CAROL: Confidence-Aware Resilience Model for Edge Federations

Shreshth Tuli*, Giuliano Casale* and Nicholas R. Jennings*†
*Department of Computing, Imperial College London, UK
†Loughborough University
Emails: {s.tuli20, g.casale}@imperial.ac.uk, n.r.jennings@lboro.ac.uk

***Abstract*—In recent years, the deployment of large-scale Internet of Things (IoT) applications has given rise to edge federations that seamlessly interconnect and leverage resources from multiple edge service providers. The requirement of supporting both latency-sensitive and compute-intensive IoT tasks necessitates service resilience, especially for the broker nodes in typical broker-worker deployment designs. Existing fault-tolerance or resilience schemes often lack robustness and generalization capability in non-stationary workload settings. This is typically due to the expensive periodic fine-tuning of models required to adapt them in dynamic scenarios. To address this, we present a confidence aware resilience model, CAROL, that utilizes a memory-efficient generative neural network to predict the Quality of Service (QoS) for a future state and a confidence score for each prediction. Thus, whenever a broker fails, we quickly recover the system by executing a local-search over the broker-worker topology space and optimize future QoS. The confidence score enables us to keep track of the prediction performance and run parsimonious neural network fine-tuning to avoid excessive overheads, further improving the QoS of the system. Experiments on a Raspberry-Pi based edge testbed with IoT benchmark applications show that CAROL outperforms state-of-the-art resilience schemes by reducing the energy consumption, deadline violation rates and resilience overheads by up to 16, 17 and 36 percent, respectively.**

***Index Terms*—Edge Federations; Service Resilience; Confidence-Aware; Generative Models; Deep Learning**

I. Introduction

The fifth industrial revolution, named Industry 5.0, marks a significant shift in the technological backbone of industrial applications such as the Internet of Things (IoT) and Artificial Intelligence (AI) [1]. It allows end-to-end integration of sensors and actuators close to the user with geographically distributed servers via a large number of intermediary smart edge nodes [2]. Such frameworks enable the deployment of latency-critical and compute-intensive AI-based IoT applications on edge devices for low response time and large-scale service delivery. It achieves this by the collective adoption of edge federations that bring together the software, infrastructure and platform services of multiple edge computing environments. Unifying the computational devices of multiple service providers enables such paradigms to accommodate sudden spikes in user demands [3]. However, due to the distributed nature of such environments, centralized management of their resources is susceptible to service downtimes in high workload settings. Thus, most edge federations employ a broker-worker topology with multiple broker nodes managing the system [4]. Here, the brokers receive tasks from the users and delegate processing to one of the worker nodes within their control, referred to as local edge infrastructures (LEIs). This leads the processing bottlenecks in such systems, *i.e.*, the broker nodes, play a vital part in resilient service delivery.

Challenges. The key challenge addressed in this paper is dealing with the diverse effects of byzantine node failures, such as increased task response time and violations of Service Level Objectives (SLOs), requiring different remediation steps to maintain system performance and reduce service downtime. This is motivated by the excessive load that AI-based IoT applications put on the limited computational capacity and working memory (up to 8GB) of edge devices [5]. Higher computational load translates to increased task processing times and SLO violation rates. Excessive memory load is typically dealt with using storage mapped virtual memory, which in most settings is a network-attached disk [6]. This also leads to higher response times due to time-consuming data transfers over congested backhaul edge networks [7]. All these issues lead to persistent resource contention and faulty behavior that adversely affects system QoS. Furthermore, without resource intensive cloud backends to rely on in edge federations, it becomes challenging to deploy modern deep learning based management solutions in resource constrained edge devices. In typical broker-worker federations, if a worker node fails, a broker could act as a worker and complete the task or allocate the same task to another worker [4], [8]. However, if a broker fails, all active tasks within the LEI and all incoming tasks that are sent to the broker are impacted. This makes broker resilience crucial in large-scale deployments. However, the problem of broker resilience is hard as maintaining hot-redundancy by replicating running task instances makes edge nodes more susceptible to failures [9]. The difficulty primarily arises from the critical need for result delivery with low latency and high accuracy [10]. A vital parameter that controls this is the number of brokers in the system. For a fixed number of devices in the system, a high broker count translates into having fewer workers, reducing the average throughput of the system. Low broker count can cause bottlenecks and contentions, increasing fault frequency. So we need to consider both the increase or decrease of the number of brokers in the system. Furthermore, the statistical moments and correlations of the

workload characteristics are non-stationary and vary over time, requiring continuous steps to adapt management models. This imposes additional overheads and impacts QoS [11].

Existing solutions. Many recent works [12], [17]–[19] aim to provide effective fault tolerance or system resilience by leveraging heuristics, meta-heuristics or AI models. The heuristic and meta-heuristic methods often perform poorly in dynamic settings where workload characteristics and SLO demands are non-stationary [18], [19]. AI-based methods typically utilize reinforcement learning (RL) or neural networks as surrogate models to predict future system states and estimate their QoS. The predicted QoS values indicate the chances of future broker or worker breakdowns and proactively manage the broker-worker topology to avoid service downtimes. These methods leverage fault-aware scheduling [16], preemptive migration based load-balancing [17] or auto-scaling techniques [19]. RL or surrogate models trained on large datasets enable such techniques to be effective even in unseen settings by exploration and neural network fine-tuning. This is a process where the model utilizes the data generated during execution to update the neural network parameters in an online fashion. However, these solutions have several limitations. Most AI-based approaches are designed for cloud setups with GPUs for faster training of the underlying neural networks and result generation [22]. To deploy these models in resource-constrained edge settings, various model compression and parameter neural network splitting are required, adversely impacting their performance [6]. This also impacts the periodic model fine-tuning, increasing the training times in distributed neural network settings [23]. Together, these factors mean that neural network fine-tuning process consumes large portions of the computational and memory resources of edge devices. This impacts the execution of the management tasks running in the broker nodes, further leading to contentions in broker nodes. To resolve this, there is a need for a lightweight solution that parsimoniously fine-tunes AI methods.

Key insights and our contributions. For a lightweight broker resilience model, we develop a method that uses a neural network to predict system QoS and also indicates when to fine-tune the network to adapt to non-stationary settings. The novel insight is using a generative network as a surrogate model to optimize QoS. Unlike prior work that use traditional feed-forward or recurrent neural networks to predict QoS [16], [18], using specific generative models not only allows us to estimate QoS of future states, but also an indicator of the prediction confidence. Examples of such a generative network with low memory footprint are recently-proposed models by the AI theory community such as Generative Optimization Networks (GONs) [24] or SpareVAEs [25], which provide an alternative to Generative Adversarial Networks (GANs) [26] or Variational Auto-encoders (VAEs) [27] with much lower memory footprint and therefore suitable for edge execution. Models with a reduced footprint of this kind pave the way to the use of GAN-type methods in edge devices with resource footprint constraints. In our recent work [24][1], we have demonstrated an application of GONs to fault-detection finding up to 32% improvement in fault detection scores when a local edge infrastructure runs a GON in its broker to detect faults within its edge devices. Contrary to that work, we focus here instead on the application of GONs to the problem of broker resilience, which is not touched upon in [24].

In the proposed work, an offline trained GON model with labelled data generated from a system with similar behavior as that of the training dataset would typically give high confidence scores, whereas as the system behavior changes, the confidence score declines (more details in Section III). Dynamic thresholding techniques allow us to decide confidence thresholds below which we fine-tune the network parameters. Thus, we only run fine-tuning measures when required, significantly reducing overheads and improving system QoS. In this work we present CAROL: Confidence Aware Resilience Model for edge federations. CAROL is the first system that uses a GON in edge brokers to reactively run topology optimization to optimize system QoS. The GON model is trained using log traces on DeFog benchmarks [28]. Extensive experiments on a Raspberry-Pi based federated edge cluster show that CAROL performs best in terms of QoS metrics. To test the generalization of the model, we test on different benchmarks as workloads, namely AIoT [29]. Specifically, CAROL reduces energy consumption, SLO violation rates and resilience overheads by up to 16, 17 and 36 percent, respectively, compared to state-of-the-art baselines.

II. Related Work

We list in Table I the prior work, dividing such methods into two classes: heuristic and meta-heuristic methods (rows 1-5 of Table I) and AI-based methods (rows 6-10). Many of these methods use simple strategies to deal with broker failures or are unable to efficiently adapt in non-stationary environments.

Heuristic and Meta-heuristic methods. Most fault prevention techniques for cloud and edge computing employ some form of heuristics or meta-heuristic approaches. Methods like DYVERSE [12] use dynamic vertical scaling in multi-tenant edge systems to manage resources assigned to IoT applications to improve scalability. It uses an ensemble of three heuristics (system-aware, community-aware and workload-aware) to dynamically allocate priority scores to the active applications in the system. DYVERSE uses AI-based face detection and online game workload to validate in a controlled edge computing environment. However, for broker failures, it allocates the worker with the least CPU utilization as the next broker of the same LEI. Federated Distributed MapReduce (FDMR) [15] is another MapReduce based framework that uses integer linear programming for fault-tolerant distributed task scheduling. Another approach, namely Distributed IoT Service Provisioning (DISP) [13] technique, balances load across edge and fog nodes by comparing CPU utilization and response time metrics for each node. However, due to the

[1] https://github.com/imperial-qore/GON.

TABLE I
COMPARISON OF RELATED WORKS WITH DIFFERENT PARAMETERS (✓MEANS THAT THE CORRESPONDING FEATURE IS PRESENT).

Work	IoT	Approach	Consider Broker Resilience	QoS Prediction	Performance parameters				
					Energy	Response Time	SLO Violations	Overheads	Memory
DYVERSE [12]	✓	Heuristic	✓		✓		✓	✓	
DISP [13]		Heuristic				✓	✓		
LBM [14]	✓	Heuristic	✓	✓	✓				
FDMR [15]		Meta-Heuristic				✓	✓		
ECLB [16]	✓	Meta-Heuristic	✓	✓	✓		✓		
LBOS [17]	✓	RL		✓	✓	✓		✓	✓
ELBS [18]	✓	Surrogate Model	✓	✓	✓	✓	✓		
FRAS [19]		Surrogate Model		✓	✓		✓		✓
TopoMAD [20]		Reconstruction		✓	✓	✓	✓		
StepGAN [21]	✓	Reconstruction		✓	✓	✓	✓		
CAROL	✓	Surrogate Model	✓	✓	✓	✓	✓	✓	✓

modeling limitations, such methods do not scale for real-time operations, making them unsuitable for mission-critical edge applications. A similar approach, Load Balancing Mechanism (LBM) [14], executes user requests within edge nodes and utilizes metrics like network traffic and CPU utilization to decide the worker node that takes over as a broker in case of a failure. However, this work assumes a homogeneous edge setup and has been shown to often perform poorly in heterogeneous environments [30]. The Energy-efficient Checkpointing and Load Balancing (ECLB) [16] technique uses Bayesian methods to classify host machines into three categories: overloaded, underloaded and normal execution. This classification is used to decide appropriate task migrations to reduce the number of overloaded hosts. However, this model only considers computational overloads and does not consider other fault types like node thrashing or network failures that could lead to broker nodes being compromised. We use the best performing methods, DYVERSE and ECLB, as baselines in our experiments, as demonstrated in prior work [12], [16].

AI-based methods. Recently, several resilience models have been proposed that leverage AI methods like RL, surrogate or reconstruction modeling. An RL based approach is Load Balancing and Optimization Strategy (LBOS) [17] that allocates the resources using RL. The reward of the RL agent is calculated as a weighted average of multiple QoS metrics to avoid system contention by balancing the load across multiple compute nodes. The values of the weights are determined using genetic algorithms. LBOS observes the network traffic constantly, gathers the statistics about the load of each edge server, manages the arriving user requests and uses dynamic resource allocation to assign them to available edge nodes. However, RL approaches are known to be slow to adapt in dynamic settings [31]. Most other approaches use neural networks as a surrogate model. For instance, Effective Load Balancing Strategy (ELBS) [18] is a recent framework that offers an execution environment for IoT applications and creates an interconnect among cloud and edge servers. The ELBS method uses the priority scores to proactively allocate tasks to edge nodes or worker nodes as brokers to avoid system failures. It uses a fuzzy inference system to calculate the priority scores of different tasks based on three fuzzy inputs: SLO deadline, user-defined priority, and estimated task processing time. The priority values are generated by a neural network acting in the capacity of a surrogate of QoS scores. Another similar method is the Fuzzy-based Real-Time Auto-scaling (FRAS) [19] technique that leverages a virtualized environment for the recovery of IoT applications that run on compromised or faulty edge nodes. Here, FRAS executes each IoT application in a virtual machine (VM) and performs VM autoscaling to improve execution speed and reduce execution costs. The VM autoscaling decisions making involves inference of system QoS using a fuzzy recurrent neural network as a surrogate model. A major drawback of such surrogate modeling methods is that their parameters need to be periodically fine-tuned to adapt to dynamic environments, giving rise to high overheads. Other methods in this category generate a reconstruction of the system state and use the deviation from the input to indicate the likelihood that the state is faulty. For instance, TopoMAD [20] uses a topology-aware neural network that is composed of a Long-Short-Term-Memory (LSTM) and a variational autoencoder (VAE) to detect faults. However, the reconstruction error is only obtained for the latest state, limiting them to using reactive fault recovery policies. Other methods use slight variations of LSTM networks with either dropout layers [32], causal Bayesian networks [33] or recurrent autoencoders [34]. A GAN-based approach that uses a stepwise training process, StepGAN [21], converts the input time-series into matrices and executes convolution operations to capture temporal trends. These methods use various thresholding techniques like Peak Over Threshold (POT) [35] or Kernel Density Estimation (KDE) [36]. However, such techniques are not agnostic to the number of hosts or workloads as they assume a maximum limit of the active tasks in the system. Moreover, even though more accurate than heuristic based approaches, deep learning models such as deep autoencoders, GANs and recurrent networks are adaptive and accurate, but have a high memory footprint that adversely affects system performance. From this category, we

test the above mentioned approaches on the testbed described in Section V and use the empirically best techniques based on our experiments as baselines: LBOS, ELBS, FRAS, Topo-MAD and StepGAN.

III. Methodology

A. Environment Assumptions and Problem Formulation

System Model. As is commonplace in federated edge computing environments, we assume a system with heterogeneous nodes configured in a broker-worker fashion [37]. The assignment of edge nodes as brokers or workers and the allocation of all workers to one of a broker defines the topology of the system. We denote the number of edge nodes by H. As is common in edge federations, we assume that the broker nodes of LEIs are interconnected and we allow data sharing among brokers to facilitate transferring management tasks of workers across LEIs. We also assume a bounded timeline of execution of tasks within the federated environment, which we divide into fixed-sized scheduling intervals, where I_t denotes the t-th interval (t ranging from 0 to T). We also consider that the edge broker can sample the resource utilization metrics of all hosts within its LEI group at any time including CPU, RAM, disk/network bandwidth, some additional fault-related metrics including consumption of the swap space, disk buffers, network buffers, disk and network I/O waits.

Fault Model. As per prior work [38], [39], we consider a byzantine failure model for the edge nodes in our setup [40]. We assume that all failures are recoverable, *i.e.*, the machines can be rebooted to resume execution from their last working state, by frequently updating snapshots of the active hosts in the system. As in prior work, edge hosts in the same LEI are connected to the same power supply and unrecoverable faults like outages are ignored [39]. Failures can occur for a variety of reasons, including software or hardware defects, such as an infinite loop or long-running uninterruptible computation, resource exhaustion (thrashing), under-performing hardware (throttling), external events such as a slow computer network, misconfiguration, and compatibility problems. We realize this using an existing fault injection module [38] to create different fault types like CPU overload, RAM contention, Disk attack and DDOS attack. We specifically restrict our attention to faults that manifest in the form of resource over-utilization; for instance, a DDOS attack could lead to contention at the network interface. A broker or worker node may become unresponsive due to this resource over-utilization [41]. This work aims to find the best system topology in terms of estimated QoS in case one or more broker nodes fail. This is crucial as broker failures lead to service downtimes from all nodes in the LEI. In case of worker failures, we simply rerun tasks on the worker with the least resource utilization in the LEI. As this work takes a step in the complementary direction of traditional load-balancing and autoscaling methods, existing fault-tolerance techniques may be leveraged for worker management [12], [16].

Workload Model. All the tasks in our system are generated by users and transferred to the edge federation using gateway devices. We assume that gateway devices send tasks to the closest broker in terms of network latency, breaking ties uniformly at random. We assume a bag-of-tasks workload model, where a set of independent tasks enter each LEI at the start of each scheduling interval. Each task has an associated (soft) SLO deadline.

Formulation. In this work, we assume that the edge brokers run software solutions to manage their worker nodes. This includes making scheduling decisions for all incoming and active tasks in the system. Such solutions may include various fault tolerance and resilience models. We denote the set of brokers by B and workers by W. We also refer to these as broker and worker layers as per prior work [4]. In this work, in each scheduling interval, we check which broker or worker nodes are unresponsive (inactive) and update the system topology to optimize QoS. We encode the undirected topology graph of the federated edge environment at the start of the interval I_t as G_{t-1}. Formally, at the start of the interval I_t, all active nodes in the system are utilized to generate the topology G_t to execute tasks in I_t. The new graph G_t may be the same as G_{t-1} in case the active node set is unchanged.

To generate the graph G_t, we utilize the performance metrics of the previous interval, denoted by M_{t-1}, and the scheduling decision by S_t. The performance metrics include resource utilization and QoS metrics such as energy consumption and response time. For S_t we assume an underlying scheduler in the system independent from the proposed fault-tolerance solution. Using the input graph topology G_t, performance metrics M_{t-1} and an input scheduling decision S_t, the model needs to predict the performance metrics for the current interval, *i.e.*, M_t and a confidence score C_t. Using this model, we optimize over the topology space to find the optimal G_t for interval I_t. Without loss in generality, whenever unambiguous, we drop the subscripts for the sake of simplicity. Hence, we only refer to the inputs and outputs as G, M, S and C.

B. CAROL Model

Node-Shift. We assume that the network has tens of broker and worker nodes. In our work, the number of brokers and the system topology is known to all nodes in the federation. Whenever a broker node fails, we consider the worker nodes of the corresponding broker as being "orphaned". In such an event, either one of the worker nodes is shifted to the broker layer, or another node in the broker layer manages the orphaned nodes. This shift of nodes from worker layer to broker layer gives rise to the name "node-shift" and is similar to the fault-tolerance schemes used in software defined networks (SDNs) [42]. Considering an input topology G, there are multiple types of node shifts that may increase, decrease or even keep the broker count static. Three types of *worker-to-broker* node-shift mechanisms considered in this work are described below with a visualization presented in Figure 1 instead. We similarly consider the respective counterparts as the *broker-to-worker* node-shifts.

- *Type 1:* If a broker node fails, two of the orphaned nodes may be shifted to the broker layer and the remaining

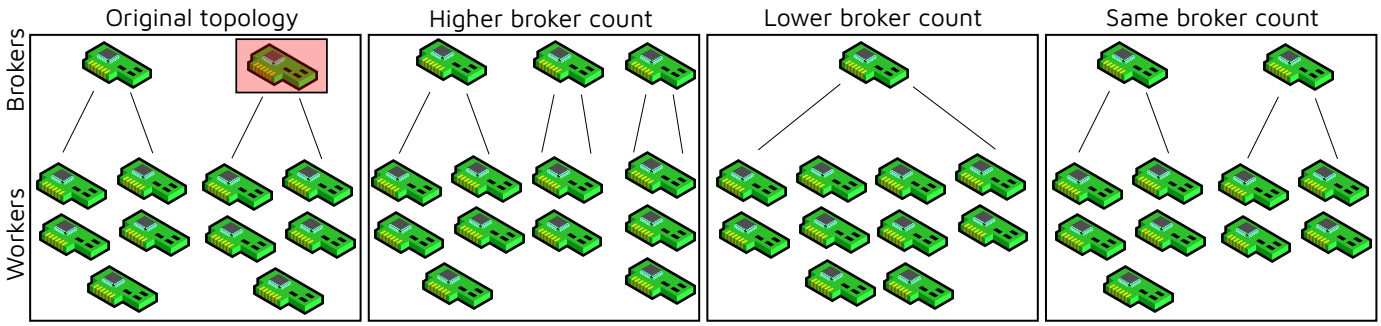


Fig. 1. Possible node-shifts in CAROL. Nodes marked in red are the high consumption brokers that break-down.

orphaned nodes may be evenly distributed among these two new brokers. This increases the broker count by one, also increasing the management capacity of the system for higher throughput at the broker layer.

- *Type 2:* The orphaned nodes could be assigned to another active broker in the system. This decreases the broker count than before failure and increases the computational throughput of the worker layer.
- *Type 3:* One of the orphaned worker nodes may be assigned to be the broker for the other worker nodes. This keeps the same number of broker nodes as before the failure. The three node-shift types may be utilized to trade off the throughputs of the broker and worker layers.

However, in heterogeneous edge federations, where nodes with disparate resource capacities may be present, the choice of the worker node to shift to the broker layer becomes vital. Moreover, with heterogeneous broker nodes, the choice of the broker nodes to which the orphaned nodes are assigned affects the QoS of the system. Another crucial factor is the overhead corresponding to the node-shift operations. This is due to the initialization of the broker management systems and synchronization of the system topology with other brokers in the federation.

Another critical aspect in finding the best graph topology is the workload heterogeneity across broker nodes. As shown in Figure 1, the node operations can lead to a different number of worker nodes and subsequently computational throughput of disparate LEIs. A single node-shift step might not be sufficient to manage resources in case of different computational loads across LEIs. Thus, a sequence of node-shifts may need to be tested to discover the topology with optimal QoS. However, generating QoS scores for a large number of node-shift sequences may not be feasible in latency-critical settings where generating QoS scores by execution or simulation might be a time-consuming activity. This is because a node-shift entails transfer of broker level data to the worker node and initializing management software containers, both of which give rise to a higher latency.

Confidence-Aware Model. To eschew the costly execution of multiple node-shift sequences and observe their effects on QoS, we need a lightweight model that mimics the behavior of a computationally expensive simulation or execution [43]. Such models are referred to as "surrogate models" in the literature [43]. Using a surrogate model enables such methods to run optimization in the input space and generate decisions, such as node-shift sequences that give G_t, giving high estimated QoS. Recent surrogate optimization techniques use gradient-based optimization compared to evolutionary search strategies that facilitate quick convergence [31]. However, in discrete search spaces, such as graph topologies, such techniques typically select the discrete point close to the converged point in the continuous space. This may often lead to non-optimal solutions [43, §19.1]. Another drawback of such methods is that the parameters of the neural networks need to be periodically fine-tuned, leading to high overheads [44]. To reduce such overheads, it is crucial to fine-tune models only when the system or workload configurations change. To do this, we take motivation from confidence-aware deep learning [45] and a recently proposed class of generative models, called Generative Optimization Networks (GONs) [24], to integrate in CAROL a memory efficient QoS surrogate. This is used to predict a confidence score, such that for low confidence the model can be fine-tuned with online generated data.

GON based Neural Network. GANs are based on a pair of neural networks, a generator and a discriminator where the generator takes random noise samples and outputs samples from a data distribution as inputs. The discriminator predicts a likelihood score for the input belonging to the target distribution. Unlike traditional GANs, we leverage a memory-efficient Generative Optimization Network (GON) [24]. GONs are similar to GANs, but do not use the generator, significantly reducing the memory footprint of the neural network. We now describe the working of the GON model and confidence-based QoS prediction.

Consider a discriminator network D with parameters θ that takes as input graph topology G, performance metrics M and scheduling decision S. The output of D, *i.e.*, $D(M, S, G; \theta)$ is a likelihood score of G, M and S belonging to a distribution. When trained for a dataset generated from the normal execution of an edge federation, the GON model predicts a high score for an input tuple (M, S, G) in the distribution of the dataset and a low score for an unseen tuple. This

Algorithm 1 Minibatch stochastic gradient based training of GON model in CAROL. Input is dataset Λ and hyperparameters m and γ.

Require: Dataset $\Lambda = \{M_t, S_t, G_t\}_{i=0}^{T}$

1: **for** number of training iterations **do**

2: Sample minibatch of size m performance metrics $\{M^{(1)}, \ldots, M^{(m)}\}$ from Λ.

3: Sample minibatch of size m noise samples $\{Z^{(1)}, \ldots, Z^{(m)}\}$.

4: Generate new samples $\{Z^{*(1)}, \ldots, Z^{*(m)}\}$ by running the following till convergence

$$Z \leftarrow Z + \gamma \cdot \nabla_Z \log\big(D(Z, S, G; \theta)\big).$$

5: Update the discriminator by ascending the stochastic gradient.

$$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} \big[\log\big(D(M^{(i)}, S, G; \theta)\big) + \log\big(1 - D(Z^{*(i)}, S, G; \theta)\big)\big].$$

allows us to translate the likelihood score output as a measure of the confidence of the network. Essentially, for dynamic systems, when the confidence score drops below a threshold, it indicates us to fine-tune the model. This enables parsimonious model fine-tuning, giving us an effective compromise between prediction performance and training overheads.

To generate the performance scores for an input graph topology G, we can randomly initialize M and maximize the discriminator output by ascending the stochastic gradient

$$M \leftarrow M + \gamma \nabla_M \log\big(D(M, S, G; \theta)\big), \tag{1}$$

where γ denotes the step size in the optimization loop. We use log-likelihood instead of likelihood scores for training stability [26]. Thus, starting from an input (S, G) pair, the above optimization loop converges us to give performance metrics M^* and a confidence score of $D(M^*, S, G)$.

Offline Model Training. To train the GON model D, we first collect an execution trace $\Lambda = \{M_t, S_t, G_t\}_{i=0}^{T}$. A summary of the training process is presented in Algorithm 1. We leverage an adversarial training process where we use the generated data (say Z^*) as fake samples and datapoints from Λ (say M) as real samples and train the model using the binary cross-entropy loss

$$L = \log\big(D(M, S, G; \theta)\big) + \log\big(1 - D(Z^*, S, G)\big), \tag{2}$$

where $(M, S, G) \in \Lambda$. Thus, the model is trained to optimize the likelihood scores for fake and real samples in an adversarial fashion using the same loss function. This adversarial style training has a two-fold goal. First, the model learns to generate new samples (Z^*) such that the tuple (Z^*, S, G) belongs to the data distribution. Thus, with sufficient training, converged Z^* are close estimates of the performance metrics for inputs G and S. Second, for previously seen datapoints, *i.e.*, (M, S, G) tuples, the predicted confidence scores are high. If the (S, G) input is unseen during training, then the converged confidence score $D(Z^*, S, G)$ would also be low. In practice, for the scheduling interval I_t, instead of starting from a random noise sample Z, we initialize M as M_{t-1} and then converge $D(M, S_t, G_t)$ to M_t. This exploits the temporal correlation between subsequent system states, facilitating quick convergence of (1).

Algorithm 2 The CAROL resilience model.

Require:
Trained GON Model D; Running dataset Γ
Broker set B; Worker set W

1: **for** $t = 0$ to T **do**
2: S_t from underlying scheduler
3: M_t from edge system
4: $G_t \leftarrow \text{NodeShift}(G_{t-1})$ ▷ Initialize topology
5: **for each** broker $b \in B$ **do**
6: **if** b fails **do**
7: $G \in N(G_t, b)$ ▷ Random node-shift
8: $G_t \leftarrow \text{TabuSearch}(G, \Omega)$ ▷ Tabu Search
9: **if** no broker in B fails **do**
10: Add datapoint (M_t, S_t, G_t) to Γ ▷ Save datapoint
11: $C \leftarrow D(M_t, S_t, G_t; \theta)$ ▷ Confidence score
12: $POT \leftarrow \text{PeakOverThreshold}(C)$ ▷ POT calculation
13: **if** $C < POT$ **do**
14: $L = \log(D(M, S, G; \theta)) + \log(1 - D(Z*, S, G))$
15: Fine-tune D using L and `Adam` ▷ Fine-tune
16: Clear Γ
17: Configure topology as G_t
18: Schedule tasks using S_t =0

Finding optimal edge topology. Having a surrogate model, at each scheduling interval I_t, we can now run optimization of the QoS by taking a convex combination of the QoS metrics in M_t. We denote this objective function as $O(M_t)$. Now, if a node fails in I_{t-1}, then we can create the graph topology G_t by a node-shift operation on G_{t-1} (line 4 in Alg. 2). However, to decide the optimal node-shift operation, we can run a local search in the topology space. We select the tabu search algorithm due to its deterministic nature and empirically faster convergence for the specific optimization problem we consider [46]. Thus, in case of a node failure, we start from graph G_{t-1}, apply a random node shift to generate G and use tabu search to generate the graph G_t such that the QoS score is optimized (line 8 in Alg. 2). We perform this iteratively for each failed broker node to achieve the final graph topology. If no broker failed in I_{t-1}, then we add the datapoint to a running dataset Γ (line 10 in Alg. 2).

The local neighbors of a graph G for a failed node, say b, is obtained by performing all possible node-shift operations. We denote this neighbourhood set of G by $N(G, b)$ (line 7 in Alg. 2). The QoS score is obtained using the GON model described previously, where we evaluate M_t using (1) over $D(M, S_t, G)$ and use the objective function $O(M_t)$ for our local search. For notational convenience, we define the objective score of a graph G, parameterized by the GON model D, scheduling decision S_t and objective function O

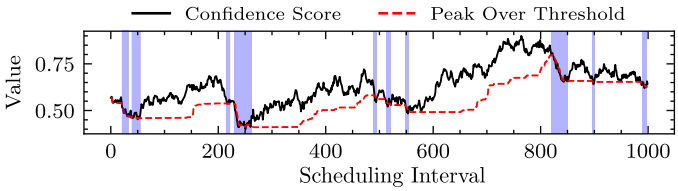


Fig. 2. Visualizing the confidence scores and POT threshold values for 1000 scheduling intervals in CAROL. We use the testbed and fault injection module described in Section IV. The blue bands represent intervals where the confidence fell below threshold value and the model was fine-tuned with the latest collected data.

as $\Omega(G; D, S_t, O)$. As the network size increases, the number of possible node-shift operations also increases. Thus, we use a tabu list of a fixed size L in our search scheme.

Confidence Aware Fine-Tuning. After performing optimization in the topology space and obtaining G_t for interval I_t, we also evaluate the confidence score $D(M_t, S_t, G_t; \theta)$ (line 11 in Alg. 2). We then use the Peak Over Threshold (POT) method [35] that dynamically chooses threshold values below which we fine-tune our model (line 12). POT uses extreme value theory to generate a threshold value from a continuous record of past confidence scores. It does this by checking the peak values reached for any period during which values fall below a certain threshold. This threshold is dynamically updated based on incoming data to ensure that the model adapts to non-stationary settings. In any interval, if the confidence score falls below the POT threshold, CAROL fine-tunes the GON network with the latest collected dataset Γ (line 15). A visualization of the training process is shown for a thousand scheduling intervals in Figure 2. The figure clearly demonstrates that the model trains the GON network only when there are dips in the confidence scores, allowing our technique to have much lower fine-tuning overheads compared to other methods that tune their neural networks at each scheduling interval.

IV. Implementation

A. GON Network.

It is crucial that the D network is able to capture the correlation between system topology and scheduling decisions with the performance metrics to effectively predict QoS and confidence scores. The model used in our approach is a composite neural network shown in Figure 3 that infers the correlations between performance metrics, scheduling decision and graph topology to generate the discriminator output. Considering we have p tasks running as a sum of new and active tasks, the scheduling decisions of these are encoded as one-hot vectors of size $|H|$ (number of hosts in the system). We thus get a matrix of scheduling decisions (S) of size $[p \times |H|]$. Also, we collect performance metrics of each host $i \in H$ denoted by M_i that includes resource utilization metrics CPU, RAM, Disk, Bandwidth consumption with QoS metrics such as energy consumption and SLO violation rates. We denote the resource utilization metrics of host i as u_i and QoS metrics as q_i. We

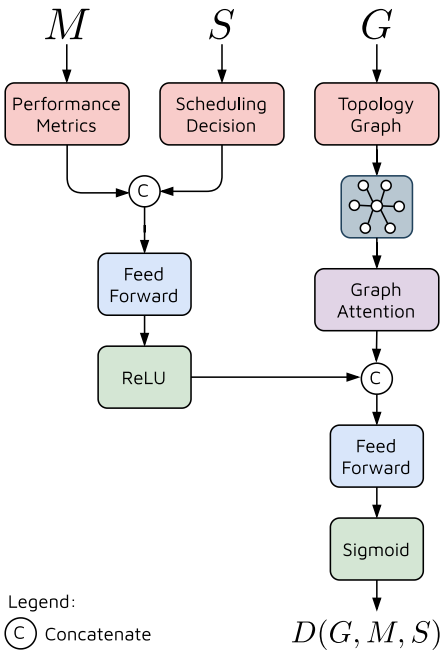


Fig. 3. Neural network used in CAROL. The three inputs to the model are shown in red. Feed-forward, graph operations and activations are shown in blue, purple and green.

also include the task resource utilization consumption with SLO deadlines, denoted as t_i. Thus, $M_i = [u_i, q_i, t_i]$. The M_i vectors for all hosts are stacked together to form the matrix M. To encode inputs M and S, we use feed-forward layers to bring down the dimension size of the input and ReLU activation:

$$E^{\{M,S\}} = \text{ReLU}(\text{FeedForward}([W, S])). \tag{3}$$

The topology of the edge federation is represented as a graph with nodes as edge hosts and edges corresponding to edge groups. All edge workers are connected via undirected edges to their respective broker, and all brokers are connected to every other broker. The resource utilization characteristics of each edge host are then used to populate the feature vectors of the nodes in the graph. We denote the feature vector of host i as e_i that are computed from u_i. We encode this graph G using a graph attention network [47] for scalable computation over the input graph. The motivation behind using a graph attention network is to allow computation over the graph to be agnostic to the number of nodes in the system topology. Graph attention operation performs convolution operation for each node over its neighbors and uses dot product self-attention to aggregate feature vectors. Thus, we transform the input graph G using *graph-to-graph* updates as:

$$e_i = \sigma\Big(\sum_{j \in n(i)} W^q \cdot \tanh(W\ u_i + b)\Big), \tag{4}$$

where W^q are the attention weights obtained by the weight matrix W [47] and $n(i)$ are the neighbors of host i in graph G. The stacked representation for all hosts (e_i) is represented as

E^G. Now, we pass this representation through a feed-forward layer with sigmoid activation as

$$D(M, S, G; \theta) = \text{Sigmoid}(\text{FeedForward}([E^{\{M,S\}}, E^G])). \tag{5}$$

Here, the sigmoid function allows us to bring the output in the range $[0, 1]$, the same as that of the true normalized window.

B. Objective function.

For generating the objective function $O(M_t)$ as part of Ω in Alg. 2, we use a convex combination of the energy consumption and SLO violation rates of the system as is commonly used in prior work to estimate system performance [31], [44]. For the energy consumption (q_i^{energy}) and SLO violation rates (q_i^{slo}) of each host i, we generate scores for the complete system

$$\begin{aligned} q^{energy} &= \sum_{i \in H} q_i^{energy}, \\ q^{slo} &= \sum_{i \in H} q_i^{slo}. \end{aligned} \tag{6}$$

We then compute the QoS score as

$$O(M) = \alpha \cdot q^{energy} + \beta \cdot q^{slo}. \tag{7}$$

This is motivated from prior work [31] where α and β (such that $\alpha + \beta = 1$) are hyper-parameters that can be set by users in line with the application requirements. Typically, in real-life settings, these values are around 0.5 [31], [44]. For an energy-constrained setting, a higher α value is used, whereas for latency-critical tasks, a higher β value is used.

C. Testbed.

Our testbed consists of 16 Raspberry Pi 4B nodes, 8 with 4GB and another 8 with 8GB RAM each. This allows the setup to have heterogeneous nodes with different memory capacities. We consider the same federated topology as a starting point in our experiments. Here, we have 4 LEIs with brokers as 8GB nodes and workers symmetrically distributed from the 4GB and remaining 8GB nodes. The devices run the Ubuntu-18.04 LTS operating system. All WAN and LAN links were 1 Gbps. To emulate each LEI being in geographically distant locations, we use the `NetLimiter` tool to tweak the communication latency among brokers nodes using the model described in [48]. To emulate the gateway devices sending tasks that affect the load distribution across LEIs, we use the gateway mobility model described in [49].

D. Creating the Training Dataset.

To generate a normal execution trace as the dataset to train the GON model, we ran the widely-used DeFog benchmark applications [28] on our testbed. Specifically, we use the Yolo, PocketSphinx and Aeneas workloads. We build CAROL as a layer on top of the state-of-the-art surrogate modeling-based scheduler (GOBI) [31] and execute these tasks as Docker containers. Our runs are divided into five minute long scheduling intervals. We run these traces for 1000 intervals where we periodically change the graph topology every ten intervals. We collect the performance metrics, scheduling decision and graph topology to generate the training dataset $\Lambda = \{M_t, S_t, G_t\}_{t=0}^{1000}$ with 100 different graph topologies.

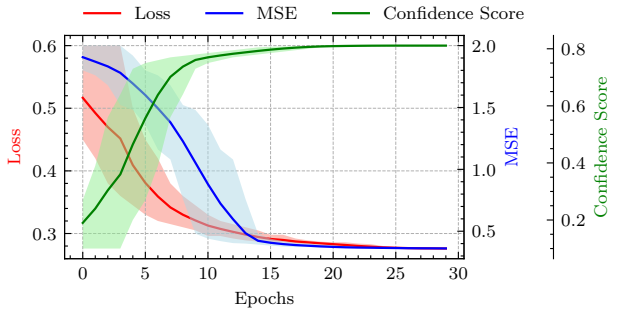


Fig. 4. Training plots for the GON model. The model converges in 30 training epochs.

E. Training Details and Hyperparameter Selection.

For GON, we only change the number of layers in the feed-forward networks in Figure 3 (keeping a fixed layer size of 128 nodes). We choose the model memory footprint based on grid search so to minimize the Mean-Square-Error (MSE) between the predicted performance metrics and the corresponding values in the dataset. A GON network with a lower parameter size is prone to underfitting and high MSE. A large number of parameters increase the training and fine-tuning time, subsequently increasing the scheduling time due to the slower optimization based generation in (1). To avoid high MSE values and scheduling times, we use a GON model that consumes $\sim$1GB RAM. A more exhaustive sensitivity analysis is given in Section V-E. For training, we randomly split the dataset into 80% training and 20% testing data. We use a learning rate of 10^{-4} with a weight decay of 10^{-5} in the `Adam` optimizer for optimizing the loss function. The training curves showing loss and F1 scores on the test set are shown in Figure 4. We use the early stopping criterion for convergence. The minibatch size and tabu list size were also obtained using grid search and the values used in our experiments were 32 and 100. More details in Section V-E.

F. Fault Injection Module.

To generate broker failures at test time, we use an existing fault-injection module [38]. We create faults of the type: CPU overload, RAM contention, Disk attack and DDOS attack. In a CPU overload attack, a simple CPU hogging application is executed that creates contention of the compute resources. In RAM contention attack, a program is run that performs continuous memory read/write operations. In disk attacks, we run the IOZone benchmark that consumes a large portion of the disk bandwidth. In a DDOS attack, we perform several invalid HTTP server connection requests causing network bandwidth contention. More details are given in [38]. We generate faults using a Poisson distribution with the rate $\lambda_f = 0.5$, sampled uniformly at random from the attack set.

These attacks were performed to cause the byzantine failure of broker nodes.

G. Broker Failure Detection.

In our implementation, all broker nodes periodically (every 30 seconds) `ping` each other to test which hosts are active in the system. Five Internet Control Message Protocol (ICMP) packets are sent using the *ping* utility to every other broker and the node waits for a response with a timeout counter set to 10 seconds. If the broker responds, we run the audit checking procedure that verifies the signed log entries since the previous audit [50]. If a broker is unresponsive or the audit check fails, an entry is made in a shared PostgreSQL database. For a broker $b \in B$, if all other brokers report it as unresponsive, then b is assumed to be compromised. A 'reboot' command is run on this broker in a non-blocking fashion using the `ssh` utility.

H. Node-shift Implementation.

All sharing of resource utilization characteristics across brokers uses the `rsync` utility. For each worker node, an entry is kept corresponding to the IP address of the broker node to which the worker is assigned. At a node-shift event, the IP address of the new broker is updated for each orphaned node. The worker nodes refresh the broker IP addresses at the start of each scheduling interval. When a worker is assigned as a broker node, it runs the broker management software as a Docker container.

I. Broker Recovery.

As described in Section III, we assume temporary failures in our setup. Thus, after a broker node fails, it takes 1-5 minutes for this node to restore to the previous state and resume computation/management. To ensure smooth execution of the system, we use the Virtual Router Redundancy Protocol (VRRP) to assign a set of virtual IPs to the broker nodes in the system. We implement this using the `keepalived` high-availability toolkit.[2] As soon as a failed node comes back online, we add it to the graph topology and assign it as a worker in the closest active broker as per network latency. This is performed at the time of graph topology initialization at the start of each interval (line 4 of Alg. 2).

V. Performance Evaluation

We compare the CAROL fault resilience method against a heuristic baseline DYVERSE [12], meta-heuristic baseline ECLB [16], an RL baseline LBOS [17], two surrogate modelling based methods ELBS [18] and FRAS [19], and two reconstruction models TopoMAD [20] and StepGAN [21] (more details in Section II). As TopoMAD and StepGAN are only fault-detection methods, we supplement them with the priority based load-balancing policy from the next best baseline, *i.e.*, FRAS. We use hyperparameters of the baseline models as presented in their respective papers. We train all deep learning models using the PyTorch-1.8.0 [51] library.

[2]Keepalived. `https://github.com/acassen/keepalived`. Accessed 7 November 2021.

A. Workloads.

To test the generalization capability of the various resilience models and their adaptive capacity, we do not use the DeFog benchmarks as workloads at test time. Instead, we use the *AIoTBench* applications [29]. AIoTBench is an AI-based edge computing benchmark suite that consists of various real-world computer vision application instances. The seven specific application types correspond to the neural networks they utilize. These include three typical heavy-weight networks: ResNet18, ResNet34, ResNext32x4d, as well as four light-weight networks: SqueezeNet, GoogleNet, MobileNetV2, MnasNet.[3] This benchmark is used due to its volatile utilization characteristics and heterogeneous resource requirements. The benchmark consists of 50,000 images from the COCO dataset as workloads [52]. To evaluate the proposed method in a controlled environment, we abstract out the users and IoT layers described in Section III and use a discrete probability distribution to generate tasks as container instances. Thus, at the start of each scheduling interval, we create new tasks from a Poisson distribution with rate $\lambda_t = 1.2$, sampled uniformly from the three applications. The Poisson distribution is a natural choice for a bag-of-tasks workload model, common in edge environments [53]. Our tasks are executed using Docker containers. We run all experiments for 100 scheduling intervals, with each interval being five minutes long, giving a total experiment time of 8 hours 20 minutes. We average over five runs and use diverse workload types to ensure statistical significance of our experiments.

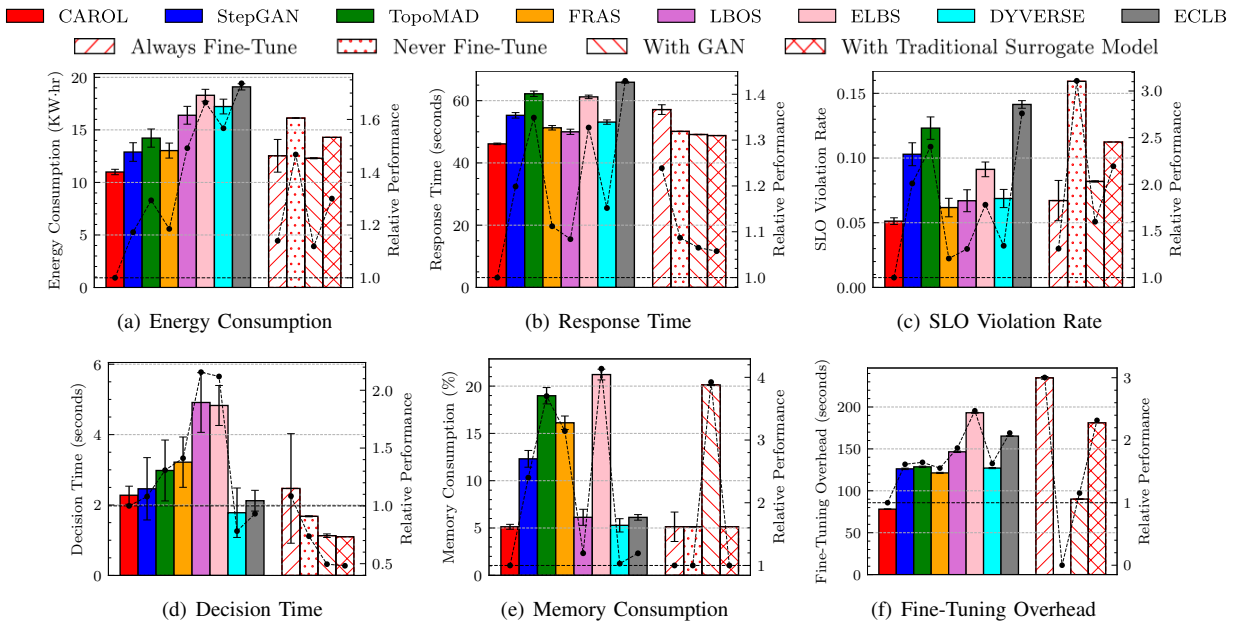
B. Evaluation Metrics.

We measure the energy consumption of the federated setup, average response time and SLO violation rate of completed tasks. We consider the relative definition of SLO (as in [31]) where the deadline is the 90^{th} percentile response time for the same application on the state-of-the-art method StepGAN that has the highest F1 scores among the baselines. We also consider the average inference time of the models, that is the time to decide the steps for system resilience, such as deciding the required node-shift operations. We also compare the memory consumption of the techniques and the fine-tuning overhead averaged over the scheduling intervals.

C. Comparison with Baselines.

We now present results comparing CAROL with baselines. For our experiments, we use $\alpha = \beta = 0.5$ in (7) as per prior work [31], [44]. Figure 5(a) compares the total energy consumption of all models and shows that CAROL is able to execute tasks with minimum energy consumption in the federated edge environment. The presence of the average energy consumption metric of all the edge nodes within the QoS score calculation as shown in (7) forces the model to take energy-efficient node-shift decisions. It results in the minimum number of active hosts with the remaining hosts in standby

[3]AIoTBench: `https://www.benchcouncil.org/aibench/aiotbench/index.html`. Accessed: 5 November 2021.



(a) Energy Consumption (b) Response Time (c) SLO Violation Rate

(d) Decision Time (e) Memory Consumption (f) Fine-Tuning Overhead

Fig. 5. Comparison of CAROL with baseline methods and ablated models. The y-axis on the left shows the absolute values and the one on the right shows relative performance with respect to CAROL. The results corresponding to the ablated models are shown as hatched bar plots.

mode to conserve energy. CAROL reduces energy consumption by 16.45% compared to StepGAN, the model having minimum energy consumption across all baseline methods.

Figure 5(b) shows the average response time per task in seconds. The response time is measured as the time difference between the timestamps of task creation and result generation and is a crucial metric to compare the service throughput offered by the edge federation. The figure demonstrates that CAROL reduces this metric by 8.04% compared to the best baseline FRAS.

Figure 5(c) shows the fraction of SLA violations out of the total completed tasks in the 100 scheduling intervals. The figure shows that CAROL has the lowest SLO violation rate of only 5.12%, 17.01% lower than the least violation rates among the baselines, *i.e.*, 6.17% of the FRAS method. Allowing node-shift at each interval with the SLO violation rate in the QoS score calculation ensures that the topology minimizes response time and subsequently the SLO violation rates on an average. Moreover, CAROL is given the SLO deadline for each task that it utilizes to decide the number of worker nodes to allocate per task to minimize the SLA violation metric in the QoS score as shown in (7).

Figure 5(d) shows the comparison of average time to decide the fault-tolerance steps for each model. The figure shows that CAROL has a lower decision time compared to all AI-based methods. LBOS and ELBS have the highest decision time due to the time-consuming weighted round-robin and match-making algorithms used in these methods. The least decision time is of the DYVERSE heuristic method. However, CAROL has only 6.77% higher decision time than DYVERSE.

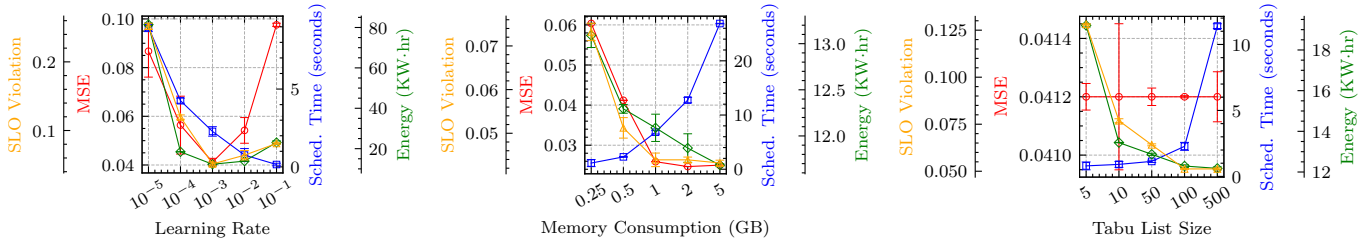
Figure 5(e) shows the average percentage memory utilization of the fault-tolerance models. The surrogate and generative methods have high memory consumption. ELBS has the highest memory consumption due to the resource intensive fuzzy neural networks in this approach. The RL based method, LBOS, has the lowest memory footprint among the AI baselines due to a lightweight Q-table used in the Q learning model in LBOS. Compared to the heuristic methods, CAROL has memory consumption (∼5%), thanks to the memory-efficient GON model.

Figure 5(f) shows the overheads for fine-tuning the AI models and dynamically updating the priority scores in the heuristic models. The overhead is measured for the complete experiment, *i.e.*, over 100 scheduling intervals. The figure shows that CAROL has the lowest overhead of 78.12 seconds, only 0.78 seconds on an average per five-minute interval. This is 35.62% lower than the baseline with the lowest overhead, *i.e.*, FRAS that takes 121.35 seconds to update its model periodically. The significantly lower overhead of CAROL is due to the economic confidence-based model fine-tuning.

Overall, we observe that the ability to approximate the QoS scores for a given graph of the system topology and scheduling decision enables the GON-based generative model to predict the confidence scores and accurately predict QoS scores for any given state of the system. This, with the node-shifting techniques in CAROL, allows us to optimize the QoS of the federated environment.

D. Ablation Analysis

To study the relative importance of each component of the model, we exclude every major component one at a time and observe how it affects the performance of the scheduler. An overview of this ablation analysis is given in Fig 5 as hatched bars. First, we consider the CAROL model without the confidence scores, fine-tuning the GON network at every



(a) Learning Rate (b) Memory Size (c) Tabu List Size

Fig. 6. Sensitivity Analysis.

scheduling interval (*Always Fine-Tune* model). We also consider CAROL without fine-tuning the GON network in any interval (*Never Fine-Tune* model). Next, we consider a model with a traditional GAN network instead of the GON based confidence and QoS prediction (*With GAN* model). Lastly, we consider the CAROL model with a traditional feed-forward surrogate network instead of GON (*With Traditional Surrogate* model). We report the following findings:

- Without the confidence-aware model fine-tuning, the overheads significantly increase, causing a higher average decision time and response time of applications.
- Without ever fine-tuning the model, it does not adapt in non-stationary settings giving rise to poor QoS scores.
- Without the GON model, and using a GAN instead, the decision time is reduced as we do not need to optimize in the topology space. However, the memory consumption increases from 5% to 30%, which impacts the running applications and broker management. This causes the SLO violation rate to increase by 6%.
- Without the GON model, and using a traditional feed-forward model, the decision time is reduced again, but at the cost of higher fine-tuning overheads.

We observe that the confidence-aware model fine-tuning and using the GON network have the maximum impact on performance values, accounting to nearly 70% of the total impact.

E. Sensitivity Analysis

Figure 6 provides a sensitivity analysis of the performance of the CAROL model with learning rate, *i.e.*, γ in (1), memory consumption and size of the tabu list. This sensitivity analysis highlights the trade-off between the QoS scores and scheduling time as we increase either of these parameters. The variation of performance metrics with the learning rate is more straightforward. The scheduling time of CAROL decreases as we increase the learning rate. This is due to larger jumps in the optimization steps. However, for a high learning rate of $\gamma \geq 10^{-2}$, the model is unable to converge to the optima and hence we see the increase in the MSE scores, energy consumption and SLO violation rates for these values. The QoS scores are best for the learning rate of $\gamma = 10^{-3}$ and hence have been used in our experiments.

As the number of layers in the GON model increases, so does the memory footprint. This increases the scheduling time as it takes longer to generate samples by running optimization in the input space. However, a higher layer count improves the prediction performance by giving lower MSE scores, leading to lower energy consumption and SLO violation rates due to the more accurate QoS prediction. However, we see marginal improvement in energy consumption and SLO violation rates for memory consumption >1GB, but large increase in the scheduling time of the model. Thus, we use the model with 1GB of memory footprint (3 layers) in our experiments. Finally, increasing the size of the tabu list increases the scheduling time and gives better energy and SLO scores. We use a 100 size tabu list in our experiments.

VI. Conclusions and Future Work

We have developed a novel method for resilient computing in edge federations. Our method uses a lightweight generative network as a surrogate model to efficiently and precisely map performance metrics like energy consumption and SLO violation rates for a given graph topology and scheduling decision. In the event of a broker failure, CAROL reactively optimizes the graph topology to accommodate the orphaned worker nodes. The topology is chosen by running a tabu search to optimize the QoS scores predicted by the surrogate model. CAROL uses the discriminator output as a confidence score that allows us to run model fine-tuning steps only when confidence scores drop below running thresholds. This parsimonious fine-tuning gives rise to 35.6% lower overheads compared to the current state-of-the-art. Performance evaluation of a real edge test-bed with AI and IoT based benchmark applications show that the low overheads in CAROL can improve energy consumption and SLO violation rates by 16.5% and 17.0% compared to the state-of-the-art.

The current approach is able to achieve optimal performance due to its lower fine-tuning overheads compared to prior work. Our methods are best suited to highly dynamic systems. For stationary settings, we propose to extend the current reactive model to a proactive scheme that is able to prevent node failures. However, proactive optimization may entail higher computation for improved predictive performance. Such challenges would be addressed in the future.

Software Availability

The code and relevant result reproduction scripts are available at `https://github.com/imperial-qore/CAROL`.

REFERENCES

[1] P. K. R. Maddikunta, Q.-V. Pham, B. Prabadevi, N. Deepa, K. Dev, T. R. Gadekallu, R. Ruby, and M. Liyanage, "Industry 5.0: a survey on enabling technologies and potential applications," *Journal of Industrial Information Integration*, p. 100257, 2021.

[2] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things journal*, vol. 1, no. 1, pp. 22–32, 2014.

[3] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres *et al.*, "The reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, no. 4, pp. 4–1, 2009.

[4] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya, "Fogbus: A blockchain-based lightweight framework for edge and fog computing," *Journal of Systems and Software*, 2019.

[5] T. Hao, Y. Huang, X. Wen, W. Gao, F. Zhang, C. Zheng, L. Wang, H. Ye, K. Hwang, Z. Ren *et al.*, "Edge AIBench: towards comprehensive end-to-end edge computing benchmarking," in *International Symposium on Benchmarking, Measuring and Optimization*. Springer, 2018, pp. 23–30.

[6] J. Shao and J. Zhang, "Communication-computation trade-off in resource-constrained edge inference," *IEEE Communications Magazine*, vol. 58, no. 12, pp. 20–26, 2020.

[7] K. Bilal, E. Baccour, A. Erbad, A. Mohamed, and M. Guizani, "Collaborative joint caching and transcoding in mobile edge networks," *Journal of Network and Computer Applications*, vol. 136, pp. 86–99, 2019.

[8] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2016, pp. 168–178.

[9] S. Li and T. Lan, "Hotdedup: Managing hot data storage at network edge through optimal distributed deduplication," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 247–256.

[10] S. P. Ahuja, S. Mani, and J. Zambrano, "A survey of the state of cloud computing in healthcare," *Network and Communication Technologies*, vol. 1, no. 2, p. 12, 2012.

[11] S. Ristov, T. Fahringer, D. Peer, T.-P. Pham, M. Gusev, and C. Mas-Machuca, "Resilient techniques against disruptions of volatile cloud resources," in *Guide to Disaster-Resilient Communication Networks*. Springer, 2020, pp. 379–400.

[12] N. Wang, M. Matthaiou, D. S. Nikolopoulos, and B. Varghese, "DYVERSE: dynamic vertical scaling in multi-tenant edge environments," *Future Generation Computer Systems*, 2020.

[13] D. Rathod and G. Chowdhary, "Scalability of m/m/c queue based cloud-fog distributed internet of things middleware," *International Journal of Advanced Networking and Applications*, vol. 11, no. 1, pp. 4162–4170, 2019.

[14] H. A. Khattak, H. Arshad, S. ul Islam, G. Ahmed, S. Jabbar, A. M. Sharif, and S. Khalid, "Utilization and load balancing in fog servers for health applications," *EURASIP Journal on Wireless Communications and Networking*, vol. 2019, no. 1, p. 91, 2019.

[15] T. Gouasmi, W. Louati, and A. H. Kacem, "Exact and heuristic mapreduce scheduling algorithms for cloud federation," *Computers & Electrical Engineering*, vol. 69, pp. 274–286, 2018.

[16] A. Sharif, M. Nickray, and A. Shahidinejad, "Fault-tolerant with load balancing scheduling in a fog-based iot application," *IET Communications*, vol. 14, no. 16, pp. 2646–2657, 2020.

[17] F. M. Talaat, M. S. Saraya, A. I. Saleh, H. A. Ali, and S. H. Ali, "A load balancing and optimization strategy (lbos) using reinforcement learning in fog computing environment," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–16, 2020.

[18] F. M. Talaat, S. H. Ali, A. I. Saleh, and H. A. Ali, "Effective load balancing strategy (ELBS) for real-time fog computing environment using fuzzy and probabilistic neural networks," *Journal of Network and Systems Management*, vol. 27, no. 4, pp. 883–929, 2019.

[19] M. Etemadi, M. Ghobaei-Arani, and A. Shahidinejad, "A cost-efficient auto-scaling mechanism for IoT applications in fog computing environment: a deep learning-based approach," *Cluster Computing*, pp. 1–16, 2021.

[20] Z. He, P. Chen, X. Li, Y. Wang, G. Yu, C. Chen, X. Li, and Z. Zheng, "A spatiotemporal deep learning approach for unsupervised anomaly detection in cloud systems," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[21] Y. Feng, Z. Liu, J. Chen, H. Lv, J. Wang, and J. Yuan, "Make the rocket intelligent at iot edge: Stepwise gan for anomaly detection of lre with multi-source fusion," *IEEE Internet of Things Journal*, 2021.

[22] M. Roopaei, P. Rad, and M. Jamshidi, "Deep learning control for complex and large scale cloud systems," *Intelligent Automation & Soft Computing*, vol. 23, no. 3, pp. 389–391, 2017.

[23] W. Luping, W. Wei, and L. Bo, "CMFL: Mitigating communication overhead for federated learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 954–964.

[24] S. Tuli, S. Tuli, G. Casale, and N. R. Jennings, "Generative optimization networks for memory efficient data generation," *Advances in Neural Information Processing Systems, Workshop on ML for Systems*, 2021.

[25] M. Ashman, J. So, W. Tebbutt, V. Fortuin, M. Pearce, and R. E. Turner, "Sparse gaussian process variational autoencoders," *arXiv preprint arXiv:2010.10177*, 2020.

[26] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[27] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *International Conference on Learning Representations (ICLR)*, 2014.

[28] J. McChesney, N. Wang, A. Tanwer, E. de Lara, and B. Varghese, "DeFog: fog computing benchmarks," in *The 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 47–58.

[29] C. Luo, F. Zhang, C. Huang, X. Xiong, J. Chen, L. Wang, W. Gao, H. Ye, T. Wu, R. Zhou *et al.*, "AIoT bench: towards comprehensive benchmarking mobile and embedded device intelligence," in *International Symposium on Benchmarking, Measuring and Optimization*. Springer, 2018, pp. 31–35.

[30] Z. Nezami, K. Zamanifar, K. Djemame, and E. Pournaras, "Decentralized edge-to-cloud load balancing: Service placement for the internet of things," *IEEE Access*, vol. 9, pp. 64 983–65 000, 2021.

[31] S. Tuli, S. R. Poojara, S. N. Srirama, G. Casale, and N. R. Jennings, "COSCO: Container Orchestration Using Co-Simulation and Gradient Based Optimization for Fog Computing Environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 101–116, 2022.

[32] L. Girish and S. K. Rao, "Anomaly detection in cloud environment using artificial intelligence techniques," *Computing*, pp. 1–14, 2021.

[33] Y. Gan, S. Dev, D. Lo, and C. Delimitrou, "Sage: Leveraging ml to diagnose unpredictable performance in cloud microservices," *ML for Computer Architecture and Systems*, 2020.

[34] S. Chouliaras and S. Sotiriadis, "Detecting performance degradation in cloud systems using lstm autoencoders," in *International Conference on Advanced Information Networking and Applications*. Springer, 2021, pp. 472–481.

[35] A. Siffer, P.-A. Fouque, A. Termier, and C. Largouet, "Anomaly detection in streams with extreme value theory," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1067–1075.

[36] E. Martin and A. Morris, "Non-parametric confidence bounds for process performance monitoring charts," *Journal of process control*, vol. 6, no. 6, pp. 349–358, 1996.

[37] J. B. Weissman and A. S. Grimshaw, "A federated model for scheduling in wide-area systems," in *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 1996, pp. 542–550.

[38] K. Ye, Y. Liu, G. Xu, and C.-Z. Xu, "Fault injection and detection for artificial intelligence applications in container-based clouds," in *International Conference on Cloud Computing*. Springer, 2018, pp. 112–127.

[39] V. Sivagami and K. Easwarakumar, "An improved dynamic fault tolerant management algorithm during VM migration in cloud data center," *Future Generation Computer Systems*, vol. 98, pp. 35–43, 2019.

[40] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine fault tolerance, from theory to reality," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2003, pp. 235–248.

[41] X. Vasilakos, W. Featherstone, N. Uniyal, A. Bravalheri, A. S. Muqaddas, N. Solhjoo, D. Warren, S. Moazzeni, R. Nejabati, and D. Sime-

onidou, "Towards Zero Downtime Edge Application Mobility for Ultra-Low Latency 5G Streaming," in *2020 IEEE Cloud Summit*. IEEE, 2020, pp. 25–32.

[42] N. Samarji and M. Salamah, "A fault tolerance metaheuristic-based scheme for controller placement problem in wireless software-defined networks," *International Journal of Communication Systems*, vol. 34, no. 4, p. e4624, 2021.

[43] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for optimization*. Mit Press, 2019.

[44] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks," *IEEE Transactions on Mobile Computing*, 2020.

[45] J. Moon, J. Kim, Y. Shin, and S. Hwang, "Confidence-aware learning for deep neural networks," in *International Conference on Machine Learning*. PMLR, 2020, pp. 7034–7044.

[46] A. M. Connor and K. Shea, "A comparison of semi-deterministic and stochastic search techniques," in *Evolutionary Design and Manufacture*. Springer, 2000, pp. 287–298.

[47] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," *International Conference on Learning Representations*, 2018.

[48] K. Gilly, S. Alcaraz, N. Aknin, S. Filiposka, and A. Mishev, "Modelling edge computing in urban mobility simulation scenarios," in *2020 IFIP Networking Conference (Networking)*. IEEE, 2020, pp. 539–543.

[49] V. Looga, Z. Ou, Y. Deng, and A. Ylä-Jääski, "Mammoth: A massive-scale emulation platform for internet of things," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, vol. 3. IEEE, 2012, pp. 1235–1239.

[50] A. Haeberlen, P. Kouznetsov, and P. Druschel, "The case for byzantine fault detection." in *HotDep*, 2006.

[51] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, pp. 8026–8037, 2019.

[52] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.

[53] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 12, pp. 3590–3605, 2016.