UniExit: A Unified Latency-Aware Early-Exit Runtime for Heterogeneous Edge AI

Yuhang Guan
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
asterism178@163.com

Huanle Zhang
Shandong University
Qingdao, China
dtczhang@sdu.edu.cn

Xiaomin Ouyang
The Hong Kong University of Science and Technology
Hong Kong, China
xmouyang@cse.ust.hk

Zhidan Liu*
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
zhidanliu@hkust-gz.edu.cn

arXiv:2611.09159v1 [cs.DC] 6 Nov 2026

Abstract

Edge applications increasingly combine CNN-based perception with edge-deployable Transformer-based language models, but deploying such heterogeneous workloads on resource-constrained devices is challenging because inference latency varies with both input difficulty and runtime hardware state. Early-exit inference reduces computation by terminating at intermediate stages, yet existing methods are often model-specific and rely on static latency profiles that become stale under dynamic edge conditions. This paper presents UniExit, a unified latency-aware early-exit runtime for heterogeneous edge AI. UniExit abstracts CNN blocks and Transformer layers as ordered stages and applies a common latency-aware routing policy while retaining model-specific quality mechanisms. It predicts the cost of reaching the next exit from runtime state and inference progress to make budget-aware continue-or-exit decisions. UniExit combines runtime-state-aware latency prediction, cache-based prediction reuse, trend-aware fallback, and CNN-specific exit stabilization to provide low-overhead and robust online control. We implement UniExit on multiple edge and edge–cloud platforms and evaluate it with representative CNN and Transformer-based language-model workloads. Across single-workload, concurrent heterogeneous, and edge–cloud settings, UniExit consistently improves deadline satisfaction and latency–accuracy trade-offs over existing progressive inference baselines, while reducing redundant exit-module computation and task-level energy consumption.

1 Introduction

With advances in edge computing and on-device hardware, AI inference is increasingly moving from cloud servers to edge devices, closer to users, sensors, and data sources [17, 42, 50]. As edge applications become more intelligent and interactive, they are evolving from single-model deployments to heterogeneous AI workloads [4, 14] that combine CNN-based visual perception [57] with edge-deployable Transformer-based language models for language understanding and interaction [44, 62]. These workloads differ in task semantics, computation patterns, and latency–accuracy requirements, yet share a common deployment need: *delivering timely and reliable inference under resource-constrained and dynamic edge environments*. Representative scenarios include in-vehicle perception [13], interactive AI assistants [44, 62], augmented reality [29, 35], and context-aware mobile sensing [1, 27], where latency directly affects responsiveness, user experience, and reliability.

A key opportunity in latency-critical deployments is that *not all inputs require the same amount of computation*. Some visual inputs can be recognized from shallow features [49], while others require deeper semantic representations [18, 25]; similarly, some language-model tokens can be predicted from intermediate layers [9, 52], while others require deeper contextual refinement [54]. Full-model inference ignores this variability and executes the entire model for every input or token, wasting computation on easy cases. Progressive inference [43, 45, 57], or early-exit inference, addresses this inefficiency by inserting candidate exits into a neural network and terminating inference once an intermediate prediction is reliable. Compared with model pruning [31] and computation offloading [24, 30, 38], early-exit inference provides a flexible on-device mechanism for runtime adaptation while preserving the option of deeper inference when needed.

Existing early-exit techniques have been studied for both CNNs [5, 12, 49] and Transformer-based language models [9, 54]. CNN methods typically attach branch classifiers to intermediate convolutional or residual blocks [49], while language-model methods use intermediate prediction heads or layer-wise confidence signals to decide whether later Transformer layers can be skipped [9, 52, 63]. These designs demonstrate the potential of adaptive computation, but they are usually developed for specific model families and tightly couple confidence estimation with **model-specific exit policies**. As a result, existing early-exit systems provide limited support for managing heterogeneous edge AI workloads under a unified runtime objective, even though CNN and language-model workloads increasingly coexist on the same edge platforms.

In this paper, we make a key observation: *CNNs and Transformer-based language models expose a common staged decision structure during early-exit inference*, despite their different task semantics and computation patterns. CNNs execute a sequence of convolutional or residual blocks, while Transformer-based language models execute a sequence of Transformer layers. At selected stages, both expose candidate outputs that can be returned upon termination. Early-exit inference can therefore be viewed as a runtime-control problem

*Corresponding author.

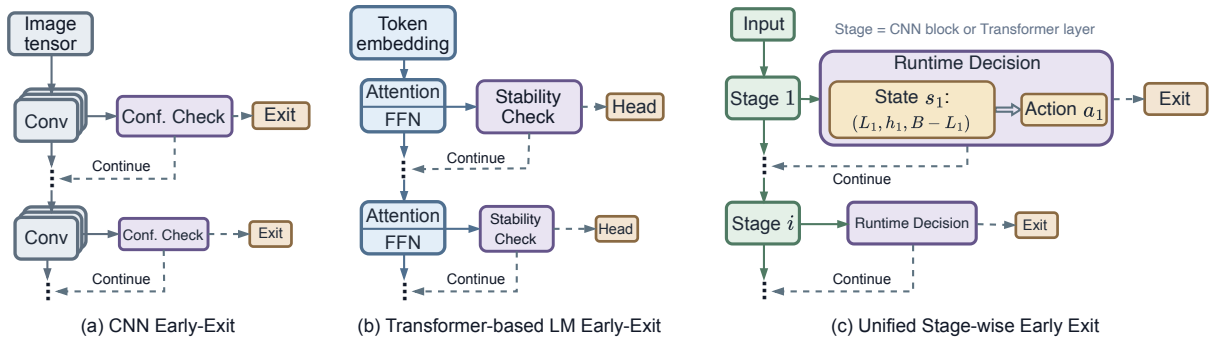


Figure 1: From model-specific early exits to a shared stage-wise runtime view. (a) CNN early-exit methods use branch classifiers and confidence checks. (b) Transformer-based language-model methods use intermediate heads and stability checks. (c) UniExit abstracts both as staged runtime decisions driven by execution progress, runtime state, and latency budget, without using model-specific confidence or stability signals for routing.

driven by execution progress, predicted remaining cost, runtime state, and latency constraints, while keeping model-specific exit and quality mechanisms unchanged.

However, making such runtime decisions reliable on edge devices is challenging because inference latency is not static. Many early-exit systems [28, 32, 37] rely on offline profiling to construct static latency tables for different exit paths and then use these tables to estimate online execution cost [20, 57, 61]. This design assumes that segment latency remains stable during deployment. In practice, edge devices dynamically adjust CPU, GPU, and memory-controller states [58] to balance performance, power, and temperature [15, 47, 57], and may also experience resource contention from co-running workloads [4, 14, 34]. Consequently, the same model segment can exhibit different latency across time. As a result, **static latency profiles can become stale**, leading to unnecessary exit execution, premature exits, or deadline violations.

To address these challenges, we propose UniExit, a **unified latency-aware early-exit runtime** for dynamic edge environments. UniExit provides a common stage-wise interface and shared latency-aware routing for CNN and language-model workloads, while retaining model-specific exit and quality mechanisms. Rather than relying solely on static latency profiles, UniExit incorporates runtime state and inference progress into exit decisions. At each candidate exit, UniExit estimates whether continuing to the next stage can still satisfy the latency budget. If the predicted cost fits within the remaining budget, UniExit bypasses the current exit module and continues deeper inference; otherwise, it executes the current exit and returns an early result. In this way, UniExit selects the exit position according to real-time latency feasibility, independently of model-specific quality signals.

UniExit makes this runtime practical through three lightweight mechanisms. First, it trains **runtime-state-aware latency predictors** that estimate stage and exit-module latency using hardware frequencies, resource utilization, and computation-unit identifiers. Second, it uses **cache-based prediction reuse** to avoid redundant predictor invocations under recurring runtime states, reducing online decision overhead. Third, it incorporates **trend-aware fallback** to make routing more conservative when abrupt hardware-state changes may invalidate recent predictions. Beyond the shared runtime, UniExit also includes a CNN-specific offline stabilization objective that reduces confidence degradation and prediction flips across CNN exits, while preserving the same online interface used for language-model workloads.

We implement UniExit on multiple edge and edge–cloud platforms and evaluate it with representative CNN and Transformer-based language-model workloads. Our evaluation covers isolated single-workload inference under static and dynamic hardware states, concurrent heterogeneous workloads, runtime overhead, ablation studies, and CNN-based edge–cloud split inference. Across these settings, UniExit consistently improves deadline satisfaction and latency–accuracy trade-offs over existing progressive inference baselines. It achieves over 95% deadline satisfaction in single-workload and edge–cloud settings, remains robust under concurrent heterogeneous workloads, and reduces redundant exit-module computation and task-level energy consumption.

This paper makes the following contributions:

- We formulate CNN and Transformer-based language-model early-exit inference as a shared stage-wise runtime-control problem, with common latency-aware routing and model-specific quality mechanisms. This abstraction improves runtime code reuse and reduces duplicated logic and integration complexity.
- We design UniExit, a latency-aware early-exit runtime that incorporates runtime state, inference progress, and latency budgets to make adaptive exit decisions under dynamic edge environments.
- We make the runtime practical through lightweight latency prediction, cache-based reuse, fallback routing under volatile hardware states, and CNN-specific exit stabilization.
- We implement UniExit on multiple edge and edge–cloud platforms and evaluate it across single-workload, concurrent heterogeneous, and edge–cloud settings, showing that UniExit delivers consistently stronger deadline-aware inference across deployment settings, with better latency–accuracy trade-offs and lower computation and energy overhead.

2 Motivation

2.1 Early-Exit Inference in Edge AI

In latency-critical edge inference, *not all inputs or tokens require the same amount of computation* [4, 45, 57]. Some images can be recognized from shallow visual features, while harder ones require deeper

semantic representations [18, 25, 49]. Likewise, some language-model tokens can be predicted from intermediate Transformer layers, while others need deeper contextual refinement [9, 54, 63]. Full-model execution for every input or token therefore wastes computation on easy cases.

Early-exit inference reduces redundant computation through input-dependent conditional execution [12, 45, 57]. Instead of executing the full model, it inserts intermediate exits that return a prediction once the current representation is sufficiently reliable. As shown in Figure 1(a), CNN early-exit networks attach branch classifiers to selected convolutional or residual blocks [18, 25, 49]. Each branch produces a tentative prediction and confidence score from the current feature representation. If the confidence exceeds a threshold, the image exits early and skips the remaining blocks; otherwise, it continues deeper. This image-level branch execution saves computation for easy samples while retaining deeper paths for harder ones when the latency budget allows.

Figure 1(b) illustrates a related early-exit pattern in Transformer-based language models, but with a different execution granularity [9, 37, 54]. Since Transformers are organized as ordered layers, intermediate prediction heads or confidence/stability signals can evaluate the current hidden state after selected layers [9, 52, 63]. During generation, the exit policy decides whether a token can be emitted from the current layer or should continue deeper for refinement. In latency-constrained deployments, this decision must also respect the remaining time budget, motivating the runtime abstraction introduced later. Unlike CNN classification, language-model early exit is often token-level, allowing easy and hard tokens within the same sequence to follow different computational depths.

2.2 Limitations of Existing Early-Exit Systems

Early-exit inference provides an effective mechanism for adaptive computation, but existing systems are often built around specific model families, exit modules, and latency assumptions [28, 32, 48]. This design style creates a *mismatch* with emerging edge AI scenarios, where CNN-based perception and Transformer-based language-model workloads may coexist on the same device or within the same edge–cloud pipeline. We identify two key limitations.

Limitation 1: Exit control is model-specific. Existing methods usually couple confidence estimation, exit policy, and accuracy–latency tradeoff within a model-specific pipeline. CNN early-exit methods typically attach branch classifiers to intermediate convolutional or residual blocks and make image-level exit decisions based on branch confidence [25, 49], sometimes together with latency or deadline constraints [37, 48, 57]. In contrast, Transformer-based language-model methods use layer-wise heads, token-level stability, or intermediate logits to decide whether the current token should exit or continue to deeper layers [9, 52, 54]. Although both pipelines follow an *exit-or-continue* pattern, their control logic is embedded in different model structures and execution granularities. This makes it difficult to reuse exit policies across workloads or expose a shared runtime controller for heterogeneous edge inference.

Limitation 2: Latency awareness relies on static profiles. Even when latency constraints are considered, many early-exit systems estimate execution cost using offline latency profiles. A typical system profiles each block, layer, exit module, or remaining path under a fixed hardware state, then uses the latency table to determine whether continuing to a deeper exit is affordable under the remaining deadline [28, 32, 48]. This approach is simple and efficient, but it implicitly assumes that profiled latency remains valid during deployment. This assumption is fragile on edge devices, where dynamic voltage and frequency scaling (DVFS), thermal effects, background workloads, memory-frequency changes, and resource contention can change execution latency over time [6, 11, 33, 60]. Once the profiled latency becomes stale, the remaining-latency estimate can mislead the exit policy: an optimistic estimate may continue too deeply and miss the deadline, while a conservative estimate may trigger premature exits and reduce prediction quality.

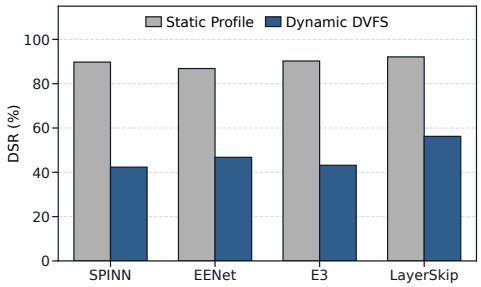


Figure 2: CNN and language-model early-exit methods suffer substantial DSR drops under dynamic hardware states.

To quantify the impact of stale latency profiles, we conduct a preliminary study with representative early-exit methods for both CNN and Transformer-based language-model workloads. Specifically, we evaluate SPINN [28], EENet [20], and E3 [32] as CNN early-exit methods, and LayerSkip [9] as a Transformer-based language-model early-exit method on an edge-deployable small language model; detailed experimental settings are described in §7.1. This preliminary study focuses on DVFS-induced hardware variability. As shown in Figure 2, all methods achieve high deadline satisfaction rates (DSRs) under static hardware conditions, but their DSRs drop substantially under dynamic DVFS conditions. The degradation appears in both CNN and language-model workloads, indicating that the problem is not tied to a particular model family. These results show that static latency profiles can support effective exit decisions in controlled environments, but become unreliable when hardware performance changes dynamically at runtime [55].

These limitations suggest two requirements for edge early-exit inference. First, *exit decisions should be online latency-aware*, accounting for current execution progress, remaining latency budget, and runtime hardware state instead of relying solely on static profiles. Second, *such control should be reusable across model families*, rather than redesigned separately for CNN and language-model workloads. These requirements motivate the unified stage-wise view introduced next.

2.3 A Unified Stage-Wise View

CNN and Transformer-based language-model early-exit systems differ in model structure and exit semantics, but both execute through sequential model blocks. We define a stage as a model block with strong internal data dependencies and use its boundary as a runtime decision point. A stage can correspond to a residual block in CNNs or a Transformer layer in language models. Such boundaries

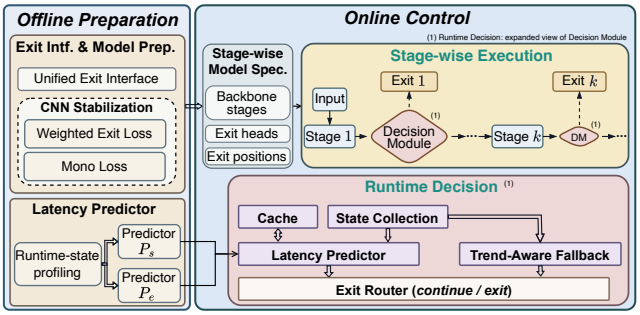


Figure 3: Overview of UniExit. Offline preparation builds exit interfaces, stabilization, and latency predictors; online execution invokes a decision module before candidate exits. The Runtime Decision block expands this module by monitoring runtime state, predicting next-exit cost, and routing each decision to continue or exit.

provide a consistent decision granularity while avoiding runtime control inside asynchronous kernels or overlapping execution regions that could disrupt execution concurrency. At each candidate exit, the current output is returned if continuing would exceed the remaining latency budget.

We therefore abstract early-exit inference as a *staged runtime decision process*. For each inference request, B denotes the application-specified latency budget, or relative deadline. As illustrated in Figure 1(c), at stage i, the runtime observes

$$s_i = (L_i, h_i, B - L_i), \tag{1}$$

where L_i, h_i, and $B - L_i$ denote elapsed latency, runtime state, and remaining latency budget, respectively. The runtime then chooses

$$a_i \in \{exit, continue\}. \tag{2}$$

On exit, the system executes the current exit module and returns y_i; otherwise, it proceeds to stage $i + 1$. Exit modules and outputs remain model-specific, while runtime monitoring, latency estimation, budget tracking, and routing are shared across workloads.

This abstraction directly addresses the two limitations above. It provides a common interface for model-specific exits and makes routing aware of execution progress, remaining budget, and hardware state rather than static latency profiles alone. This unified stage-wise view forms the basis for UniExit, a latency-aware early-exit runtime for heterogeneous edge AI workloads.

3 System Overview

3.1 Design Goals

Motivated by the limitations and requirements identified above, the design of UniExit targets four design goals:

- **Unified control.** Provide CNN and language-model workloads with a shared stage-wise latency-aware routing interface while preserving model-specific quality mechanisms.
- **Online latency awareness.** Make exit-or-continue decisions using elapsed latency, runtime state, and the remaining latency budget, rather than relying only on static latency profiles.
- **Low overhead.** Keep latency prediction and exit routing lightweight enough that online control does not offset the computation saved by early exits.
- **Runtime robustness.** Maintain deadline satisfaction under dynamic hardware states and workload interference, avoiding both overly optimistic continues that miss deadlines and overly conservative exits that reduce prediction quality.

3.2 UniExit Architecture

Figure 3 shows the architecture of UniExit. To realize the goals above, UniExit separates model-specific exit preparation from shared runtime control. In the offline phase, UniExit prepares exit modules and runtime-state-aware latency predictors for different workloads. For CNN workloads, UniExit trains early-exit branches with CNN-specific stabilization to improve exit reliability; for Transformer-based language models, it reuses layer-wise exit heads or pretrained early-exit models. In parallel, UniExit collects latency measurements under representative runtime states and trains compact predictors for estimating stage and exit-module latency.

During online inference, UniExit executes the model as ordered stages and invokes a lightweight decision module before each candidate exit. The lower Runtime Decision block in Figure 3 expands the internal logic of this module. It uses the elapsed latency L_k, the predicted cost of reaching the next candidate exit $\hat{\Delta L}_{k\to k+1}$, the current runtime state, and the latency budget B. UniExit continues to deeper computation when $L_k + \hat{\Delta L}_{k\to k+1} + \Delta S \le B$, and exits at the current stage otherwise, where ΔS is a safety margin for volatile runtime conditions. This decision placement avoids unnecessary exit-module execution when the remaining budget still allows deeper inference.

To keep the runtime lightweight and robust, UniExit integrates cache-based prediction reuse and trend-aware fallback. The cache reduces repeated predictor invocations under recurring runtime states, while fallback makes routing more conservative when runtime states become volatile. Together, they enable robust, low-overhead latency-aware control across heterogeneous edge workloads. The following sections describe these components in detail.

4 Offline Support for Runtime Control

UniExit separates workload-specific preparation from shared online control. The offline phase exposes candidate exits through a common interface, trains lightweight stage and exit latency predictors from runtime-state profiles, and applies CNN-specific stabilization when needed. This keeps the online control path unified while allowing workload-specific preparation where necessary. Table 1 summarizes how each offline component supports online runtime.

4.1 Exit Interface and Model Preparation

UniExit supports heterogeneous early-exit workloads through a common interface that exposes ordered stages, executable candidate exits, prediction outputs, and exit-latency metadata. It requires no shared confidence or stability signals, as UniExit 's routing does not use model-specific quality signals.

Table 1: Offline support for online runtime control.

Component	Offline output	Runtime role
Exit interface	Exit module, prediction, metadata	Common execution interface
Latency predictors	P_s and P_e	Next-exit cost estimation
CNN stabilization	WEL + Mono losses	Stable CNN exit signals

For CNN workloads, candidate exits are implemented as branch classifiers attached to selected convolutional or residual blocks. When exit k is selected, its branch classifier maps h_k to logits $z_k = g_k(h_k)$ and returns the prediction; bypassed classifiers are not executed. Each CNN branch classifier consists of a 1×1 convolution, normalization, global average pooling, and a linear classifier. Its architecture is automatically constructed from the feature dimensions at the corresponding exit.

For Transformer-based language-model workloads, UniExit reuses layer-wise exit heads or pretrained early-exit models. When an exit layer is selected, its hidden representation is mapped to the output through the model's exit head, while model-specific output semantics and training remain unchanged.

UniExit supports models whose exits are trained with confidence-, entropy-, margin-, patience-, or stability-based mechanisms without using these quality signals for routing. Adapting a new architecture only requires mapping its stages and exits to the common interface and profiling their latency; the latency predictors, cache, fallback, and routing rule remain unchanged. Model-specific quality mechanisms can remain as orthogonal components when needed.

4.2 Runtime-State-Aware Latency Predictor

UniExit trains lightweight stage and exit latency predictors offline and uses them online to estimate the remaining latency between candidate exits. Unlike static-profile-based methods [20, 28, 32] that directly use fixed latency tables during deployment, UniExit learns a compact mapping from runtime state to execution latency. Here, runtime state includes the current hardware frequency, resource utilization, and the computation unit being predicted. This design lets the online controller adapt exit decisions when performance changes due to DVFS variation or resource contention.

Predictor decomposition. UniExit uses two predictors for each workload: a stage-latency predictor P_s and an exit-latency predictor P_e. The intuition is that backbone stages and exit modules exhibit different latency patterns. CNN blocks and Transformer layers dominate the sequential execution path, and their latency is affected by computation intensity, memory access, execution depth, and hardware state. Exit modules, such as CNN branch classifiers or language-model exit heads, are auxiliary networks with different computation structures and shorter execution paths. Modeling both with a single predictor may mix heterogeneous latency distributions and reduce prediction accuracy. Therefore, UniExit predicts backbone-stage latency and exit-module latency separately while keeping both predictors compact.

Runtime-state features. CNN and language-model predictors use six- and five-dimensional feature vectors, respectively. The first three features describe the current frequency state of CPU, GPU, and External Memory Controller (EMC). We normalize each frequency by its maximum supported value:

$$\hat{f}_{\mathrm{CPU}} = \frac{f_{\mathrm{CPU}}}{f_{\mathrm{CPU}}^{\max}}, \quad \hat{f}_{\mathrm{GPU}} = \frac{f_{\mathrm{GPU}}}{f_{\mathrm{GPU}}^{\max}}, \quad \hat{f}_{\mathrm{EMC}} = \frac{f_{\mathrm{EMC}}}{f_{\mathrm{EMC}}^{\max}}. \tag{3}$$

The next two features capture runtime load. We use CPU and GPU utilization, denoted as u_{CPU} and u_{GPU}, to reflect resource contention from co-running tasks. They are normalized as

$$\hat{u}_{\mathrm{CPU}} = \frac{u_{\mathrm{CPU}}}{100}, \quad \hat{u}_{\mathrm{GPU}} = \frac{u_{\mathrm{GPU}}}{100}. \tag{4}$$

These utilization features allow the predictor to distinguish lightly loaded and contended runtime states even when their DVFS frequencies are identical.

For CNNs, the final feature is a normalized computation-unit identifier ID_ℓ, identifying the block or stage group for stage prediction and the exit module for exit prediction. Language-model predictors omit this feature because Transformer layers are structurally identical. Overall, the predictor input is

$$\mathbf{x}_\ell = \left[\hat{f}_{\mathrm{CPU}}, \hat{f}_{\mathrm{GPU}}, \hat{f}_{\mathrm{EMC}}, \hat{u}_{\mathrm{CPU}}, \hat{u}_{\mathrm{GPU}}, \mathrm{ID}_\ell\right]. \tag{5}$$

Profiling and training. During offline profiling, UniExit runs the instrumented model under representative runtime states and records per-stage and per-exit latency. Varying hardware frequencies and utilization levels produces regression samples for the stage and exit predictors. The stage predictor maps $\mathbf{x}_\ell$ to $\hat{L}_\ell$, while the exit predictor maps the corresponding exit feature to $\hat{E}_k$; both are trained with a standard regression loss. Each predictor is a compact two-hidden-layer MLP with 64 and 32 hidden neurons and about 2.5K parameters, keeping latency estimation lightweight on the online decision path. Predictors are calibrated per workload and platform; cross-device transfer is outside this paper's scope.

Remaining-latency estimation. At runtime, the online controller uses the predictors to estimate whether continuing from the current exit to the next candidate exit is affordable under the remaining latency budget. For candidate exit k, UniExit estimates the cost of reaching and evaluating the next candidate exit as

$$\hat{\Delta L}_{k\to k+1} = \sum_{\ell \in \mathcal{P}_{k\to k+1}} \hat{L}_\ell + \hat{E}_{k+1}, \tag{6}$$

where $\mathcal{P}_{k\to k+1}$ denotes the backbone stages between exits k and $k+1$, and $\hat{E}_{k+1}$ is the predicted latency of the next exit module. The online router uses this estimate to decide whether deeper computation is affordable under the current budget.

4.3 CNN-Specific Exit Stabilization

UniExit uses deeper stages when the latency budget allows, which implicitly assumes that deeper exits provide more reliable predictions. CNN early-exit networks may violate this assumption: due to changes in receptive field and semantic abstraction across depths, a sample that is confident or correct at an early exit may become less confident or even incorrect at a later exit [2, 21, 25]. Such non-monotonic behavior can destabilize latency-aware routing. We therefore add a CNN-specific offline stabilization objective, while keeping the online exit interface unchanged.

Weighted exit loss (WEL). For the k-th CNN exit, the branch classifier produces logits $z_k = g_k(h_k)$ from the intermediate feature h_k. Its cross-entropy loss is

$$\mathcal{L}_k^{\mathrm{ce}} = -\log \frac{\exp(z_{k,y})}{\sum_{c=1}^{C} \exp(z_{k,c})}, \tag{7}$$

where y is the ground-truth label and C is the number of classes. Instead of weighting all exits equally, UniExit assigns larger weights to deeper exits:

$$\mathcal{L}_{\mathrm{weighted}} = \sum_{k=1}^{K} w_k \cdot \mathcal{L}_k^{\mathrm{ce}}, \quad w_k = w_0 + \alpha \cdot (k-1). \tag{8}$$

where K is the number of exits excluding the final model output, w_0 is the initial weight, and α controls the linear growth rate. This reduces the dominance of shallow branches over the shared backbone and encourages deeper exits to learn more stable representations.

Monotonicity regularization (Mono). To reduce confidence degradation across exits, UniExit regularizes the correct-class confidence of adjacent exits. Let $p_k = \text{softmax}(z_k)$ and $p_{k,y}$ be the ground-truth class confidence at exit k. The regularization is

$$\mathcal{L}_{\text{mono}} = \sum_{k=1}^{K-1} \max(0, p_{k,y} - p_{k+1,y}). \tag{9}$$

This penalty discourages later exits from assigning lower confidence to the correct class than earlier exits, without enforcing strict monotonicity for every sample.

Overall objective. The final CNN training objective is

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{final}} + \mathcal{L}_{\text{weighted}} + \lambda_{\text{mono}} \cdot \mathcal{L}_{\text{mono}}, \tag{10}$$

where $\mathcal{L}_{\text{final}}$ preserves full-model accuracy and λ_{mono} controls the regularization strength. This objective stabilizes CNN exits for online routing while preserving the common interface shared with Transformer-based language-model workloads.

5 Online Runtime Control

During inference, UniExit follows the online control path in Figure 3 to make per-stage routing decisions under the current latency budget. At each candidate exit, the controller collects runtime context, estimates next-exit cost, applies cache reuse and fallback as needed, and routes the model to continue or exit.

5.1 Runtime State Collection

UniExit maintains two lightweight sources of runtime context for online latency prediction and exit routing:

- **Hardware-state monitor.** UniExit runs a background thread that periodically samples CPU/GPU/EMC frequencies and CPU/GPU utilization. On Jetson platforms, it obtains CPU frequency via Linux CPUFreq, GPU frequency/utilization via Devfreq, CPU utilization from procfs accounting deltas, and EMC frequency via BPMP. Since some EMC nodes require root access, a privileged process publishes the active EMC frequency to a shared state file. At candidate exits, the inference thread non-blockingly reads the monitor's in-memory snapshot to construct the predictor input in Eq. (5), avoiding synchronous device queries.
- **Inline latency timer.** UniExit measures elapsed latency with a high-resolution monotonic clock, synchronizing CUDA before request execution and at each candidate exit so that preceding GPU kernels complete before timing. The timer is read again after latency prediction to include state retrieval and controller overhead in the remaining-budget calculation. Upon exit, UniExit synchronizes after the exit head and records end-to-end latency. Synchronization is limited to candidate exits rather than every layer to avoid unnecessary GPU serialization.

Together, the monitored state and inline latency measurement provide the context required by the latency predictor and exit router.

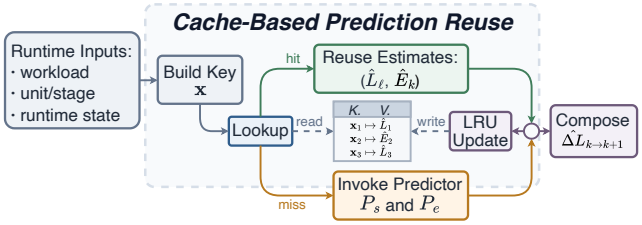


Figure 4: Cache-based prediction reuse in UniExit. The cache is queried before the stage predictor P_s and exit predictor P_e; hits reuse recent latency estimates, while misses invoke the corresponding predictor and update the cache.

5.2 Latency-Aware Exit Routing

At each candidate exit k, UniExit invokes the routing decision before executing the current exit module. Particularly, if the remaining budget still allows deeper execution, UniExit can bypass the current exit module and avoid unnecessary exit overhead. Let B denote the application-specified latency budget, L_k denote the elapsed latency when reaching exit k, and $\hat{\Delta}L_{k\to k+1}$ denote the predicted latency of reaching and evaluating the next candidate exit. UniExit continues to the next stage when $L_k + \hat{\Delta}L_{k\to k+1} + \Delta S \le B$, and executes the current exit otherwise, where ΔS is a safety margin used under runtime uncertainty. Therefore, the routing rule to take an action is

$$a_k = \begin{cases} \text{Continue}, & L_k + \hat{\Delta}L_{k\to k+1} + \Delta S \le B, \\ \text{Exit}, & \text{otherwise}. \end{cases} \tag{11}$$

This rule turns early-exit inference into a deadline-aware runtime control problem. When the predicted next-exit cost fits within the remaining budget, UniExit preserves the opportunity for deeper computation and potentially higher accuracy. When continuing would risk violating the deadline, UniExit executes the current exit module and returns an intermediate result.

Although UniExit currently focuses on latency-aware routing, its stage-wise control interface could incorporate energy constraints. An additional predictor could estimate the energy cost of reaching the next exit, while the router tracks the remaining energy budget. Under a joint latency–energy policy, execution would continue only if both predicted costs fit their respective budgets; otherwise, it would favor an earlier exit. Such energy-aware prediction and routing are left for future work.

5.3 Cache-Based Prediction Reuse

Although the latency predictors in UniExit are lightweight, online prediction can still become non-negligible when decisions are triggered frequently, especially for token-level language-model exits or workloads with many candidate exits. Meanwhile, edge hardware states often exhibit *temporal locality*: CPU/GPU/EMC frequencies and utilization may remain stable for short intervals, and similar runtime states may recur across consecutive inputs or tokens. This creates an opportunity to reuse recent latency predictions instead of invoking the MLP predictor at every decision point. The cache does not replace the predictor or act as an offline profiling-based LUT; it is a bounded, on-demand runtime cache storing only recent states and their predictor outputs.

Cache lookup and update. As shown in Figure 4, UniExit inserts a lightweight cache before both the stage predictor P_s and the exit predictor P_e.

Each entry maps a composite key to a predicted latency. The key combines the workload ID, unit type (stage or exit), and the predictor input in Eq. 5; CNNs use the full $\mathbf{x}_\ell$, while language models omit ID_ℓ because Transformer layers share the same structure. Normalized features are used directly without quantization or bucketization, and hash lookup provides expected $O(1)$ access.

Each value is a predicted stage latency $\hat{L}_\ell$ or exit latency $\hat{E}_k$. At candidate exit k, cache hits reuse the estimates needed for $\hat{\Delta L}_{k\to k+1}$ and bypass the MLP; misses invoke P_s or P_e and cache the result.

Bounded reuse. To keep the cache small and predictable, UniExit uses a least-recently-used (LRU) replacement policy [3]. In our implementation, the cache holds 128 entries and occupies about 3 KB; when full, it evicts the least recently accessed entry. This matches the short-term locality of edge runtime states, where recent frequency and utilization configurations are likely to reappear in nearby inference requests. The cache is transparent to model execution: it does not alter the forward path, exit heads, or confidence signals, but only reduces repeated predictor invocations on the online decision path.

Interaction with fallback. Cache reuse is most effective under static or low-variability runtime states. When the trend-aware module detects volatile hardware behavior, UniExit avoids blindly trusting cached predictions. In such cases, the router either increases the safety margin ΔS or uses a conservative historical estimate, as described in §5.4. This keeps cache acceleration from compromising deadline robustness under rapidly changing edge conditions.

5.4 Trend-Aware Fallback

Cache reuse and lightweight prediction are effective when runtime states are stable or change gradually. However, abrupt DVFS transitions, thermal throttling, or resource contention can make latency estimates overly optimistic and cause deadline misses. To improve robustness, UniExit uses a trend-aware fallback mechanism.

The fallback module monitors recent runtime-state variations using two indicators: fluctuation magnitude σ_f and fluctuation period τ_f. Here, σ_f captures the magnitude of recent frequency or utilization changes, while τ_f captures how rapidly such changes occur. When $\sigma_f > \sigma_{\text{thr}}$ and $\tau_f < \tau_{\text{thr}}$, UniExit treats the environment as volatile and augments the safety margin ΔS. The base margin is set from an upper percentile of the offline prediction-error distribution, *e.g.*, 95th or 99th percentile. Online, the trend-aware module records periodic worst-case latency and adds the gap between this worst-case latency and the current prediction as extra compensation.

Fallback is triggered only under volatile runtime states and does not replace normal latency prediction. Instead, it complements the predictor by making routing more conservative in tail cases, reducing deadline misses caused by sudden performance drops.

6 Implementation

We implement UniExit as a runtime layer on top of PyTorch-based CNN and Transformer-based language-model inference pipelines. The runtime includes four main components: *a runtime-state monitor*, *an inline latency timer*, *a cache-assisted latency prediction module*, and *a latency-aware exit router*. The monitor samples CPU/GPU/EMC frequencies and CPU/GPU utilization, and exposes the latest state to the inference thread through shared memory. The inline timer measures elapsed latency at stage and exit boundaries, providing accurate per-request timing for routing decisions.

Table 2: Hardware platforms used in our evaluation.

Platform	CPU	GPU	Memory / Compute
Xavier NX	6-core Carmel	384-core Volta	8 GB / 6.8 TFLOPS
Orin Nano	6-core A78AE	1024-core Ampere	8 GB / 10.0 TFLOPS
Cloud Server	2×32-core Xeon	RTX 5880 Ada	48 GB / 69.3 TFLOPS

The latency prediction module uses compact two-hidden-layer MLPs for stage- and exit-latency estimation. Predictors are trained offline from runtime-state profiling data and invoked online only on cache misses. The cache indexes predictions by workload ID, unit type, and normalized predictor features, including stage/exit IDs for CNN workloads, and uses LRU replacement to bound memory usage. The exit router combines elapsed latency, predicted remaining latency, the latency budget, and the fallback safety margin to decide whether to continue or execute the current exit.

For CNN workloads, UniExit inserts branch classifiers at selected convolutional or residual blocks and optionally applies CNN-specific exit stabilization during offline training. For Transformer-based language-model workloads, UniExit reuses layer-wise exit heads or pretrained early-exit variants. This implementation keeps model-specific exits separate from the shared online runtime, allowing the same latency-aware controller to operate across CNN and language-model workloads.

7 Evaluation

7.1 Experimental Setup

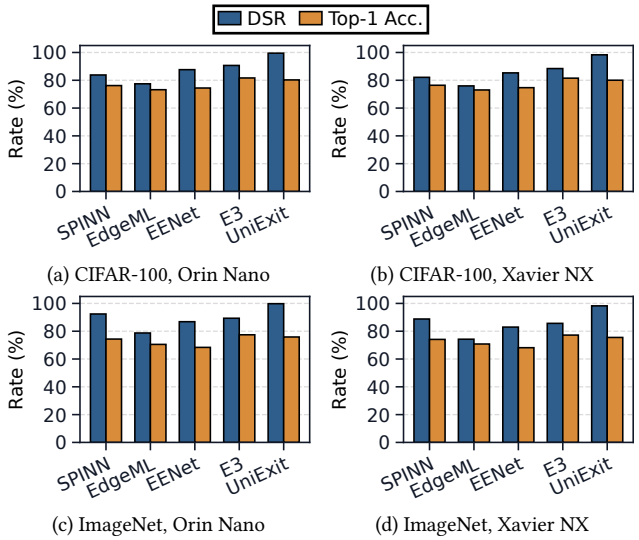
Testbed. We evaluate UniExit on Jetson Xavier NX, Jetson Orin Nano, and an edge–cloud platform with an NVIDIA RTX 5880 Ada GPU. Table 2 summarizes their hardware specifications; shared memory and system services reduce usable embedded-device memory by approximately 2 GB from nominal capacity.

Workloads and models. For CNN workloads, we use ResNet-50 [16] and MobileNetV2 [46] to cover both computation-intensive and lightweight perception models. Exit points are inserted at FLOPs-equidistant locations, and ResNet-50 is used as the default CNN backbone unless otherwise noted.

For language-model workloads, we use the 16-layer LLaMA 3.2-1B [40], which runs fully on the target platforms without quantization. We focus on this edge-deployable 1B-scale model because larger models require quantization, offloading, or external acceleration, confounding the runtime-control problem. For standalone UniExit, all 16 layers are candidate exits and share the model's original final LM head.

Datasets. We use standard datasets for both perception and language-model workloads:

- ***CNN inference.*** We use CIFAR-100 [26] and ImageNet [8]. Training sets are used for supervised training, and test sets are used to generate inference requests. CIFAR-100 images keep their original 32×32 resolution, while ImageNet images are resized to 224×224. To increase input diversity, we augment 30% of test samples with random cropping and horizontal flipping.



(a) CIFAR-100, Orin Nano (b) CIFAR-100, Xavier NX

(c) ImageNet, Orin Nano (d) ImageNet, Xavier NX

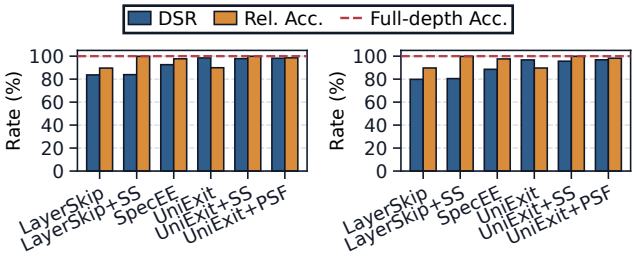
Figure 5: Static single-workload performance of CNN early-exit methods across datasets and edge devices.

- ***Language-model inference.*** We use WikiText-103 [39], a standard language-modeling benchmark, for LLaMA 3.2-1B inference. Each input generates a fixed-length output of 64 tokens. We use the full-depth model output on the same inputs as the reference and measure the output consistency of early-exit inference against this reference.

Baselines. We compare UniExit with representative early-exit architectures and decision methods:

- ***CNN early-exit architectures.*** We compare with SPINN [28] and E3 [32], which represent progressive CNN architectures with intermediate branches. For fairness, branch positions and branch counts are aligned with UniExit where applicable.
- ***CNN exit-decision methods.*** We compare with EdgeML [61] and EENet [20], which optimize early-exit decisions over branch-based CNN networks. These baselines evaluate the benefit of UniExit's latency-aware runtime control beyond conventional confidence-based or static-profile-based decisions.
- ***Transformer-based language-model methods.*** We compare with LayerSkip [9] and SpecEE [54] as representative progressive inference methods for language-model workloads. Because UniExit only changes runtime routing and does not modify language-model weights or exit heads, it can be combined with LayerSkip and SpecEE to evaluate orthogonality.
- ***Concurrent DNN runtime methods.*** We compare with Pantheon [14] and HaX-CoNN [7], which target concurrent DNN execution under edge GPU and heterogeneous SoC resource contention, respectively. Both baselines evaluate UniExit against contention-aware coordination methods during concurrency.

Parameter settings. For CNN training, we use SGD with an initial learning rate of 0.1, weight decay of 1×10^{-4}, and a batch size of 64. CIFAR learning rates decay at 50% and 75% of training epochs, while ImageNet learning rates decay every 30 epochs. The inference batch size is 1 to emulate frame-by-frame edge workloads. Unless otherwise specified, the relative deadline for each workload is set to its full-depth inference latency under the corresponding hardware



(a) Jetson Orin Nano (b) Jetson Xavier NX

Figure 6: Static single-workload performance of Transformer-based language-model early-exit methods. The dashed line denotes the full-depth reference for relative accuracy.

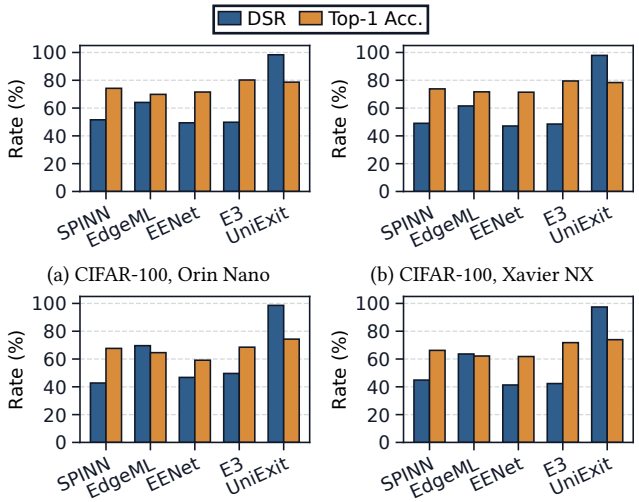
setting. Under static maximum-power operation, the deadlines are 15 ms for CNN inference and 70 ms for language-model TPOT; under dynamic DVFS, they are 16 ms and 118 ms, respectively. The same deadline is used across methods within each workload and setting for fair comparison. Predictor MLPs use two hidden layers with 64 and 32 neurons. For UniExit-specific parameters, the base safety margin ΔS is set from an upper percentile of the offline prediction-error distribution and augmented online using the gap between periodic worst-case latency and current prediction. For CNN stabilization, validation search selects $w_0 = 0.5$ and $\alpha = 0.2$, and $\lambda_{\text{mono}} = 0.1$. Baseline-specific parameters are tuned on validation settings to maximize their performance.

Metrics. All results are averaged over five runs. We report: (i) **deadline and latency**: *Deadline Satisfaction Rate (DSR)*, defined as the fraction of requests meeting their deadlines for CNNs and the fraction of inter-token generation intervals meeting their per-token deadlines for language models (TPOT DSR), and *latency saving rate*; (ii) **prediction quality**: *CNN top-1 accuracy* and *language-model relative accuracy*, defined as the percentage of generated tokens matching the full-depth reference at corresponding positions on the same inputs; (iii) **runtime efficiency**: *latency prediction error* and *FLOPs* relative to the full backbone; and (iv) **resource cost**: task-level *energy consumption*, computed by integrating device power over the inference interval.

7.2 Single-Workload Performance

We first evaluate UniExit under isolated single-workload execution for both CNN and Transformer-based language-model workloads. We consider two controlled settings: (1) *static*, with CPU/GPU/EMC frequencies locked at maximum, and (2) *controlled dynamic DVFS*, where all three frequencies are varied jointly. Orin Nano provides 22/8/4 CPU/GPU/EMC levels and Xavier NX 25/15/6. Starting from maximum frequencies, all dimensions decrease simultaneously by one level every 10 ms; dimensions reaching the minimum early remain there until all reach their minima, after which the sequence reverses analogously to the maxima. This down–up trace repeats throughout execution and is identical across methods, isolating DVFS variation without co-running workloads.

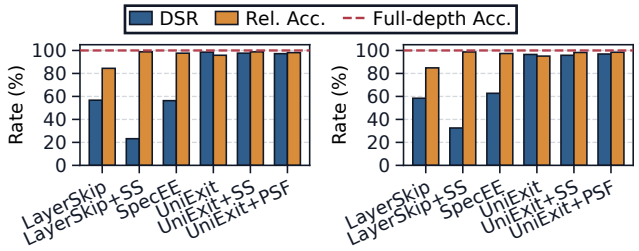
Static hardware states. Figure 5 shows that UniExit achieves the highest and most stable DSR, approaching 100% across datasets and devices, while outperforming most baselines in accuracy. SPINN, EENet, and E3 also achieve high DSR in some settings but remain sensitive to device- and deadline-dependent latency gaps between



(a) CIFAR-100, Orin Nano (b) CIFAR-100, Xavier NX

(c) ImageNet, Orin Nano (d) ImageNet, Xavier NX

Figure 7: CNN early-exit performance under controlled dynamic DVFS across datasets and devices.



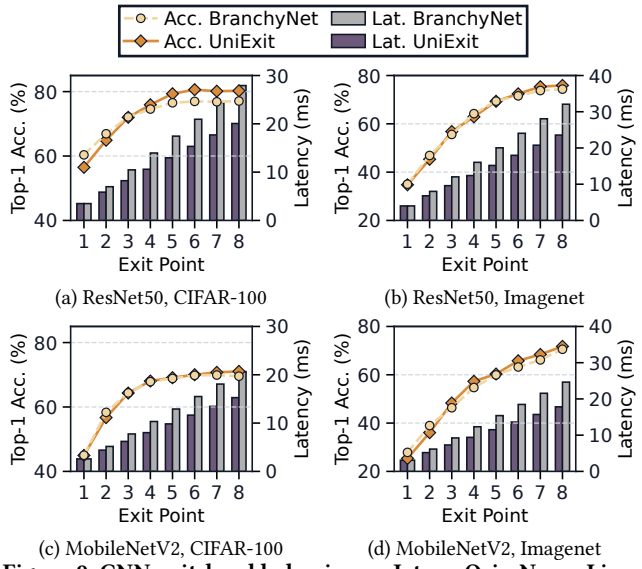
(a) Jetson Orin Nano (b) Jetson Xavier NX

Figure 8: LLM early-exit performance under controlled dynamic DVFS. Dashed line: full-depth relative accuracy.

adjacent exits; EdgeML is less stable due to online-update overhead. UniExit outperforms SPINN, EdgeML, and EENet in most settings and remains close to E3, which benefits from accuracy-oriented feature enhancement. Overall, combining stable CNN exits with latency-aware routing improves the accuracy–deadline trade-off, showing that even under static hardware, routing must determine whether the next exit is reachable within the remaining budget.

Figure 6 reports the corresponding results for Transformer-based language-model workloads. Standalone UniExit achieves the most stable DSR by making latency-aware routing decisions, but its relative accuracy is lower than accuracy-oriented methods such as LayerSkip and SpecEE, since it does not explicitly model token-level output quality. We further combine UniExit with LayerSkip's self-speculative mechanism (UniExit w/ SS) and SpecEE's probability-shift feature predictor (UniExit w/ PSF). Both combinations improve relative accuracy toward the full-depth reference while preserving high DSR, showing that UniExit is orthogonal to existing accuracy-aware early-exit mechanisms.

Controlled dynamic-DVFS states. Figure 7 shows that dynamic DVFS substantially degrades existing CNN early-exit methods. Static-profile-based methods such as SPINN, EENet, and E3 suffer large DSR drops because offline latency tables become stale when hardware frequencies change. EdgeML adapts online, but its updates are too coarse-grained to track fast DVFS variations.



(a) ResNet50, CIFAR-100 (b) ResNet50, Imagenet

(c) MobileNetV2, CIFAR-100 (d) MobileNetV2, Imagenet

Figure 9: CNN exit-level behavior on Jetson Orin Nano. Lines show top-1 accuracy and bars show cumulative latency.

In contrast, UniExit maintains high DSR across all CNN settings while preserving strong top-1 accuracy, showing that runtime-state-aware prediction and deadline-aware routing remain effective. The enlarged gap under dynamic DVFS confirms that stale latency profiles are a major source of deadline misses.

Figure 8 reports Transformer-based language-model results under dynamic DVFS; the dashed line marks the full-depth relative-accuracy reference. LayerSkip and SpecEE suffer clear DSR degradation because their routing does not adapt to runtime latency changes. Standalone UniExit maintains substantially higher DSR by adapting to current runtime state and remaining budget, though with lower relative accuracy than accuracy-oriented methods. The UniExit w/ SS and UniExit w/ PSF variants recover relative accuracy toward the reference while preserving high DSR, confirming latency-safe routing for accuracy-aware early-exit mechanisms.

Exit-level behavior. We use the static setting to isolate exit-level behavior, since dynamic DVFS mainly changes latency values rather than selected-exit accuracy. Figures 9 and 10 further explain where the latency–accuracy gains of UniExit come from. For CNN workloads, we compare against a BranchyNet-style execution pattern [49], which evaluates each encountered branch classifier before

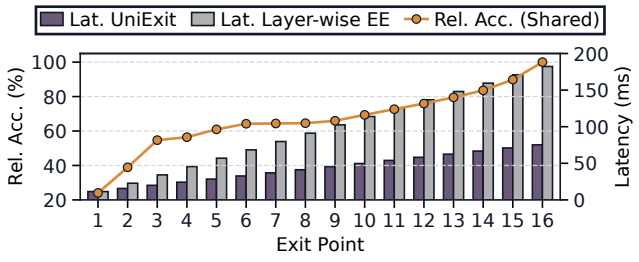
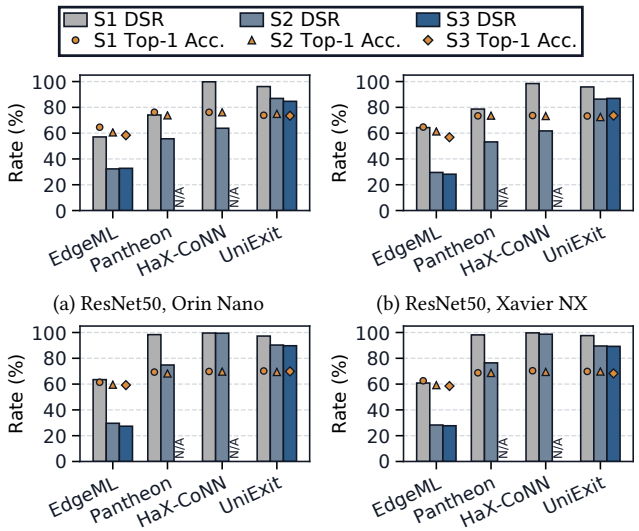


Figure 10: Exit-level behavior of the Transformer language model on Orin Nano. Bars show cumulative latency; the line shows *shared* relative accuracy at selected exits.



(a) ResNet50, Orin Nano (b) ResNet50, Xavier NX

(c) MobileNetV2, Orin Nano (d) MobileNetV2, Xavier NX

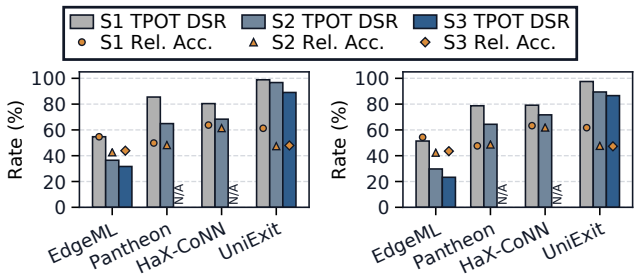
Figure 11: CNN performance under heterogeneous concurrent workloads: DSR and Top-1 accuracy. S1: CNN+CNN; S2: CNN+CNN+Map+Audio; S3: CNN+LLM+Map+Audio, with LLaMA 3.2-1B as the LLM.

deciding whether to exit or continue. As shown in Figure 9, UniExit consistently reduces cumulative latency across exit points by deciding whether to bypass an exit module before executing it, avoiding repeated branch-evaluation overhead. With two stages between adjacent exits, UniExit advances 1.5× faster than BranchyNet, further amplifying latency savings at deeper exits. In terms of accuracy, UniExit is slightly lower at a few shallow exits but becomes comparable to or better than BranchyNet at later exits, indicating that CNN-specific exit stabilization improves deeper-exit reliability.

For Transformer-based language-model workloads, we compare against layer-wise EE [19] that evaluates intermediate output heads along the layer sequence. As shown in Figure 10, UniExit and layer-wise EE share the same relative accuracy at the selected exit layer because they use the same intermediate language-model output. The difference is execution cost: layer-wise EE evaluates intermediate heads along the way, while UniExit skips unselected heads and executes only the selected exit. UniExit thus preserves the same exit-level relative accuracy while substantially reducing cumulative latency, enabling about 2.0× higher inference progress per unit time than layer-wise EE. These results show that UniExit improves the latency–accuracy trade-off by spending latency budget on backbone progression rather than redundant exit-module evaluation.

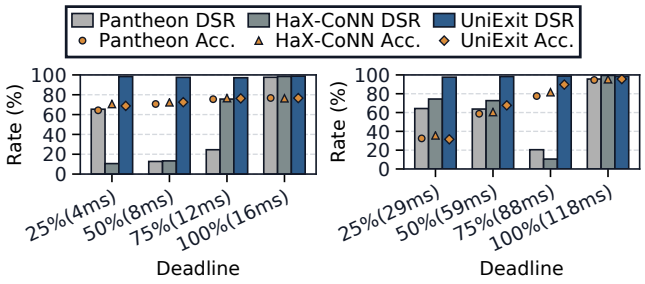
7.3 Concurrent Heterogeneous Workloads

Under platform-managed DVFS, we evaluate four settings: heterogeneous workloads, time-varying workloads, LUT-based prediction, and deadline sensitivity. Heterogeneous experiments run ResNet-50, MobileNetV2, and LLaMA 3.2-1B on Orin Nano and Xavier NX under three scenarios: two same-type models, two same-type models+Map+Audio, and CNN+LM+Map+Audio. Other settings use ResNet-50 and LLaMA 3.2-1B on Orin Nano. Time-varying execution spans six equal periods repeating single model, two same-type models, and two same-type models+Map+Audio twice. The LUT



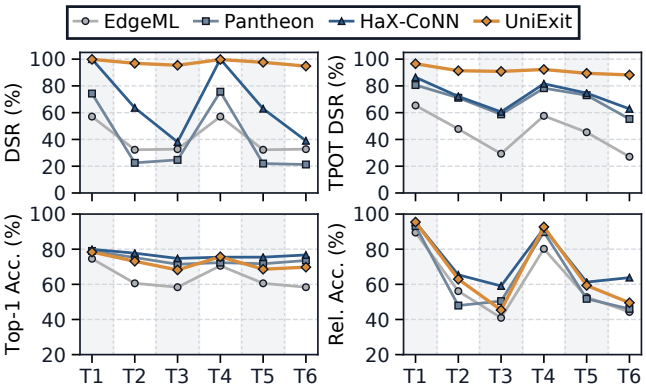
(a) LLaMA-1B, Orin Nano (b) LLaMA-1B, Xavier NX

Figure 12: LLM performance under heterogeneous concurrent workloads: TPOT DSR and relative accuracy to full-depth inference. S1: LLM+LLM; S2: LLM+LLM+Map+Audio; S3: CNN+LLM+Map+Audio, with ResNet-50 as the CNN.



(a) CNN (b) LLM

Figure 13: Deadline sensitivity under single-model inference: CNN DSR/Top-1 accuracy at 4–16 ms; LLM TPOT DSR/relative accuracy at 29–118 ms TPOT.



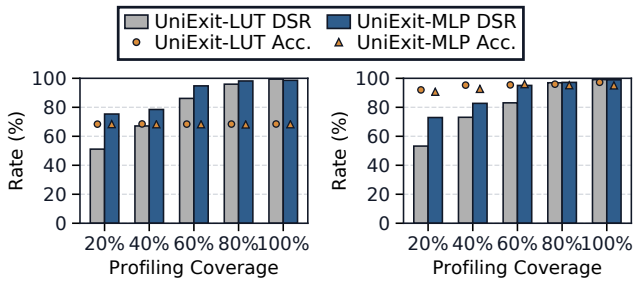
(a) CNN (b) LLM

Figure 14: Performance under time-varying contention over six periods: T1/T4, single model; T2/T5, two models; T3/T6, two models+Map+Audio. CNN reports DSR/Top-1 accuracy and LLM reports TPOT DSR/relative accuracy.

study replaces the MLP with a DVFS- and utilization-indexed LUT under single-model execution, while deadline sensitivity evaluates methods at 25%, 50%, 75%, and 100% of the default deadline.

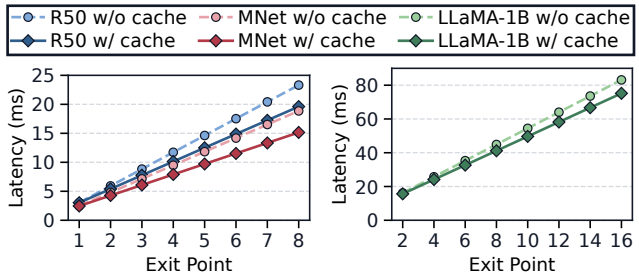
Figures 11 and 12 show that UniExit maintains high DSR and stable prediction quality under S1. Although heavier CPU/GPU/EMC contention and DVFS variation in S2–S3 reduce DSR, UniExit avoids severe degradation, preserves reasonable prediction quality, and remains robust under heavy heterogeneous workloads.

Figure 13 shows that UniExit maintains about 98% DSR for both workloads as the deadline tightens from 100% to 25%, whereas



(a) CNN (b) LLM

Figure 15: Profiling-coverage sensitivity of LUT- and MLP-based prediction. CNN reports DSR/Top-1 accuracy; LLM reports TPOT DSR/relative accuracy.



(a) CNN models (b) LLaMA3.2-1B

Figure 16: Latency reduction from cache-based prediction reuse. Enabling cache lowers average inference latency across exit points by avoiding redundant predictor invocations.

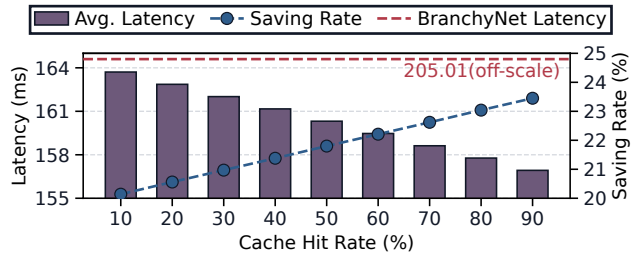


Figure 17: Effect of cache hit rate under dynamic DVFS. Higher cache reuse reduces average inference latency and increases latency saving over branch-based early exit.

Pantheon and HaX-CoNN degrade substantially and fluctuate more. Looser deadlines enable deeper exits and higher Top-1/relative accuracy, while DSR saturation at 100% narrows the gap. Overall, UniExit adapts reliably across latency budgets while preserving quality comparable to the baselines.

Figure 14 shows baseline DSR drops from single-model execution (T1) to two same-type models (T2) and further under Map+Audio contention (T3), recovers at T4, and declines again in T5–T6. In contrast, UniExit maintains high DSR and adapts quickly across transitions while preserving comparable Top-1/relative accuracy.

Figure 15 shows that LUT DSR approaches the MLP as profiling coverage increases, whereas the MLP performs better under limited coverage, indicating better generalization to insufficiently profiled states; Top-1 and relative accuracy remain comparable. Table 3 shows that LUT lookup is faster, but its memory reaches 7.80/171.57 MiB at 100% coverage versus 0.24 MiB for the MLP.

Table 3: Memory and runtime overhead of LUT- and MLP-based latency prediction across profiling coverage. *S./I.*: synchronized/independent CPU-frequency scaling; *C./L.*: CNN/LLM. Memory is reported as S./I. and latency as C./L..

Coverage	LUT-based Prediction		MLP-based Prediction	
	Mem. (MiB, S./I.)	Lat. (ms, C./L.)	Mem. (MiB, S./I.)	Lat. (ms, C./L.)
20%	1.56 / 34.32	0.36 / 0.14	0.24 / 0.24	0.60 / 0.18
40%	3.12 / 68.63	0.33 / 0.17	0.24 / 0.24	0.61 / 0.22
60%	4.68 / 102.95	0.29 / 0.15	0.24 / 0.24	0.61 / 0.23
80%	6.24 / 137.26	0.24 / 0.13	0.24 / 0.24	0.60 / 0.23
100%	7.80 / 171.57	0.17 / 0.11	0.24 / 0.24	0.62 / 0.23

Table 4: Latency prediction error under dynamic DVFS. Each cell reports MAE (ms) / MAPE (%) / P95 APE (%). C: CNN; LM: language model; R50: ResNet-50; MNet: MobileNetV2; C100: CIFAR-100; INet: ImageNet; WT103: WikiText-103.

Type	Model	Data	Dev.	Stage	Exit
C	R50	C100	Orin	0.112 / 1.08 / 3.42	0.086 / 0.94 / 3.05
C	R50	C100	Xavier	0.147 / 1.31 / 3.86	0.113 / 1.17 / 3.48
C	R50	INet	Orin	0.168 / 1.42 / 4.11	0.129 / 0.98 / 3.29
C	R50	INet	Xavier	0.214 / 1.71 / 4.67	0.163 / 1.53 / 4.25
C	MNet	C100	Orin	0.094 / 1.14 / 3.36	0.071 / 0.88 / 2.91
C	MNet	C100	Xavier	0.126 / 1.38 / 3.92	0.096 / 1.12 / 3.37
C	MNet	INet	Orin	0.139 / 1.18 / 3.51	0.104 / 0.84 / 3.07
C	MNet	INet	Xavier	0.181 / 1.79 / 4.74	0.138 / 1.47 / 4.19
LM	LLaMA-1B	WT103	Orin	0.230 / 1.62 / 5.07	0.342 / 1.32 / 4.20
LM	LLaMA-1B	WT103	Xavier	0.689 / 2.87 / 6.47	0.479 / 1.46 / 4.71

Table 5: Offline deployment cost on Jetson Orin Nano; training time and FP32 size include both stage and exit predictors.

Metric	CNN	LM
DVFS/utilization states	17,600	17,600
Profiling records	52,800	52,800
Profiling time	≈2.73 h	≈29.63 h
Two-MLP training time	≈5.15 min	≈5.14 min
FP32 parameter size	20.01 KiB	19.51 KiB

Thus, the MLP offers better low-coverage DSR with fixed memory while avoiding exhaustive profiling.

Overall, UniExit provides stable latency-aware control across models, devices, contention patterns, and deadlines, while MLP-based prediction improves adaptability to limited profiling and dynamic runtime states.

7.4 Runtime Efficiency and Overhead

Latency prediction accuracy. Table 4 reports the error of UniExit's runtime-state-aware predictors under dynamic DVFS. Stage and exit predictors are separated because backbone stages and exit modules exhibit different latency distributions. For CNN workloads, MAPE stays below 1.8% and P95 APE below 4.8% across models, datasets, and devices. The language-model workload shows higher stage-latency error on Xavier NX due to its more complex DVFS and memory behavior, but the worst MAPE remains below 3.3%. These errors are low enough for per-exit routing, with tail cases handled by fallback.

Offline deployment cost. Table 5 summarizes the one-time profiling and predictor-training cost on Jetson Orin Nano. These costs are

Table 6: FLOPs ratio relative to the original backbone.

Method	ResNet-50	MobileNetV2	LLaMA 3.2-1B
Ref. EE	133.9%	145.7%	418.8% / 248.8%
UniExit	101.4%	111.6%	100.0%

Ref. EE denotes BranchyNet-style CNN and layer-wise language-model early exit. For LLaMA 3.2-1B, 418.8% and 248.8% correspond to 16 and 8 exits, respectively.

Table 7: Effect of fallback under dynamic DVFS. FB: Fallback; Values are DSR (%); Gain is the absolute improvement.

Dataset	Device	w/o FB	w/ FB	Gain
CIFAR-100	Orin	94.18	98.24	+4.06
	Xavier	93.62	97.65	+4.03
ImageNet	Orin	94.37	98.56	+4.19
	Xavier	92.91	97.68	+4.77
WikiText-103	Orin	92.82	97.41	+4.59
	Xavier	90.72	96.46	+5.74

incurred once per workload–platform pair and amortized across inference runs; a new platform requires recalibration but no changes to the shared runtime. CNN exit stabilization is performed only when preparing a new CNN early-exit model.

Code reuse and integration cost. Shared code accounts for 47.2% of the runtime (52.4% excluding the platform adapter), while CNN and language-model integration requires only 84 and 152 model-specific SLOC, respectively. Compared with separate runtimes, the shared design reduces logical statements and control-flow branches by 9.3% and 17.8%, respectively.

Cache effectiveness. Figures 16 and 17 evaluate cache-based prediction reuse under static and dynamic settings. Under static hardware, caching reduces average latency across exits, particularly deeper ones with more accumulated predictor calls. Under dynamic DVFS, different change periods yield different hit rates; greater reuse further reduces latency and increases savings over branch-based early exit. Thus, caching remains effective when states are stable or recur over short intervals. Prediction quality remains unaffected because caching skips only redundant predictor calls without altering model execution or exit semantics.

Computation and energy overhead. Table 6 shows that conventional early-exit architectures incur substantial computation by executing intermediate exits: *Ref. EE* increases FLOPs to 133.9% for ResNet-50, 145.7% for MobileNetV2, and much higher for LLaMA 3.2-1B due to costly intermediate language-model heads. In contrast, UniExit routes before executing exit modules and skips unselected exits, keeping FLOPs near the original backbone.

Figure 18 shows that these computational savings reduce task-level energy consumption. UniExit consumes less energy than branch-heavy CNN baselines and achieves the lowest energy consumption for language-model workloads, confirming that latency-aware routing reduces both computation and energy overhead.

7.5 Ablation Studies

Effect of fallback. Table 7 shows that fallback consistently improves DSR under dynamic DVFS. Without it, UniExit may underestimate remaining latency when frequencies drop abruptly or cross-exit latency changes faster than the predictor can capture. Fallback increases DSR by 4.03%–5.74% using a more conservative safety margin or historical latency estimate. The largest gain occurs for the language-model workload on Xavier NX, which exhibits more variable runtime states. This confirms that fallback handles volatile hardware states rather than replacing latency prediction.

Table 8: Effect of CNN-specific exit stabilization. Values are top-1 accuracy (%). Equal: equal-weighted exit training; WEL: weighted exit loss; Mono: monotonicity regularization.

Model	Data	Equal	WEL	Mono	WEL+Mono
ResNet-50	CIFAR-100	76.19	78.64	77.52	**80.26**
	ImageNet	74.35	75.12	74.86	**75.85**
MobileNetV2	CIFAR-100	71.67	72.18	71.96	**72.59**
	ImageNet	68.90	69.31	69.17	**69.68**

Effect of CNN-specific exit stabilization. Table 8 evaluates two CNN stabilization losses. *Equal* denotes conventional equal-weighted exit training; *WEL* and *Mono* denote weighted exit loss and monotonicity regularization, respectively. Both improve early-exit accuracy, and their combination performs best across all model–dataset pairs. They are complementary: WEL reduces shallow-exit dominance during shared-backbone training, while Mono stabilizes correct-class confidence across adjacent exits.

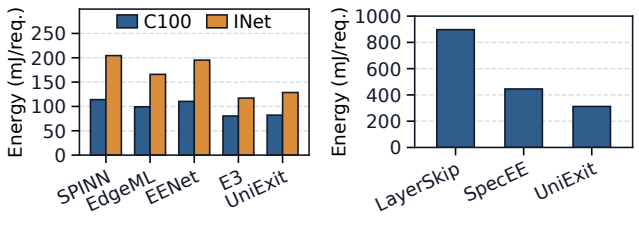
Figure 19 evaluates exit stability through the fraction of test samples whose maximum correct-class confidence drop exceeds different thresholds. Compared with BranchyNet-style training, UniExit consistently reduces large confidence drops across both datasets and backbones, making deeper CNN exits more reliable when the online router continues within the latency budget.

Together, these ablations show that robust routing requires both runtime-level fallback and model-level exit stabilization.

7.6 Generalization to Edge–Cloud Inference

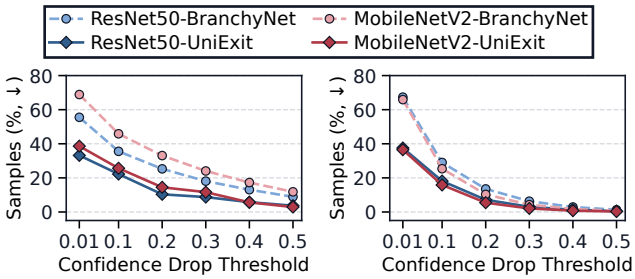
We assess UniExit's generalization beyond edge-only execution via CNN-based edge–cloud split inference. We focus on CNNs because their feature offloading follows standard split inference, whereas language-model offloading requires transferring hidden states or KV caches and adds scheduling variables [23]. For simplicity, we split the early-exit CNN midway: Jetson Orin Nano runs the first part and the cloud the remainder. To avoid routing errors from inaccurate transmission estimates, the edge uploads intermediate features, obtains measured transmission feedback, then triggers cloud-side inference with an aligned start signal. After offloading, the edge continues local inference in parallel and retains the edge-side result as a fallback. If the cloud result is not received before the deadline, UniExit returns the edge-side result as output.

Figure 20 shows that UniExit maintains DSR above 95% across models, datasets, and feature-compression ratios, with most settings



(a) CNN workloads (b) LM workload

Figure 18: Inference energy on Jetson Orin Nano across CNN and language-model workloads.



(a) CIFAR-100 (b) ImageNet

Figure 19: Confidence degradation across CNN exits. UniExit reduces the fraction of samples whose maximum correct-class confidence drop exceeds each threshold.

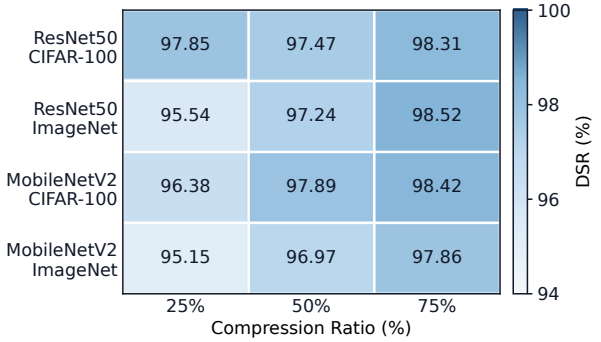


Figure 20: Edge–cloud generalization of UniExit. The heatmap reports DSR under different CNN workloads and intermediate-feature compression ratios.

near or above 97%. Higher compression generally improves DSR by reducing transmission latency and preserving more time for cloud execution. Minor non-monotonic variations suggest that end-to-end timing also depends on model-specific computation and transmission variability. Overall, UniExit remains effective in edge–cloud execution with measured transmission feedback.

8 Related Work

Early-exit inference for CNNs and Transformer-based models. Early-exit networks lower average inference cost by exposing intermediate predictions and terminating easy inputs. For CNNs, BranchyNet [49] attaches side branches, and MSDNet [18] uses dense connectivity and multi-scale representations for anytime prediction; later work studies branch placement, confidence thresholds, exit training, and accuracy–computation trade-offs [25, 45]. For Transformers, DeeBERT [52], FastBERT [36], PABEE [63], and BERxiT [53] use intermediate classifiers, entropy criteria, patience, or improved fine-tuning. LayerSkip [9] and SpecEE [54] use layer-wise exits with self-speculation or probability-shift features for efficient generation. Although effective model-side mechanisms, their exit heads, confidence signals, and rules remain model-family specific. Energy-oriented studies adapt execution to energy budgets in energy-harvesting or batteryless settings, including HarvNet [22], nonuniformly compressed multi-exit networks for intermittent inference [51], and FreeML [10]. In contrast, UniExit extends model-side optimization to latency/deadline-aware, system-level online routing in dynamic edge environments.

Edge-oriented progressive inference systems. Progressive systems also target edge or edge–cloud deployment under latency, accuracy, energy, or budget constraints. SPINN [28] combines progressive CNN inference with device–cloud execution and jointly considers early exit and model splitting; EdgeML [61] adapts policies to changing environments; and EENet [20] and E3 [32] optimize exit scheduling or progressive architectures for budgeted inference. Pantheon [14] and HaX-CoNN [7] study concurrent DNN execution under mobile edge GPU and heterogeneous SoC contention. Although motivating runtime coordination of early-exit and concurrent inference, many rely on offline profiles, static tables, or periodic policy updates that become inaccurate under DVFS changes, thermal effects, and contention, especially per input or token; most also focus on CNN/DNNs. UniExit instead uses runtime state, inference progress, and latency budgets for online per-exit decisions through one staged interface for CNN and Transformer-based language-model workloads.

Latency-aware edge inference and offloading. Broader edge inference studies model partitioning [24, 30], offloading [30, 59], latency prediction [41, 56], and hardware-aware adaptation [58]. Neurosurgeon [24] partitions DNNs between mobile devices and the cloud, Edgent [30] combines partitioning with model right-sizing, and other work addresses distributed inference, latency modeling, and DVFS-aware adaptation [41, 56, 58, 59]. These systems expose the importance of hardware state, network conditions, and contention, but usually operate per model, request, or partition rather than within the early-exit loop. They lack runtime-state-aware, deadline-driven control at every candidate exit across heterogeneous model families; UniExit complements them with online runtime-state-aware latency estimation and remaining-budget tracking for per-exit routing in heterogeneous edge AI.

9 Conclusion

We present UniExit, a unified latency-aware early-exit runtime for heterogeneous edge AI workloads. It treats CNN blocks and Transformer layers as ordered stages with common latency-aware routing while retaining model-specific exit-head design and quality mechanisms. Runtime-state-aware latency prediction, cache reuse, trend-aware fallback, and CNN-specific exit stabilization enable low-overhead online decisions under dynamic edge conditions. Evaluations on edge and edge–cloud platforms show improved deadline satisfaction and latency–accuracy trade-offs, with less redundant computation and energy overhead, across representative CNN and Transformer-based language-model workloads. The results validate a shared stage-wise runtime-control abstraction for latency-aware early-exit inference across heterogeneous edge workloads.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their constructive feedback. This work is supported by the National Natural Science Foundation of China under Grant 62572416, The HKUST-HKUST(GZ) Cross-campus Research Collaboration "1+1+1" Joint Funding Program under G_2025_052, and Key Areas Project of the Guangdong Provincial Department of Education under Grant No. 2026ZDZX1022. The corresponding author is Zhidan Liu.

References

[1] Ahmed A. Al-Saedi, Veselka Boeva, and Emiliano Casalicchio. 2022. Context-Aware Edge-Based AI Models for Wireless Sensor Networks: An Overview. *Sensors* 22, 15 (2022), 5544.

[2] Arian Bakhtiarnia, Qi Zhang, and Alexandros Iosifidis. 2021. Improving the Accuracy of Early Exits in Multi-Exit Architectures via Curriculum Learning. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*. 1–8.

[3] Laszlo A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal* 5, 2 (1966), 78–101.

[4] Jiahe Cao, Xiaomeng Li, Qiang Liu, Tao Han, Ning Zhang, and Weisong Shi. 2026. EdgeServing: Deadline-Aware Multi-DNN Serving at the Edge. *arXiv preprint arXiv:2605.05527* (2026).

[5] Yichong Chen, Zifeng Niu, Manuel Roveri, and Giuliano Casale. 2025. CEED: Collaborative Early Exit Neural Network Inference at the Edge. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. 1–10.

[6] Haolin Chu, Xiaolong Zheng, Liang Liu, and Huadong Ma. 2023. nnPerf: Demystifying DNN Runtime Inference Latency on Mobile Platforms. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*. 125–137.

[7] Ismet Dagli and Mehmet E. Belviranli. 2024. Shared Memory-contention-aware Concurrent DNN Execution for Diversely Heterogeneous System-on-Chips. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 243–256.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 248–255.

[9] Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bolei Wasti, Liangzhen Lai, Ahmed Mahmoud, Bilge Acun, Saurabh Agarwal, Adina Roman, Ahmed Aly, Beidi Chen, Carole-Jean Wu, Elena Presani, Mary Brooks, Jeff Clune, Saman Hosseini, Leshem Choshen, and Michael Auli. 2024. LayerSkip: Enabling Early Exit Inference and Self-Speculative Decoding. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. 12622–12642.

[10] Pietro Farina, Subrata Biswas, Eren Yıldız, Khakim Akhunov, Saad Ahmed, Bashima Islam, and Kasım Sinan Yildirim. 2024. Memory-efficient Energy-adaptive Inference of Pre-Trained Models on Batteryless Embedded Systems. In *Proceedings of the 2024 International Conference on Embedded Wireless Systems and Networks (EWSN)*. 82–93.

[11] Ziyan Fu, Yongheng Deng, Li Li, Yingjun Wu, Zhibo Wang, Su Yao, Yaoxue Zhang, and Ju Ren. 2025. Malva: A Jitter-Aware Online Pruning Framework for DNN Inference Tasks. In *Proceedings of the IEEE/ACM International Symposium on Quality of Service (IWQoS)*. 1–10.

[12] Saeed Khalilian Gourtani, Ehsan Aghapour, Nirvana Meratnia, Andy Pimentel, and Anuj Pathania. 2025. Early-Exit DNN Inference on HMPSoCs. In *Proceedings of the IEEE International Conference on Edge Computing and Communications (EDGE)*. 75–82.

[13] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A Survey of Deep Learning Techniques for Autonomous Driving. *Journal of Field Robotics* 37, 3 (2020), 362–386.

[14] Lixiang Han, Zimu Zhou, and Zhenjiang Li. 2024. Pantheon: Preemptible Multi-DNN Inference on Mobile Edge GPUs. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 465–478.

[15] Yunchu Han, Zhaojun Nan, Sheng Zhou, and Zhisheng Niu. 2025. DVFS-Aware DNN Inference on GPUs: Latency Modeling and Performance Analysis. In *Proceedings of the IEEE International Conference on Communications (ICC)*. 1274–1279.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[17] Kyle Hoffpauir, Jacob Simmons, Nikolas Schmidt, Rachitha Pittala, Isaac Briggs, Shanmukha Makani, and Yaser Jararweh. 2023. A Survey on Edge Intelligence and Lightweight Machine Learning Support for Future Applications and Services. *Journal of Data and Information Quality* 15, 2 (2023), 20:1–20:30.

[18] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. 2018. Multi-Scale Dense Networks for Resource Efficient Image Classification. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[19] Lianming Huang, Shangyu Wu, Yufei Cui, Ying Xiong, Haibo Hu, Xue Liu, Tei-Wei Kuo, Nan Guan, and Chun Jason Xue. 2026. RAEE: A Robust Retrieval-Augmented Early Exit Framework for Efficient Inference. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[20] Fatih Ilhan, Ka-Ho Chow, Sihao Hu, Tiansheng Huang, Selim Tekin, Wenqi Wei, Yanzhao Wu, Myungjin Lee, Ramana Kompella, Hugo Latapie, Gaowen Liu, and Ling Liu. 2024. Adaptive Deep Neural Network Inference Optimization with EENet. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. 1362–1371.

[21] Metod Jazbec, James Urquhart Allingham, Dan Zhang, and Eric Nalisnick. 2023. Towards Anytime Classification in Early-Exit Architectures by Enforcing Conditional Monotonicity. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*. 56138–56168.

[22] Seunghyeok Jeon, Yonghun Choi, Yeonwoo Cho, and Hojung Cha. 2023. HarvNet: Resource-Optimized Operation of Multi-Exit Deep Neural Networks on Energy Harvesting Devices. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*. 42–55.

[23] Bodon Jeong, Hongsu Byun, Youngjae Kim, Weikuan Yu, Kyungkeun Lee, Jihoon Yang, and Sungyong Park. 2026. DUAL-BLADE: Dual-Path NVMe-Direct KV-Cache Offloading for Edge LLM Inference. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*.

[24] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 615–629.

[25] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. 2019. Shallow-Deep Networks: Understanding and Mitigating Network Overthinking. In *Proceedings of the International Conference on Machine Learning (ICML)*. 3301–3310.

[26] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[27] Nicholas D. Lane, Petko Georgiev, and Lorena Qendro. 2015. Can Deep Learning Revolutionize Mobile Sensing?. In *Proceedings of the International Workshop on Mobile Computing Systems and Applications (HotMobile)*. 117–122.

[28] Stefanos Laskaridis, Stylianos I. Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D. Lane. 2020. SPINN: Synergistic Progressive Inference of Neural Networks over Device and Cloud. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. 37:1–37:15.

[29] Lik-Hang Lee, Tristan Braud, Simo Hosio, and Pan Hui. 2021. Towards Augmented Reality Driven Human-City Interaction: Current Research on Mobile Headsets and Future Challenges. *Comput. Surveys* 54, 8 (2021), 165:1–165:38.

[30] En Li, Zhi Zhou, and Xu Chen. 2018. Edge Intelligence: On-Demand Deep Learning Model Co-Inference with Device-Edge Synergy. In *Proceedings of the Workshop on Mobile Edge Communications (MECOMM)*. 31–36.

[31] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and Quantization for Deep Neural Network Acceleration: A Survey. *Neurocomputing* 461 (2021), 370–403.

[32] Changyao Lin, Zhenming Chen, Ziyang Zhang, and Jie Liu. 2025. E3: Early Exiting with Explainable AI for Real-Time and Accurate DNN Inference in Edge-Cloud Systems. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*. 385–397.

[33] Chengdong Lin, Kun Wang, Zhenjiang Li, and Yu Pu. 2023. A Workload-Aware DVFS Robust to Concurrent Tasks for Mobile Devices. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. 19:1–19:16.

[34] Neiwen Ling, Kai Wang, Yuxin He, Guoliang Xing, and Dantong Xie. 2021. RT-mDL: Supporting Real-Time Mixed Deep Learning Tasks on Edge Platforms. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[35] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge Assisted Real-Time Object Detection for Mobile Augmented Reality. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. 25:1–25:16.

[36] Weijie Liu, Peng Zhou, Zhiruo Wang, Zhe Zhao, Haotang Deng, and Qi Ju. 2020. FastBERT: A Self-distilling BERT with Adaptive Inference Time. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. 6035–6044.

[37] Yuxiao Liu, Rui Han, Qinglong Zhang, Haiting Hou, Chi H. Liu, and Lydia Y. Chen. 2025. On Scheduling Early-Exit Layers for Model Pipeline in 6G-Based Edge Inference. *IEEE Network* 39, 5 (2025), 131–137.

[38] Yuer Ma, Yanyan Wang, and Bin Tang. 2025. Joint Optimization of Model Partitioning and Resource Allocation for Multi-Exit DNNs in Edge-Device Collaboration. *Electronics* 14, 8 (2025), 1647.

[39] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer Sentinel Mixture Models. In *International Conference on Learning Representations (ICLR)*.

[40] Meta AI. 2024. Llama 3.2 Model Card. https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/MODEL_CARD.md. Accessed: 2026-06-02.

[41] Saeejith Nair, Saad Abbasi, Alexander Wong, and Mohammad Javad Shafiee. 2022. MAPLE-Edge: A Runtime Latency Predictor for Edge Devices. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 3659–3667.

[42] Dat Ngo, Hyun-Cheol Park, Bongsoon Kang, Youhuizi Li, and Lanyu Xu. 2025. Edge Intelligence: A Review of Deep Neural Network Inference in Resource-Limited Environments. *Electronics* 14, 12 (2025), 2495.

[43] Xian Peng, Xin Wu, Lianming Xu, Li Wang, and Aiguo Fei. 2024. DistrEE: Distributed Early Exit of Deep Neural Network Inference on Edge Devices. In *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*. 3116–3121.

[44] Guanqiao Qu, Qiyuan Chen, Wei Wei, Zhengyi Lin, Xianhao Chen, and Kaibin Huang. 2024. Mobile Edge Intelligence for Large Language Models: A Contemporary Survey. *IEEE Communications Surveys & Tutorials* 27 (2024), 3820–3860.

[45] Haseena Rahmath P, Vishal Srivastava, Kuldeep Chaurasia, Roberto G. Pacheco, and Rodrigo S. Couto. 2024. Early-Exit Deep Neural Network: A Comprehensive Survey. *Comput. Surveys* 57, 3 (2024), 75:1–75:37.

[46] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 4510–4520.

[47] Prashanthi S.K, Sai Anuroop Kesanapalli, and Yogesh Simmhan. 2022. Characterizing the Performance of Accelerated Jetson Edge Devices for Training Deep Learning Models. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 3 (2022), 44:1–44:26.

[48] Srinivasan Subramaniyan, Rudra Joshi, Xiaorui Wang, and Marco Brocanelli. 2025. SEEB-GPU: Early-Exit Aware Scheduling and Batching for Edge GPU Inference. In *Proceedings of the ACM/IEEE Symposium on Edge Computing (SEC)*. 9:1–9:16.

[49] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2016. BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks. In *Proceedings of the International Conference on Pattern Recognition (ICPR)*. 2464–2469.

[50] Xubin Wang, Zhiqing Tang, Jianxiong Guo, Tianhui Meng, Chenhao Wang, Tian Wang, and Weijia Jia. 2025. Empowering Edge Intelligence: A Comprehensive Survey on On-Device AI Models. *Comput. Surveys* 57, 9 (2025), 228:1–228:39.

[51] Yawen Wu, Zhepeng Wang, Zhenge Jia, Yiyu Shi, and Jingtong Hu. 2020. Intermittent Inference with Nonuniformly Compressed Multi-Exit Neural Network for Energy Harvesting Powered Devices. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[52] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. 2246–2251.

[53] Ji Xin, Raphael Tang, Yaoliang Yu, and Jimmy Lin. 2021. BERxiT: Early Exiting for BERT with Better Fine-Tuning and Extension to Regression. In *Proceedings of the Conference of the European Chapter of the Association for Computational Linguistics (EACL)*. 91–104.

[54] Jiaming Xu, Jiayi Pan, Yongkang Zhou, Siming Chen, Jinhao Li, Yaoxiu Lian, Junyi Wu, and Guohao Dai. 2025. SpecEE: Accelerating Large Language Model Inference with Speculative Early Exiting. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 467–481.

[55] Yuwei Yan, Yikun Hu, Qinyun Cai, Wangdong Yang, and Kenli Li. 2025. DIDS: A Distributed Inference Framework with Dynamic Scheduling Capability. *Future Generation Computer Systems* 162 (2025), 107459.

[56] 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 Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 81–93.

[57] Ziyang Zhang, Yang Zhao, Ming-Ching Chang, and Jie Liu. 2025. E4: Energy-Efficient DNN Inference for Edge Video Analytics via Early Exiting and DVFS. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 1165–1173.

[58] Ziyang Zhang, Yang Zhao, Huan Li, Changyao Lin, and Jie Liu. 2024. DVFO: Learning-Based DVFS for Energy-Efficient Edge-Cloud Collaborative Inference. *IEEE Transactions on Mobile Computing* 23, 10 (2024), 9042–9059.

[59] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.

[60] Zhihe Zhao, Neiwen Ling, Nan Guan, and Guoliang Xing. 2023. Miriam: Exploiting Elastic Kernels for Real-Time Multi-DNN Inference on Edge GPU. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*. 97–110.

[61] Zhihe Zhao, Kai Wang, Neiwen Ling, and Guoliang Xing. 2021. EdgeML: An AutoML Framework for Real-Time Deep Learning on the Edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation (IoTDI)*. 133–144.

[62] Yue Zheng, Yuhao Chen, Bin Qian, Xiufang Shi, Yuanchao Shu, and Jiming Chen. 2025. A Review on Edge Large Language Models: Design, Execution, and Applications. *Comput. Surveys* 57, 8 (2025), 209:1–209:35.

[63] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. 2020. BERT Loses Patience: Fast and Robust Inference with Early Exit. In *Advances in Neural Information Processing Systems (NeurIPS)*. 18330–18341.