# FailLite: Failure-Resilient Model Serving for Resource-Constrained Edge Environments

Li Wu

University of Massachusetts Amherst USA

Walid A. Hanafy University of Massachusetts Amherst

USA

David Irwin University of Massachusetts Amherst USA Jesse Milzman Army Research Laboratory USA Tarek Abdelzaher University of Illinois at Urbana-Champaign USA

Prashant Shenoy University of Massachusetts Amherst USA

# ABSTRACT

Model serving systems have become popular for deploying deep learning models for various latency-sensitive inference tasks. While traditional replication-based methods have been used for failureresilient model serving in the cloud, such methods are often infeasible in edge environments due to significant resource constraints that preclude full replication. To address this problem, this paper presents FailLite, a failure-resilient model serving system that employs (i) a heterogeneous replication where failover models are smaller variants of the original model, (ii) an intelligent approach that uses warm replicas to ensure quick failover for critical applications while using cold replicas, and (iii) progressive failover to provide low mean time to recovery (MTTR) for the remaining applications. We implement a full prototype of our system and demonstrate its efficacy on an experimental edge testbed. Our results using 27 models show that FailLite can recover all failed applications with 175.5ms MTTR and only a 0.6% reduction in accuracy.

#### **1** INTRODUCTION

In recent years, deep learning-based systems have revolutionized many domains, such as autonomous driving, augmented reality, personal assistants, and Internet of Things analytics [20, 22, 37]. The broad adoption of deep learning models is attributed to the availability of accelerators, such as Nvidia GPUs and Google TPUs. To train a Deep Neural Network (DNN) model, users often rely on cloud data centers (e.g., Amazon AWS and Google Cloud) to access their powerful DNN accelerators via simple application interfaces. For example, services like Amazon SageMaker allow users to upload their data and use a simple user interface to select the models to train [3]. In contrast to DNN training, which is largely a batch workload, runtime use of trained models-often called model serving or model inference-tends to be latency-sensitive in nature. To meet the latency requirement of such applications, users often rely on edge data centers placed at the network edge [35, 36]. The advances in edge accelerators (e.g., edge GPUs and TPUs) have made edge computing a popular choice for deploying latency-sensitive models serving applications [23, 37].

With the advances in AI, researchers have addressed many challenges in building model serving systems. For instance, researchers have addressed the latency challenges by optimizing the network overheads [12, 15], processing latency [5, 7, 23, 34, 39, 48, 49], cost [2, 6, 25, 46], energy efficiency [11, 24, 40, 45], and addressing challenges of workload dynamics [1, 2, 47]. However, there is limited

research on the failure resiliency of model serving systems, especially in resource-constrained environments, such as the edge.

The traditional approach for ensuring fault tolerance of an application task is to replicate it on additional servers and to fail over to a backup server when the primary fails. This approach to crash fault tolerance is commonly used for critical cloud-based services. Cloud-based model serving, which is useful when tight latency requirements are not needed, has also employed replication to satisfy demand and latency constraints and to ensure failure resiliency. For instance, a recent effort [39] utilized backup requests to multiple replicas and cross-server cancellation to ensure that all requests are processed within their latency requirements. Which such replication may be feasible in cloud settings where resources are abundant and can be scaled on demand, edge environments are often highly resource constrained, which may preclude replication of deployed models for fault tolerance. Moreover, edge computing clusters are often deployed across multiple geographically distributed locations and may face many single points of failure issues, where the entire cluster becomes out of service or unreachable.

Designing a failure-resilient model serving system at the edge involves several key challenges. First, such a system must address resource constraints where traditional replication and failover solutions may not be feasible. Second, the interactive nature of model serving requires low mean time to recovery (MTTR) in the event of a failure, such that users are minimally impacted. Third, the systems should be able to handle concurrent server failures or the failure of an edge location by restoring applications at other suitable edge sites.

To address this problem, we present FailLite, a failure-resilient machine learning inference system for edge computing clusters. FailLite exploits several well-known accuracy-resource trade-offs for deep learning models, such as choosing the right-sized DNN model [1, 2, 9, 11, 19, 47] and compression techniques, such as quantization and pruning [13, 27, 45], all of which ignificantly decrease resource requirements and latency with a minor reduction in accuracy. The core insight of FailLite is to provide fault tolerance using the concept of *heterogeneous replication*, where we use a smaller variant of the original model as a failover replica, ensuring high resiliency in such resource-constrained environments. Our second insight is that applications can vary in terms of their criticality and failure resilience needs, allowing FailLite to intelligently choose the level of failure resilience on a per-model basis. Specifically, FailLite can intelligently use proactive replication

Conference'17, July 2017, Washington, DC, USA

and warm replicas for the most important models, while using cold replicas with progressive failover for low downtimes for the remaining models. Such an approach can provide good accuracy upon failover despite using smaller models as replicas, while providing low mean time to recovery.

In designing, implementing, and evaluating FailLite, our paper makes the following contributions:

- We present the design of FailLite, a failure-resilient model serving system for resource-constrained environments. To our knowledge, FailLite is the first effort to tackle this issue.
- (2) We present a two-step failover approach that utilizes heterogeneous replication, considers applications' criticality levels, and progressively loads cold backups to optimize their MTTR.
- (3) We implemented FailLite as a framework agnostic control plane with failure detection and failure-resilient policies. We integrated FailLite with the Nvidia Triton Inference Server, an open-source model serving framework to show the utility of our design. We also integrated our FailLite with a custom-built discrete event simulation platform for large-scale evaluations.
- (4) Our experimental evaluation of FailLite of a real test bed using 27 machine learning models shows that in resource constraint environments, FailLite can recover all failed applications under 175.5 ms MTTR only for 0.6% reduction in accuracy. In addition, our large-scale simulation with 69 DNN models shows that in the extreme scenario, where 50% of the edge sites fail simultaneously, FailLite achieves at least 39.3% higher recovery rate compared to the full-size baseline.

#### 2 BACKGROUND AND MOTIVATION

In this section, we provide a background on model serving, failures and fault-tolerance techniques, and trade-offs in DNN models. Lastly, we motivate the need for FailLite by highlighting the key challenges in designing failure-resilient model serving systems.

## 2.1 Model Serving on the Edge

Edge Computing brings cloud-like computational and storage resources to the network's edge and provides users with low-latency applications [35, 36]. Although edge computing was pioneered more than a decade ago, the rise of AI made it more critical due to the resource requirements and performance objective of AI applications that often go beyond the most sophisticated on-device capabilities [20, 37]. The model serving process, also known as model inference, can be described as follows: a user or a sensor node sends input data  $x_i$  to a specific ML inference application, and the application responds with output data  $y_i$ , where  $y_i = f(x_i)$ . The function  $f(\cdot)$  may represent a single model inference or a pipeline of different ML models. Users often rely on model serving frameworks (e.g., Nvidia Triton [30] and Kubeflow [18]) to run their DNN models as they provide many management and monitoring capabilities. Model serving services benefit from edge servers' low latency, the availability of powerful DNN accelerators, and data privacy capabilities to provide users with real-time processing capabilities. For example, applications such as IoT analytics, video streaming, and AR/VR systems heavily rely on edge resources to cover their high computing demand [33, 38].



Figure 1: Approaches to utilize a failover replica.

#### 2.2 Network and Resource Failures

Network and compute resources are prone to multiple types of failures, such as transient, crash, or byzantine failures, impacting the model serving systems' reliability [10, 17]. Resilient execution ensures service availability and correctness in the event of faults and aims to minimize metrics such as downtime or mean time to recovery (MTTR). In this work, we focus on network and server crash failures that make an edge server or a cluster inaccessible or unreachable. Crash failures result from various factors, including human errors, power outages, battery drain, hardware malfunctions, software misconfigurations, and security attacks [41]. Addressing crash failures is critical in edge computing since they are irrecoverable and more likely to occur than in cloud computing, as edge computing resources typically operate in a less controlled environment with limited to no resource redundancy. In the context of model serving systems, little or no work has addressed the resiliency challenges and the effect of server or network crashes. To our knowledge, researchers have only addressed failures in distributed training and inference scenarios [14, 21, 28, 42, 43] but not the entire model's failure. Finally, we note that addressing byzantine failure from adversarial clients (e.g., DNN Perturbation attacks [26]) is outside the scope of this work.

## 2.3 Failover Replication

Replication is a key strategy in ensuring resilient systems, where components are backed with a failover replica. In the case of model serving systems, researchers have used traditional failover replica approaches, where a model can have a hot (also referred to as activeactive), warm (active-passive), or cold failover backup. Figure 1 highlights the difference between hot, warm, and cold backups. For instance, in [39], the authors have used a hot backup (see Figure 1a) where each request is issued twice, and the extra request is canceled once a response is available. To avoid wasting compute cycles, warm backups are used where the DNN models are loaded to memory but do not process any requests. In contrast to hot and warm backups, which are loaded to memory, cold backups are just cached to disk and loaded to memory only when the failure occurs.

Despite the advantages and trade-offs involved with traditional failover replication techniques, edge environments are resourceconstrained, and such failover replication methods fall short in resilient execution. For example, using a hot backup needlessly increases the system's load and average response time, while resources may not allow all applications to have a warm backup. Moreover, unlike cloud resources, adding more resources to edge environments introduces high-cost overheads and may not be feasible due to space and power constraints. To overcome such resource limitations, one approach is to limit the failure recovery to a set



Figure 2: Accuracy-Resource Tradeoff (a) and Loading time (b) across DNN models.

of critical applications or reduce the quality of service, commonly known as *graceful service degradation* [8, 10, 29, 50]. Our proposed FailLite utilizes this method to increase the resiliency of model serving systems by inter-playing the accuracy-resiliency trade-offs.

#### 2.4 DNN Models Trade-offs

Recent research has highlighted the accuracy-memory trade-offs of DNNs [1, 2, 9, 11, 19, 24, 47], where increases in accuracy come at an exponential increase in resource requirements. Figure 2 demonstrates the accuracy-memory trade-offs and the loading time across different pre-trained model families from Pytorch [31]. Figure 2a shows the trade-off between model size and accuracy (we report normalized accuracy to the biggest model) in four different model families. As shown, the figure highlights that significant reductions in memory requirements introduce limited reductions in accuracy. For example, ConvNext-Tiny is  $5.1 \times$  smaller compared to ConvNext-Large but only reduces accuracy by 1.89%. Figure 2b also highlights a key aspect of DNNs where loading time is a function of memory size, where accurate models are typically large, thus requiring higher loading time.

This unique trade-off has motivated users to address runtime issues. For example, the authors of [1, 47] have used model switching to address workload dynamics by replacing the DNN with a smaller variant at peak hours, allowing all requests to abide by their required SLO. Moreover, Clover [19] used model switching to reduce the energy consumption when the energy is generated from brown energy sources. In contrast to these approaches that focus on workload dynamics, FailLite utilizes model variants in optimizing the resiliency of model serving systems by using a compressed or smaller size failover replica, which significantly decreases the resource requirements for the failover replicas, an approach we denote as *heterogeneous replication*.

#### 2.5 Failure Resilience Challenges on the Edge

FailLite leverages the DNN model's flexibility to ensure resilient model serving in case of crash failures in resource constrained edge environments. Designing FailLite involves addressing the following research challenges:

- **C1 Model Selection.** The first challenge is model selection; naively choosing the smallest variant unnecessarily reduces accuracy. In contrast, selecting larger or identical replicas restricts the capacity to provide failover replicas. Therefore, the system must weigh the trade-offs across applications and other factors when choosing the failover models.
- **C2 Balancing Accuracy-MTTR Trade-offs.** The second challenge is that while offering failover backups for all applications highly decreases the MTTR, adding backups for all applications in resource-constrained scenarios unnecessarily reduces the accuracy for applications that fail since resources are allocated to backups of functioning applications. In contrast, cold backups enable us to load the most accurate models, but activating cold backup models takes more time, significantly increasing the MTTR, as loading large models from disk is a lengthy process (see Figure 2b).
- **C3 Model Placement.** The third challenge is that the system must optimize the placement decisions while considering a unique set of constraints while optimizing the selection decisions. The system must adhere to the performance requirements of both the primary and failover models, as well as the resource limits. Additionally, the placement must address correlated failures, where co-located edge servers are more likely to fail in a correlated manner.
- **C4 Failure Detection and Fast Failover.** Finally, the system must promptly detect failures and implement the recovery plan, quickly restoring applications' functional behavior. Furthermore, the system must notify clients of the new location and the selected DNN model.

# **3 FAILLITE DESIGN**

To address the above challenges, we propose FailLite, a failureresilient model serving system for the resource-constrained edge. Inside FailLite, the core is a two-step proactive and progressive failover approach, which intelligently provides failover replicas in a proactive and dynamic manner to minimize the MTTR while ensuring minimal accuracy degradation. In this section, we first give an overview of the two-step approach, then detail each of the two steps, and lastly explain how these two steps sit together within FailLite and how to extend them to tolerate geographically correlated edge failures.

## 3.1 FailLite Overview

The key insight of FailLite is leveraging the flexibility in DNN models by using the concept of *heterogeneous replication*, where we allow the failover backup to be a smaller model, reducing the failover replica resource requirements. This heterogeneous replication allows FailLite to increase the number of failover replicas, enhancing the end-to-end model serving resiliency. However, as highlighted in Section 2.5, designing a fault tolerance model serving system requires addressing issues of model selection and placement



Figure 3: Overview of FailLite's two-step approach.

and balancing the accuracy-MTTR trade-offs. To do so, FailLite features a two-step fault tolerance process, illustrated in Figure 3. As shown, initially, FailLite places warm failover replicas for a set of critical applications while considering available resources and performance requirements. In the second step, FailLite uses a progressive failover process in which cold failover replicas are loaded in an iterative manner from disk to memory based on the servers that have failed. As we will demonstrate in Section 5, our proposed two-step approach overcomes the limitations of singular methods (i.e., only using warm or cold backups). Our hypothesis is that using our two-step approach enables FailLite to maximize accuracy during edge server failures by preloading warm backups for critical applications while reserving some capacity for cold backups. This progressive loading of cold backups on demand allows FailLite to enhance the system's accuracy while minimizing MTTR. Next, we provide an overview of these two steps.

1) Proactive Failover. FailLite uses a configurable parameter to split the free capacity between the proactive and the progressive backups (see Figure 3). FailLite takes into account the usable capacity (free capacity - reserved capacity) and applications that require a warm backup (e.g., critical applications) and selects model variants per application while permitting all applications to have a cold backup (i.e., backups stored on servers' disks). FailLite leverages the flexibility of DNN models to optimize the warm backup model selection and placement with a goal of maximizing the total accuracy while ensuring the placement of the warm backups. FailLite assumes that each application has multiple DNN variants of different resource requirements and accuracy, and models with smaller sizes often have less accuracy (see Figure 2a).

2) Progressive Failover. When a server fails, FailLite needs to determine and load models for the application without a warm backup in a dynamic fashion. The key challenge in this step is that the model selection and placement becomes a real-time decision and highly affects the MTTR. In addition, loading cold backups often has a high loading and warm-up time, a function of the model size, as illustrated in Figure 2b. To overcome this challenge, we propose a progressive model selection and placement approach. FailLite first uses a greedy heuristic that maximizes the accuracy under available capacity to perform backup model selection and placement. Next, FailLite loads these models in a progressive manner. It instantaneously loads the smallest model variant per application to minimize the MTTR and then loads the selected larger model to maximize the accuracy. Although the application may experience higher performance degradation when using the smallest model, it greatly decreases the service downtime. As shown in Figure 2b, two model variants from the same model family can have up to an order of magnitude difference in their loading time.

Table 1: Notations used in FailLite.

Notation	Definition
Ν	set of applications.
S	set of servers.
$K:K\subseteq N$	set of applications that require a warm backup.
$\alpha \in [0, 1]$	fault-tolerance parameter.
$c_k^r$	resource capacity of server $k$ of type $r$ .
ni	set of models for application <i>i</i> .
$a_{ij}$	normalized accuracy of model <i>j</i> of application <i>i</i> .
$d_{ii}^{r}$	resource demand of model <i>j</i> of application <i>i</i> of type <i>r</i> .
$p_i$	primary server of application <i>i</i> .
$q_i$	request rate of application <i>i</i> .
liik	latency of running model <i>j</i> of application <i>i</i> on server <i>k</i> .
$L_i$	latency constraints of application <i>i</i> .
$x_{iik} \in \{0, 1\}$	1 when model <i>i</i> for application <i>i</i> assigned to server <i>k</i> .

where,  $i \in K$ ,  $j \in n_i$ , and  $k \in S$ .

Taking the four applications in Figure 3 as an example, FailLite loads warm backups for two applications while the others have cold backups. By reserving capacity for cold backups, FailLite can utilize server 2 for failing over applications from servers 1 and 3. If server 3 fails, FailLite switches the two applications to a warm backup on server 1 and to a cold backup on server 2, respectively.

#### 3.2 **Proactive Failover**

Our proactive fault tolerance step employs users' preference (e.g., using application criticality level) as well as the required fault tolerance degree of non-critical applications. FailLite assumes that users define a set of applications that must have a warm failover replica while allowing all models to have cold failover replicas. However, naively allocating all available capacity to warm backups may curb the overall resiliency of the systems, as FailLite may utilize oversized failover replicas, leaving no room for other applications. To address this issue, FailLite reserves some of the capacity for cold replicas determined by a configuration parameter. Warm Backup Model Selection and Placement Problem. We formulate our warm backup model selection and placement as an Integer Linear Programming (ILP) optimization problem. Our optimization problem considers N applications running on S edge servers and a set of applications  $K : K \subseteq N$  that must have a warm backup. We assume that the primary instances are placed and each server has a free capacity  $c_k^r$  where r represents different resource types (e.g., GPU memory, CPU RAM, and compute cycles). We assume that users or system administrators define a system-wide failover parameter  $\alpha$ , where a higher  $\alpha$  reserves more resources for cold backups, improving the overall system's resilience, while a small  $\alpha$  maximizes the accuracy of the critical applications. Furthermore, our settings assume that each application *i* has a set of model variants  $n_i$  with varying resource demands  $d_{ij}^r$  and accuracy  $a_{ij}$ . We note that since different applications have different accuracy ranges, we normalize the accuracy, where  $a_{ij} = a_{ij}/max(a_{ij})$ , a common approach in such situations [1]. The latency of running variant j of application i on server  $s_k$  is denoted as  $l_{iik}$ . The notations used in the problem are summarized in Table 1. We define our problem as a maximization problem as follows:

$$\max_{\substack{x_{ijk}^* \\ ijk}} \sum_{i \in K} \sum_{j \in n_i} \sum_{k \in S} a_{ij} \cdot q_i \cdot x_{ijk}$$
(1)

FailLite: Failure-Resilient Model Serving for Resource-Constrained Edge Environments

Algorithm 1: FailLite\_Heuristic()

**Input:** Affected Applications N', Available Servers S'. Output: Model Selection X and Placement Y 1 **Initialization**:  $X \leftarrow \{\}$  and  $Y \leftarrow \{\}$ ; 2  $C^r = \sum_k^{S'} c_k^r //$  Compute Available Capacity. 3  $D^r = \sum_k^{N'} d_{max}^r //$  Compute Max Demand. 4  $\delta = C^r / D^r / /$  Compute demand ratio. 5 for  $i \in N'$  do  $K[i] = \operatorname{match}(n_i, \delta) // \text{ Select variants close to } \delta.$ 7 for  $i \in N'$  do // Iterate over model variants for  $j \in [X[i]..1]$  do 8 k = WorstFit(j, S') / / Model Placement9 if  $k \neq \phi$  then 10 // Feasible placement was found. Y[i] = k;11 X[i] = j12 13 for  $i \in N'$  do // Increase model accuracy.  $X[i] = upgrade_model(X[i], Y[i]);$ 14 15 return X, Y

$$\sum_{i \in K} \sum_{j \in n_i} x_{ijk} \cdot d_{ij}^r \le c_k^r, \quad \forall k, \ \forall r$$
(2)

$$\sum_{i \in K} \sum_{j \in n_i} \sum_{k \in S} x_{ijk} \cdot d_{ij}^r \le (1 - \alpha) \sum_{k \in S} c_k^r, \quad \forall r$$
(3)

$$\sum_{j \in n_j} x_{ijp_i} = 0, \quad \forall i$$
(4)

$$\sum_{j \in n_j} \sum_{k \in S} x_{ijk} = 1, \quad \forall i$$
(5)

$$\sum_{j \in n_j} \sum_{k \in S} l_{ijk} \cdot x_{ijk} \le L_i, \quad \forall i$$
(6)

$$x \in \{0,1\}\tag{7}$$

The objective function (Equation 1) maximizes the effective accuracy [1, 47] by considering the normalized accuracy and applications' request rate  $q_i$  by only considering the set of applications K. Equation 2 ensures the total resource demands of all warm failover backups assigned to a server do not exceed its available capacity. Equation 3 ensures that the capacity reserved for cold backups is respected. Equation 4 is a primary backup independence constraint, where failover replicas do not share the same server with the primary. Equation 5 ensures each application has one backup. Equation 6 limits backup placement to servers and variants that meet the latency SLO. Lastly, Equation 7 ensure that  $x_{iik}$  is binary.

#### 3.3 **Progressive Failover**

In the proactive model selection and placement step, FailLite leverages an ILP to select and place model variants across servers. Despite the similarities between the proactive approach in Section 3.2, which can be used to select and place the failover backups by considering the subset of affected applications K', the location of cold replicas, and the available resources. The key challenge in formulating this problem as an ILP is that solving ILPs typically takes large amounts of time. Although this may be suitable for the proactive step, it highly increases the MTTR, a key objective

of FailLite. Moreover, as mentioned earlier, cold backups often exhibit a high loading and warm-up time, which can be in orders of seconds (see Figure 2b).

To address these challenges, we propose a model selection and placement heuristic, described in Algorithm 1, and a progressive model loading approach. When FailLite detects a failure, Algorithm 1 takes the list of impacted applications that have no warm backup to fail over and available servers to select the model variants (Line 2-6) and to decide their placement (Line 7-14). First, it computes the available resource capacity and maximum resource demands and computes a resource ratio  $\delta$ , where  $\delta \geq 1$  denotes that resources are mostly sufficient to place full backups for all applications, while  $\delta < 1$  means that some applications may face accuracy degradation. Algorithm 1 then selects the model variant that matches the  $\delta$  (e.g., when  $\delta$  is 0.5, the match() function selects a model whose resource demands are 50% of the full-size model). The algorithm then iterates over the applications and their model variants (from the selected model to smaller) and tries to place it on a server in a worst-fit manner until a feasible server and model variant is found. Lastly, Algorithm 1 tries to upgrade the selected model if the selected server can fit a more accurate model. After determining the model variant and its placement, FailLite first loads the smallest variant, ensuring a low MTTR, and after the connection is restored, it upgrades it to the selected model.

## 3.4 Putting It Together

Figure 4 presents the architecture of FailLite, featuring a centralized controller that implements failure detection and our proposed two-step failover policies. FailLite utilizes an agent sitting at edge servers to facilitate the two-step failover process. Here, we present the workflow of FailLite and demonstrate how the proactive and progressive failover approaches are utilized.

First, when a new application arrives (①), FailLite uses the predetermined placement of primary replica (e.g., by the lowest latency or user selection) and uses the proactive failover approach to select and place warm backup models. In either case, FailLite informs the FailLite agent to retrieve the model from the disk and load it to the GPU memory according to the policy (②). At runtime, FailLite uses a heartbeat signal to determine the state of the model serving servers. When FailLite detects a failure, it follows the progressive failover approach and determines the location of the cold backup using our proposed heuristic approach(③). In the case of cold backups, FailLite progressively loads the model, allowing the system to have both a low MTTR and a high accuracy(④). Once the failover process completes, FailLite informs the load balancer or clients of the new location to reroute their inference requests(⑤).

#### Extending FailLite to Geographically Correlated Failures.

Although the aforementioned policy focuses on single failure tolerance, geographically corrected failures within or across edge sites are commonplace in edge environments. To tolerate these failures, FailLite's proactive failover can allow applications to have multiple warm backups (i.e., updating Equation 5 as a multi-replica constraint). Moreover, FailLite can extend the primary independence constraint (Equation 4) to edge site independence, where primary and failover backup do not share the same edge cluster or create an exclusion list where applications in an edge site are



Figure 4: Overview of FailLite Architecture.

only allowed to have backups in a subset of the listed edge sites. Importantly, our progressive failover approach can adapt to these widespread failures by selecting models and placing them on the fly. In Section 5.6, we evaluate FailLite behavior in edge site failures.

## 4 IMPLEMENTATION

In this section, we detail the prototype implementation of FailLite. Our current prototype controller is integrated with the Nvidia Triton Inference Server [30]. However, it is not limited to Triton's design or interfaces. We implemented FailLite using Python and ~5kSLOC. FailLite is available at (url-hidden). In the remainder of this section, we will detail the implementation of FailLite and how the proposed architecture is integrated with the Nvidia Triton Inference Server.

**FailLite Failure Detection.** At the heart of FailLite, is our failure detection approach. The FailLite agent utilizes a periodical *push alive message*, sent every *T*ms, which our experiments use T = 20 ms. If the controller does not receive two consecutive messages, it initiates our proposed failover process.

**FailLite Failover.** We implemented our proposed two-step failover approach, where it places warm backups for critical applications (i.e., proactive failover) and reacts to failure detection by loading the failover replicas for other applications. We implemented our proposed ILP for the proactive placement step using the Python interface of Gurobi v12.0.0. In contrast, our progressive failover heuristic Algorithm 1) is implemented in Python and initiated to load models for applications without a warm backup. Lastly, FailLite asynchronously informs the applications' client of the failover inference server and model using push notifications implemented using websockets v15.0.

**FailLite Data Store**. FailLite utilizes a data store to collect system state (e.g., utilization), performance metrics (e.g., response time), and application profiles (e.g., memory requirements and service time) across model variants. FailLite also maintains the locations of the primary and backups of applications. Our implementation utilizes Redis v7.4.2, an in-memory key-value data store, which allows replication and periodic checkpoints, enhancing the resiliency of our controller.

**Integrating FailLite and Triton.** We build the worker node on top of Nvidia's open-source enterprise model inference system, Nivida Triton Inference Server v24.12-py3 [30], and TensorRT v10.8. We run Triton as a Docker container using Docker v27.5.1, where we allocate all accelerator resources to the container. Triton can serve multiple model inferences within one server, collect system and inference metrics, and support model management at runtime. Triton utilizes a local model repository where models are placed on the disk before loading. To load/unload models, Triton presents a Load/Unload{ID} API, where ID is the application ID, and each application model variant has a unique ID. To differentiate the applications, each inference model is indexed with appID and model variant (e.g., *AppID\_MVar*).

Our implementation augments the Triton server with two key features. First, with failure detection capabilities, where FailLite agent implements a periodic heartbeat where it pushes a keep alive message every 50ms. Second, FailLite dynamically configures the running models to enhance the system's resiliency. To do so, our controller utilizes Triton API to load and unload models as per the failover policy. However, since Triton assumes that all models are locally available, our FailLite agent manages the local model repository by fetching the needed model variants (e.g., from the cloud).<sup>1</sup>

## **5 EVALUATION**

In this section, we evaluate the performance of FailLite and our proposed policies using a real-world testbed. We also augment our results with large-scale simulations highlighting the supremacy and the scalability of our solution. In doing so, we answer the following research questions:

- How does FailLite react to an edge server crash failure? How does our two-step approach address the limitation of traditional failover approaches?
- What is the performance of FailLite with different resource constraints, application configurations, model families, and edge site failures.
- What is the overhead and scalability of FailLite?

## 5.1 Experiment Setup

**Experimental Testbed**. We deploy FailLite on a local testbed consists of 7 Dell PowerEdge R630, each equipped with a 40-core Xeon E5-2660v3 CPU running on Ubuntu 22.04. Each server has 256 GB of memory, a 400 GB Intel 730 SSD, and up to 10 Gbps networking speed. Additionally, 6 of these servers include an NVIDIA A2 GPU with 1280 CUDA cores and 16 GB of GPU memory. We assume these six servers represent three edge sites, each with two servers, and we use a server as a controller node.

**DNN Workloads**. We evaluate FailLite with 16 model families PyTorch [31], totaling 69 DNN models. PyTorch models include their accuracy and model requirements (number of parameters and FLOPs) across these models and different architectures. We

<sup>&</sup>lt;sup>1</sup>We assume that servers typically have ample storage and may maintain many cold replicas. Although addressing storage limitations is beyond the scope of this paper, one solution is to assume that servers rely on a cloud-based model repository where they download models on the fly or use a replication where application models are replicated on multiple servers or sites.

assume that each architecture represents a family, and we calculate the accuracy by normalizing the accuracy to the most accurate model within a family. We categorize these model families into three classes: small, medium, and large, according to their maximum resource demands difference between the largest and smallest models. Our experiments first focus on deploying a number of DNN models from 5 model families, including Mobilenet, Shufflenet, Convnext, Efficientnet, and Regnet. We later expand our analysis to include all model families using simulation.

We compile these models using TensorRT and use Triton's Model Analyzer to profile all the model variants. Triton Model Analyzer measures the GPU memory, compute utilization, and average inference time for batch size 1. Lastly, we note that our approach can utilize static analysis techniques to estimate the memory and processing times [4, 16, 32].

**FailLite** *Deployment*. We deploy FailLite on the edge testbed. Each server acts as a worker node and serves inference requests. Each server hosts a Triton inference server within a container and uses TensorRT as a backend. Furthermore, we disable the auto completion and set the model loading thread of 10 to reduce the model loading overhead introduced by Triton. Each worker node reports the heartbeat to the controller every 20 ms. The FailLite controller runs on one edge server and detects edge crash failure every 100 ms.

In our real experiments, we first deploy the primary of these DNN applications using the worst-fit algorithm, resulting in around 50% utilization in GPU memory and GPU. Our setup deploys a client on the same server as the primary. Moreover, unless otherwise mentioned, we assume that users provide a set K of critical applications that must have a warm failover replica, which we set as 50%, and we set the fault tolerance parameter  $\alpha$  as 0.1. Lastly, to make results comparable across settings, we fix the applications and control the available capacity via controlling a headroom parameter, which we range from 50% to 10%, representing different resource constraints, where unless otherwise mentioned, we assume that headroom is 20%. Moreover, we extend the evaluation of FailLite through large-scale simulations. We utilize all the model families from Py-Torch [31] and expand the setup to 100 servers. We utilize the model profiles and MTTR numbers collected from the real testbed to ensure the fidelity of our results. To overcome the scalability issues of the ILP solver, our solutions utilize the proposed heuristic (Algorithm 1).

**Failure Injection**. We inject crash failures of edge servers by stopping the Triton inference container. When the container is down, the FailLite Agent stops sending the heartbeat to the controller. Note that we do not consider transient failures, where failed servers are shortly recovered. In our experiments, we run model inference in the normal state for 60 seconds before injecting crash failures and collect another 60 seconds of data afterward.

**Evaluation Metrics**. We focus on three metrics to evaluate the performance of FailLite: (i) *Recovery Rate* (%): The percentage of DNN applications that were affected by the failure and were recovered, (ii) *Mean Time To Recovery (MTTR)* (s): The duration between the failure detection and notifying the client of the new application location, which includes model loading time for cold



Figure 5: FailLite behavior across different types of backups, showing the advantages of warm backups and our proposed progressive failover.

backups. Note that we only consider applications that were recovered; non-recovered applications have an infinite MTTR, and (iii) *Accuracy Reduction (%)*: The accuracy reduction of backup compared to primary. Note that we only consider applications that were recovered; non-recovered applications have zero accuracy.

**Baselines**. To the best of our knowledge, no existing model serving systems address the failure resiliency in resource constraint environments. Therefore, we compare FailLite to the traditional approaches in failover replication that do not consider the accuracyresource trade-off and try to use full-size models. Note that since these policies do not consider different model variants, they have no accuracy loss, but not all applications can be recovered. In doing so, we consider three baselines:

- **Full-Size-Warm**: This policy places full-size warm backups to minimize the MTTR of DNN applications. It first considers the set of critical application *K*, then tries to place full-size backup for others.
- Full-Size-Cold: This policy considers all applications and places cold backups. In case of failure, this policy first loads backups for the *K* critical applications, then randomly loads other failover replicas.
- Full-Size-Warm (*K*): Lastly, this policy considers both warm and cold backups, where it only places warm backups for the *K* critical applications, while allowing all application to have cold backups

## 5.2 FailLite in Action

First, we evaluate the behavior of FailLite in failure recovery, where we consider a simple scenario where a single application is running and show the failure recovery process across different approaches. Figure 5 shows the behavior of FailLite when consider a model serving application using the Convnext model family. The x-axis represents the experiment type, where we inject the failure at the 4th second, and the y-axis shows the clients' response time. The figure shows the difference in behavior across types of failover backups, where a warm backup (aside from its size) will have a small failover time, where clients can quickly restore their







Figure 7: Recovery rate and MTTR of FailLite. The accuracy loss is 0.6%.

behavior and reissue the queued requests after failure, restoring the normal behavior after ~300ms. In contrast, when using the cold backups, the model loading time (see Figure 2b) is dependent on the model size. For example, Figure 5 shows that the small model of size 158MB, requires 594ms to load, while the large one of size 806MB, requires 2294ms to load, increasing the MTTR between small and large by a 3.86×. Lastly, the figure shows the benefits of our progressive approach, where it reduces the MTTR while retaining the accuracy of the large. We note that, in this progressive approach, the client is oblivious to the switch, where FailLite does the change using the same network interface, yielding a small spike in the response time.

Next, we extend our experimental scenario, where we consider 5 model families and load models to cover 50% of the compute and memory resources, resulting in 46 different applications. Figure 6 shows the behavior of FailLite compared to the Full-Size-Warm(K), which considers warm backups for K = 50% critical application. We also assume that the headroom (usable space to load backups) is 20%. Figure 6 shows the behavior of FailLite and Full-Size-Warm(*K*) for three selected scenarios, highlighting the advantages of FailLite, but we summarize the results across all applications in Figure 7. Figure 6a shows a scenario where FailLite utilizes a warm backup of small size to circumvent the resource limitation, maintaining a small MTTR (85 ms), in a scenario where a full-sized model cannot be loaded. Figure 6b shows another decision made by FailLite, where it decides only to load a smaller-sized model, allowing the application to be recovered despite the resource limitation. In this case, FailLite will face a 1316 ms MTTR, as loading the application from disk takes more time. Lastly, Figure 6c shows the behavior of FailLite in a scenario where both approaches can secure a cold failover backup. As shown, although both approaches can recover the full-model, FailLite can reduce the MTTR by 72%.

Figure 7 shows the aggregate behavior of FailLite and three baseline policies in terms of recovery rate (Figure 7a) and (Figure 7b). We note that we do not report the accuracy reduction, as

baselines use full-size models, leading to no accuracy reduction, while FailLite exhibits a 0.6% accuracy reduction. The figure reports the average recovery rate and MTTR across six runs where we fix the application placement and, each time, we inject a failure on one of our six servers.

As shown in Figure 7a, in all scenarios, FailLite was able to maintain a high recovery rate, where it was able to load a failover backup for all applications. In contrast, other baselines could not host a failover for all applications. Figure 7b completes the picture by showing the MTTR across baselines. The figures highlight the tradeoffs of the choices made by the Full-size baselines. For instance, the Full-Size (Warm) policy was able to provide only a failover replica (in case of failure) for 85.9% and 76% of the applications on average and in the worst case. In contrast, although the Full-Size (Cold) was able to overcome this limitation, Figure 7b shows that this approach has a high average MTTR, 15.8× larger than that of the Full-size (Warm). Moreover, the figure shows that despite how Full-Size-Warm(*K*) may address the limitation of either approach, it still falls short in comparison to FailLite. For instance, Figure 7a shows how FailLite has a recovery rate higher by 7.7% and an average MTTR better by 2×.

**Key Takeaways**. FailLite proposed a two-step approach, heterogeneous replication, and progressive loading address the limitation of traditional failover techniques. Our results show that compared to Full-Size-Warm(K), FailLite is able to maintain a 100% recovery rate and decrease the MTTR by 2×, only for an 0.6% accuracy loss.

## 5.3 Impact of Resource Constraints

Despite the ability of FailLite to enhance failure resiliency in resource-constrained environments, the quantity of available resources influences the decisions made by FailLite. For example, when resources are already overloaded, there is no room to enhance the system's resiliency. In contrast, in low utilization scenarios, traditional failover techniques shall behave quite well. In this section, we evaluate the performance of FailLite and depict its ability to optimize its decisions in different resource contention scenarios. We simulate 10 edge sites with 100 servers in total and deploy the same mixture of DNN models as the real experiments across these servers. To make results comparable across settings, we fix the applications to 640 and control the available capacity via controlling a headroom parameter, which we range from 50% to 10%, representing different resource constraints.

Figure 8 shows the behavior of FailLite and traditional failover baselines across different headroom settings. Figure 8a shows the

FailLite: Failure-Resilient Model Serving for Resource-Constrained Edge Environments



Figure 8: Impact of resource constraints on the decisions and performance of FailLite and our baselines. We introduce resource constraints by changing the headroom available for failover backups.



Figure 9: Impact of K.

recovery rate across different settings, highlighting the ability of FailLite to recover all applications, even in highly constrained settings. In contrast, the traditional failover approach has struggled, especially for small headrooms. For instance, when headroom is 10%, the full-warm, full-cold, and full-warm-k are only able to recover 50.5%, 79.8%, and 66% of the applications. Nonetheless, increased headroom increases the recovery rate, where at 50%, all approaches, except for Full-Warm, are able to recover 100% of the applications.

In addition, Figure 8b shows the runtime behavior of different policies. As expected, the Full-Warm backup has a low and stable MTTR, as the headroom size has no effect when only considering warm backups, yielding a 50 ms average failover time. In contrast, policies that depend on cold backups (i.e., Full-Cold and Full-Warm-K) have a much higher increase in MTTR, as higher headroom allows for failover replicas for larger models, increasing the MTTR. For instance, the average MMTR increases from 784 ms to 1742 ms between 10% and 30% headroom. The figure also highlights how FailLite addresses the limitation in these policies, where FailLite achieves a low MTTR by allowing all applications within K to have a warm backup while progressively loading cold failover models, resulting in ~134 MTTR. Finally, Figure 8c shows the accuracy; as noted earlier, utilizing full backups have no accuracy loss, and we only considered restored applications. As shown, the high recovery rate and smaller MTTR, come at a cost of accuracy reductions, where at a headroom of 10% FailLite loses 4.52% of the accuracy of the models.

**Key Takeaways**. Although resource limitation highly affects the failure resiliency of traditional failover approaches, FailLite was able to circumvent the key limitations of these approaches and achieve a 100% recovery rate for a 4.52% reduction in accuracy.



MTTR (b) and Accuracy Reduction (c).

## 5.4 Impact of Applications' Criticality

Another key factor in the decision of FailLite is the application criticality. Figure 9 shows the effect of FailLite across different K configurations, where we change K from 0% to 100% using the large-scale simulation setup described earlier. As shown, the curve highlights the accuracy-MTTR trade-off in failure-resilient modelserving systems. For instance, when all applications are critical (K = 100), FailLite needs to place warm backups for all applications, reducing the overall accuracy. On the other hand, when none of the applications require a warm backup (K = 0), FailLite can maximize the accuracy by loading the largest model for the affected applications at the cost of higher MTTR. Lastly, we note that although how users value different parameters is beyond the scope of this paper, we highlight that a balance point exists when K = 60%.

**Key Takeaways**. The decisions of FailLite introduces an accuracy-MTTR trade-offs, where the decisions of FailLite at K introduces a low MTTR, while higher K, comes with a performance desegregation.

#### 5.5 Impact of Model Family

We analyze the impact of different model families on the performance of FailLite. We categorize model families into three classes according to the maximal difference of resource demand between model variants: Small, Medium, and Large. For example, we consider Mobilenet model family as Small as the maximal resource demand difference is 12MB, and consider the Convnext model family as Large as its 648MB demand difference. Figure 10 shows the performance of FailLite and other baselines when all applications are composed exclusively from one model family class. Lastly, we note that since model families have different resource requirements,

Wu et al.

Conference'17, July 2017, Washington, DC, USA



the total number of applications changes across scenarios ranging between 3264 and 402 across the Small and Large classes.

Figure 10a shows that FailLite is able to achieve a higher recovery rate when the demand difference is bigger, while full-size baselines struggle. For instance, Full-Cold can only recover 65% of the applications in the large class while FailLite can recover all. When the demand difference is small, FailLite can still outperform the Full-Warm-K policy by 15% in recovery rate. Figure 10b also highlights the effect of model family class on the MTTR, where increases in model size introduce higher MTTR across all baselines. However, it is worth mentioning that even in the Large class, FailLite MTTR value was 214 ms on average. Figure 10c shows the accuracy reduction, although accuracy reduction is not comparable across scenarios, FailLite exhibits a maximum of 7.1% reduction in accuracy. Lastly, as noted earlier, traditional policies have no accuracy loss.

**Key Takeaways**. With a larger difference in resource demand between model variants, FailLite is able to further optimize the resiliency of DNN models while achieving a 214ms MTTR.

## 5.6 Impact of Edge Site Failure

Site-wide failures are common in the edge, as network and power systems do not have high levels of redundancy. To evaluate the effect of edge-wide failures, we utilize our large-scale simulation testbed of 100 servers and group these servers into ten sites of size 10. We then explore the effect of site failures by failing 1 - 7 of the 10 edge sites. Similar to previous experiments, we repeat these experiments multiple times to ensure the stability of our results. Figure 11 shows the recovery rate and MTTR of FailLite and other approaches. We note that as mentioned in Section 3.4, we add a site independence constraint, where the warm backup is placed in a different site. As shown in Figure 11, FailLite is able to maintain 100% failure recovery until 50% of the sites fail enhancing the recovery rate by 7.9% and 39.3% compared to Full-Cold, for the single edge site and seven edge sites failures scenarios, respectively. Figure 11b also highlights how the MTTR changes across scenarios, where increases in the number of failed sites decreases the ability to load large models, decreasing the MTTR for all approaches.

**Key Takeaways**. The capabilities of FailLite can be extended to edge-site resilience. For instance, when 50% of the edge sites fail, FailLite increases the recovery rate by at least 39.3% compared to the full-size policies.



Figure 12: Scalability of FailLite heuristic model selection and placement. We fix the number of servers as 500, applications as 1000, and model variants as 4, and show the effect of changing each.

## 5.7 FailLite Overheads

In this section, we highlight the runtime overheads experienced by FailLite. Our failure detection process typically took 65ms per our configuration, while informing the clients with the location of the failover replica was around 10ms. Figure 12 highlights the scalability of our proposed heuristic, where even in very large scale settings (e.g., 3000 applications or 1000 servers), it took less than 4 seconds. Lastly, we note that the largest overhead source was the model loading time, which was not controlled by FailLite as it was a function of the model size as we shown in Figure 2b.

## 6 RELATED WORK

**Model Serving.** The importance of model serving systems has encouraged many researchers to address its latency [5, 7, 7, 12, 23, 34, 39, 48, 49], cost [2, 6, 25, 46], energy efficiency [11, 24, 40, 45], and addressing challenges of workload dynamics [1, 2, 24, 40, 47]. A common idea of these papers is exploring the accuracy-resource trade-offs where choosing the right-sized DNN model [1, 2, 9, 11, 19, 47] or utilizing flexible DNN architecture (e.g., multi-exit DNNs [24] and Slimmable DNNs [44]). Although similar to many of these papers, FailLite exploits the accuracy-resource trade-offs, and we utilize it to enhance the resiliency of model serving systems and highlight a new accuracy-resiliency trade-off.

**Resilient Model Serving.** Although researchers have addressed failure resiliency issues in previous work, they often have different assumptions or objectives than FailLite. For instance, the authors of [28, 42, 43] highlighted that as models become larger, the model inference systems will often utilize multi-nodes to process inference requests, where failures of any of the utilized nodes affect the model resiliency. To address these issues, the authors often use early-exit and skip-connection models to ensure a result is available. Our approach, in contrast, focuses on a different scenario, where we assume that models are deployed on a single server and failures affect the entire model

**Resiliency in Resource Constraint Environments** Graceful degradation is commonly used in scenarios with resource contention. For instance, Defcon [29] improves reliability by incorporating services' criticality and allocating resources per this criticality. In contrast to this paper, where placement decisions are binary, FailLite addresses the Failure Resiliency in model serving systems, where the decisions are far more complex and the system can utilize DNNs' flexibility by choosing different model sizes and selecting which applications have a failover replica.

FailLite: Failure-Resilient Model Serving for Resource-Constrained Edge Environments

#### 7 CONCLUSION

In this paper, we proposed FailLite, a failure-resilient model serving system that employs (i) a *heterogeneous replication* where failover models are smaller variants of the original model, and (ii) an intelligent approach that uses warm replicas to ensure quick failover for important applications while using cold replicas and (iii) *progressive failover* to provide low mean time to recovery for the remaining applications. Our evaluation has demonstrated that our two-step approach enables FailLite to maximize worst-case accuracy (i.e., accuracy during system failures) for critical and non-critical applications while minimizing MTTR. Our results using the 27 models show that FailLite can recover all failed applications with 175.5ms MTTR and only introduce 0.6% reduction in accuracy. Our future work will address the failure resiliency in heterogeneous environments and when deploying model serving pipelines.

#### REFERENCES

- [1] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. 2024. Proteus: A High-Throughput Inference-Serving System with Accuracy Scaling. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (La Jolla, CA, USA) (ASPLOS '24). Association for Computing Machinery, New York, NY, USA, 318–334. https://doi.org/10.1145/3617232.3624849
- [2] Sohaib Ahmad, Hui Guan, and Ramesh K. Sitaraman. 2024. Loki: A System for Serving ML Inference Pipelines with Hardware and Accuracy Scaling. In Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (Pisa, Italy) (HPDC '24). Association for Computing Machinery, New York, NY, USA, 267–280. https://doi.org/10.1145/3625549.3658688
- [3] Amazon Web Services. 2025. Amazon SageMaker. https://aws.amazon.com/ sagemaker/ Accessed: 2025-04-09.
- [4] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. 2017. Neuralpower: Predict and Deploy Energy-efficient Convolutional Neural Networks. In Asian Conference on Machine Learning.
- [5] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). USENIX Association, Boston, MA, 613–627. https://www. usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw
- [6] Qiang Fan and Nirwan Ansari. 2019. On cost aware cloudlet placement for mobile edge computing. *IEEE/CAA Journal of Automatica Sinica* 6, 4 (2019), 926–937. https://doi.org/10.1109/JAS.2019.1911564
- [7] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati
- [8] Zhishan Guo, Kecheng Yang, Sudharsan Vaidhun, Samsil Arefin, Sajal K. Das, and Haoyi Xiong. 2018. Uniprocessor Mixed-Criticality Scheduling with Graceful Degradation by Completion Rate. In 2018 IEEE Real-Time Systems Symposium (RTSS) 373–383. https://doi.org/10.1109/RTSS.2018.00052
- [9] Matthew Halpern, Behzad Boroujerdian, Todd Mummert, Evelyn Duesterwald, and Vijay Janapa Reddi. 2019. One Size Does Not Fit All: Quantifying and Exposing the Accuracy-Latency Trade-Off in Machine Learning Cloud Service APIs via Tolerance Tiers. In 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE Computer Society, Los Alamitos, CA, USA, 34–47. https://doi.org/10.1109/ISPASS.2019.00012
- [10] Walid A. Hanafy et al. 2023. Failure-Resilient ML Inference at the Edge through Graceful Service Degradation. In *IEEE Military Communications Conference (MIL-COM)*. 144–149. https://doi.org/10.1109/MILCOM58377.2023.10356302
- [11] Walid A. Hanafy, Tergel Molom-Ochir, and Rohan Shenoy. 2021. Design Considerations for Energy-Efficient Inference on Edge Devices. In Proceedings of the Twelfth ACM International Conference on Future Energy Systems (Virtual Event, Italy) (e-Energy '21). Association for Computing Machinery, New York, NY, USA, 302–308. https://doi.org/10.1145/3447555.3465326
- [12] Walid A. Hanafy, Limin Wang, Hyunseok Chang, Sarit Mukherjee, T. V. Lakshman, and Prashant Shenoy. 2023. Understanding the Benefits of Hardware-Accelerated Communication in Model-Serving Applications. In Proceedings of IEEE/ACM 31st International Symposium on Quality of Service (IWQoS'23).
- [13] Soheil Hashemi, Nicholas Anthony, Hokchhay Tann, R. Iris Bahar, and Sherief Reda. 2017. Understanding the Impact of Precision Quantization on the Accuracy and Energy of Neural Networks. In Proceedings of the Conference on Design,

*Automation & Test in Europe* (Lausanne, Switzerland) (*DATE '17*). European Design and Automation Association, Leuven, BEL, 1478–1483.

- [14] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. 2023. Understanding and Mitigating Hardware Failures in Deep Learning Training Systems. In Proceedings of the 50th Annual International Symposium on Computer Architecture (Orlando, FL, USA) (ISCA '23). Association for Computing Machinery, New York, NY, USA, Article 70, 16 pages. https: //doi.org/10.1145/3579371.3589105
- [15] Jin Huang, Colin Samplawski, Deepak Ganesan, Benjamin Marlin, and Heesung Kwon. 2020. CLIO: Enabling Automatic Compilation of Deep Learning Pipelines across IoT and Cloud. In Proceedings of the 26th Annual International Conference on Mobile Computing and Networking (London, United Kingdom) (MobiCom '20). Association for Computing Machinery, New York, NY, USA, Article 58, 12 pages. https://doi.org/10.1145/3372224.3419215
- [16] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. 2018. Predicting the Computational Cost of Deep Learning Models. In 2018 IEEE International Conference on Big Data (Big Data).
- [17] Israel Koren and C. Mani Krishna. 2021. *Fault-Tolerant Systems* (second edition ed.). Morgan Kaufmann.
- [18] Kubeflow. 2025. Kubeflow: The Machine Learning Toolkit for Kubernetes. https: //www.kubeflow.org/ Accessed: 2025-04-14.
- [19] Baolin Li, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2023. Clover: Toward Sustainable AI with Carbon-Aware Machine Learning Inference Service. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Denver, CO, USA) (SC '23). Association for Computing Machinery, New York, NY, USA, Article 20, 15 pages. https://doi. org/10.1145/3581784.3607034
- [20] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. 2020. Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing. *IEEE Transactions* on Wireless Communications 19, 1 (2020), 447–457. https://doi.org/10.1109/TWC. 2019.2946140
- [21] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. 2017. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Denver, Colorado) (SC '17). Association for Computing Machinery, New York, NY, USA, Article 8, 12 pages. https://doi.org/ 10.1145/3126908.3126964
- [22] Jinyang Li, Yizhuo Chen, Tomoyoshi Kimura, Tianshi Wang, Ruijie Wang, Denizhan Kara, Yigong Hu, Li Wu, Walid A. Hanafy, Abel Souza, Prashant Shenoy, Maggie Wigness, Joydeep Bhattacharyya, Jae Kim, Guijun Wang, Greg Kimberly, Josh Eckhardt, Denis Osipychev, and Tarek Abdelzaher. 2024. Acies-OS: A Content-Centric Platform for Edge AI Twinning and Orchestration. In 2024 33rd International Conference on Computer Communications and Networks (ICCCN). 1–9. https://doi.org/10.1109/ICCCN61486.2024.10637580
- [23] Qianlin Liang, Walid A. Hanafy, Ahmed Ali-Eldin, and Prashant Shenoy. 2023. Model-Driven Cluster Resource Management for AI Workloads in Edge Clouds. ACM Trans. Auton. Adapt. Syst. 18, 1, Article 2 (mar 2023), 26 pages. https: //doi.org/10.1145/3582080
- [24] Qianlin Liang, Walid A. Hanafy, Noman Bashir, Ahmed Ali-Eldin, David Irwin, and Prashant Shenoy. 2023. DèLen: Enabling Flexible and Adaptive Model-Serving for Multi-Tenant Edge AI. In Proceedings of the 8th ACM/IEEE Conference on Internet of Things Design and Implementation (San Antonio, TX, USA) (IoTDI '23). Association for Computing Machinery, New York, NY, USA, 209–221. https: //doi.org/10.1145/3576842.3582375
- [25] Qianlin Liang, Prashant Shenoy, and David Irwin. 2020. AI on the Edge: Characterizing AI-based IoT Applications Using Specialized Edge Architectures. In 2020 IEEE International Symposium on Workload Characterization (IISWC). 145–156. https://doi.org/10.1109/IISWC50251.2020.00023
- [26] Yun Lin, Haojun Zhao, Ya Tu, Shiwen Mao, and Zheng Dou. 2020. Threats of Adversarial Attacks in DNN-Based Modulation Recognition. In *IEEE INFOCOM* 2020 - IEEE Conference on Computer Communications. 2469–2478. https://doi. org/10.1109/INFOCOM41043.2020.9155389
- [27] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. 2020. AutoCompress: An Automatic DNN Structured Pruning Framework for Ultra-High Compression Rates. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 04 (Apr. 2020), 4876–4883. https://doi.org/10.1609/aaai.v34i04.5924
- [28] Ayesha Abdul Majeed, Peter Kilpatrick, Ivor Spence, and Blesson Varghese. 2022. CONTINUER: maintaining distributed DNN services during edge failures. In 2022 IEEE International Conference on Edge Computing and Communications (EDGE). IEEE, 143–152.
- [29] Justin J. Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, Shuyang Shi, Tina Luo, David Ke Hong, Sankaralingam Panneerselvam, Hans Ragas, Svetlin Manavski, Weidong Wang, and Francois Richard. 2023. Defcon: Preventing Overload with Graceful Feature Degradation. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). USENIX Association, Boston, MA, 607–622. https://www.usenix.org/conference/osdi23/presentation/

Conference'17, July 2017, Washington, DC, USA

Wu et al.

meza

- [30] NVIDIA. 2024. Triton Inference Server. https://developer.nvidia.com/tritoninference-server Accessed: 2025-04-13.
- [31] PyTorch. 2024. TorchVision Models and pre-trained weights. https://pytorch. org/vision/stable/models.html Accessed: 2025-04-14.
- [32] Qi, Evan R. Sparks, and Ameet S. Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *The International Conference on Learning Representations (ICLR'17).*
- [33] Mohammad Salehe, Zhiming Hu, Seyed Hossein Mortazavi, Iqbal Mohomed, and Tim Capes. 2019. VideoPipe: Building Video Stream Processing Pipelines at the Edge. In Proceedings of the 20th International Middleware Conference Industrial Track (Davis, CA, USA) (Middleware '19). Association for Computing Machinery, New York, NY, USA, 43–49. https://doi.org/10.1145/3366626.3368131
- [34] Colin Samplawski, Jin Huang, Deepak Ganesan, and Benjamin M. Marlin. 2020. Towards Objection Detection Under IoT Resource Constraints: Combining Partitioning, Slicing and Compression. In Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things (Virtual Event, Japan) (AIChallengeloT '20). Association for Computing Machinery, New York, NY, USA, 14–20.
- [35] Mahadev Satyanarayanan. 2017. The Emergence of Edge Computing. Computer 50, 1 (2017), 30–39. https://doi.org/10.1109/MC.2017.9
- [36] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing* 8, 4 (2009), 14–23. https://doi.org/10.1109/MPRV.2009.82
- [37] Mahadev Satyanarayanan, Nathan Beckmann, Grace A. Lewis, and Brandon Lucia. 2021. The Role of Edge Offload for Hardware-Accelerated Mobile Devices. In Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications (Virtual, United Kingdom) (HotMobile '21). Association for Computing Machinery, New York, NY, USA, 22–29. https://doi.org/10.1145/3446382.3448360
- [38] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (Huntsville, Ontario, Canada) (SOSP '19). 322–337. https://doi.org/10.1145/3341301.3359658
- [39] Jonathan Soifer, Jason Li, Mingqin Li, Jeffrey Zhu, Yingnan Li, Yuxiong He, Elton Zheng, Adi Oltean, Maya Mosyak, Chris Barnes, Thomas Liu, and Junhua Wang. 2019. Deep Learning Inference Service at Microsoft. In 2019 USENIX Conference on Operational Machine Learning (OpML 19). Santa Clara, CA, 15–17.
  [40] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael
- [40] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. 2020. ALERT: Accurate Learning for Energy and Timeliness. In 2020 USENIX Annual Technical Conference (USENIX ATC 20). USENIX Association, 353–369.
- [41] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, et al. 2024. {SuperBench}: Improving Cloud {AI} Infrastructure Reliability with Proactive Validation. In 2024 USENIX Annual Technical Conference (USENIX ATC 24). 835–850.
- [42] Ashkan Yousefpour, Siddartha Devic, Brian Q Nguyen, Aboudy Kreidieh, Alan Liao, Alexandre M Bayen, and Jason P Jue. 2019. Guardians of the deep fog: Failure-resilient DNN inference from edge to cloud. In Proceedings of the first international workshop on challenges in artificial intelligence and machine learning for internet of things. 25–31.
- [43] Ashkan Yousefpour, Brian Q Nguyen, Siddartha Devic, Guanhua Wang, Aboudy Kreidieh, Hans Lobel, Alexandre M Bayen, and Jason P Jue. 2020. Resilinet: Failureresilient inference in distributed neural networks. arXiv preprint arXiv:2002.07386 (2020).
- [44] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. 2018. Slimmable Neural Networks. arXiv:1812.08928 [cs.CV]
- [45] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. GOBO: Quantizing Attention-Based NLP Models for Low Latency and Energy Efficient Inference. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 811–824. https://doi.org/10.1109/MICRO50266.2020. 00071
- [46] Chengliang Zhang, Minchen Yu, wei wang, and Feng Yan. 2020. Enabling Cost-Effective, SLO-Aware Machine Learning Inference Serving on Public Cloud. *IEEE Transactions on Cloud Computing* (2020), 1–1. https://doi.org/10.1109/TCC.2020. 3006751
- [47] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-asa-Service Systems. In 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20). USENIX Association. https://www.usenix.org/conference/ hotCloud20/presentation/zhang
- [48] Shuai Zhang, Sheng Zhang, Zhuzhong Qian, Jie Wu, Yibo Jin, and Sanglu Lu. 2021. DeepSlicing: Collaborative and Adaptive CNN Inference With Low Latency. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2175–2187. https://doi.org/10.1109/TPDS.2021.3058532
- [49] Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. 2019. Hetero-Edge: Orchestration of Real-time Vision Applications on Heterogeneous Edge Clouds. In IEEE INFOCOM 2019 - IEEE

Conference on Computer Communications. 1270–1278. https://doi.org/10.1109/ INFOCOM.2019.8737478

[50] Jie Zou, Xiaotian Dai, and John Mcdermid. 2023. Graceful Degradation with Condition- and Inference-Awareness for Mixed-Criticality Scheduling in Autonomous Systems. In Proceedings of Cyber-Physical Systems and Internet of Things Week 2023 (San Antonio, TX, USA) (CPS-IoT Week '23). Association for Computing Machinery, New York, NY, USA, 215–220. https://doi.org/10.1145/ 3576914.3587511