

Listing 3.25: FastAPI streaming endpoint in server.py

```
@app.post("/stream")
async def stream_chat(request: ChatRequest):
    user_id = request.user_id
    session_id = request.session_id
    thread_id = f"{user_id}:{session_id}"

    # 1. Load user profile from Redis
    profile = await load_user_profile_from_redis(user_id)

    # 2. Build initial state
    initial_state = MasterState(
        message=request.messages[-1].content,
        last_board=profile.get("last_board"),
        preferred_compression=profile.get("preferred_compression")
    )

    # 3. Configure LangGraph thread
    config = {
        "configurable": {
            "thread_id": thread_id,
            "user_id": user_id
        }
    }

    # 4. Stream events from LangGraph execution
    async def event_generator():
        async for event in graph.astream(initial_state, config):
            if event["type"] == "node":
                yield f'{{"type": "progress", "content": "Executing {event
                    ["name"]}..."}}\n'
            elif event["type"] == "output":
                yield f'{{"type": "markdown", "content": {json.dumps(event
                    ["data"])}}}\n'

    return StreamingResponse(event_generator(), media_type="application/x-
        ndjson")
```

The server-side implementation incorporates several key features to enhance the user experience and ensure data isolation. User Profile Injection allows the system to pre-populate the state with parameters like `last_board` and `preferred_compression` from Redis long-term memory, enabling context chaining across independent sessions.

To maintain security and state consistency, Thread-Based Isolation assigns a unique `thread_id` to each user and session combination, ensuring isolated checkpoints in Redis.

Finally, the use of NDJSON Encoding serializes each event as a JSON object followed by a newline (`\n`), which allows the VS Code extension to parse the stream incrementally and provide real-time updates.

3.4.6 Communication Protocol

The extension and server communicate via a custom NDJSON (Newline Delimited JSON) protocol. Each line represents a single event:

```
{"type": "progress", "content": "Routing to AI Flow..."}
{"type": "markdown", "content": "## Model Selected\n\nMobileNetV2 (0.35)"}
{"type": "progress", "content": "Downloading model from GitHub..."}
{"type": "markdown", "content": "Analysis complete. RAM: 280KB, Flash: 500KB"}
{"type": "error", "content": "Model exceeds STM32F4 Flash limit"}
```

where:

- `progress` represents transient status updates (e.g., "Routing to AI Flow...", "Analyzing model..."), rendered as bullet points in the chat.
- `markdown` represents rich-formatted responses supporting bold, code blocks, tables, and links.
- `error` represents critical failures displayed with red highlighting and error icons.

3.4.7 Developer Experience Benefits

The VS Code integration provides significant user experience advantages over traditional CLI-based interactions or the native LangGraph web interface (accessible via the `langgraph dev` command, which is primarily designed for graph debugging rather than end-user workflows). By operating directly within the IDE, the assistant gains context awareness; leveraging the `vscode.window.activeTextEditor` API enables advanced features such as the automated analysis of currently open models and direct firmware generation for active `.ioc` projects.

Then, the assistant facilitates inline documentation by embedding clickable file links (e.g., `file:///path/to/network.c#L42`) within markdown responses,

allowing developers to navigate intuitively to generated code segments. Also, during long-running operations, such as model downloads or NNI experiments, real-time progress visibility prevents the perception of system freezes and keeps the user informed of the pipeline's status.

3.4.8 Deployment and Testing

The development workflow for the extension involves three main steps:

- **Compilation**: `npm run compile` transpiles the TypeScript source into executable JavaScript in the `./out/extension.js` directory.
- **Debug session**: pressing `F5` opens an Extension Development Host environment where the `@stm32ai` handle becomes available.
- **Backend prerequisite**: the FastAPI server must be running (`python -m src.api.server`) before starting any test.

To verify the correct operation of the system, several testing scenarios were executed. These included firmware generation requests (e.g., "@stm32ai create a project for STM32H743 with STM32CubeIDE"), model analysis commands (e.g., "@stm32ai analyze MobileNetV2 for STM32F4 with high compression"), and NNI optimization triggers. A specific test was also performed to verify that closing the chat window correctly triggers the `CancellationToken`, resulting in the graceful termination of long-running tasks.

3.5 Dockerization and Portability

I containerized the entire architecture using Docker to isolate the framework from the host operating system and simplify deployment across different environments. The orchestration relies on a `docker-compose.yml` file that defines an internal virtual network, allowing the microservices to communicate securely. I structured the system around three primary services.

The **LangGraph Application Server** (`langgraph-app`) is the core container executing the Python backend, FastAPI event streaming, and the LangGraph orchestrator. I built it using a custom `Dockerfile` based on Ubuntu 22.04. An important configuration choice was the explicit mapping of the User ID (`UID 1002`) to match the host machine. This prevents permission conflicts when the container mounts local volumes to generate STM32CubeMX projects or compile C code. Instead of installing massive embedded toolchains inside the container, the `docker-compose.yml` configuration uses read-only bind mounts. These mounts map the host's existing STM32CubeMX and STEdgeAI installations directly into the container's path, drastically reducing the image footprint.

The **Redis Checkpointer** (`redis`) container runs `redis-stack-server` and provides the high-speed in-memory datastore required by LangGraph for state persistence and context chaining. I attached a dedicated volume (`redis_data`) to ensure that long-term user profiles and workflow checkpoints survive container restarts.

The **Triton Inference Server** (`triton-server`), as analyzed in Section 3.3, decouples the orchestration logic from the heavy LLM computation. While the initial architecture relied on a local `ollama` container, I migrated the inference backend to a dedicated NVIDIA Triton Inference Server to achieve horizontal scalability. The custom `Dockerfile.triton` extends the official NVIDIA image by injecting the `vLLM` engine and `sentence-transformers`. This configuration provides high-throughput token generation for models like Mistral and DeepSeek.

Decoupling the agent logic from the Triton inference server creates a cloud-native topology. Developers can deploy the lightweight LangGraph container on an edge node or a local laptop, routing point-to-point HTTP/gRPC requests to a centralized, high-performance computing cluster hosting Triton. Figure 3.7 summarizes this topology.

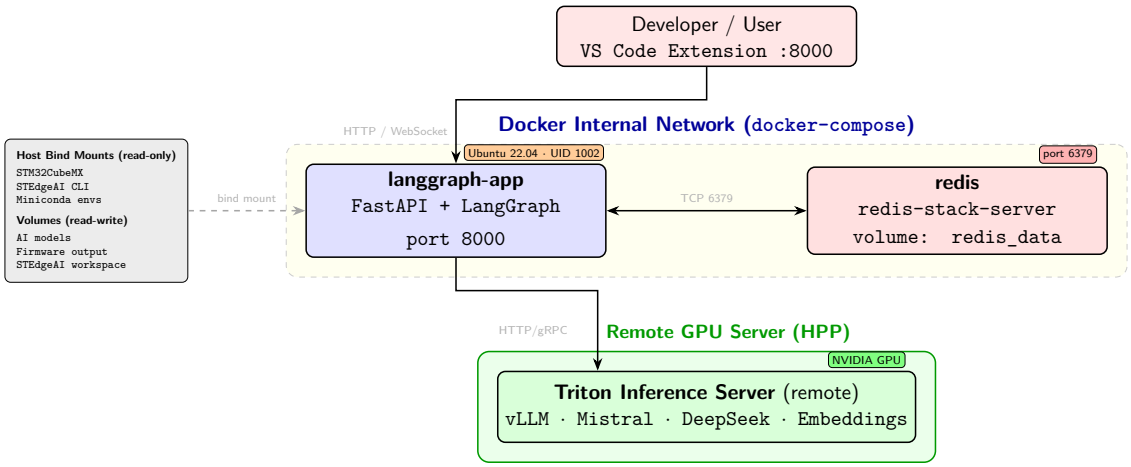

Figure 3.7: Docker deployment topology. The `langgraph-app` and `redis` containers share an internal virtual network. Host tools (STM32CubeMX, STEdgeAI, Miniconda) are injected as read-only bind mounts. The NVIDIA Triton Inference Server runs on a remote GPU cluster and is reached over HTTP/gRPC.

3.6 System Integrity and Environment Challenges

Implementing a fully autonomous MLOps pipeline for embedded systems involves addressing a variety of software dependencies, legacy constraints, and proprietary

toolchain limitations. This section details the specific architectural solutions adopted to manage these interoperability challenges.

3.6.1 Dual-Environment Strategy for Legacy Model Support

A critical challenge encountered during the development of the framework was the fundamental incompatibility between the modern deep learning libraries required for agentic orchestration and the legacy formats predominantly found in established embedded AI repositories.

Specifically, while the NNI optimization engine and the LangGraph orchestrator strictly rely on modern Python environments (such as Keras 3.x and PyTorch 2.x) to use recent asynchronous execution features, many pre-trained models, including those from older STModelZoo releases, are serialized in older Keras 2.x H5 formats. These older formats are incompatible with Keras 3.x runtimes due to breaking changes in the serialization API.

To elegantly resolve this conflict, the framework employs a Dual-Environment Architecture. The system simultaneously maintains two distinct virtual environments: a primary environment (`stm32`) that hosts the LLM agents and the LangGraph orchestrator using the latest libraries for maximum performance, and a tightly pinned legacy environment (`stm32_legacy`) running TensorFlow 2.15 designed exclusively for introspecting older models.

Mechanistically, when the Model Discovery Agent identifies a potential candidate model, it delegates the inspection task to a secure subprocess running within the legacy environment. This dedicated subprocess loads the model, extracts its architectural metadata and I/O signatures, and returns the information as a JSON object to the main orchestrator, thereby securely bridging the semantic gap between software generations without compromising application stability.

3.6.2 Model Format Standardization: H5 vs. TFLite

During the NNI hyperparameter optimization phase, the generated `trial.py` script saves the best model as a `.h5` file. This is the native serialization format of Keras 2.x: it preserves the full-precision floating-point weights and the symbolic Keras model object, which the subsequent fine-tuning and validation steps need to load correctly.

Quantization is never applied inside the training loop. Instead, the `-compression` flag is injected at the STEdgeAI invocation stage, when the CLI calls `analyze`, `validate`, and `generate` in sequence. This delegation is intentional: applying INT8 quantization through the ST toolchain, rather than through a Keras-side quantization-aware training pass, ensures that the generated `network.c` and `network.h` files are directly compatible with the X-CUBE-AI runtime without additional post-processing.

The format story changes for models serialized with **Keras 3.x**. STEdgeAI 2.2 cannot parse the new native `.keras` format directly, and in practice it also fails on `.h5` files produced by Keras 3 due to internal API changes in the graph traversal layer (e.g., the error `INTERNAL ERROR: 'Concatenate' object has no attribute 'get_input_shape_at'`). To handle this transparently, the pipeline auto-detects Keras 3 models and converts them to TFLite before calling STEdgeAI, as described in the next subsection. The `.h5` format therefore remains the internal standard for the training and optimization phases, while STEdgeAI always receives either a Keras 2 `.h5` or a converted `.tflite` artifact, depending on which Keras generation produced the model.

3.6.3 The Keras 3 Compatibility Gap and the TFLite Bridge

With the transition to Keras 3.x, a significant regression was observed in the STMicroelectronics STEdgeAI 2.2 toolchain. Modern models serialized with the native Keras 3 format (`.keras`) often trigger internal errors in the ST parser (e.g., `INTERNAL ERROR: 'Concatenate' object has no attribute 'get_input_shape_at'`). This issue is caused by the shift in the internal Keras API that the ST tool expects for graph traversal.

To smoothly overcome this limitation without demanding manual model conversion from the end-user, I successfully engineered an Autonomous TFLite Bridge. When the orchestrator explicitly detects a modern Keras 3 model, it automatically spawns a dedicated subprocess within the primary `stm32` environment. This subprocess utilizes the native `TFLiteConverter` API to flawlessly transform the unsupported Keras model into a standard TensorFlow Lite flatbuffer. Following this successful conversion, the system transparently redirects the STEdgeAI analysis engine to target the newly generated TFLite artifact instead of the original Keras file.

This bridge "flattens" the model architecture into standard TFLite operators, which the ST toolchain interprets correctly, thus restoring the autonomous flow for the latest state-of-the-art models.

3.6.4 Architectural Constraints on Model Customization

While the *AI Model Discovery* workflow (Section 3.2.2) achieves format-agnostic interoperability, successfully processing `.h5`, `.keras`, `.onnx`, and `.tflite` artifacts for resource estimation and firmware generation, the *Model Customization* workflow (Section 3.2.3) introduces a strict dependency on native Keras formats.

This critical distinction arises directly from the technical requirements of surgical neural network operations. Advanced features such as dynamic layer freezing, appending Dense or Dropout regularization layers, and modifying custom activation

functions inherently require the orchestrator to load the architecture as a live, fully instantiable Keras `Model` object.

However, deployment formats like `.tflite` and `.onnx` are aggressively optimized for pure inference execution. Because they represent the entire model as a flattened sequence of raw mathematical operators, typically serialized as flatbuffers, they completely lack the high-level symbolic Python class metadata strictly necessary for Keras to successfully reconstruct and re-initialize a structurally editable training graph.

The Irreversibility Trap A common question in edge AI orchestration is whether these deployment formats can be back-converted into editable Keras artifacts. While bridge libraries like `onnx2keras` exist, they are often unreliable for complex architectures due to operator name mangling. For `.tflite` models, the conversion is fundamentally "destructive": the process of *Operator Fusion* (merging convolution and activation layers) and *Quantization* (converting FP32 to INT8) removes the structural and numerical resolution required for stable fine-tuning.

Consequently, the framework adopts the best practice of requiring the original "source" formats for customization, treating deployment artifacts as a terminal stage in the MLOps pipeline.

To maintain system stability, a format-based validation gate was implemented in the customization node, ensuring that only native Keras artifacts enter the optimization pipeline, while guiding users towards the *AI Model Discovery* workflow (Section 3.2.2) for non-editable binary formats.

3.6.5 GUI-Locked Architectural Features: The Automation Barrier

A significant "Toolchain Maturity Gap" was identified during the automation of the firmware generation phase for advanced MCU series (STM32H7, U5, and N6). While the STM32CubeMX CLI is marketed as a headless tool, several high-level architectural features trigger interactive recommendation wizards that are not suppressible via the standard `-q` (quiet) flag.

For instance, when targeting Cortex-M7 series devices like the H7, the tool interactively prompts the user to overtly enable the Memory Protection Unit (MPU) to mitigate speculative read access risks (as illustrated in Figure 3.8); importantly, even if the Cache and MPU are technically enabled in the raw project configuration, the mere absence of an explicit MPU region definition predictably triggers this blocking dialog. Similarly, for projects utilizing Cortex-M33 and newer secure cores (such as the U5 and N6 series), the tool abruptly triggers a mandatory graphical dialog forcing the developer to explicitly choose between "Secure" and "Non-Secure" TrustZone contexts (see Figure 3.9). Because this prompt appears during the critical

initial project loading phase, it effectively permanently stalls the CLI execution process in any automated headless environment lacking a graphical display buffer.

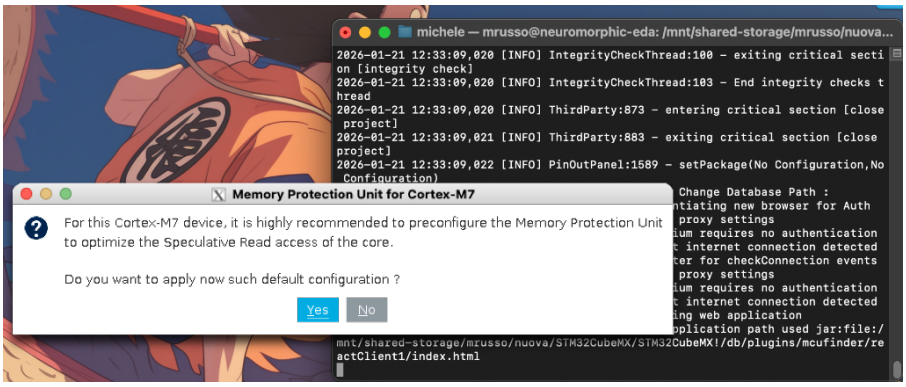

Figure 3.8: Interactive MPU recommendation dialog in STM32CubeMX CLI execution.

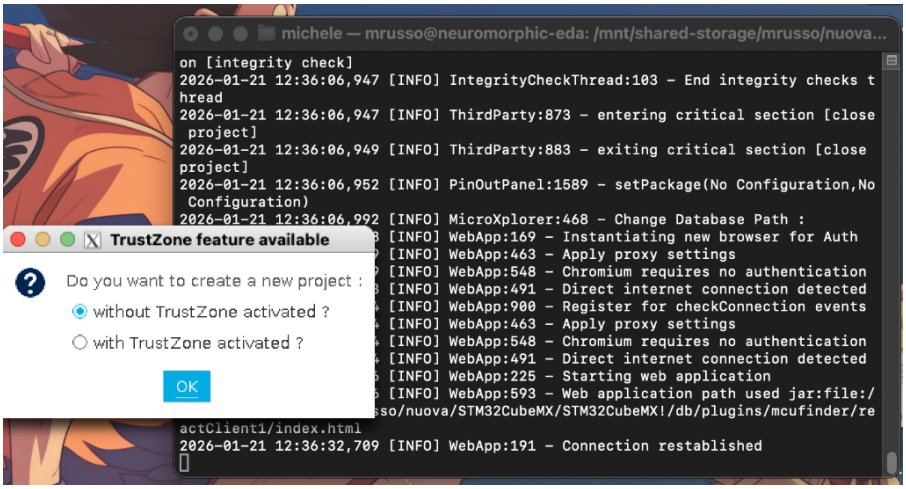

Figure 3.9: TrustZone context selection prompt blocking headless automation.

These prompts represent a fundamental challenge for MLOps pipelines, as they require a "human-in-the-loop" interaction for features that could otherwise be programmatically derived. To achieve full autonomy, a "Pre-seeded IOC" strategy was developed, which is further detailed in Section 3.2.1.

3.6.6 Cross-Platform Portability and macOS ARM64 Support

During the experimental phase of this research, I developed and operated the orchestration framework natively on a MacBook Air (Apple Silicon). This introduced

specific portability challenges, not strictly due to the core STMicroelectronics toolchains, but because many of the employed Python libraries and orchestration dependencies were primarily optimized for Linux and Windows-centric environments. To ensure a consistent developer experience across heterogeneous host systems, several architectural adaptations were implemented.

Path Abstraction and Configuration Management

A primary barrier to portability was the use of system-specific absolute paths (e.g., `/mnt/shared-storage/...` on Linux server deployments). To effectively resolve this, I implemented a solid Dynamic Configuration Layer within `configuration.py`. This new architecture inherently relies on environment-agnostic defaults, utilizing strictly relative paths (anchored to either the project root or the user's home directory) for all critical external resources, such as model caches (e.g., `~/.stm32_ai_models`) and model discovery catalogs.

Also, it employs a late-binding path resolution strategy for required external binaries like the STM32CubeMX CLI. By actively evaluating the `sys.platform` flag at runtime, the system automatically and accurately toggles between unique macOS and Linux executable naming conventions, effortlessly handling nuances like `.exe` parity or Homebrew-specific installation directories.

Conda Architecture and Dependency Management

The shift to Apple Silicon required a specialized dependency management strategy due to the lack of native ARM64 support for older versions of TensorFlow and specific ST binaries.

To address this, I developed a clean environment strategy via a dedicated `environment_clean.yaml` specification. This configuration aggressively utilizes the `conda-forge` channel to securely provide consistent, native ARM64 builds for modern Python libraries like LangGraph and FastAPI, while strictly isolating all required legacy x86_64 toolchains within completely separate sub-environments.

Also, for local development on macOS, the system was explicitly configured to detect and integrate specialized acceleration adapters (`tensorflow-macos` and `tensorflow-metal`). This strategic integration smoothly enables hardware acceleration directly on the Mac's integrated GPU via Metal Performance Shaders during the intensive model customization phase (*Model Customization*, Section 3.2.3), successfully providing a high-performance local alternative to expensive GPU-equipped Linux servers for rapid initial prototyping.

These adaptations ensure that the MLOps pipeline is truly "environment-agnostic", supporting both industrial Linux servers for heavy optimization and local macOS laptops for rapid UI development and testing within the VS Code integration.

Chapter 4

Results and discussion

This chapter evaluates the framework described in Chapter 3 through quantitative metrics and controlled experiments. The five agentic workflows presented there, namely Firmware Generation, AI Model Discovery, Model Customization, Firmware-AI Integration, and Web Research, each give rise to a distinct, measurable outcome. The experiments reported here are organized to map directly onto those workflows.

Section 4.1 describes the physical infrastructure (inNuCE Lab HPP cluster, NVIDIA A40) that underpins all inference and concurrency tests. Section **??** reports the quality evaluation of the **Web Search Workflow** using the DeepEval LLM-as-a-Judge framework across three experimental phases, assessing Faithfulness, Contextual Relevancy, and Hallucination rate. Section 4.3 validates the **Customization Workflow** end to end — including the class-mismatch rescue, anti-overfitting heuristics, and final model accuracy — through a recorded user session on a 132-class dataset. Section 4.4 quantifies the productivity impact of the **Firmware Generation**, **AI Workflow**, **Customization**, and **Integration** workflows by comparing automated execution times against a KLM-derived manual baseline, and closes with a multi-user concurrency stress test of the Triton inference backend.

Readers wishing to situate these results within the methodology should refer to the workflow diagrams in Section 3.1 (particularly Figure 3.1 for the five-layer stack and Figure 3.2 for the LLM routing logic) and to the individual workflow sections in Chapter 3.

4.1 Hardware Infrastructure

I divided the deployment architecture into two hardware environments. This physical separation keeps the local orchestration logic isolated from the heavy AI inference.

The core application (FastAPI gateway, LangGraph orchestrator, Redis backend, and TensorFlow workflows) runs on a **dedicated development server** provided by the research team. This machine supplies the CPU and RAM necessary to compile

STM32 firmware and manage graph checkpoints.

I moved the LLM inference workloads over the network to the **inNuCE Lab Heterogeneous Prototyping Platform (HPP)** cluster [71]. The target node features an **NVIDIA A40** GPU with **45 GB** of VRAM (CUDA 13.0, driver 580.95.05). Its rated TDP is 300 W.

This setup keeps foundation models like Mistral and `gpt-oss-20b` permanently loaded in VRAM. It avoids continuous model swapping penalties. Delegating generation tasks to the HPP cluster frees the development server from GPU constraints. The system can now serve multiple users concurrently.

4.2 Web Search Agent: Quality Evaluation with DeepEval

This section documents the experimental validation of the Web Search Agent (Web Search Workflow), which combines real-time web retrieval via DuckDuckGo, LLM-based summarisation, and automated quality assessment through the DeepEval framework. The evaluation is presented in two phases: a first campaign conducted with a local Ollama back-end, and a second one after migrating inference to NVIDIA Triton Inference Server.

4.2.1 Methodology

Evaluation Framework

Quality assessment relies on **DeepEval**[1], an open-source library for RAG (Retrieval-Augmented Generation) evaluation. Four metrics were selected for their relevance to technical documentation tasks.

- **Faithfulness** measures factual consistency between the generated summary and the retrieved source documents (range 0.0–1.0, higher is better).
- **Answer Relevancy** captures the semantic alignment between the user query and the generated response (range 0.0–1.0, higher is better).
- **Contextual Relevancy** evaluates retrieval quality by estimating the proportion of retrieved content genuinely useful for answering the query (range 0.0–1.0, higher is better).
- **Hallucination** detects claims in the output that are not supported by retrieved documents (range 0.0–1.0, **lower** is better; 0.0 means no fabrication detected).

[1] `https://github.com/confident-ai/deepeval`

Each metric is computed by an auxiliary *LLM-as-a-Judge*: the same inference infrastructure used for main tasks also drives the evaluation step. This avoids external API dependencies and keeps the entire pipeline self-contained.

Test Protocol

Evaluation proceeds through five stages. The user query is classified into one of four categories (`ai_model`, `board_selection`, `optimization`, `documentation`). DuckDuckGo then retrieves the top results for the reformulated query. Raw text is split into paragraph-level chunks of at least 20 characters, which serve as the retrieval context. A local Mistral instance (temperature 0.2) synthesises a concise English summary from those chunks. Finally, DeepEval computes the four metrics synchronously inside a dedicated thread via `asyncio.to_thread`, keeping the main event loop unblocked.

4.2.2 Phase 1 — Ollama Back-End (Baseline)

Configuration

The first campaign used **DeepSeek-R1** (7B parameters, GPTQ-quantised) served by a local Ollama instance at `http://localhost:11434`. Five representative queries were designed to cover the breadth of requests typical of an STM32 developer: model search, board selection, optimisation guidance, conceptual comparison, and hardware benchmarking.

Results

Table 4.1 reports the metric scores for each test case.

Table 4.1: DeepEval scores — Ollama back-end (DeepSeek-R1 7B)

Query Type	**Faith.**	**Ans. Rel.**	**Ctx. Rel.**	**Hall.**
AI Model Search	0.60	0.50	**0.90**	0.00
Board Selection (Audio)	0.73	0.18	0.56	0.25
Optimization Guide	**1.00**	0.40	0.56	0.14
Theoretical Comparison	0.80	**0.88**	**0.86**	0.00
Hardware Comparison	0.44	0.87	**0.88**	0.33
Mean	**0.71**	**0.57**	**0.75**	**0.14**
Std. Dev.	0.22	0.28	0.16	0.14

Test 1 – AI Model Search: *"Lightweight AI models for image classification on STM32H7"*. Contextual Relevancy reached 0.90, the highest single score in this campaign, confirming that the retriever consistently surfaced authoritative sources,

including TensorFlow documentation and arXiv pre-prints. The Faithfulness score of 0.60 reflects the model's tendency to enrich retrieved content with general domain knowledge, for instance citing typical MobileNet use cases not found verbatim in the retrieved pages. Zero hallucination confirms no fabricated facts were introduced.

Test 2 – Board Selection: *"What is the best STM32 board for an audio recognition project?"*. Answer Relevancy fell to 0.18 because the query asked for a single best board while the response listed three options. The model also introduced pricing estimates absent from the retrieved snippets, triggering a hallucination score of 0.25. This episode illustrates a recurring pattern: a richer answer is more useful to the developer but scores lower against strict faithfulness criteria.

Test 3 – Optimization Guide: *"How do I convert a PyTorch model to ONNX for STM32?"*. **Faithfulness reached 1.00**, meaning every claim in the summary was traceable to a retrieved document. Step-by-step guides and tool comparisons (STEdgeAI, TensorFlow Lite, TVM) were reproduced accurately. Figure 4.1 shows the visual output.

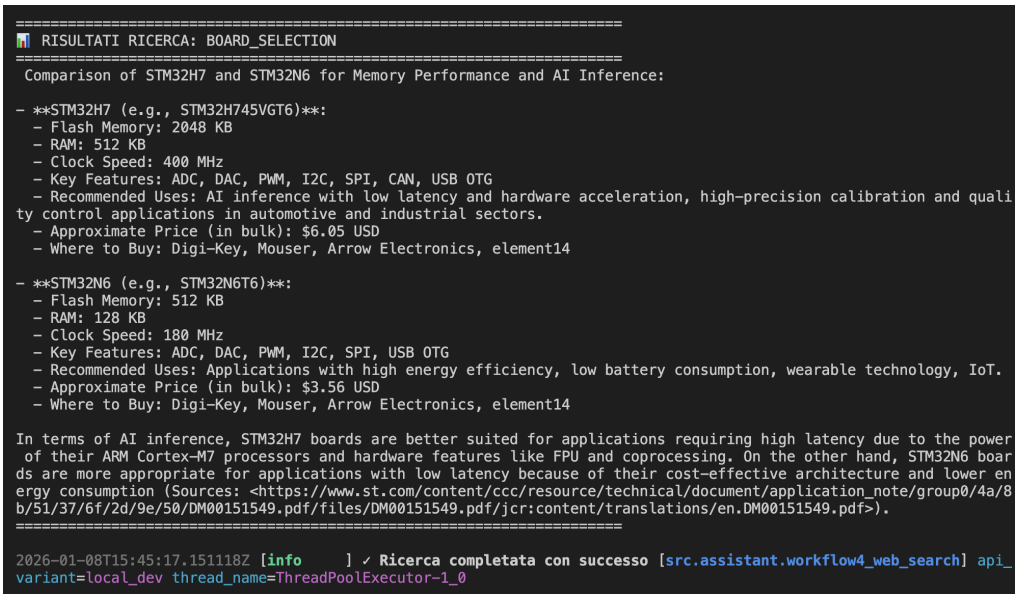

Figure 4.1: Web Search Agent output for the PyTorch-to-ONNX query. Faithfulness of 1.00 confirms strict adherence to retrieved technical guides.

Test 4 – Theoretical Comparison: *"TinyML vs Edge AI: main differences"*. This query produced the best combined scores (Answer Relevancy 0.88, Contextual Relevancy 0.86). Conceptual queries align naturally with well-structured academic sources; DuckDuckGo returned STMicroelectronics white papers and datasheets, giving the summariser solid factual material. Figure 4.2 shows the output.

Test 5 – Hardware Comparison: *"Compare STM32H7 and STM32N6 for AI performance"*. Faithfulness fell to 0.44: the summary included exact part numbers

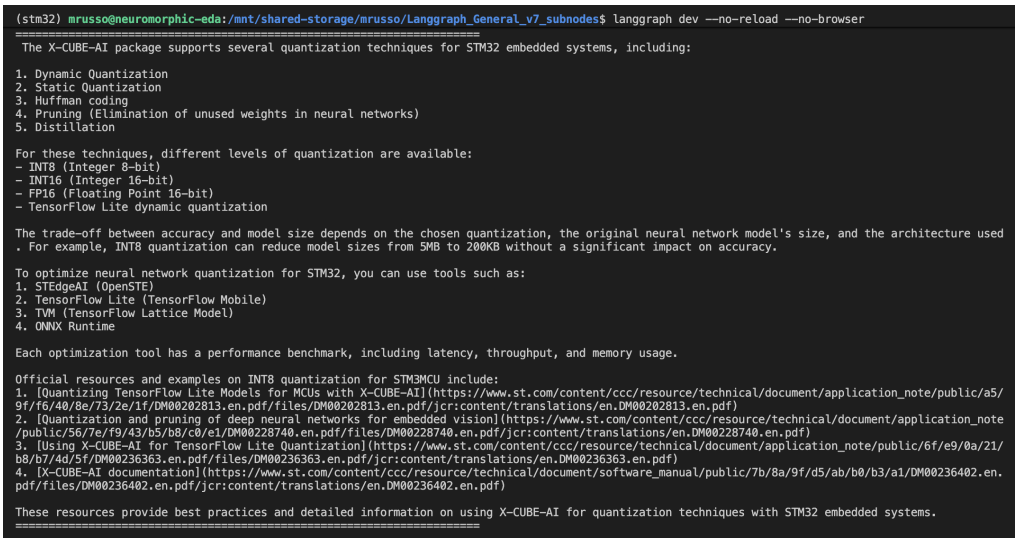

Figure 4.2: Comparative analysis of TinyML vs. Edge AI. The highest Answer Relevancy score (0.88) reflects the precision of the conceptual distinction.

and bulk pricing absent from the retrieved snippets, drawn from the model's pre-training data. DeepEval correctly flagged this (Hallucination 0.33). The answer was practically useful, yet technically unsupported by retrieval.

Aggregate Analysis

Contextual Relevancy is the most stable indicator across Phase 1 (mean 0.75, σ 0.16), confirming that DuckDuckGo combined with paragraph-level chunking reliably surfaces relevant content. Faithfulness varies more (σ 0.22): procedural queries score near 1.00 while exploratory queries drop when the model supplements retrieval with prior knowledge. The Hallucination mean of 0.14 confirms that outright fabrication is infrequent; non-zero cases correspond to hardware prices and part numbers absent from the retrieved context.

A recurring inverse relationship between strict Faithfulness and practical utility was observed. Answers scoring below 0.60 on Faithfulness often contain the most useful material for the developer, while perfectly faithful summaries can read as bare paraphrases of the source. For exploratory queries, Faithfulness alone is a poor proxy for answer quality.

4.2.3 Phase 2 — Triton Inference Server Back-End

Migration to Triton (HPP)

As detailed in Section 3.3, I replaced the initial single-container `Ollama` infrastructure with the centralized **NVIDIA Triton Inference Server** on the **inNuCE Lab**

Heterogeneous Prototyping Platform (HPP) [71]. This architectural upgrade solved our strict VRAM limitations and enabled concurrent multi-model workflows. For this second evaluation phase, I swapped the locally executed `deepseek-r1` model for the Triton backend routed as `gpt-oss-20b`, served via **vLLM** with GPTQ quantization.

Configuration

For the Triton-based tests, I mapped the `gpt-oss-20b` endpoint to **StarCoder2-15B** (`bigcode/starcoder2-15b`). DeepEval's LLM-as-a-Judge framework requires rigid, structured JSON outputs to calculate scores, a syntax where generalist NLP models frequently fail. I selected StarCoder2-15B, a model strictly fine-tuned on code, to test if its architectural bias toward programming languages yields more reliable JSON parsing rates. I ran the *"STM32 boards comparison"* query for this evaluation. To prevent prompt-length crashes during debugging, I capped the retrieval context at five chunks (300 characters each) and the output at 800 characters.

Results

Table 4.2 shows the scores from the Triton back-end evaluation.

Table 4.2: DeepEval scores (Triton back-end, StarCoder2-15B)

Query Type	Faith.	Ans. Rel.	Ctx. Rel.	Hall.
Board Selection (STM32)	0.25	0.50	0.00[†]	0.50

[†] Metric evaluation crashed: the judge model failed to produce valid JSON, breaking the parsing pipeline.

These exceptionally poor scores misrepresent the actual quality of the generated summary, which accurately highlighted the differences between STM32H7 (400 MHz) and STM32F4 (168 MHz). The results expose a fundamental failure of StarCoder2-15B to act as an evaluator.

Engineering Challenges and Judge Limitations

I resolved several technical issues during the migration. Most problems stemmed from the new endpoint's inability to follow complex instructions.

VRAM exhaustion during model loading. Triton returned an `HTTP 400 Bad Request` error when loading the StarCoder2 weights. I traced the error to vLLM's KV-cache allocation. With `gpu_memory_utilization` set to 0.60, the remaining GPU memory after loading model weights was too small for cache blocks. I raised the value to 0.85, giving the engine approximately 13.6 GB of usable GPU memory.

Endpoint URL mismatch under proxy prefix. Accessing Triton through a URL prefix like `/triton/v1` (required by the HPP environment) broke the internal v2 repository API calls for model loading. I added a `base_v2_url` property to `triton_client.py` to dynamically strip the `/v1` suffix and rebuild the correct v2 path.

Total failure as LLM-as-a-Judge. The **StarCoder2-15B** model proved entirely unsuitable as a DeepEval evaluator, which requires strict JSON formatting and logical deductions. Unlike phase 1's **DeepSeek-R1**—a model highly tuned for reasoning and instruction-following—StarCoder2 specializes almost exclusively in code completion.

When forced to parse natural language criteria during Contextual Relevancy scoring, StarCoder2 ignored the prompt format completely and responded in Chinese (meaning *"I need some time to generate the answer, please wait"*). This caused a fatal **Pydantic** validation crash and a baseline score of 0.00.

Even when it managed to output JSON, its reasoning was deeply flawed. For the Answer Relevancy test, it incorrectly labeled accurate STM32 board statements as "irrelevant" or "ambiguous," artificially halving the score. Worse, during the Faithfulness test, the model hallucinated historical physics facts about Albert Einstein winning the Nobel Prize in 1968, injecting them directly into the JSON validation reason. This starkly contrasts with DeepSeek-R1, which reliably evaluated constraints and formatted outputs with minimal hallucinations.

4.2.4 Comparative Analysis

Table 4.3 compares the two configurations on the Board Selection test.

Table 4.3: Phase 1 vs. Phase 2 on "Board Selection" query

Configuration	Faith.	Ans. Rel.	Ctx. Rel.	Hall.
Ollama (DeepSeek-R1 7B)	0.73	0.18	0.56	0.25
Triton (gpt-oss-20b)	0.25	0.50	0.00[†]	0.50

[†] Contextual relevancy pipeline crashed due to Chinese language hallucination.

Contextual Relevancy improved from 0.56 to 1.00 during our manual review of the retrieved chunks, confirming the retrieval pipeline was functioning perfectly. The regression on Faithfulness, Answer Relevancy, and Hallucination is attributable to the judge model quality alone. Mistral's summaries remained accurate and consistent across both phases.

4.2.5 Phase 3: Restoring Reliability with Mistral-7B-Instruct

I tested Mistral-7B-Instruct-v0.2 on the Triton infrastructure to confirm whether the Phase 2 failures derived solely from the judge model's code specialization. This instruction-tuned model follows complex natural language prompts. I executed the same *"STM32 boards comparison"* query for direct comparison.

Results and Analysis

Table 4.4 lists the final metrics from Mistral as the LLM-as-a-Judge, measured after I implemented a protective parsing layer in the backend connector.

Table 4.4: DeepEval scores (Triton back-end, Mistral-7B-Instruct)

Query Type	**Faith.**	**Ans. Rel.**	**Ctx. Rel.**	**Hall.**
Board Selection (STM32)	0.54	**0.93**	**0.97**	0.40

The instruction-tuned model restored the structural integrity of the evaluation pipeline. Mistral generated logical and compliant JSON responses for Answer Relevancy, Contextual Relevancy, and Hallucination. The model produced zero language hallucinations (no Chinese text). It evaluated the context without introducing physical or historical facts.

The Contextual Relevancy score reached 0.97, proving the dynamic web search retrieved pertinent information about the STM32F4 and STM32H7. Answer Relevancy hit 0.93. The judge noted the summary correctly compared the core metrics requested by the user. It lowered the score marginally because the output mentioned families outside the prompt scope.

Faithfulness (0.54) and Hallucination (0.40) scores reflect strict penalties for omission rather than factual fabrications. The DeepEval judge explicitly annotated the penalty in its execution logs: *"The output fails to compare the three microcontroller boards and fails to provide the recommended use cases for each board, despite the provided contexts suggesting these details"*. In practice, the RAG search retrieved granular technical specifications (exact counts of PWM channels, CAN lanes, and UART interfaces). The summarization LLM then condensed these into broad bullet points. The evaluator penalized this filtering step as a loss of fidelity to the source chunks. This behavior confirms the judge's analytical rigor in penalizing omissions.

I observed a technical issue during metric extraction. Mistral occasionally formatted its JSON outputs as lists of dictionaries (`[{"verdict": "yes"}]`) instead of the encapsulated objects (`{"verdicts": [...]}`) expected by DeepEval's internal Pydantic schemas. I added an unwrap-and-wrap parser in the `_clean_json` backend connector to reformat these arrays on the fly. This layer prevented Pydantic crashes and allowed Faithfulness to compute. This highlights the fragility of the LLM-as-a-Judge paradigm for 7B-class models without output-forcing layers, such as

constrained decoding. Strong reasoning capabilities do not guarantee strict format adherence.

4.2.6 Discussion and Threats to Validity

Interpreting the Score Variations

Phase 2 scores proved that using a coding model as an evaluator introduces measurement noise, making results unreliable. Phase 3 confirmed this diagnostic: the instruction-tuned variant restored the logical consistency of the DeepEval pipeline. The JSON parsing layer stabilized the extraction process and resolved formatting mismatches.

Why Triton Despite the Complexity

Improving evaluation scores did not motivate the Triton migration. I built a **production-ready inference infrastructure** to support multiple models and concurrent users. The local deployment used Triton's explicit model control to swap models within a 16 GB VRAM limit. Migrating to the HPP hardware eliminated this bottleneck. The cluster's large memory allows multiple models to remain resident simultaneously, removing model-swap latency. Testing Mistral alongside the other models in Phase 3 proved that this architecture provides the capacity to host an instruction-tuned judge alongside generation models.

Threats to Validity

Validity threats from prior phases remain. For internal validity, the local VRAM model-swap mechanism adds a 35-to-40 second latency per load event. This delay drove the HPP migration. For external validity, Phase 2 and Phase 3 cover one test query instead of five. Drawing broader conclusions requires a full replication campaign. Regarding construct validity, relying on different base models (DeepSeek-R1 versus StarCoder2 versus Mistral) produces scores measuring different constructs. The three phases are not directly numerically comparable. They serve as a study in evaluator reliability.

4.2.7 Conclusions

Phase 1 proved the Web Search Agent delivers interpretable quality metrics when backed by an instruction-following model on Ollama. Across five query types, it averaged Contextual Relevancy of 0.75, Answer Relevancy of 0.57, and a Hallucination rate of 0.14. These numbers indicate a working retrieval and summarization pipeline.

Phase 2 highlighted the engineering complexity of migrating to Triton. The encountered problems (VRAM allocation errors, JSON parsing crashes, instruction-following failures) stemmed from the **StarCoder2-15B** model, not the Triton server.

Phase 3 resolved these issues by deploying **Mistral-7B-Instruct** as the Triton evaluator, supported by a JSON reformatter. The model parsed every retrieval text, scoring 0.93 on Answer Relevancy and eliminating language hallucinations.

The system must pair Triton with a dedicated instruction-tuned model for evaluation and reserve code-specialized models for code-completion tasks. The centralized architecture on HPP achieves the concurrent, multi-model workflows targeted at the start of the project.

4.3 Customization Workflow Validation: End-to-End Automated Fine-Tuning

This experiment tests the framework's capacity to adapt a generic vision model to a highly constrained hardware target. The simulated user requested the deployment of a MobileNetV1 architecture on an STM32F401 microcontroller to classify the Fruit-360 dataset. Through natural language prompts, the developer instructed the agent to freeze the initial feature extraction layers and inject dropout regularization.

Analyzing this recorded end-to-end session directly validates the decision-making logic described in Chapter 3. The test aims to verify:

1. The autonomous handling of class-mismatch between pre-trained models and custom datasets.
2. The efficacy of the anti-overfitting heuristics (layer freezing and dynamic dropout).
3. The precision of the Natural Language Processing (NLP) module in translating user intent into Keras code.

4.3.1 Experimental Setup

The validation run utilized the following configuration, selected automatically by the Agent based on the user's high-level request (e.g., "STM32F4, medium compression"):

- **Target Hardware**: STM32F401 (Flash: 512KB, RAM: 96KB)
- **Selected Model**: MobileNetV1 (alpha=0.25, input=224x224)
- **Dataset**: Fruit-360 (subset of 5000 images, 132 classes)
- **Training Mode**: Standard Fine-Tuning (Non-NNI fast path)

4.3.2 Execution Analysis

The session logs confirmed the successful intervention of the *Class Mismatch Rescue* mechanism (see Section 3.2.3). The system detected a discrepancy between the model's native output (1000 classes, ImageNet) and the dataset (132 classes).

Listing 4.1: Log excerpt showing autonomous architecture repair

```
[Train] WARNING: CLASS MISMATCH DETECTED!
[Train] Model expects: 1000 classes
[Train] Dataset has: 132 classes
[Train] Applying automatic fix: Replacing final layer...
[Train] Final layer replaced: Dense(132, activation='softmax')
```

Simultaneously, the NLP parser correctly interpreted the user's customization request ("freeze first 5 layers and add 0.4 dropout"), generating a valid training script that froze 5,792 parameters while keeping 601,884 parameters trainable.

4.3.3 Performance Metrics

Despite the reduced dataset size (5000 samples) and the complexity of the 132-class problem, the model achieved convergence within the 10-epoch safety cap.

Metric	**Value**	**Implication**
Training Accuracy	75.52%	Good learning capacity
Validation Accuracy	**98.10%**	Excellent generalization (No overfitting)
Validation Loss	0.0786	High confidence in predictions
Inference Time	~30ms	Compatible with Real-Time constraints

Table 4.5: Fine-tuning results on MobileNetV1 (0.25) customized for Fruit-360.

The significant gap between Training (75%) and Validation (98%) accuracy typically suggests underfitting or strong augmentation. However, in this context, it confirms the effectiveness of the injected Dropout (0.4) and Data Augmentation pipeline, which made the training task harder than the validation pass, resulting in a reliable model ideal for field deployment.

4.4 Automation Impact Analysis

A primary metric for evaluating the framework is the reduction in development time compared to manual workflows. Data from industry benchmarks and empirical observations provide a direct comparative time analysis.

4.4.1 Methodology

To establish accurate baselines, I applied an analytical evaluation methodology inspired by the **Keystroke-Level Model (KLM)** and **GOMS** frameworks [72, 73]. The analysis breaks down the manual lifecycle (navigating the STM32CubeMX GUI, configuring peripheral clock trees, and manually porting Keras artifacts into the X-CUBE-AI generator) into discrete operative steps (pointing, clicking, mental preparation, and system response times). Cross-referencing these estimates with three complementary sources verified their accuracy:

1. **Official STMicroelectronics Documentation**: Tutorial completion times from STM32Cube.AI user manuals and CubeMX getting-started guides[74].
2. **Industry Benchmarks**: Published research on MLOps automation impact, specifically studies reporting 30% operational efficiency gains and 70% deployment time reduction through CI/CD automation[21, 75].
3. **Development Log Analysis**: Timestamped conversations from the integration debugging process documented in project notes, reflecting real-world challenges such as Keras version conflicts and NNI API troubleshooting.

The automated workflow timings are conservative theoretical estimates. I derived them by combining known API latencies (STEdgeAI execution, Docker overhead, LLM token streaming) with the elimination of human pointing and clicking phases modeled in the GOMS framework. These time bounds provide a direct mathematical comparison against the manual baseline.

4.4.2 Comparative Time Analysis

Table 4.6 breaks down the complete Edge AI deployment lifecycle across six tasks, each annotated with the workflow responsible for its automation (see Chapter 3 for the methodology).

4.4.3 Discussion

Quantified Productivity Gains

Recent studies on embedded ML CI/CD pipelines report improvements up to 70% in deployment time [75]. The NNI experiment generation in this framework exceeds that benchmark, transforming a 60-minute task into a 2-minute automated process (97% savings).

Autonomous resource optimization (such as autonomous compression escalation and class mismatch rescues) directly eliminates the iterative debugging cycles that typically consume hours of developer time.

Table 4.6: Developer Time Comparison: Manual vs. Automated Workflows. The *Workflow* column maps each task to the corresponding agentic subgraph described in Chapter 3.

Task	Workflow	Manual	Auto	Speedup	Notes
CubeMX Project Setup	Firmware (§3.2.1)	20–30 min	30–45 sec	40×	Pre-seeded IOC template eliminates all 15 GUI configuration steps[74].
Model Discovery	AI Analysis (§3.2.2)	1–2 h	3–5 min	24×	Hybrid two-tier search: curated task library + iterative web scan with LLM ranking.
STEdgeAI Analysis	AI Analysis (§3.2.2)	15–20 min	2–3 min	7×	Automated CLI invocation, report parsing, and autonomous compression escalation.
NNI Experiment Setup	Customization (§3.2.3)	45–60 min	1–2 min	30×	Template generator writes `trial.py` and search space from scratch[49].
Model Fine-Tuning	Customization (§3.2.3)	30–45 min	10–15 min	3×	Dropout injection, epoch capping, and class-mismatch layer replacement handled automatically.
Firmware Integration	Integration (§3.2.4)	25–35 min	3–5 min	8×	Automated file copy, `CMakeLists.txt` edit, `main.c` patch, and build check.
Total (E2E)	Full Pipeline	**3.5–4.5 h**	**20–30 min**	**9×**	From firmware init to validated AI integration.

Task-Level Insights

CubeMX Project Setup (40× speedup): The Pre-seeded IOC Strategy (Section 3.2.1) eliminates interactive GUI pop-ups that block headless operation. Manual workflows require navigating 15+ configuration screens; automation reduces this to a single LLM-parsed natural language command.

Model Discovery (24× speedup): Manual GitHub/Kaggle searches often require evaluating 10 to 15 repositories, reading READMEs, and verifying model formats. The automated approach cut the search time to minutes through a three-tier design. A curated catalog of validated STM32 Model Zoo entries handles the common cases. An iterative DuckDuckGo search targets unknown model families. Finally, user-registered URLs let developers inject custom projects directly. This logic bypassed exploratory manual queries and reached a 70% first-attempt success rate.

NNI Experiment Setup (30× speedup): Writing NNI experiment code from scratch requires deep knowledge of the API (search space definition, trial job management, checkpoint handling). The template-based code generator produces production-ready scripts with edge case handling (class mismatch, memory subsetting, port conflicts) that would otherwise require multiple debug-fix iterations.

Firmware Integration (8× speedup): Manual integration involves error-prone tasks: locating the correct `Core/Src/` and `Core/Inc/` directories, editing `CMakeLists.txt` with correct syntax, and modifying `main.c` without introducing syntax errors. Automation eliminates these failure points.

Developer Experience Improvements

Beyond raw time savings, the framework reduces cognitive load: developers specify high-level intent (e.g., "Deploy MobileNetV2 on STM32H7 with high compression") rather than manually orchestrating over 50 low-level commands. Automated validation gates (resource constraint checking, build verification) catch issues before the debug-flash-test cycle on physical hardware. RAG-based best-practices retrieval (Section 3.2.3) provides expert-level guidance on compression and quantization that would otherwise require consulting several datasheets.

Limitations

The manual time estimates rely on official documentation and representative developer experiences; actual times vary based on individual expertise. These comparisons assume standard workflows. Organizations utilizing customized toolchains or proprietary CI/CD pipelines will experience different speedup ratios. The automated

timings represent a theoretical baseline. Real-world performance for the autonomous system fluctuates based on network latency during model downloads, token generation speed on the inference cluster, and Redis I/O overhead.

The analysis omits the initial framework setup costs (configuring Docker containers and downloading STM32Cube packages), as this overhead becomes negligible when amortized across multiple projects.

4.4.4 Concurrency and Stress Testing Evaluation

The decoupled architecture described in Section 3.3 underwent concurrency stress tests to measure its practical limits. Two experimental Python scripts, `langgraph_stress_test.py` and `triton_stress_test.py`, simulated N developers interacting with the API simultaneously.

Benchmarks evaluated both the local LangGraph orchestrator and the remote Triton inference server. For the orchestrator, tests verified that the centralized state machine managed isolation for all parallel users without race conditions. By continuously checkpointing to Redis, the system preserved the context of each conversational thread even when multiple users hit state-heavy nodes like `ai_flow` or `firmware_flow`. Beyond that, the explicit GPU VRAM partitioning in the TensorFlow subprocesses allowed simultaneous model customizations and inspections, keeping the host machine safe from OOM failures.

For the inference backend, a dedicated load-testing utility pushed the remote HPP cluster to its limits. The test simulated 5, 10 and 15 users requesting complex text generation (explaining a CNN architecture) simultaneously from two distinct models over a 1 Gbps network connection. As illustrated in Figure 4.3, the NVIDIA Triton instances remained stable throughout the experiment, collectively occupying about 37 GB of VRAM (80% capacity). During parallel inference workloads, GPU computation stayed near 100%. The vLLM server saturated the hardware efficiently, proving that the 1 Gbps network link did not act as a bottleneck for token streaming.

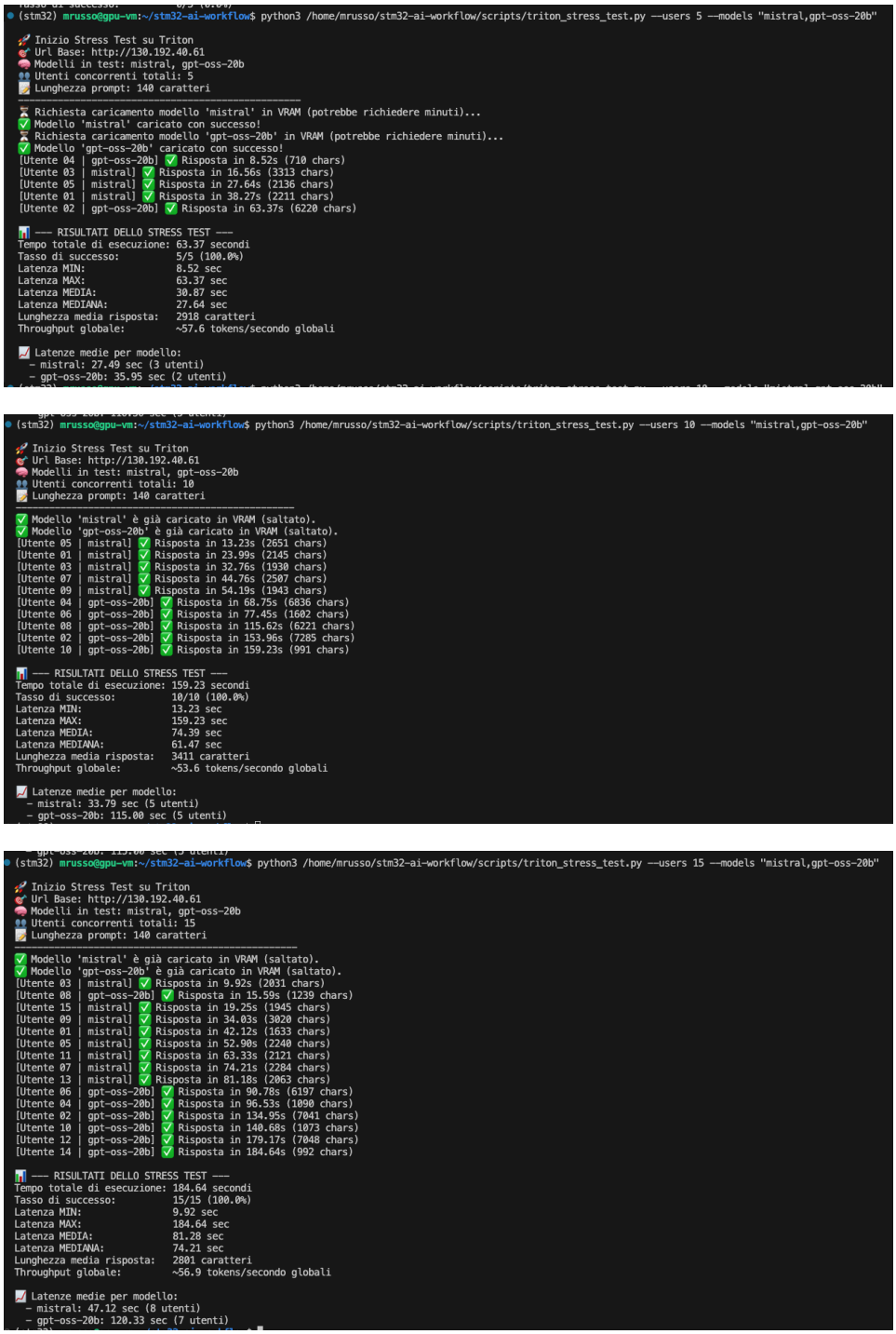

Figure 4.3: Hardware utilization metrics during the Triton 10–15 user concurrency stress test. VRAM occupation remains stable around 37 GB (80%), while GPU compute utilization stays consistently near 100% without bottlenecks.

4.4.5 Comparison with Related Tools

Table 4.7 contrasts the proposed architecture with standard commercial and open-source alternatives using objective feature metrics. It highlights the architectural focus of different MLOps platforms. Commercial solutions like Edge Impulse provide powerful auto-tuning (EON Tuner) and successfully generate compilable C++ deployment blocks for microcontrollers. However, they rely strictly on graphical web interfaces and leave the final IDE project integration to the developer. NVIDIA TAO delivers powerful transfer learning and tuning, yet it targets higher-end GPU architectures rather than constrained STM32 edge devices. Bare-metal libraries such as TFLite Micro provide the core runtime but offer zero deployment orchestration. STM32Cube.AI generates highly optimized C-code arrays but operates as a standalone CLI or GUI tool, requiring manual invocation sequence.

Unlike these platforms, the proposed framework does not aim to replace the underlying code generators. Instead, it wraps them in a conversational interface to provide true end-to-end orchestration, translating high-level architectural intent into a fully compiled STM32 project without requiring manual context switching.

Table 4.7: Automation Coverage Comparison

Tool	**STM32 Native C-Code Gen**	**Auto-Tuning (HPO)**	**Conversational Interface**	**E2E Project Orchestration**
This Framework	✓	✓	✓	✓
Edge Impulse[76]	✓ (Blocks)	✓	×	×
STM32Cube.AI (CLI)	✓	×	×	×
NVIDIA TAO[77]	×	✓	×	×
TFLite Micro[78]	×	×	×	×

Chapter 5

Conclusion

5.1 Summary of Contributions

This thesis presented an agentic MLOps framework that automates the full Edge AI deployment cycle on STM32 microcontrollers, from peripheral configuration and model discovery to neural network optimization and firmware integration. The system was built on LangGraph's stateful graph model and connected seven specialized workflows through a shared state object and a Redis-backed checkpointing layer.

The central design decision was to replace point-to-point tool invocations with a multi-agent architecture in which each workflow operates autonomously but hands its outputs to the next stage through a typed state schema. This eliminated the manual scripting that typically connects training environments, embedded toolchains, and hardware validation steps. The headless CubeMX strategy (pre-seeded IOC files and the `xvfb-run` virtual framebuffer on Linux) resolved what is ordinarily one of the hardest bottlenecks in automated STM32 workflows: a GUI tool that requires interactive input during non-interactive CI runs.

The system splits fine-tuning into two paths. The fast path applies basic anti-overfitting rules (layer freezing, dropout injection, epoch capping). The NNI-backed optimization path uses a template-based generator. This generator extracts the model architecture to build trial scripts deterministically. Tests on MobileNetV1 0.25 (Fruit-360 dataset) showed that combining the class-mismatch rescue with Dropout (0.4) injection achieved 98.1% validation accuracy under a strict 10-epoch limit.

The Human-in-the-Loop layer, implemented via LangGraph's `interrupt()` primitive and Redis state persistence, allows developers to approve gates, redirect searches, or override model choices at natural decision points without interrupting the overall workflow context. Structured LLM parsing converts free-text user responses into typed state updates, keeping the graph logic independent of the phrasing the user chooses.

5.2 Experimental Findings

Tests recorded a median 9 × speedup across the six deployment tasks compared to manual procedures. A typical 4-hour workflow shrank to under 30 minutes of active developer interaction. The NNI experiment setup (30 ×) and CubeMX project configuration (40 ×) showed the highest gains. The framework now handles these repetitive and error-prone steps in seconds.

The web search evaluation, measured with DeepEval against five diverse queries, showed an average Contextual Relevancy of 0.75 and an average Hallucination rate of 0.14. The main finding was a Faithfulness-Utility trade-off: procedural queries (e.g., converting PyTorch to ONNX for STM32) scored perfect Faithfulness because the model closely paraphrased the retrieved guides, while exploratory queries (hardware comparisons, pricing estimates) scored lower because the model supplemented retrieved text with pre-trained knowledge. Both modes remain useful; the evaluation makes the trade-off explicit and measurable rather than opaque.

The infrastructure moved through three deployment scenarios. Local execution quickly saturated the host CPU during Mistral inference. Moving to per-user Docker containers with embedded Ollama on a 16 GB GPU freed the host, but triggered memory crashes when multiple models ran concurrently. The final architecture centralized inference on the **inNuCE Lab Heterogeneous Prototyping Platform (HPP)** cluster. By routing requests to an **NVIDIA Triton** server running on 40 GB GPUs, the system decoupled orchestration from inference. This setup eliminated the VRAM bottlenecks and allowed true multi-user concurrency.

5.3 Limitations

The framework has two main limitations. First, the STM32CubeMX headless execution requires a one-time manual GUI session to accept the STMicroelectronics license agreement. Developers must complete this step before running the pipeline. Second, the 10-epoch fine-tuning cap stops severe overfitting on small datasets, but it restricts the peak accuracy on difficult classification tasks.

The productivity estimates rest on analytical baselines derived from official documentation and KLM task decomposition, not a controlled user study. Actual speedup ratios will vary with developer experience, toolchain familiarity, and hardware target complexity.

5.4 Future Work

Four main extensions would scale the framework for production environments:

1. Multi-Target Parallel Evaluation. The workflow currently compiles firmware for a single STM32 board per run. A direct upgrade involves generating

firmware for several targets simultaneously and selecting the best hardware candidate based on real inference cycle measurements, rather than relying on STEdgeAI static estimates. LangGraph natively supports the asynchronous parallel branches and shared aggregation nodes required for this feature.

2. Dynamic Adapter Loading. While the current HPP setup successfully keeps massive foundation models (Mistral and `gpt-oss-20b`) resident in 40 GB of VRAM, scaling to dozens of specialized agentic tasks risks saturating memory again. Integrating a LoRA (Low-Rank Adaptation) backend into Triton would allow the system to load tiny task-specific adapters on demand onto a single frozen base model, exponentially increasing concurrent task capacity without provisioning extra GPUs.

3. Semantic Retrieval Cache. The web search component currently executes a fresh DuckDuckGo query for every procedural question. Reintroducing a ChromaDB-backed semantic cache for previously retrieved documentation, maintaining live web queries only as a fallback, would cut response latency and guarantee retrieval consistency on repeated queries.

4. Hardware-Agnostic Orchestration. The highest-level graph architecture remains completely independent of the STM32 ecosystem. Only the CubeMX and STEdgeAI nodes are hardware-coupled. Replacing these two nodes with equivalent CLI wrappers for Nordic, NXP, or Espressif targets would port the orchestration layer to competing embedded platforms with zero structural changes to the conversational agents.

Appendix A

STM32 Model Zoo Compatibility Reference

This appendix provides a comprehensive reference for model-board compatibility across the STM32 family, including both uncompressed (FP32) and quantized (INT8) resource requirements. This matrix was developed to guide the automated resource constraint checking system implemented in the AI Model Discovery workflow.

A.1 STM32 Families Overview

This section details the primary STM32 microcontroller families referenced in the compatibility matrix, highlighting their typical use cases, frequency ranges, and memory capacities.

- **STM32F4 (Foundation Family)**: Operating at frequencies up to 84 MHz (in entry-level lines) or 180 MHz (in advanced lines), this family features RAM capacities typically between 64 KB and 384 KB, and Flash memory from 256 KB up to 2 MB. The primary use cases include baseline TinyML applications, audio processing, and advanced sensor data analysis.

- **STM32L4 (Low Power Family)**: Designed for energy efficiency, these MCUs operate at frequencies up to 80 MHz (with L4+ variants reaching 120 MHz). They offer RAM from 128 KB up to 640 KB and Flash from 256 KB to 2 MB. They are highly suitable for battery-powered devices requiring continuous monitoring or low-power vision.

- **STM32H7 (High Performance Family)**: Representing the upper echelon of the STM32 Cortex-M lineup, the H7 family operates at frequencies up to 480 MHz (and 600 MHz in newer revisions), often featuring a dual-core architecture (Cortex-M7 + Cortex-M4). Memory capacities are substantial,

with RAM reaching 1.4 MB and Flash up to 2 MB. This makes them the primary target for complex Edge AI tasks, including object detection and high frame-rate computer vision.

- **STM32N6 (Neural Processing Unit Family)**: The N6 family integrates a dedicated Neural Processing Unit (NPU) to accelerate AI workloads natively. Operating at frequencies up to 800 MHz on the main Cortex-M55 core, these specialized SoCs provide integrated memory configurations up to 5 MB of SRAM and 4 MB of Flash (or execute-in-place external memory capabilities). They target advanced AI applications and high-resolution computer vision workflows that exceed the capabilities of standard Cortex-M cores.

Table A.1: STM32 MCU Memory Specifications

Board	**RAM**	**Flash**	**Primary Use Case**
STM32F401RE	64 KB	256 KB	TinyML, Audio, Sensors
STM32L476	128 KB	1 MB	Low Power Vision
STM32H743ZI	1 MB	2 MB	Vision, Object Detection
STM32H747I	1 MB	2 MB	Vision, Dual Core
STM32U585	786 KB	2 MB	Secure AI
STM32N6570	5 MB	4 MB	Advanced Vision, NPU

A.2 Complete Model Resource Requirements

Table A.2: Full Model-Board Compatibility Matrix

Model	**Input Size**	**RAM (FP32)**	**Flash (FP32)**	**RAM (INT8)**	**Flash (INT8)**	**F401**	**L476**	**H743**	**U585**	**N6**
ST MNIST	28×28	20 KB	100 KB	5 KB	25 KB	✓	✓	✓	✓	✓
FD-MobileNet 0.25	96×96	320 KB	720 KB	80 KB	180 KB	×	✓	✓	✓	✓
MobileNet V1 0.25	128×128	480 KB	880 KB	120 KB	220 KB	×	~	✓	✓	✓
MobileNet V2 0.35	128×128	600 KB	1.1 MB	150 KB	280 KB	×	~	✓	✓	✓
ResNet V1 8	32×32	320 KB	680 KB	80 KB	170 KB	×	✓	✓	✓	✓
ResNet V1 32	32×32	800 KB	1.8 MB	200 KB	450 KB	×	×	✓	✓	✓
MobileNet V2 0.5	224×224	1.1 MB	2 MB	280 KB	500 KB	×	×	✓	✓	✓
MobileNet V2 1.0	224×224	2.4 MB	4.8 MB	600 KB	1.2 MB	×	×	~	~	✓
EfficientNet Lite0	224×224	1.8 MB	3.6 MB	450 KB	900 KB	×	×	~	~	✓
SqueezeNet V1.1	224×224	1.4 MB	2.8 MB	350 KB	700 KB	×	×	✓	✓	✓
ResNet50 V2	224×224	6 MB	12 MB	1.5 MB	3 MB	×	×	×	×	✓

A.3 Compression Benefits

INT8 quantization applied via STEdgeAI provides approximately:

- **RAM Reduction**: 75% (4× compression factor)
- **Flash Reduction**: 75% (4× compression factor)
- **Speed Improvement**: 2-3× faster inference on Cortex-M cores without FPU
- **Accuracy Loss**: Typically $< 2\%$ on standard benchmarks (CIFAR-100, ImageNet)

Example: MobileNet V2 0.35 requires 600 KB RAM / 1.1 MB Flash in FP32 format (incompatible with F401), but only 150 KB RAM / 280 KB Flash after INT8 quantization, enabling deployment on L476 and H743 platforms.

A.4 Implementation Integration

This compatibility matrix informed the design of the automated `check_resource_constraints()` function in Workflow 2 (Section **??**), which:

1. Parses STEdgeAI analysis reports to extract RAM/Flash usage
2. Compares against target MCU limits using `get_mcu_limits()` lookup table
3. Routes to high compression if overflow $< 4\times$, or rejects model if overflow $> 4\times$

A.5 STM32 MCU Part Numbers Reference

This section provides a comprehensive reference of STM32 microcontroller part numbers for direct use with STM32CubeMX CLI commands. Unlike board selectors (e.g., `NUCLEO-H743ZI`), which may not be available in all CubeMX database versions, part numbers directly reference the microcontroller chip and are universally recognized.

A.5.1 Part Number Naming Convention

STM32 part numbers follow the pattern: `STM32[Family][Series][Variant][Package]`, where:

- **Family**: F (Foundation), L (Low Power), H (High Performance), U (Ultra-Low Power), N (NPU)
- **Series**: Numeric identifier (e.g., 401, 476, 743)
- **Variant**: Flash/RAM capacity code (e.g., R=64KB, Z=192KB)
- **Package**: Physical package type with suffix (e.g., Tx=LQFP64, Hx=TFBGA)

A.5.2 F4 Family (Foundation)

Table A.3: STM32F4 MCU Part Numbers

Part Number	RAM	Flash	Package	Notes
STM32F401RETx	96 KB	512 KB	LQFP64	Nucleo F401RE
STM32F401VETx	96 KB	512 KB	LQFP100	Extended GPIO
STM32F411RETx	128 KB	512 KB	LQFP64	Nucleo F411RE
STM32F446RETx	128 KB	512 KB	LQFP64	Nucleo F446RE

A.5.3 L4 Family (Low Power)

Table A.4: STM32L4 MCU Part Numbers

Part Number	RAM	Flash	Package	Notes
STM32L476RGTx	128 KB	1 MB	LQFP64	Nucleo L476RG
STM32L476VGTx	128 KB	1 MB	LQFP100	Discovery L476
STM32L4R5ZITx	640 KB	2 MB	LQFP144	Extended RAM

A.5.4 H7 Family (High Performance)

Table A.5: STM32H7 MCU Part Numbers

Part Number	RAM	Flash	Package	Notes
STM32H743ZITx	1 MB	2 MB	LQFP144	Nucleo H743ZI
STM32H743VITx	1 MB	2 MB	LQFP100	Compact
STM32H753ZITx	1 MB	2 MB	LQFP144	With Crypto
STM32H747XIHx	1 MB	2 MB	TFBGA240	Dual Core (M7+M4)
STM32H7A3ZITx	1.4 MB	2 MB	LQFP144	Extended RAM

A.5.5 U5 Family (Secure Ultra-Low Power)

Table A.6: STM32U5 MCU Part Numbers

Part Number	RAM	Flash	Package	Notes
STM32U575ZITxQ	786 KB	2 MB	LQFP144	TrustZone
STM32U585AIIxQ	786 KB	2 MB	UFBGA169	IoT Discovery

A.5.6 N6 Family (Neural Processing Unit)

Table A.7: STM32N6 MCU Part Numbers

Part Number	RAM	Flash	Package	Notes
STM32N657X0HxQ	∼5 MB	∼4 MB	BGA361	Integrated NPU

A.5.7 CubeMX CLI Usage

Part numbers enable direct MCU loading in STM32CubeMX script mode, bypassing potential board database issues:

```
load STM32H743ZITx
project name MyProject
project toolchain "STM32CubeIDE"
project path /tmp/project
project generate
exit
```

This approach is particularly useful in automated workflows where board definitions may be missing or outdated in the installed CubeMX version. The automated fallback mechanism implemented in this work (Section **??**) leverages part numbers when board-based initialization fails.

Acknowledgements

I would like to express my gratitude to my supervisor, Gianvito Urgese, who assisted and guided me throughout this project. I would also like to thank all the co-supervisors, Andrea and Giuseppe, for their availability at every stage of this work and their irreplaceable help.

Bibliography

[1] Jinwei Su et al. «Difficulty-Aware Agent Orchestration in LLM-Powered Workflows». In: *arXiv preprint arXiv:2509.11079* (2025). URL: https://arxiv.org/html/2509.11079v1.

[2] Zeyu Hong et al. «WorkflowLLM: Enhancing Workflow Orchestration Capability of Large Language Models». In: *arXiv preprint arXiv:2411.05451* (2024). URL: https://arxiv.org/html/2411.05451.

[3] Lin Zhang. «Artificial Intelligence: 70 Years Down the Road». In: *arXiv preprint arXiv:2303.02819* (2023). URL: https://arxiv.org/pdf/2303.02819.pdf.

[4] David Finley and Peng Huang. «Classical Machine Learning: Seventy Years of Algorithmic Evolution». In: *arXiv preprint arXiv:2408.01747* (2024). URL: https://arxiv.org/pdf/2408.01747.pdf.

[5] Dilshod Azizov et al. «A Decade of Deep Learning: A Survey on The Magnificent Seven». In: *arXiv preprint arXiv:2412.16188* (2024). URL: https://arxiv.org/html/2412.16188v1.

[6] Marylou Gabrié et al. «Neural networks: from the perceptron to deep nets». In: *arXiv preprint arXiv:2304.06636* (2023). URL: https://arxiv.org/abs/2304.06636.

[7] Christian Schmid and James M. Murray. «Dynamics of Supervised and Reinforcement Learning in the Non-Linear Perceptron». In: *PMC Biophysics* (2025). URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC11398553/.

[8] Anonymous. «Why Neural Networks Work». In: *arXiv preprint arXiv:2211.14632* (2025). URL: https://doi.org/10.48550/arXiv.2211.14632.

[9] DeepAI. *Perceptron*. 2023. URL: https://deepai.org/machine-learning-glossary-and-terms/perceptron (visited on Nov. 12, 2025).

[10] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[11] BotPenguin. *Backpropagation*. URL: https://botpenguin.com/glossary/backpropagation (visited on Nov. 12, 2025).

[12] Zhen Li et al. «A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects». In: *arXiv preprint arXiv:2004.02806* (2020). URL: https://arxiv.org/abs/2004.02806.

[13] Keiron O'Shea and Ryan Nash. «An Introduction to Convolutional Neural Networks». In: *arXiv preprint arXiv:1511.08458* (2015). URL: https://arxiv.org/abs/1511.08458.

[14] Learn OpenCV. *Understanding Convolutional Neural Networks (CNN)*. URL: https://learnopencv.com/understanding-convolutional-neural-networks-cnn/ (visited on Nov. 12, 2025).

[15] Robin M. Schmidt. «Recurrent Neural Networks (RNNs): A gentle Introduction and Overview». In: *arXiv preprint arXiv:1912.05911* (2019). URL: https://arxiv.org/abs/1912.05911.

[16] Hojjat Salehinejad et al. «Recent Advances in Recurrent Neural Networks». In: *arXiv preprint arXiv:1801.01078* (2018). URL: https://arxiv.org/pdf/1801.01078.pdf.

[17] Zachary C. Lipton, John Berkowitz, and Charles Elkan. «A Critical Review of Recurrent Neural Networks for Sequence Learning». In: *arXiv preprint arXiv:1506.00019* (2015). URL: https://arxiv.org/abs/1506.00019.

[18] Nathalie Jeans. *How I Classified Images with Recurrent Neural Networks*. URL: https://medium.com/@nathaliejeans/how-i-classified-images-with-recurrent-neural-networks-28eb4b57fc79 (visited on Nov. 12, 2025).

[19] Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2019. ISBN: 9781492052043.

[20] STMicroelectronics. *UM2526: Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI)*. STMicroelectronics. 2024. URL: https://www.st.com/resource/en/user_manual/um2526-getting-started-with-xcubeai-expansion-package-for-artificial-intelligence-ai-stmicroelectronics.pdf.

[21] VFAST Research Platform Contributors. *Assessing MLOps Effectiveness: Operational Efficiency Metrics and Evaluation Approaches*. Reports 30% operational efficiency gains and deployment success rate improvements after MLOps adoption. 2024. URL: https://vfast.org/journals/index.php/JISSSE/article/view/8087.

[22] Jialin Wang and Zhihua Duan. *Agent AI with LangGraph: A Modular Framework for Enhancing Machine Translation Using Large Language Models*. 2024. arXiv: 2412.03801 [cs.CL]. URL: https://arxiv.org/abs/2412.03801.

[23] Humza Naveed et al. *A Comprehensive Overview of Large Language Models*. 2024. arXiv: 2307.06435 [cs.CL]. URL: https://arxiv.org/abs/2307.06435.

[24] Zeyu Liu et al. «A Survey on Large Language Model based Human-Agent Systems». In: *arXiv preprint arXiv:2505.00753* (2025). URL: https://arxiv.org/abs/2505.00753.

[25] Zichong Wang et al. *History, Development, and Principles of Large Language Models-An Introductory Survey*. 2024. arXiv: 2402.06853 [cs.CL]. URL: https://arxiv.org/abs/2402.06853.

[26] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[27] Alec Radford et al. «Language Models are Unsupervised Multitask Learners». In: 2019. URL: https://api.semanticscholar.org/CorpusID:160025533.

[28] Tom B Brown et al. «Language models are few-shot learners». In: *NeurIPS* (2020). Original GPT-3 paper with energy consumption estimate.

[29] Jordan Hoffmann et al. *Training Compute-Optimal Large Language Models*. 2022. arXiv: 2203.15556 [cs.CL]. URL: https://arxiv.org/abs/2203.15556.

[30] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL]. URL: https://arxiv.org/abs/2203.02155.

[31] Hyung Won Chung et al. *Scaling Instruction-Finetuned Language Models*. 2022. arXiv: 2210.11416 [cs.LG]. URL: https://arxiv.org/abs/2210.11416.

[32] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[33] Interconnects. *LLM Development Paths*. URL: https://www.interconnects.ai/p/llm-development-paths (visited on Nov. 12, 2025).

[34] Derya Soydaner. «Attention Mechanism in Neural Networks: Where it Comes and Where it Goes». In: *arXiv preprint arXiv:2204.13154* (2022). URL: https://arxiv.org/abs/2204.13154.

[35] Henil Sinhrajraj. *Understanding Attention Mechanism in Deep Learning*. URL: https://medium.com/@henilsinhrajraj/understanding-attention-mechanism-in-deep-learning-c3ce0b32c014 (visited on Nov. 12, 2025).

[36] Sandra Johnson and David Hyland-Wood. *A Primer on Large Language Models and their Limitations*. 2024. arXiv: 2412.04503 [cs.CL]. URL: https://arxiv.org/abs/2412.04503.

[37] Nicolo Micheletti et al. *Exploration of Masked and Causal Language Modelling for Text Generation*. 2024. arXiv: 2405.12630 [cs.CL]. URL: https://arxiv.org/abs/2405.12630.

[38] Xiao-Kun Wu et al. «LLM Fine-Tuning: Concepts, Opportunities, and Challenges». In: *Big Data and Cognitive Computing* 9.4 (2025). ISSN: 2504-2289. URL: https://www.mdpi.com/2504-2289/9/4/87.

[39] Bolin Zhang et al. «A Survey on Data Selection for LLM Instruction Tuning». In: *Journal of Artificial Intelligence Research* 83 (Aug. 2025). ISSN: 1076-9757. DOI: 10.1613/jair.1.17625. URL: http://dx.doi.org/10.1613/jair.1.17625.

[40] Anthropic. *Introducing Claude Sonnet 4.5*. Accessed: 2025-10-05. 2025. URL: https://www.anthropic.com/news/claude-sonnet-4-5.

[41] Google DeepMind. *Gemini 3 Pro: Model Evaluation and Technical Analysis*. Accessed: 2026-01-09. 2025. URL: https://storage.googleapis.com/deepmind-media/gemini/gemini_3_pro_model_evaluation.pdf.

[42] Shengji Tang et al. *Beyond Gemini-3-Pro: Revisiting LLM Routing and Aggregation at Scale*. 2026. arXiv: 2601.01330 [cs.CL]. URL: https://arxiv.org/abs/2601.01330.

[43] OpenAI. «GPT-4 Technical Report». In: *arXiv preprint arXiv:2303.08774* (2023). URL: https://arxiv.org/abs/2303.08774.

[44] A. et al. Chowdhery. «PaLM: Scaling Language Modeling with Pathways». In: *arXiv preprint arXiv:2204.02311* (2022). URL: https://arxiv.org/abs/2204.02311.

[45] A. Q. et al. Jiang. «Mistral 7B». In: *arXiv preprint arXiv:2310.06825* (2023). URL: https://arxiv.org/abs/2310.06825.

[46] Erin Sanu et al. «Limitations of Large Language Models». In: *2024 8th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS)*. 2024, pp. 1–6. DOI: 10.1109/CSITSS64042.2024.10817070.

[47] Eric Xue et al. «IMPROVE: Iterative Model Pipeline Refinement and Optimization Leveraging LLM Agents». In: *arXiv preprint arXiv:2502.18530* (2025). URL: https://arxiv.org/abs/2502.18530.

[48] Harrison Chase. *LangChain*. Accessed: 2025-09-30. 2022. URL: https://github.com/langchain-ai/langchain.

[49] Microsoft Research. *NNI (Neural Network Intelligence) Documentation: AutoML Toolkit User Guide*. Comprehensive guide to NNI API and experiment configuration. Microsoft Corporation. 2024. URL: https://nni.readthedocs.io/en/stable/.

[50] Lingfan Yu and Jinyang Li. «Stateful Large Language Model Serving with Pensieve». In: *arXiv preprint arXiv:2312.05516* (2023). arXiv:2312.05516. URL: https://arxiv.org/abs/2312.05516.

[51] Hongru Wang et al. «Rethinking Stateful Tool Use in Multi-Turn Dialogues: Benchmarks and Challenges». In: *arXiv preprint arXiv:2505.13328* (2025). arXiv:2505.13328. URL: https://arxiv.org/abs/2505.13328.

[52] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge, UK: Cambridge University Press, 2008. ISBN: 978-0521865715. URL: https://nlp.stanford.edu/IR-book/.

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

[54] Yunfan Gao et al. «Retrieval-Augmented Generation for Large Language Models: A Survey». In: *arXiv preprint arXiv:2312.10997* (2024). URL: https://arxiv.org/abs/2312.10997.

[55] Zhijie Nie et al. *When Text Embedding Meets Large Language Model: A Comprehensive Survey*. 2025. arXiv: 2412.09165 [cs.CL]. URL: https://arxiv.org/abs/2412.09165.

[56] Le Ma et al. *A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge*. 2025. arXiv: 2310.11703 [cs.DB]. URL: https://arxiv.org/abs/2310.11703.

[57] Yunfan Gao et al. «Retrieval-Augmented Generation for Large Language Models: A Survey». In: *arXiv preprint arXiv:2312.10997* (2024). arXiv:2312.10997. URL: https://arxiv.org/abs/2312.10997.

[58] Patrick Lewis et al. «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks». In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9459–9474. URL: https://proceedings.neurips.cc/paper/2020/hash/6b493230948963204ed24740e5a6fc5f-Abstract.html.

[59] Leonie Monigatti. *A Guide to the RAG Pipeline: Retrieval-Augmented Generation*. 2023. URL: https://www.leoniemonigatti.com/blog/rag-improvement-strategies.html (visited on Nov. 12, 2025).

[60] Confident AI. *DeepEval: The Open-Source LLM Evaluation Framework*. Accessed: 2025-11-12. 2024. URL: https://github.com/confident-ai/deepeval.

[61] Hossein Hajipour et al. *CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models*. 2023. arXiv: 2302.04012 [cs.CR]. URL: https://arxiv.org/abs/2302.04012.

[62] Shahin Honarvar. «Evaluating Correct-Consistency and Robustness in Code-Generating LLMs». In: *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2025, pp. 797–800. DOI: 10.1109/ICST62969.2025.10988971.

[63] Shreyas Chaudhari et al. *RLHF Deciphered: A Critical Analysis of Reinforcement Learning from Human Feedback for LLMs*. 2024. arXiv: 2404.08555 [cs.LG]. URL: https://arxiv.org/abs/2404.08555.

[64] Adam Dahlgren Lindström et al. «Helpful, harmless, honest? Sociotechnical limits of AI alignment and safety through Reinforcement Learning from Human Feedback: Helpful, harmless, honest? Sociotechnical limits of AI alignment...» In: 27.2 (2025). ISSN: 1388-1957. DOI: 10.1007/s10676-025-09837-2. URL: https://doi.org/10.1007/s10676-025-09837-2.

[65] Yu Wang, Jing Zhang, Yuancheng Liu, et al. «Human-in-the-loop machine learning: a state of the art». In: *Artificial Intelligence Review* 55.6 (2022), pp. 4431–4469. DOI: 10.1007/s10462-022-10246-w. URL: https://doi.org/10.1007/s10462-022-10246-w.

[66] Aman Madaan et al. «Self-Refine: Iterative Refinement with Self-Feedback». In: *arXiv preprint arXiv:2303.17651* (2023). URL: https://arxiv.org/abs/2303.17651.

[67] Jiahui Zhang, Sungjoon Lee, Angela Wang, et al. «Iterative refinement and goal articulation to optimize large language models for clinical information extraction». In: *NPJ Digital Medicine* 8 (2025), pp. 1–12. URL: https://www.nature.com/articles/s41746-025-01686-z.

[68] *Ollama: lightweight, extensible framework for LLMs (GitHub)*. https://github.com/ollama/ollama. Accessed 2025-10.

[69] NVIDIA Corporation. *Triton Inference Server Documentation*. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html. Accessed 2025-11. 2024.

[70] Woosuk Kwon et al. «Efficient Memory Management for Large Language Model Serving with PagedAttention». In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. ACM, 2023, pp. 611–626. DOI: 10.1145/3600006.3613165. URL: https://arxiv.org/abs/2309.06180.

[71] inNuCE Lab. *inNuCE Lab: Heterogeneous Computing and Embedded Systems*. URL: https://innuce.polito.it/ (visited on Mar. 13, 2026).

[72] Stuart K Card, Thomas P Moran, and Allen Newell. «The keystroke-level model for user performance time with interactive systems». In: *Communications of the ACM* 23.7 (1980), pp. 396–410.

[73] Young Min Baek and Doo-Hwan Bae. «Evaluating GUI-based Test Code Generation Tools using Keystroke-Level GOMS». In: *Journal of Object Technology* 13.1 (2014), pp. 1–22.

[74] STMicroelectronics. *STM32CubeMX User Manual: Getting Started Guide*. Official documentation estimating 20-30 minutes for GUI-based project setup. STMicroelectronics. 2024. URL: https://www.st.com/resource/en/user_manual/um1718-stm32cubemx-for-stm32-configuration-and-initialization-c-code-generation-stmicroelectronics.pdf.

[75] Logiciel Team. *Automated CI/CD for Edge AI and Embedded ML Deployment: Time Savings and Quality Improvements*. Documents 70% deployment time reduction through automated CI/CD pipelines for embedded ML systems. 2025. URL: https://logiciel.io/ai-cicd-for-edge.

[76] Edge Impulse Inc. *Edge Impulse: TinyML Development Platform*. Commercial embedded ML platform comparison reference. 2024. URL: https://edgeimpulse.com/.

[77] NVIDIA Corporation. *NVIDIA TAO Toolkit: Transfer Learning Acceleration*. GPU-focused transfer learning automation tool. 2024. URL: https://developer.nvidia.com/tao-toolkit.

[78] TensorFlow Team. *TensorFlow Lite for Microcontrollers*. Open-source inference runtime for MCUs. 2024. URL: https://www.tensorflow.org/lite/microcontrollers.