End-to-End Model Generation with Large Language Models for Adaptive IoT Application Deployment

Abstract

The deployment of AI-powered applications on resource-constrained edge devices presents a significant software engineering challenge. While Pruning and Neural Architecture Search (NAS) have shown promise in optimizing model efficiency, their application in edge device deployment is often limited by low levels of automation. In this paper, we introduce LLM-based Adaptive Model Generation (LAMDA), a novel framework that tackle the challenge of adaptive AI deployment as an automated model generation problem. LAMDA empowers an LLM to perform end-to-end design, refinement, and optimization of DNNs to meet the specific hardware constraints. Our approach makes two primary contributions. First, we introduce a serialization technique that transforms complex DNN computation graphs into a structured textual representation that makes model architectures comprehensible and manipulable by an LLM. Further, to ground the generation process in real-world hardware constraints, we integrate a feedback-driven optimization loop. This loop leverages an empirical performance model, trained to correlate architectural patterns with on-device latency, enabling the LLM to reason about and optimize for non-functional requirements. To mitigate architectural "hallucinations", we incorporate context management and validation to ensure valid generation. We evaluate LAMDA through extensive experiments on public benchmarks and real-world edge devices. The results demonstrate that our framework can autonomously generate and adapt DNNs that satisfy deployment-specific accuracy and latency constraints, significantly advancing the state-of-the-art in automated software adaptation for the AI-enabled edge devices.

1 Introduction

Modern software systems are increasingly built by integrating pre-trained Deep Neural Networks (DNNs) as functional components [36], often sourced from large-scale Model-as-a-Service (MaaS) platforms like Hugging Face, which hosts nearly 800,000 models [22, 32]. This trend is prominent in the Internet of Things (IoT) domain, which drives the integration of intelligence into billions of resource-constrained edge devices. While these repositories accelerate development, they present a critical software engineering challenge: a gap between the near-infinite supply of generic AI models and the practical demands of deploying them onto the heterogeneous, resource-constrained edge devices. This creates a significant gap between model availability and practical, performance-aware deployment, where non-functional requirements (NFRs)-such as accuracy, latency and energy consumption-are paramount. For example, deploying a standard object detection model to a pedestrian counter on a Jetson Nano (Fig. 1) is challenging as the model's latency often violates the hardware's operational limits. The IoT intensifies this challenge, as applications like smart surveillance and continuous health monitoring require reliable, low-latency, and energy-efficient operation directly on edge devices. Fulfilling these non-functional requirements is critical for the practicality and user

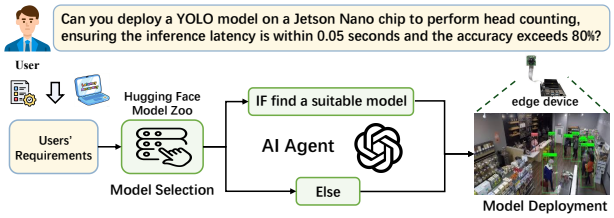

Figure 1: An example of leveraging an LLM-based agent to automate the adaptation and deployment of an AI component (a DNN model) for a real-world edge application.

adoption of such systems. The conventional engineering response is an ad-hoc process of manual model tuning – a trial-and-error cycle of layer pruning or channel resizing [19, 31]. This approach is not only labor-intensive and reliant on scarce expertise but is also fundamentally unscalable. It fails to provide the reliable, automated engineering process needed for the rapid evolution of AI-enabled systems. To resolve this, the community has invested heavily in automated methods like Neural Architecture Search (NAS) [14]. However, NAS and related techniques such as pruning [13, 37] are not a *panacea*. These methods are overwhelmingly search-based, confining their exploration to a predefined, hand-crafted search space. More critically, they optimize for unreliable proxy metrics like FLOPs or parameter counts, which correlate poorly with real-world on-device metrics such as latency and energy consumption. This creates a fundamental disconnection between the optimization objective and the deployment reality, resulting in architectures that are theoretically efficient but practically suboptimal.

These limitations reveal the absence of a tractable method to synthesize a DNN directly from a specification of its hardware-grounded NFRs. This motivates to shift from *searching* for an architecture to *generating* one. This paper investigates a transformative approach by asking: *Can we reframe hardware-aware model adaptation as a generative software synthesis problem, and leverage a Large Language Model (LLM) as the core synthesis engine?*

Challenges. Answering this question requires overcoming multifaceted challenges that lie at the intersection of natural language processing, software engineering, and systems. Using an LLM to synthesize a complex, non-textual software artifact like a DNN is not straightforward as it faces the following challenges: *Semantic Representation.* How do we bridge the modality gap between an LLM's textual world and the graph-based, semantically rich structure of a DNN? Since a naive serialization is insufficient, the LLM must comprehend the architectural semantics, constraints, and the non-linear impact of structure on performance. *Compositional Synthesis.* Real-world DNNs are vast, often exceeding an LLM's context window. How can we generate a large, structurally coherent artifact compositionally, ensuring that locally generated segments assemble into a globally valid and high-performance model? *Grounding and Verification.* LLMs are prone to "hallucination" [33], generating syntactically or semantically invalid code. How do we ground the

LLM's generative process in reality, ensuring its outputs are not only executable but are also rigorously verified against the target hardware and NFRs?

In this paper, we introduce LAMDA, an end-to-end framework that operationalizes this generative paradigm. LAMDA empowers an LLM to function as an automated AI software architect, directly synthesizing and refining DNNs to meet user-defined NFRs. To tackle the challenges above, our framework integrates a multi-stage methodology. We devise a structured textual representation that provides an efficient, bijective (one-to-one) mapping from a neural network to a unique string, making architectures *natively* comprehensible to LLMs. To manage complexity, we leverage this canonical representation within a hardware-aware, compositional generation process. Crucially, we embed this in an automatic closed-loop verification and refinement system, where the LLM's proposals are continuously compiled, profiled on the target device, and the performance feedback is used to guide subsequent generation steps. This loop corrects hallucinations and steers the synthesis process towards NFR-compliant solutions. Our primary contributions are:

(1) An End-to-End Generative Framework for Neural Architectures. We propose a novel framework that shifts the paradigm from searching a fixed space to directly synthesizing hardware-aware DNNs. To our knowledge, this is the first work to treat hardware-aware DNN design as an end-to-end generative software engineering task driven by an LLM.

(2) LLM-Powered, Hardware-Aware Generation Method. We design a methodology that fine-tunes an LLM-using Low-Rank Adaptation (LoRA) and prompt engineering-to serve as a neural architecture generator. The method explicitly trains the LLM to comprehend and reason about the complex, non-linear relationship between architectural patterns and empirical on-device latency.

(3)A Closed-Loop Verification and Refinement System. We introduce an adaptive deployment system with a closed-loop feedback mechanism. This system ensures the generated models are not only syntactically valid and executable but are also rigorously optimized to satisfy user-defined latency and accuracy constraints for the target task and device.

(4) Comprehensive Experimental Validation. We conduct extensive experiments demonstrating that LAMDA generates models that meet latency targets with high precision (0.14 MAPE) and significantly outperform state-of-the-art NAS and pruning methods. For instance, our framework can reduce a model's latency to just 7.7% of its original time while preserving 96.6% of its accuracy, showcasing a superior balance of performance and efficiency.

2 Related Work

Model Pruning. Model pruning techniques aim to reduce a DNN's complexity by removing redundant parameters, thereby creating a more efficient derivative of an existing model [16, 21]. These methods are broadly categorized as unstructured pruning, which zeroes out individual weights but requires specialized hardware for acceleration [20], and structured pruning, which removes entire filters or channels [25]. While effective at reducing parameter counts, pruning is an indirect and often unreliable strategy for satisfying specific performance NFRs. The core limitation is its reliance on parameter count as the primary optimization metric, which, like FLOPs, is an unreliable proxy for on-device latency [37]. Consequently, pruning often fails to systematically navigate the trade-off space between model accuracy and hardware performance, leading to models that either violate deployment constraints or suffer unacceptable accuracy degradation. It is a post-hoc optimization heuristic rather than a generative method for engineering models against first-class performance requirements.

Neural Architecture Search. NAS represents a significant advance to automate the design of DNNs, moving beyond manual engineering. Early methods relied on resource-intensive, multi-trial evaluations [28, 39], while more recent approaches employ weight-sharing supernets to amortize training costs [7, 10, 26]. These methods aim to navigate a predefined design space to find an optimal architecture. However, NAS exhibits two critical limitations. First, it is often a semi-automated process requiring significant manual effort to design the search space and interpret the results, thus falling short of a true end-to-end automation solution. Second, and more fundamentally, there is a disconnect between its optimization objectives and the real-world deployment requirements. NAS typically optimizes for proxy metrics like FLOPs, which are poor indicators of on-device NFRs such as latency or memory bandwidth [13]. This misalignment results in architectures that are theoretically optimal but perform sub-optimally on the target hardware, necessitating further manual, post-hoc refinement.

LLMs for Model Synthesis. Large language models have demonstrated strong capabilities in software automation and AI model design. In code completion, tools such as CodeX [8] and SBLLM [18] can automatically produce function-level code, thereby improving development efficiency. In model design scenarios, recent works like EvoPrompting [6], GENIUS [38], and LLMatic [29] leverage LLMs to generate or iteratively refine neural architectures, marking a promising shift toward generative topology discovery.

However, these LLM-guided generators operate under a fundamentally different paradigm from our LAMDA framework. First, their goal is typically high-level, topology-level discovery—producing novel architecture blueprints—whereas LAMDA focuses on fine-grained, hardware-in-the-loop specialization within a given architecture family. Second, their search space is largely abstract and unconstrained, relying on free-form prompts or textual specifications. By contrast, LAMDA uses a domain-specific language (DSL) to represent architectures with guaranteed syntactic validity and semantic correctness, ensuring every candidate is deployable. Third, their performance feedback is proxy-based (e.g., FLOPs, parameter counts) or entirely open-loop, lacking direct measurement on target hardware. LAMDA instead incorporates closed-loop, hardware-in-the-loop feedback, allowing the system to meet strict numerical constraints such as latency and memory usage. Fourth, their output is a novel blueprint that may require further engineering to deploy, whereas LAMDA directly synthesizes models that are fully constraint-compliant and deployment-ready.

This structured comparison clarifies that LAMDA is complementary to prior LLM-guided generation methods: by combining a DSL with direct hardware feedback, LAMDA uniquely achieves precise, constraint-driven synthesis for deployable AI components—capabilities that these existing approaches are not designed to provide.

3 Problem Formulation

The central challenge we address is the automated synthesis of DNN models, treated as software components, to meet specific NFRs for deployment on resource-constrained IoT hardware. This task requires navigating a vast and complex design space to produce an optimal architectural artifact. Our approach, named LAMDA, is an end-to-end framework that leverages a LLM as the core engine for this synthesis process. The overall pipeline, depicted in Figure 2, automates the generation of a deployable DNN by taking user-defined constraints, such as a maximum latency, and iteratively refining an architecture until these constraints are met.

To enable an LLM to perform this task, we first formulate the problem of learning the relationship between a system's architecture and its performance. The LLM must be trained to understand how a given architectural design will behave when deployed on specific hardware. We model this as learning a mapping $\mathcal{F}$:

$$\mathcal{F} : (\mathcal{A}, \mathcal{S}_d, \mathcal{Q}) \rightarrow y \tag{1}$$

where $\mathcal{A}$ is a structured textual representation of the neural architecture, $\mathcal{S}_d$ encapsulates the target device's specifications, and $\mathcal{Q}$ contains user requirements (e.g., desired accuracy). The function's output, y, is a key performance metric, such as the measured inference latency. A critical prerequisite for this is representing the architecture $\mathcal{A}$ in a format the LLM can process. We achieve this through a systematic serialization process, detailed in Figure 3, which converts the DNN's structure into a parsable text sequence.

The process of teaching the LLM this mapping is achieved by fine-tuning it on a dataset of known architecture-performance pairs. The fine-tuning objective is to find the optimal model parameters, resulting in a specialized LLM, $\mathcal{F}$, that minimizes the prediction error over this training data:

$$\mathcal{F} := \arg\min_{\mathcal{F}} \sum_{i=1}^{K} \mathcal{L}(y_i, \mathcal{F}(\mathcal{A}^{(i)}, \mathcal{S}d^{(i)}, \mathcal{Q}^{(i)})) \tag{2}$$

Here, $\mathcal{L}(\cdot, \cdot)$ is a suitable loss function (e.g., mean squared error) that quantifies the difference between the empirically measured performance y_i and the LLM's prediction. By minimizing this loss across a training dataset $\mathcal{D}$train of K tasks, the LLM becomes a reliable hardware-aware performance predictor.

Once the LLM $\mathcal{F}$ has been endowed with this predictive knowledge, we leverage it to perform the architectural synthesis. Given a new set of requirements for an unseen task or device, encapsulated in the input $\mathcal{X}^j := \mathcal{Q}^{(j)}, \mathcal{S}_d^{(j)}, y_j$, the framework uses the fine-tuned LLM to generate a compliant architecture $\mathcal{A}^{(j)}$:

$$\mathcal{A}^{(j)} = \mathcal{F}(\mathcal{X}^j; \theta) \tag{3}$$

where θ denotes the fine-tuned parameters of the LLM. It is important to note that Equation 3 represents the outcome of an iterative synthesis process, as illustrated in the overall pipeline (Figure 2). The LLM uses its embedded performance knowledge from Equation 2 to propose, evaluate, and refine architectural candidates in a closed loop until the user's constraints are satisfied. This formulation enables the LLM to adaptively generate novel architectures that are engineered for specific device and user needs.

4 LAMDA Framework

The LAMDA framework is a multi-stage pipeline designed to automate the synthesis of deployable DNNs that adhere to specified non-functional requirements, such as inference latency. As illustrated in Figure 2, the framework systematically transforms, optimizes, and verifies a neural network architecture. The process comprises three main stages: 1) LLM-Powered, Hardware-Aware Generation: At the core of the framework is a fine-tuned LLM that acts as a generative engine. It takes the serialized architecture, along with user-defined constraints (e.g., target device, latency budget), and synthesizes a new, optimized architectural description. For very large models, this process can be applied to architectural slices to manage context length. 2) Iterative Refinement and Verification: The generated textual description is deserialized back into a model format for validation. The framework performs syntactic checks to ensure the architecture is well-formed and then enters a closed loop of refinement. This iterative process continues until the generated model is both valid and satisfies all user-defined constraints.

4.1 LLM-based Model Structural Generator

From ONNX to a Structured Textual Representation. We formally define a given ONNX model as a directed graph $G = (V, E)$, where each node $v \in V$ represents a computational operator (e.g., convolution, activation) and each edge $e \in E$ represents the dataflow between operators. Each node v is characterized by its operator type Op, its input and output channel dimensions (C_{in}, C_{out}), and a set of operator-specific parameters θ.

Connection-Oriented Traversal. A naive sequential traversal of the graph's nodes is insufficient for capturing the topology of modern architectures with complex branching and merging paths (e.g., GoogLeNet). Such an approach would lose critical information about node connectivity. To overcome this, we employ a connection-oriented traversal strategy. This method explicitly preserves the model's structural integrity by assigning unique identifiers to the input and output edges of each node. Each node can reference multiple input IDs ($\mathcal{ID}_{in}$) but has a single output ID ($\mathcal{ID}_{out}$). This ensures that all connections, including complex skip-connections and branches, are accurately represented in the final serialized text, preventing any loss of structural information.

4.1.1 Training Data Formulation for the Generative LLM. To empower the LLM to function as a hardware-aware architecture generator, we fine-tune it on a specifically constructed dataset. This process involves careful prompt engineering to teach the LLM the complex relationship between architectural patterns, hardware constraints, and performance outcomes. For each data point in our training set, we formulate a "question-answer" pair.

Question. The Question (Input Prompt) is a carefully structured prompt provided to the LLM. It contains:

- The target constraints, including device specifications $\mathcal{S}_d$, the desired performance metric y (e.g., target latency), and other requirements $\mathcal{Q}$.
- The full serialized architecture $\mathcal{A}$ of an existing model, which serves as the baseline for modification.

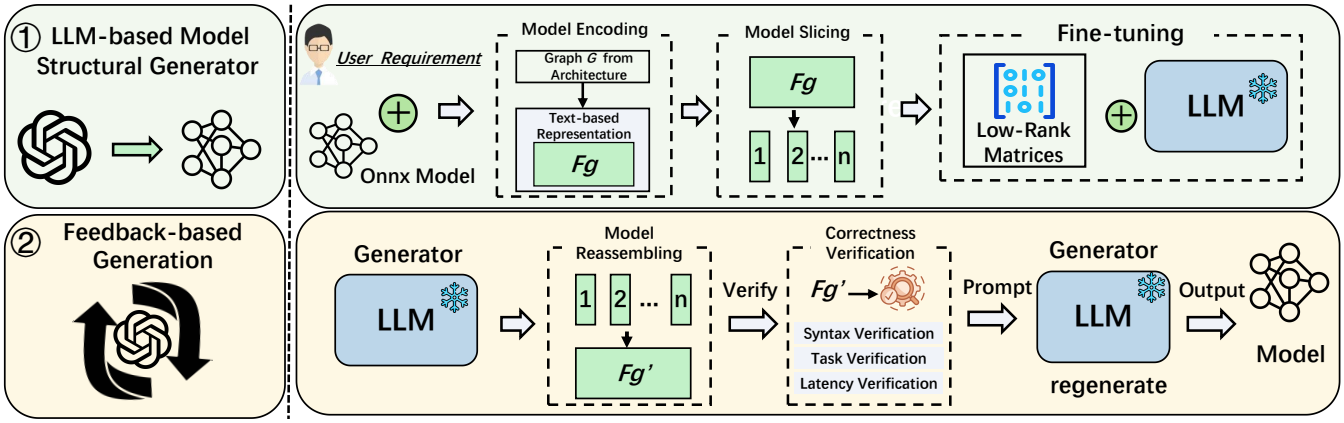

Figure 2: The end-to-end pipeline of LAMDA for the automated, constraint-driven synthesis of a deployable DNN component. The framework uses an LLM, informed by a hardware-aware performance model, to iteratively generate, verify, and refine an architecture until it meets user-defined requirements like latency.

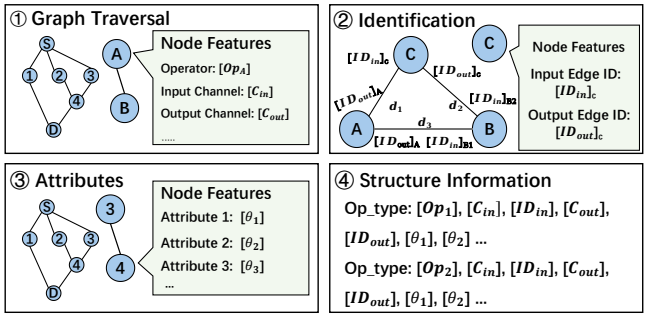

Figure 3: The architecture serialization process. A graph-based DNN is systematically traversed to convert its nodes (layers), attributes, and connectivity into a structured textual representation.

- A set of syntax rules $(\mathbf{c}_0, \mathbf{c}_1, \cdots)$ that define the grammar of a valid architectural representation, ensuring the LLM's output is well-formed.

Answer. The Answer (Output Label) is the serialized representation of a new, optimized architecture $\mathcal{A}'$ that satisfies the constraints specified in the prompt.

By fine-tuning the LLM on these pairs, it learns to transform a given architecture into a new one that meets specific, hardware-grounded performance targets, effectively acting as a constraint-aware software synthesizer.

4.1.2 Managing Context Limitations via Optimal Architectural Slicing. A primary engineering challenge in applying LLMs to large-scale software artifacts is the inherent limitation of their context window. LLMs process information in fixed-size token blocks; for instance, models like GPT-3.0 [4] are limited to 2k tokens, while more recent models like Llama 3 [12] support 8k tokens. The serialized representation of a complex DNN, such as GoogLeNet (which requires 6.6k tokens), can easily exceed these limits, making it impossible to process the entire architecture in a single pass.

To address this, we introduce a method for architectural slicing: partitioning the serialized model representation (F_g) into smaller, syntactically coherent blocks (F_i) that respect the LLM's context length. However, this decomposition introduces a significant new challenge: architectural fragmentation.

The Impact of Architectural Fragmentation. Slicing a model's serialized representation inevitably severs the dataflow dependencies (the edges) that connect the resulting segments. Reconstructing a valid and functional model requires correctly re-establishing these connections. Any mismatch in the input/output channel specifications (C_{in}, C_{out}) during reassembly can introduce subtle but critical semantic errors that corrupt the model's behavior.

The severity of this problem is directly proportional to the number of severed connections. To quantify this effect, we conducted an experiment to measure the impact of fragmentation on the quality of the final synthesized model. We systematically varied the number of cuts applied to the ResNet-18 and SqueezeNet architectures while tasking the LLM with generating a model variant that met a latency constraint of ≤ 1 second.

As shown in Figure 4, there is a clear correlation between the number of broken edges and the degradation in model quality. The y-axis shows the Mean Absolute Percentage Error (MAPE) between the target latency and the measured latency of the generated model. A higher MAPE indicates a greater failure to meet the specified non-functional requirement. This empirical evidence demonstrates that a naive slicing strategy is insufficient; minimizing the number of severed connections is critical to preserving the model's integrity and achieving the desired performance targets.

Optimal Slicing as a Constrained Optimization Problem. Based on our empirical findings, we formulate the architectural slicing task as a constrained optimization problem. The goal is to partition the serialized model in a way that minimizes the number of severed connections (edges) while ensuring that no individual slice exceeds the LLM's maximum token limit, $\mathcal{T}_{\max}$.

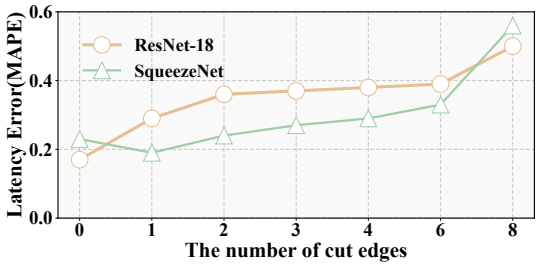

Figure 4: The impact of architectural fragmentation on model synthesis quality. As the number of edges between slices increases, the LLM's ability to generate a model that adheres to the target latency constraint degrades significantly.

To formalize this, let the serialized model be a sequence of N operators. We define the set of potential slice boundaries as the points between these operators.

Token Size of a Slice: Let $T(i, j)$ be a function that returns the number of tokens in a slice starting after operator i and ending after operator j. Cost of a Slice Boundary: Let $C(j)$ be the cost function that returns the number of edges severed if a slice boundary is placed immediately after operator j. The optimization problem is to find a sequence of slice boundaries $p_1, p_2, \ldots, p_k$ that solves:

$$\min \sum_{m=1}^{k} C(p_m) \tag{4}$$

$$\text{s.t.} \quad T(p_{m-1}, p_m) \leq \mathcal{T}_{\max}, \quad \forall m \in 1, \ldots, k \tag{5}$$

where p_0 and p_N is the start and end of the model, respectively.

Dynamic Programming Solution. This optimization problem exhibits optimal substructure, making it well-suited for a dynamic programming (DP) solution. The principle of optimality holds: an optimal partitioning of the first i operators must contain an optimal partitioning of the first j operators for some $j < i$.

We define $E[i]$ as the minimum number of cuts required to partition the prefix of the model containing the first i operators. The value of $E[i]$ can be computed via the following recurrence relation:

$$E[i] = \min_{0 \leq j < i \text{ s.t. } T(j,i) \leq \mathcal{T}_{\max}} \left(E[j] + C(i) \right) \tag{6}$$

The base case is $E = 0$, as no cuts are needed for an empty prefix. The final solution to the problem is given by $E[N]$. To reconstruct the actual slice boundaries, we store the index j that yields the minimum value for each $E[i]$.

Algorithm and Implementation. Algorithm 1 provides the pseudocode for our optimal model slicing method. The algorithm proceeds in two main phases:

Forward Pass (Iterative Computation): It iteratively computes the minimum cut costs for all possible prefixes of the model. It uses an *array*, min_{cuts}, to store the values of $E[i]$ and a second *array*, $prev_{cut}$, to store the backtracking pointers (the optimal choice of j for each i). Backtracking: After the forward pass completes, it reconstructs the optimal sequence of slice boundaries by following the pointers in the $prev_{cut}$ array backward from the end of the model (N) to the beginning.

Algorithm 1 Optimal Model Slicing via Dynamic Programming

Require: Serialized model graph G of length N, max token limit $\mathcal{T}_{\max}$

Initialize:
$min_{cuts} \leftarrow$ array of size $N + 1$, initialized to ∞
$prev_{cut} \leftarrow$ array of size $N + 1$, initialized to -1
$min_{cuts[0]} \leftarrow 0$
Forward Pass: Compute minimum cuts
for $i = 1$ to N **do**
  for $j = 0$ to $i - 1$ **do**
    if $T(j, i) \leq \mathcal{T}_{\max}$ **then**
      cost $\leftarrow min_{cuts[j]} + C(i)$
      if cost $< min_{cuts[i]}$ **then**
        $min_{cuts[i]} \leftarrow$ cost
        $prev_{cut[i]} \leftarrow j$
      end if
    end if
  end for
end for
Backtracking: Reconstruct slice boundaries
boundaries $\leftarrow$ empty list
$p \leftarrow N$
while $p > 0$ **do**
  boundaries.prepend(p)
  $p \leftarrow prev_{cut[p]}$
end while
return $min_{cuts[N]}$, boundaries

Complexity Analysis. The time complexity of the algorithm is determined by the nested loops in the forward pass. The outer loop runs N times, and the inner loop runs up to N times. The initialization and backtracking phases both take $O(N)$ time. Therefore, the overall time complexity of the algorithm is $O(N^2)$, where N is the number of operators in the model. This quadratic complexity is efficient and practical for even very large DNN architectures.

4.1.3 Learning structural information via LLM Fine-tuning. To imbue LLMs with structural knowledge inherent in a given domain, we employ a data-driven low-rank graph adaptation scheme for efficient fine-tuning. Given a training dataset $\mathcal{D}_{\text{train}} = \{(q_i, a_i)\}_{i=1}^{K}$ comprising question-answer pairs, the fine-tuning process aims to minimize a loss function $\mathcal{L}_{\text{ft}}$, quantifying the discrepancy between the predicted answer $\hat{a}$ and the ground truth answer a. Mathematically, this can be expressed as,

$$\mathcal{L}_{\text{ft}} = \text{Func}(a, \hat{a}) \tag{7}$$

where $\text{Func}(\cdot)$ represents a suitable loss function, in this paper we use Mean Squared Error (MSE). Directly fine-tuning all LLM parameters is computationally prohibitive due to their immense size. Therefore, we freeze the pre-trained LLM weights (Φ_0) and introduce a low-rank adaptation method. This method capitalizes on the observation that parameter changes, $\Delta\Phi$, during fine-tuning often reside in a low-rank subspace. For each pre-trained weight matrix $W_0 \in \Phi_0$ with dimensions $n \times p$, we introduce two low-rank matrices $L_1 \in \mathbb{R}^{n \times r}$ and $L_2 \in \mathbb{R}^{r \times p}$, where $r \ll \min(n, p)$. These

matrices approximate the weight update ΔW:

$$\Delta W \approx L_1 L_2 \tag{8}$$

Furthermore, to reduce memory footprint, we quantize the original weights W_0 using a quantization function $\text{deq}(\cdot)$. The updated weight matrix W' is then computed as:

$$W' = W_0 + \Delta W = \text{deq}(W_0) + L_1 L_2 \tag{9}$$

During fine-tuning, only L_1 and L_2 are trained, drastically reducing the number of trainable parameters and thus the computational cost. The rank r controls the trade-off between expressiveness and efficiency of the adaptation.

Model Reassembling. When a model architecture is processed in slices, the LLM generates a corresponding sequence of output segments ($\hat{a}_i$). These segments must be deserialized and integrated to reconstruct the full architectural description. The initial step is concatenation:

$$\hat{a} = \bigoplus i = 1^n \hat{a}_i \tag{10}$$

However, simple concatenation is insufficient to guarantee a valid model. Because each slice is processed independently, the LLM lacks the global context to ensure that the interfaces between the segments remain compatible. This loss of global context can introduce critical semantic errors, such as:

Mismatched tensor dimensions between connected layers. Broken dataflow dependencies (e.g., a layer expecting an input that is no longer produced). Syntactically correct but structurally invalid operator sequences. The result is an architectural description that may be textually coherent but represents a functionally invalid or non-executable model. This challenge highlights the need for a robust, post-generation validation and refinement process. To address this, the next section details our closed-loop feedback framework, which is designed to detect and correct these semantic errors, ensuring the final synthesized artifact is a valid and deployable software component.

4.2 Closed-Loop Verification and Refinement

The initial architectural artifact synthesized by the LLM is not guaranteed to be valid or compliant. To address this fundamental challenge of generative AI, we designed and implemented a closed-loop verification and automated refinement framework, the architecture of which is depicted in Figure 5. This system methodically validates each generated model and, upon detecting a non-conformance, automatically synthesizes a precise, corrective prompt to guide the LLM toward a compliant solution in its next iteration.

The framework operates through two tightly integrated stages: 1) Multi-Stage Validation: The generated artifact is first subjected to a rigorous validation process that checks for three classes of defects: syntactic correctness (e.g., well-formed structure, compatible interfaces), functional correctness (i.e., semantic alignment with task requirements), and non-functional compliance (e.g., adherence to latency constraints). Any detected failure is formally codified into a structured error report using the standardized templates shown in Table 1. 2) Automated Corrective Prompt Generation: This structured error report serves as input to the next stage, which automatically generates a high-fidelity refinement prompt. As detailed in Table 2, this component identifies the error type from the report, selects a corresponding corrective template, and instantiates

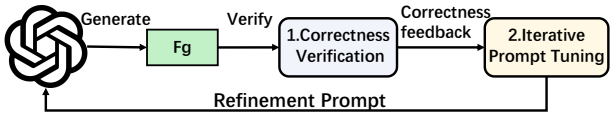

Figure 5: The feedback mechanism for improving the quality of model generation.

Table 1: A standardized template of the Correctness Feedback from the Correctness Verification.

Template
1. Syntax Error: {Op position}: "description of the issue" 2. Task Error: The model's {parameter} is "{current_value}", which does not match the required {expected_value}. 3. Requirement Error: The {requirement} is "{required_value}", but the {actual_metric} is "{actual_value}".

Table 2: A customized prompt for generating refinement prompt by identifying error types and applying the appropriate fine-tuning templates.

Prompt
Identify the type of error from the following inputs {{Correctness feedback}}. Based on the identified error select from the following templates and fill the parameters in the corresponding template: {{Available fine-tuning templates}}. To track the history of the feedback process, the historical output is available as {Chat Logs}.
Fine-tuning templates
1. **Syntax Error**: Locate the "{ordinal}" operation in the "{Chat Logs}" and modify its {parameter} from "{old_value}" to "{new_value}". 2. **Task Error**: Since the task involves {task_description}, update the {parameter} of the final operation in the "{Chat_Logs}" to "{new_value}". 3. **Requirement Error**: The current measured latency {metric} exceeds the requirement by {excess_value}. Ensure that the {aspect} is adjusted to meet the required {target_value}.
Refinement prompt
Extract F_g from {Chat_log} as "history". 1. Locate the "10th" operation in the "history". Change its input_dim from "40" to "50". 2. Since the task involves recognizing 10 different object classes, update the output_dim of the final operation in the "history". to 10. 3. The current measured latency exceeds the requirement by 0.5 seconds. Ensure that the model's latency is reduced to meet the required 1 second.

it with the specific error parameters and the historical context from the "chat logs". The resulting prompt provides a clear, unambiguous instruction for the LLM to repair the defect, creating a robust, self-correcting generation cycle.

4.2.1 Syntactic Validation: Ensuring Architectural Well-Formedness. The first stage of the validation pipeline is syntactic validation, a process analogous to a compiler's static analysis pass. It ensures the generated architecture is structurally sound and adheres to

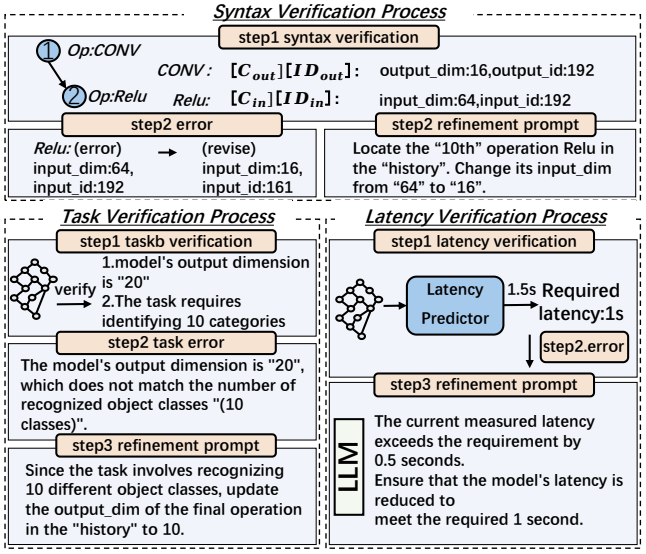

Figure 6: Examples of the *syntax verification*, the *task verification* process and the *latency verification* process.

the predefined grammar of the model representation before any functional or performance evaluation. This validation is critical for ensuring that the artifact is a well-formed component that can be correctly deserialized and analyzed in subsequent stages. The syntactic validator performs several key checks:

- Valid Operator Types: Verifies that all specified operators in the graph are legitimate and supported.
- Attribute Correctness: Ensures that all required attributes for each operator are present and their values are within valid ranges.
- Interface Compatibility: Critically, this check validates that the dataflow connections between nodes are consistent. It confirms that the output tensor dimensions of a node precisely match the expected input dimensions of its successor nodes.

As illustrated in Figure 6, a common fault is a channel mismatch, a specific type of interface incompatibility. The validator detects that a convolutional layer (CONV) with 16 output channels is connected to a ReLU layer expecting 64 input channels. This detected non-conformance triggers the automated feedback loop, which formulates a corrective prompt to guide the LLM toward generating a syntactically valid artifact.

4.2.2 Task Validation: Ensuring Functional Correctness. Once an architecture is deemed syntactically valid, it undergoes semantic validation to ensure it correctly implements the functional requirements of the specified task. This validation stage focuses on the model's primary interface for producing results: the task head. The task head is typically the final trainable layer (e.g., a linear layer) whose output dimensionality must correspond directly to the task's nature (e.g., the number of classes in a classification problem).

The validation process involves comparing the output dimension of the generated model's task head against the ground-truth task specification provided by the user. This ensures that the model's output space aligns with the problem's label space.

For example, as illustrated in Figure 6, consider a task requiring classification into 10 distinct categories. The task validator inspects the generated model and identifies that its final layer has an output dimension of 20. This constitutes a semantic mismatch, as the model is not configured to solve the specified 10-class problem. Upon detecting this non-conformance, the validator flags the error and triggers the automated feedback mechanism, which generates a corrective prompt instructing the LLM to regenerate the model with the correct output dimension.

4.2.3 Non-Functional(Latency) Validation: Enforcing Performance Constraints. The final stage of the validation pipeline verifies that the generated architecture adheres to the specified NFRs, with a primary focus on the inference latency constraint. This step is critical for ensuring that the synthesized artifact is not only functionally correct but also practical for real-world deployment on resource-constrained hardware.

Deserialization for Analysis. A prerequisite for any performance analysis is the transformation of the LLM's textual output from a token-based representation into a structured, analyzable format. This is the role of the Deserializer component. It parses the generated specification, extracting the operators, attributes, and connection topology, and instantiates them as an executable computational graph in a standard framework like PyTorch or TensorFlow. This deserialization step effectively bridges the gap from the generative text domain to the software execution domain.

Efficient Latency Estimation. Executing the model on target hardware for direct latency measurement within a rapid feedback loop would be prohibitively slow, impeding the iterative refinement process. To maintain efficiency, our framework leverages a pre-trained model latency predictor [17, 37]. This component acts as a highly efficient surrogate model, providing a reliable estimation of the on-device performance without incurring the high cost of actual deployment and measurement.

The validation process is exemplified in Figure 6. If a user specifies a latency requirement of 0.8 seconds, but the predictor estimates the generated model's latency to be 1.1 seconds, the validator flags this as a non-functional violation. This failure triggers the automated refinement process, prompting the LLM with explicit feedback to generate a new architecture that satisfies the constraint.

5 Evaluation

In this section, we evaluate the performance of LAMDA in terms of generated model latency, accuracy and LAMDA's inference time. In addition, we present a detailed analysis of the effects of its modules.

5.1 Experiment Setup

Training Dataset. The NNLQP dataset [27] is a open-source dataset that includes a variety of state-of-the-art neural network models, each with one original model and 1,999 variants. Every model is annotated with latency labels, and tested on a specific device. The dataset also features over ten different operators, such as Convolution (Conv), Pooling, and ReLU. We select *seven* most popular DNNs model to evaluate the performance of our LAMDA.

Hyperparameter Settings. The LAMDA is implemented using Python and Bash, with PyTorch [30] chosen as our DNN runnable

Table 3: Performance of generated models under 12 different latency constraints.

Base Model	Accuracy (%)	Latency (ms)	Average Required Latency (ms)	Average Accuracy (%)	Average Generated Latency (ms)	Latency Error (MAE)	Latency Error (MAPE %)	Average Inference Time of LLM (min)	Success Rate (%)
AlexNet	75.0	10.38	3.25	72.0	3.22	0.34	11.0	4.32	83.0
ResNet-18	85.5	18.22	2.57	80.0	3.09	0.77	25.0	11.08	92.0
MobileNet-V2	85.0	29.21	7.78	85.0	7.77	0.94	13.0	26.30	100.0
MnasNet	83.0	34.96	8.78	85.0	8.25	1.16	12.0	25.20	100.0
GoogLeNet	87.0	25.35	4.32	86.0	3.95	0.56	13.0	35.28	100.0
VGG-16	90.0	61.53	4.82	87.0	4.78	0.55	11.0	8.58	92.0
SqueezeNet	82.0	21.16	1.98	76.0	1.91	0.26	13.0	24.30	100.0

format. We employ Qwen-14B [1] as our foundational LLM, a transformer-based [34] model trained on ultra-large-scale pre-training data. For fine-tuning the LLM, we adjust the maximum sequence length to 8K to optimize performance for the generation task, setting the learning rate to 3e-4 and the number of training epochs to 5. For training the generated networks, we use a learning rate of 1e-3 and set the number of training epochs to 30.

Testbeds. Our experimental setup is divided into two parts: the *general evaluation* and the *real-world test-bed evaluation*. The general evaluation is performed on the aforementioned dataset using a server equipped with 12 Intel(R) Xeon(R) Silver 4210R CPUs, 4 NVIDIA Ampere A100 40GB GPUs, and 256GB of RAM. For the real-world test-bed evaluation, we deploy the generated model on a Raspberry Pi 4B demo board with 8GB of RAM to assess the performance of our LAMDA.

LAMDA Training. The model is initially fine-tuned on the NNLQP dataset [27], which comprises seven base models, each accompanied by 1,999 variants. We use Qwen-14B [1] as the base pre-trained model. The fine-tuning is conducted for 5 epochs using the PyTorch framework, with a maximum input context window of 2K tokens and a learning rate of 3×10^{-4}.

Base Model. The accuracy and latency of each base model, which has the largest number of parameters in the NNLQP dataset, are reported in Table 3. These metrics serve as a reference for the models generated by LAMDA.

Latency Constraints. To establish latency constraints for evaluation, the latency distribution of the NNLQP dataset is analyzed, excluding the training set used for fine-tuning. For each base model, 12 latencies are extracted from 12 distinct model-latency pairs.

Evaluation Dataset. A Sub-ImageNet dataset is constructed to evaluate the accuracy and latency of the models produced by our method. The dataset consists of 20 classes, each containing 1,300 training samples and 50 testing samples. All models were trained on this dataset for 30 epochs with a learning rate of 1×10^{-3}.

Evaluation Metrics. *Accuracy (Acc.):* Test **Acc.** is reported on a subset of ImageNet. To mitigate the influence of outliers, we calculate the average **Acc.** over models generated under 12 latency constraints, excluding the maximum and minimum values. *Latency (Lat.) & Error:* We use a latency predictor (NAR-Former2) for rapid evaluation and measure the actual **Lat.** on heterogeneous devices (e.g., Raspberry Pi, laptop). To evaluate precision, we report the error between the measured **Lat.** and the target constraints using Absolute Error (AE), Absolute Percentage Error (APE), and Mean Absolute Percentage Error (MAPE). *Generation Efficiency:* The average inference time taken by LAMDA to generate a model from a base model, averaged over the 12 latency constraints. *Success Rate:* The proportion of successful generations across 12 trials. A generation is considered successful if (1) the **Lat.** MAPE is within 30% of the target, and (2) the **Acc.** degradation does not exceed 5% compared to the reference model.

5.2 Key Findings

We summarize the following key findings: **1) Effective Learning of Architecture-Latency Relationship:** LAMDA successfully learns the complex relationship between a model's architecture and its performance latency on edge devices. **2) High Fidelity on Real-World Devices:** The LAMDA approach demonstrates its effectiveness in practical scenarios, achieving a MAPE of less than 0.05 between the latency predicted in simulation and the actual latency measured on real IoT devices. **3) Superiority over Existing Methods:** When operating under the same latency constraints, models generated by LAMDA consistently achieve higher accuracy compared to those produced by conventional NAS, pruning and quantization techniques.

5.3 Results Analysis

Performance of LAMDA Under Latency Constraints. To evaluate its performance, LAMDA generates models under various latency constraints, with detailed results presented in Table 3. The generated models demonstrate a strong balance between accuracy and latency, achieving an average accuracy of 81.57% on a subset of ImageNet [9] while maintaining a low MAPE of 14% relative to the target latency. The latency error is consistently low, ranging from 11% (for AlexNet and VGG-16) to 25% (for ResNet-18), confirming that LAMDA effectively adheres to specified latency requirements.

While a trade-off for lower latency is typically a slight reduction in accuracy compared to the base models in Table 3, the performance gains are significant. For instance, models derived from AlexNet are more than 3× faster than the reference model. Remarkably, in some cases, LAMDA improves both metrics. When applied to MnasNet, the generated models are not only > 4× faster but also achieve 2% higher accuracy than the base model, highlighting the method's capability to optimize models beyond simple trade-offs.

LAMDA demonstrates a substantial advantage in model generation efficiency, requiring only 35.28 minutes to produce the target

Table 4: Comparison with Pruning Methods on Various Base Models. Bold indicates the best performance.

Base Model	Required Lat. (ms)	Acc. (%) [LAMDA]	Acc. (%)[Geometri [21]]	Acc. (%)[Greg] [35]	Acc. (%)[Group-L1[25]]	Acc. (%)[Group-Pruner[16]]
AlexNet	3.2	**72.0**	59.1	64.8	64.5	58.5
ResNet-18	2.5	**80.0**	79.4	68.3	66.7	74.0
MobileNet-V2	7.7	**85.0**	73.8	69.5	66.8	68.1
MnasNet	8.7	**85.0**	65.3	69.8	67.3	67.1
GoogLeNet	4.3	**86.0**	76.7	79.0	78.7	75.9
VGG-16	4.8	**87.0**	73.2	71.3	70.4	69.8
SqueezeNet	1.9	**76.0**	68.5	68.3	66.8	68.1

Table 5: Comparison of LAMDA and AOWS under Different Latency Requirements.

Latency Constraints	0.9ms	1.5ms	1.8ms	3.5ms	3.9ms	4.3ms	4.8ms	5.1ms
Accuracy (%)[LAMDA]	**79%**	**75%**	**78%**	**87%**	**86%**	**86%**	**87%**	**87%**
Accuracy (%)[AOWS]	39%	50%	48%	63%	61%	62%	64%	64%
Δ Accuracy	+40	+25	+30	+24	+25	+24	+23	+23
Cost of LAMDA	**2.2h**	**2.1h**	**2.1h**	**2.1h**	**2.1h**	**2.5h**	**2.1h**	**2.4h**
Cost of AOWS	4.1h	4.1h	4.2h	4.2h	4.2h	4.3h	4.3h	4.3h

models, whereas ProxylessNAS [5] takes up to 8.3 days. This efficiency stems from LAMDA's reliance on the base model rather than an exhaustive architectural search space, which constitutes a major bottleneck in traditional NAS methods. Moreover, LAMDA exhibits strong reliability, achieving an average success rate of 95.14% across seven base models and reaching 100% success for AlexNet, ResNet-18, and VGG-16.

Comparison with Pruning Methods. LAMDA is compared against four representative pruning approaches—Geometric [21], Group-L1 [25], Greg [35], and Group Pruner [16]. These methods suffer from a key limitation: they require manual specification of pruning ratios, which lack direct control over latency. To enable a fair comparison under latency constraints, we perform an iterative binary search to determine a pruning ratio yielding latency error within 10% of the target.

As detailed in Table 4, the results across seven base models show that LAMDA consistently outperforms all tested pruning methods. The accuracy improvement is particularly striking for models based on AlexNet, MobileNet-V2, MnasNet, and VGG-16, where LAMDA achieves more than 10% higher accuracy under the same latency constraints. This indicates that pruning methods can be less effective at generating high-performance models for specific latency targets, especially in Model-as-a-Service (MaaS) contexts. Overall, LAMDA demonstrates superior capability in meeting latency requirements while maintaining higher model accuracy.

Comparison with NAS Methods. LAMDA is benchmarked against two NAS methods, AOWS [2] and MetaD2A+HELP [24], under latency constraints on an NVIDIA 1660 Ti GPU. Both LAMDA and the NAS methods were executed on an NVIDIA A100-40G GPU, with model accuracy evaluated on the Sub-ImageNet dataset. For NAS method, the "total cost time" denotes the overall time spent on both NAS training and search. For LAMDA, the "total cost time" refers to the overall time spent on both LLM fne-tuning and inference.

Table 6: Comparison of LAMDA and MetaD2A+HELP under Different Latency Requirements.

Latency Constraints	2.3ms	2.8ms	3.6ms	4.5ms	5.5ms
Accuracy (%)[LAMDA]	**74%**	**83%**	**84%**	**88%**	**87%**
Accuracy (%)[MetaD2A+HELP]	41%	53%	48%	54%	59%
Δ Accuracy	+33	+30	+36	+34	+28
Cost of LAMDA	**2.1h**	**2.1h**	**2.5h**	**2.5h**	**2.2h**
Cost of MetaD2A+HELP	41.2h	41.1h	41.2h	41.3h	41.4h

Table 7: Model Accuracy and Latency under Different Quantization (Lat. (ms) means Latency, Acc. (%) means Accuracy).

Base Model	FP32		FP16		INT8		LAMDA	
	Lat. (ms)	Acc. (%)	Lat. (ms)	Acc. (%)	Lat. (ms)	Acc. (%)	Lat. (ms)	Acc. (%)
AlexNet	10.38	75.0	5.55	67.6	5.25	68.6	2.77	74.1
GoogLeNet	25.35	87.0	10.60	83.3	12.21	81.5	4.04	88.9
MnasNet	34.96	83.0	20.96	79.5	22.21	78.8	8.99	84.6
MobileNet-V2	29.21	85.0	17.75	80.5	16.21	80.9	6.11	83.8
ResNet-18	18.22	85.5	10.04	82.4	9.12	80.6	2.72	81.9
VGG-16	61.53	90.0	22.36	84.9	21.34	84.6	4.25	86.9
SqueezeNet	21.16	82.0	9.57	78.3	10.21	76.7	2.18	79.3

As shown in Table 5, when compared with AOWS, LAMDA demonstrates superior efficiency and accuracy. It reduces the total cost time by approximately half while generating more accurate models. For instance, under a strict 0.9 ms latency constraint, the model generated by LAMDA is 40% more accurate than the one produced by AOWS. Similarly, Table 6 shows that on NAS-Bench-201 [11], LAMDA consistently outperforms MetaD2A+HELP, achieving higher accuracy and lower total cost time under diverse latency constraints.

These results highlight a common challenge for NAS methods: the trade-off between meeting latency targets and maintaining model accuracy. LAMDA effectively addresses this limitation, achieving superior accuracy under the same constraints with substantially lower computational overhead.

Comparison with Quantization Methods. We benchmark LAMDA against FP16 and INT8 post-training quantization across multiple base models. Since quantization produces discrete latency levels tied to hardware precision, direct comparison at identical latency is infeasible. To ensure fairness, LAMDA is configured to generate models with strictly lower latency than their quantized counterparts, and accuracy is compared under this constraint.

Models are trained in PyTorch, exported to ONNX, and converted to TensorRT format. INT8 quantization uses 100 images for calibration. As shown in Table 7, LAMDA consistently achieved the lowest latency while generally exceeding FP16 and INT8 accuracy. Notably, for irregular multi-branch architectures (e.g., GoogLeNet, MnasNet), INT8 latency exceeded FP16 due to the NVIDIA GTX 1660 Ti lacking native INT8 tensor cores. INT8 ops were instead emulated via DP4A instructions with limited throughput and high overhead, and complex topologies hindered TensorRT's operator fusion. Overall, LAMDA delivers competitive accuracy at lower latency without hardware-dependent low-precision acceleration.

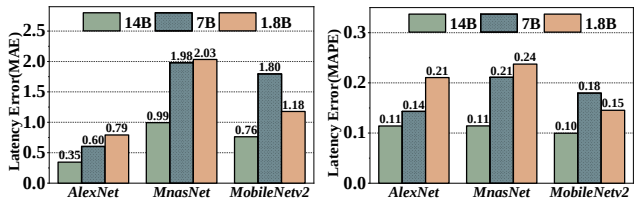

Figure 7: Impact of varying LLM parameter sizes on the latency error of models generated by LAMDA.

5.4 Generalization Across Heterogeneous Hardware and Diverse Tasks

Deployment on Heterogeneous Real-World IoT Devices. We evaluate LAMDA on diverse IoT platforms with varying architectures: Raspberry Pi 4B (ARM Cortex-A72), NVIDIA Jetson Xavier NX (Carmel ARM v8.2), and Edge Workstation (Intel Core i7-9750H CPU). Latency is measured directly on each device and used to fine-tune the LLM. For Raspberry Pi and Jetson, a ResNet-18-based model is deployed; for Workstation, a MnasNet-based model is used. As shown in Table 8, LAMDA meets latency constraints while preserving high accuracy, achieving success rates above 90% on all devices. Latency errors (MAPE) are 13.25% (Raspberry Pi), 11.51% (NVIDIA Jetson Xavier NX), and 23.40% (Workstation). These results demonstrate strong generalization of LAMDA.

Table 8: Performance Metrics by IoT Devices.

Devices	Averaged Required Lat.(ms)	Averaged Generated Lat.(ms)	Latency Error MAPE(%)	Average Acc. (%)	Success Rate (%)
Raspberry Pi 4B	190	178	13.25	84.1	100
Jetson Xavier NX	333	315	11.51	83.5	92.0
Workstation	139	123	23.40	83.0	92.0

Table 9: Performance of generated models on MNIST dataset

Base Model	Lat. (ms)	Acc. (%)	Required Lat. (ms)	Generated Lat. (ms) [LAMDA]	Acc. (%) [LAMDA]
AlexNet	10.38	92.7	4.0	3.55	98.2
GoogLeNet	25.35	98.5	4.5	4.29	97.9
MnasNet	34.96	98.8	6.5	6.88	98.9
MobileNet-V2	29.21	95.8	6.0	6.61	97.4
ResNet-18	18.22	98.7	2.0	2.21	99.1
SqueezeNet	21.16	98.5	2.5	2.54	98.7
VGG-16	61.53	99.2	5.0	4.89	98.9

Cross-Task Generalization of LAMDA. We examine the performance of LAMDA across a diverse set of computer vision tasks: image classification on MNIST dataset [23], object detection on PASCAL VOC2007 dataset [15], and semantic segmentation on CamVid dataset [3]. All experiments in this section are conducted on an NVIDIA GTX 1660 Ti GPU.

We replace the classification layer of pre-trained models for the MNIST dataset. Table 9 shows that LAMDA generates models that satisfied the latency constraints while maintaining or even improving accuracy. For example, on ResNet-18 base model, the latency is reduced from 18.22 ms to 2.21 ms (over 8× speedup) with accuracy of 99.1% . For object detection on PASCAL VOC2007, we use ResNet-18+Faster R-CNN and SqueezeNet+U-Net to fine-tune the LAMDA. Table 10 shows that our optimized ResNet-18+Faster R-CNN achieve a latency reduction from 23.53 ms to 8.07 ms (3.3× speedup) with only a minor mAP decrease from 90.9% to 89.7%. Similarly, for semantic segmentation on CamVid, a generated SqueezeNet+U-Net model by LAMDA can reduce latency from 23.28 ms to 2.28 ms (10.2× speedup) with comparable mIoU as shown in Table 11.

These results demonstrate that LAMDA can generalize across devices and tasks. LAMDA can generate required latency models with negligible performance degradation.

Table 10: Object detection on VOC2007 dataset.

Base Model	Lat. (ms)	mAP (%)	Required Lat. (ms)	Generated Lat. (ms) [LAMDA]	mAP (%) [LAMDA]
AlexNet+Faster RCNN	13.18	84.3	7.0	6.92	82.3
ResNet-18+Faster RCNN	23.53	90.9	8.5	8.07	89.7

Table 11: Semantic segmentation on CamVid dataset.

Base Model	Lat. (ms)	mIoU (%)	Required Lat. (ms)	Generated Lat. (ms) [LAMDA]	mIoU (%) [LAMDA]
GoogLeNet+U-Net	26.85	46.1	4.5	4.21	47.9
SqueezeNet+U-Net	23.28	45.5	2.5	2.28	45.3

5.5 Ablation Study

The Impact of LLM Parameter Sizes. This study investigates how the scale of the LLM affects the quality of the generated models. We compared three versions of the Qwen model [1] with 14B, 7B, and 1.8B parameters, using AlexNet, MnasNet, and MobileNet-V2 as base models. Figures 7a and 7b indicate a clear trend: larger LLMs produce better results. The 14B model consistently achieves the lowest MAE and MAPE in latency. Conversely, the 1.8B model generally yields the highest errors, confirming that larger, more capable LLMs are more effective at generating models that precisely meet latency constraints. However, the performance gains diminish as model size increases; excessively large LLMs (e.g., those exceeding 70B parameters) incur prohibitively high computational costs.

Benefits of Model Slicing. The model slicing module reduces the GPU memory consumption of LAMDA during LLM fine-tuning. As shown in Table 12, the most dramatic example is with GoogLeNet, where memory requirements were halved from 72 GB to 36 GB, enabling fine-tuning on a single A100 40GB GPU. Importantly, this saving of memory does not compromise performance. Model slicing reduces the latency error (Mean Absolute Percentage Error, MAPE)

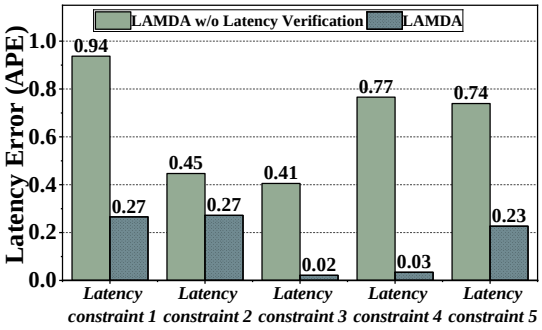

Figure 8: Latency errors (APE) of models generated by LAMDA, with and without latency verification, under varying latency constraints of the MobileNet-V2 base model.

Table 12: Comparison of LAMDA with and without model slicing, and naive fixed-size slicing.

Base Model	Memory Usage (GB)			Latency Error (MAPE)		
	LAMDA	w/o Slicing	Naive	LAMDA	w/o Slicing	Naive
AlexNet	18.2	19.2	18.2	0.14	0.11	0.16
ResNet-18	36.3	39.5	36.3	0.25	0.19	0.27
MobileNet-V2	36.3	57.6	36.3	0.13	0.11	0.13
MnasNet	36.3	57.6	36.3	0.12	0.11	0.14
GoogLeNet	36.3	72.4	36.3	0.12	0.31	0.35
SqueezeNet	36.3	41.3	36.3	0.13	0.19	0.22
VGG-16	19.1	22.2	19.1	0.12	0.11	0.25

by up to 0.06. For comparison, we also evaluate a *Naive Fixed-Size Slicing* baseline, where each slice is assigned a constant length of 2K tokens. While this naive approach achieves similar GPU memory reductions to our adaptive slicing, its neglect of the number of broken edges results in noticeably higher latency prediction errors. For example, in ResNet-18 and SqueezeNet, Naive Fixed-Size Slicing yields MAPE values of 0.27 and 0.22, respectively, compared with 0.25 and 0.13 using LAMDA. In GoogLeNet, the gap is even more pronounced, with MAPE values of 0.35 vs. 0.12. These results emphasize the advantages of LAMDA's adaptive model slicing in achieving both memory efficiency and precise latency estimation.

Benefits of Latency Verification. The latency verification module provides a critical feedback mechanism, regenerating models that fail to meet their latency constraints. To demonstrate its impact, we test it with the MobileNet-V2 base model under five different latency targets. As illustrated in Figure 8, the module consistently and significantly reduces the latency error. The most notable improvement is a reduction in APE from 0.77 to just 0.03. This result underscores the module's essential role in ensuring that the final models produced by LAMDA are both reliable and precisely aligned with the specified performance targets.

Benefits of Fine-tuning. To evaluate the impact of fine-tuning, we compare LAMDA against an in-context learning (ICL) baseline. Specifically, the ICL method uses the LLM with a prompt containing the textual representation of the base model's architecture, syntactic rules, and the latency constraint. We use SqueezeNet as the base model for this comparison and set a uniform latency constraint of 1.98 ms for both methods.

Table 13: Fine-tuning vs. In-context learning (ICL)

Method	Average Required Lat. (ms)	Latency Error (MAPE%)	Average Acc. (%)
Fine-tuning	1.98	13.0	76.0
ICL	1.98	71.0	5.0

Table 14: LAMDA vs. LAMDA with FP16

Base Model	LAMDA		LAMDA+FP16	
	Lat. (ms)	Acc.(%)	Lat. (ms)	Acc. (%)
ResNet-18	2.87	80.9	1.67	78.1
SqueezeNet	2.02	76.2	1.12	75.7

The results, presented in Table 13, reveal a stark performance disparity. The fine-tuned approach achieved a Mean Absolute Percentage Error (MAPE) of only 13.0% against the latency target, whereas the ICL method produced a far greater error of 71.0%. The difference in task performance was even more dramatic: the fine-tuned model achieved 76.0% accuracy, while the model generated via ICL reached only 5.0%. This outcome unequivocally demonstrates that fine-tuning is not merely beneficial but a necessary component of our methodology.

LAMDA Combined with Quantization. We further reduce the latency of the models generated by LAMDA by using FP16 inference and compare the latency and accuracy between LAMDA and LAMDA+FP16, as shown in Table 14. We apply FP16 quantization to two models generated by LAMDA using ResNet-18 and SqueezeNet as base architectures, respectively, to achieve further inference acceleration. For ResNet-18, the latency decreases from 2.87 ms to 1.67 ms (a 42% reduction), with only a 2.8 percentage point drop in accuracy (from 80.9% to 78.1%). SqueezeNet shows an even more pronounced latency reduction of 45% (from 2.02 ms to 1.12 ms), while suffering a negligible accuracy loss of just 0.5 percentage points. These results demonstrate that models produced by LAMDA can be effectively combined with quantization techniques to further accelerate inference while maintaining high accuracy without significant degradation.

6 Conclusions and Future work

We proposed LAMDA, the first pipeline designed to leverage LLMs to efficiently generate hardware-aware DNNs. Our extensive experiments demonstrated that LAMDA achieves strong performance and generalization across diverse model architectures and hardware. The results confirm that LLMs can effectively learn the complex relationship between a model's architecture and its on-device performance, outperforming traditional NAS methods in both accuracy and efficiency. Future work will extend LAMDA into a multi-objective optimization framework that jointly optimizes accuracy, latency, memory footprint and energy consumption. This would significantly broaden the applicability of LAMDA, empowering it to generate models tailored to a wider range of complex, real-world deployment scenarios.

References

[1] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, et al. 2023. Qwen Technical Report. arXiv:2309.16609

[2] Maxim Berman, Leonid Pishchulin, Ning Xu, Matthew B. Blaschko, and Gerard Medioni. 2020. AOWS: Adaptive and optimal network width search with latency constraints. arXiv:2005.10481 [cs.CV] https://arxiv.org/abs/2005.10481

[3] Gabriel J. Brostow, Julien Fauqueur, and Roberto Cipolla. 2009. Semantic object classes in video: A high-definition ground truth database. (2009). doi:10.1016/j.patrec.2008.04.005

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, et al. 2020. Language Models are Few-Shot Learners. (2020). arXiv:2005.14165

[5] Han Cai, Ligeng Zhu, and Song Han. 2019. Proxylessnas: Direct neural architecture search on target task and hardware. . *In International Conference on Learning Representations (ICLR)*. (2019).

[6] Angelica Chen, David Dohan, and David R. So. 2023. EvoPrompting: Language Models for Code-Level Neural Architecture Search. (2023). arXiv:2302.14838

[7] Kunlong Chen, Liu Yang, Yitian Chen, Kunjin Chen, Yidan Xu, and Lujun Li. 2023. GP-NAS-ensemble: a model for NAS Performance Prediction. (2023). arXiv:2301.09231

[8] Mark Chen, Jerry Tworek, Heewoo Jun, and Qiming Yuan. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, K. Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition* (2009), 248–255. https://api.semanticscholar.org/CorpusID:57246310

[10] Peijie Dong, Xin Niu, Lujun Li, Linzhen Xie, Wenbin Zou, Tian Ye, Zimian Wei, and Hengyue Pan. 2022. Prior-guided one-shot neural architecture search. (2022). arXiv:2206.13329

[11] Xuanyi Dong and Yi Yang. 2020. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. arXiv:2001.00326 [cs.CV] https://arxiv.org/abs/2001.00326

[12] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. 2024. The Llama 3 Herd of Models. (2024). arXiv:2407.21783

[13] Lukasz Dudziak, Thomas Chau, Mohamed Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas Lane. 2020. Brp-nas: Prediction-based nas using gcns. *Advances in Neural Information Processing Systems* 33 (2020), 10480–10490.

[14] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. *Journal of Machine Learning Research* 20, 55 (2019), 1–21. http://jmlr.org/papers/v20/18-598.html

[15] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. 2010. The PASCAL Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision* (2010). doi:10.1007/s11263-009-0275-4

[16] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. 2023. DepGraph: Towards Any Structural Pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 16091–16101.

[17] Chengquan Feng, Li Lyna Zhang, Yuanchi Liu, Jiahang Xu, Chengruidong Zhang, Zhiyuan Wang, Ting Cao, Mao Yang, and Haisheng Tan. 2024. {LitePred}: Transferable and Scalable Latency Prediction for {Hardware-Aware} Neural Architecture Search. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1463–1477.

[18] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-Based LLMs for Code Optimization. arXiv:2408.12159 [cs.SE] https://arxiv.org/abs/2408.12159

[19] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. (2015). arXiv:1510.00149

[20] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv:1510.00149 [cs.CV] https://arxiv.org/abs/1510.00149

[21] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. 2019. Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 4335–4344. doi:10.1109/CVPR.2019.00447

[22] Huggingface. 2024. *IHuggingface.* Retrieved 2024-07-30 from https://huggingface.co

[23] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. doi:10.1109/5.726791

[24] Hayeon Lee, Sewoong Lee, Song Chong, and Sung Ju Hwang. 2021. HELP: Hardware-Adaptive Efficient Latency Prediction for NAS via Meta-Learning. arXiv:2106.08630 [cs.LG] https://arxiv.org/abs/2106.08630

[25] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. arXiv:1608.08710 [cs.CV] https://arxiv.org/abs/1608.08710

[26] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. (2018). arXiv:1806.09055

[27] Liang Liu, Mingzhu Shen, Ruihao Gong, Fengwei Yu, and Hailong Yang. 2022. Nnlqp: A multi-platform neural network latency query and prediction system with an evolving database. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–14.

[28] Yu Liu, Xuhui Jia, Mingxing Tan, Raviteja Vemulapalli, Yukun Zhu, Bradley Green, and Xiaogang Wang. 2020. Search to distill: Pearls are everywhere but not the eyes. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 7539–7548.

[29] Muhammad Umair Nasir, Sam Earle, Julian Togelius, Steven D. James, and Christopher Wesley Cleghorn. 2023. LLMatic: Neural Architecture Search via Large Language Models and Quality-Diversity Optimization. (2023). arXiv:2306.01102

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[31] Adam Polyak and Lior Wolf. 2015. Channel-Level Acceleration of Deep Face Representations. *IEEE Access* 3 (2015), 2163–2175. https://api.semanticscholar.org/CorpusID:2668965

[32] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems* 36 (2024).

[33] Neeraj Varshney, Wenlin Yao, Hongming Zhang, Jianshu Chen, and Dong Yu. 2023. A Stitch in Time Saves Nine: Detecting and Mitigating Hallucinations of LLMs by Validating Low-Confidence Generation. arXiv:2307.03987 [cs.CL] https://arxiv.org/abs/2307.03987

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[35] Huan Wang, Can Qin, Yulun Zhang, and Yun Fu. 2021. Neural Pruning via Growing Regularization. arXiv:2012.09243 [cs.CV] https://arxiv.org/abs/2012.09243

[36] Shih-Yuan Yu, Yonatan Gizachew Achamyeleh, Chonghan Wang, Anton Kocheturov, Patrick Eisen, and Mohammad Abdullah Al Faruque. 2023. CFG2VEC: Hierarchical Graph Neural Network for Cross-Architectural Software Reverse Engineering *(ICSE-SEIP '23)*. IEEE Press. doi:10.1109/ICSE-SEIP58684.2023.00031

[37] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 81–93.

[38] Mingkai Zheng, Xiu Su, Shan You, Fei Wang, Chen Qian, Chang Xu, and Samuel Albanie. 2023. Can GPT-4 Perform Neural Architecture Search? (2023). arXiv:2304.10970

[39] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.