DRAGON: Decentralized Fault Tolerance in Edge Federations

Shreshth Tuli, Giuliano Casale and Nicholas R. Jennings

***Abstract*—Edge Federation is a new computing paradigm that seamlessly interconnects the resources of multiple edge service providers. A key challenge in such systems is the deployment of latency-critical and AI based resource-intensive applications in constrained devices. To address this challenge, we propose a novel memory-efficient deep learning based model, namely generative optimization networks (GON). Unlike GANs, GONs use a single network to both discriminate input and generate samples, significantly reducing their memory footprint. Leveraging the low memory footprint of GONs, we propose a decentralized fault-tolerance method called DRAGON that runs simulations (as per a digital modeling twin) to quickly predict and optimize the performance of the edge federation. Extensive experiments with real-world edge computing benchmarks on multiple Raspberry-Pi based federated edge configurations show that DRAGON can outperform the baseline methods in fault-detection and QoS metrics. Specifically, the proposed method gives higher F1 scores for fault-detection than the best deep learning (DL) method, while consuming lower memory than the heuristic methods. This allows for improvement in energy consumption, response time and service level agreement violations by up to 74, 63 and 82 percent, respectively.**

***Index Terms*—Edge Computing, Decentralized Management, Fault-Tolerance, Federated Computing, Generative Optimization Networks.**

I. Introduction

FEDERATED computing is a paradigm that brings together the software, infrastructure and platform services from disparate and distributed computing environments. This prevents resource under-utilization and facilitates the accommodation of sudden spikes in demand [2]. In the past few years, this paradigm has been extended to cloud computing, giving rise to cloud federations [3]. However, bringing the computation closer the users, *i.e.*, the edge of the network, has many benefits including lower latency, efficient communication and reduced costs [4]. Bolstered by affordable computation, recently proposed federated edge architectures enable large-scale deployment by leveraging multiple standalone edge computing providers [5]. The cost benefits of this seamless interconnect of several edge devices in a federated setup have pushed the industry from relying on cloud-only and hybrid edge-cloud deployments to edge-only ones [6]. Due to the large number of devices at the edge, this has led to an explosion in the amount of operational log data being generated. As modern applications shift to resource-intensive AI based workloads, mission critical tasks make it crucial to detect and recover from faults arising from malicious attacks, intrusion events or unexpected device breakdowns by mining this log data [7].

Challenges. Modern edge computing systems now support sophisticated Internet of Things (IoT) and AI based workloads that put excessive load on the limited working memory (up to 8GB) of edge devices [8]–[10]. In such systems, it is often possible to handle the processing limitations by effective preemption and prolonged job execution. However, memory bottlenecks are much harder to solve due to the limited performance of virtual memory in network attached storage settings [11]. This problem is further exacerbated by the latency constraints of cloud backends that make them often unsuitable for offloading IoT jobs [12]. Without cloud backends, overloaded edge devices make it hard to deploy deep learning models. All these issues lead to persistent resource contention and faulty behavior adversely affecting system QoS.

Existing solutions. Several fault-tolerance approaches have been proposed for both cloud and edge systems that use heuristics or unsupervised learning strategies [13]–[17]. Compared to supervised learning methods that rely on labeled data, unsupervised methods enable fault detection without human annotations or time-consuming log-trace analyses. In this domain, the best performing solutions are contemporary models that leverage memory-intensive deep generative models such as autoencoders (AE), recurrent neural networks and generative adversarial networks (GANs) [16]–[22]. State-of-the-art fault-tolerance approaches in the cloud use deep learning due to the ability of such models to capture patterns and trends in large amounts of data. On the other hand, the approaches designed for edge systems are dominated by either heuristic based methods [23], [24] or techniques that use powerful cloud devices as management nodes [25], [26]. This is particularly due to the memory limitations at the edge.

The models that have the best detection scores (AEs and GANs) often have a very high memory footprint, 4-6 GB for training and inference in deep learning based methods. Due to the large amount of time-series data generated by edge systems [27], the AEs and GANs used in such methods have extremely high parameter size. Some prior work aims at reducing the on-device parameter size of such models using techniques like model compression [28], neural network slimming [29] or model distribution [30]. However, such approaches usually have accuracy penalties associated with them. This creates a research gap of achieving fault-detection performance comparable with that of state-of-the-art deep

S. Tuli, G. Casale and N. R. Jennings are with the Department of Computing, Imperial College London, United Kingdom.

N. R. Jennings is also with Loughborough University, United Kingdom.
E-mails: {s.tuli20, g.casale}@imperial.ac.uk, n.r.jennings@lboro.ac.uk.

Manuscript received —; revised —.

A preliminary version of the GON framework was presented at the Machine Learning for Systems Workshop at NeurIPS 2021 [1]. The code and datasets are available at `https://github.com/imperial-qore/GON`.

generative models, within memory footprint constraints.

New insights and contributions. To develop a memory-efficient generative model, we propose a novel model called generative optimization networks (GON). We take motivation from GANs that utilize two neural networks, generator and discriminator, where the former generates new data and the latter acts as a critic to train the former [31]. The key insight of our work is that a sufficiently trained discriminator should not only indicate whether an input belongs to a data distribution, but also how to modify the input to make it resemble the target distribution more closely. This can be done using the gradient of the discriminator output with respect to its input [32] and executing gradient optimization of the output score. Having only a single network should give significant memory footprint gains, at the cost of higher training times. This is acceptable in many applications scenarios where initial model training is a one-time activity [33], [34]. The GON model is agnostic to the specific use case and can be utilized in any application where GANs are used. A preliminary version of the GON model was presented at the Machine Learning for Systems Workshop at NeurIPS 2021 [1]. Compared to the earlier work, we present a full framework namely Decentralized Fault-Tolerance using Generative Optimization Networks (DRAGON) that extends GONs using graph neural network and attention operations with also utilizing a simulator for fault-tolerant computing.

The decentralized fashion of fault-detection and remediation in DRAGON has a two-fold benefit. Firstly, there is no single point of failure in the system as the detection and remediation steps are run in each edge provider's local edge infrastructure (LEI). Secondly, it allows the distribution of fault-tolerance related management workload across multiple computing devices, facilitating scalability of the model. Specifically, each LEI runs GON in its broker to detect faults within its edge devices. DRAGON extends the GON model to also consider the system topology and leverages the updated model in a decentralized fashion with a digital-twin like simulator to converge to fault-remediation decisions. All brokers altruistically transfer and resume tasks from faulty nodes to a different device for load-balancing, commonly referred to as *preemptive migration* [35]. The GON model is trained using log traces on DeFog benchmarks [36] and executed on AIoT benchmarks [37] to test efficacy in unseen settings. DRAGON is the first system that uses a GON in edge brokers to predict faults in edge devices of each LEI and proactively runs load-balancing preemptive migration steps to remediate faults. Extensive experiments on a Raspberry-Pi based federated edge cluster show that DRAGON performs best in terms of fault-detection and QoS metrics. To test the generalization of the model, we test on three different federated configurations. Specifically, DRAGON reduces energy consumption, response time and service deadline violations by up to 74%, 63% and 82% compared to state-of-the-art baselines.

Article structure. Section II overviews related work. Section III provides a motivating example for the problem. Section IV outlines the DRAGON methodology for model training and the optimization of the scheduling decisions. A performance evaluation of the proposed method is presented in Section V. Finally, Section VI concludes.

II. Related Work

We now analyze the prior work in fault-tolerant computing. We divide the prior approaches into two categories: heuristic/meta-heuristic methods and deep learning methods.

Heuristic and Meta-Heuristic Methods. Most fault prevention techniques for cloud and edge computing employ some form of heuristics or meta-heuristic approaches. Some techniques like Medusa [13] provide fault-tolerance by creating multiple replicas of each running task across multiple cloud or edge providers allowing resilience against task failures. Other approaches like Threshold-Based Adaptive Fault Tolerance (TBAFT) [14] proactively migrate tasks from overloaded devices. The latter uses an ARIMA model to predict resource utilization metrics of host machines and checks them against preset thresholds to execute remediation steps like task migration and replication. However, in edge computing environments, having node or task level redundancy is inefficient and counterproductive for low-cost deployments [38]. Another method proposes a preference based fault management technique (PBFM) [15] that tries to balance between the QoS improvement and the migration cost using a multi-objective integer linear programming formulation. Federated Distributed MapReduce (FDMR) [3] is another MapReduce based framework that leveraged integer linear programming for fault-tolerant distributed task scheduling. Such methods do not scale for real-time operations, making them unsuitable for mission critical edge applications. Another related work for federated edge-cloud computing environments is the IoTEF [39] framework that leverages task virtualization and rerun strategies to avoid service disruption. All methods in this category are memory efficient and have lower accuracy than contemporary deep learning based methods. We use the best performing methods PBFM, IoTEF and TBAFT as baselines in our experiments, as demonstrated in prior work [14], [15], [39].

Deep Learning Methods. Recently, many unsupervised methods have started utilizing deep learning techniques for fault prevention and fault recovery. For instance, the Adaptive Weighted Gath-Geva (AWGG) [40] clustering method is a reconstruction model that detects faults using stacked sparse autoencoders to reduce detection times. Another recent class of methods applies neural networks to reconstruct the last state of the system [17], [27]. The reconstruction error has been used as an indicator of the likelihood of the current state being anomalous. For instance, TopoMAD [17] 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, which limits them to use reactive fault recovery policies. Other methods use slight variations of LSTM networks with either dropout layers [20], causal Bayesian networks [16] or recurrent autoencoders [21]. A GAN based approach that uses stepwise training process, StepGAN [22], 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) [41] or Kernel Density Estimation

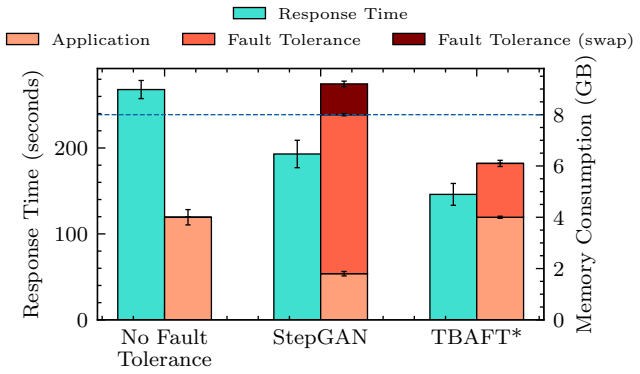


Fig. 1. Motivation. Experiments performed on four node edge cluster each with 8GB RAM: (left) No fault tolerance model is run giving high response time and only the application using memory. (middle) StepGAN fault tolerance model that improves response time, but leads to memory contention due to heavy deep neural network. (right) TBAFT heuristic model with precomputed fault labels from StepGAN to obtain memory consumption of TBAFT as a proxy.

(KDE) [42]. 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 with higher accuracy than heuristic based approaches, deep learning models such as deep autoencoders, GANs and recurrent networks are adaptive and accurate, but have high memory footprint that adversely affect system performance. From this category, we test the above mentioned approaches on the testbed described in Section V and use the empirically best techniques as baselines: AWGG, TopoMAD and StepGAN.

DRAGON is another deep learning based method that aims to resolve the challenge of high resource-utilization of other methods in the same class.

III. Motivating Example

As demonstrated by prior work [17], [27], [40], when increasing the number of edge devices and the amount of operational data, the fault-detection models tend to be more sophisticated. This entails increase in the parameter size of neural networks, subsequently causing higher memory consumption. This load on the limited working memory of edge devices also affects the functioning of the applications running on edge systems. To demonstrate this, we run simple experiments on four Raspberry Pi 4B nodes using the COSCO framework [32] and DeFog benchmark applications [36]. Here, the each node has 8GB RAM, acts as its own broker and runs our fault tolerance model in our setup. Fig. 1 shows the results corresponding to three different cases, presenting average response time in blue and memory consumption in red. The first case is when we do not run any fault tolerance model. All the memory of edge nodes is utilized by the running applications in this case. In the second case we run the StepGAN fault-tolerance model, which has a large memory overhead of 7.4GB. Even though it brings down the average response time by 28%, the model consumes 1.2GB of slow virtual memory (swap space), limiting its optimization capability. In the third case, we run the TBAFT approach, but use precomputed fault labels from the StepGAN model (denoted by TBAFT*). This shows the performance in a case when we have accuracy of the deep learning model with memory-efficiency of a heuristic approach. In this case, as no swap space is utilized, the average response time is much lower, giving 24% improvement compared to the StepGAN case. This example highlights the potential for an approach that is as memory-efficient as heuristic approaches, but that does not compromise on performance, matching fault detection scores with those of state-of-the-art deep learning techniques.

IV. DRAGON Model

A. Environment Assumptions and Problem Formulation

System Model. We assume a federated edge computing environment with heterogeneous nodes in a master-slave fashion as summarized in Fig. 2. Although our methods are generalizable to any federated setup with H hosts, in the rest of the discussion we consider three federated configurations with 16 edge nodes:

- *Configuration 1:* 16 LEI groups, each with a single edge device duplicates as a broker and edge worker. All broker nodes are connected to a Wide Area Network (WAN).
- *Configuration 2:* 4 LEI groups, each with its own broker node and 3 worker nodes. All nodes in an LEI are connected with each other in a Local Area Network (LAN).
- *Configuration 3:* 4 LEI groups with 2, 4, 4 and 8 nodes. Each LEI has its own broker node.

These configurations have been chosen to cover a wide range of heterogeneous federated setups. Configuration 1 and 2 are balanced, whereas configurations 3 is unbalanced in terms of number of nodes in an LEI.

As in traditional federated environments, we assume that all LEI groups are seamlessly connected with each other, allowing information and task sharing across groups. Further, we consider that any execution in this environment lasts for a bounded time. We divide the execution timeline 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 (e.g., 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 [43]).

Fault Model. We consider different type faults in our system that frequently occur in edge workers while executing data-intensive applications, ranging from breakdowns, malicious attacks and intrusions. We use an existing fault injection module [44] to create different fault types like CPU overload, RAM contention, Disk attack and DDOS attack. The faults 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 [45]. If a broker breaks down, we allocate the worker with least resource utilization as the broker of that group. As in prior work, edge hosts in the same LEI are connected to the same power supply and unrecoverable faults like outages are ignored [25], [38], [46]. We prevent

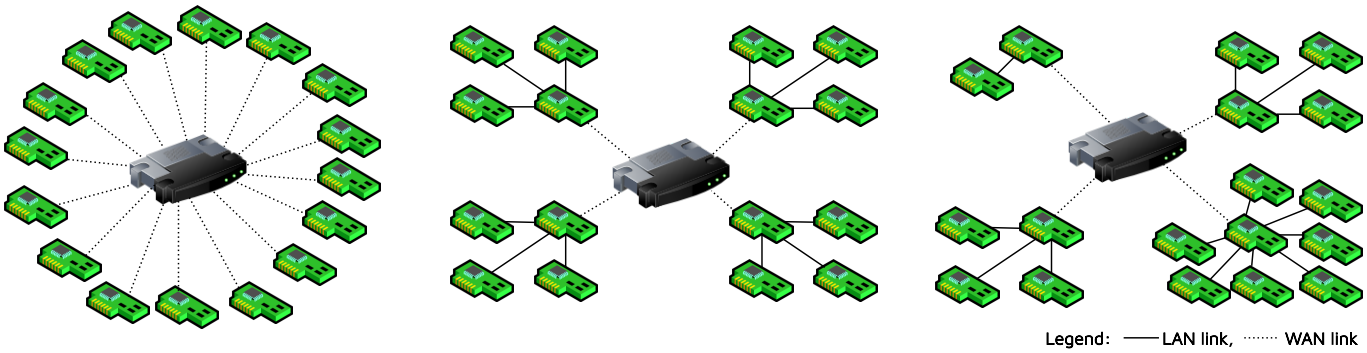


Fig. 2. Edge Federation Model with 16 edge nodes. We consider a master-slave architecture of the network where local edge infrastructure (LEI) has a broker for management. All LEIs share resources and interact with each other. The broker of each LEI is connected to the federation using a WAN link to a public-cloud router. (left) 16 homogeneous LEIs with each having only a broker node that also doubles as a worker. (middle) 4 homogeneous LEIs with each broker controlling 4 nodes. (right) 4 heterogeneous LEIs with each broker controlling 2, 4, 4 and 8 workers.

over-utilization faults by preemptively migrating tasks from faulty LEIs.

Workload Model. We assume a bag-of-tasks workload model, where a set of independent tasks enter each LEI at the start of each scheduling interval. These are generated from the users and are transferred to the edge broker via gateway devices or IoT sensors. Each task has an associated Service Level Objective (SLO) deadline.

Formulation. In this work, we assume that the edge brokers run scheduling algorithms that place each incoming task onto one of the node in the LEI. Apart from this, the brokers also decide tasks that need to be migrated to another LEI as part of fault management decisions. Formally, at the start of the interval I_t, the edge broker makes a decision denoted S_t, that is a mapping of new tasks in the interval I_t as well as the active tasks from interval I_{t-1} to its edge hosts. The edge broker also makes a decision M_t, that migrates active tasks to another LEI. All tasks being migrated to an LEI are considered as incoming tasks in that interval. Unlike new tasks, migratory tasks may resume from a saved state. We also encode the undirected topology graph of the federated edge environment as G_t.

At the beginning of interval I_t, for an input time-series of LEI system states $\{x_0, \ldots, x_t\}$, each broker needs to predict the next system state, *i.e.*, x_{t+1}. Here, the system state (x_t) consists of values for an arbitrary set of resource utilization metrics for all active tasks and hosts at the start of I_t. Instead of directly using the system states, we use a sliding window of length k to capture the temporal contextual trends:

$$W_t = \{x_{t-k+1}, \ldots, x_t\}. \tag{1}$$

We use replication padding [47] for the first k intervals, where we replicate the first system state to ensure the time-series window is always of the same size. Also, to ensure robust predictions in the DRAGON model, we normalize these time-series windows using min-max scaling.

Using the input window W_t, graph topology G_t and an input scheduling decision S_t, the model needs to predict whether there is likely to be a fault in the next system state, *i.e.*, x_{t+1}. For S_t we assume that there is an underlying scheduler in the system independent from the proposed fault-tolerance solution. In lieu of directly predicting the fault label (denoted as y_t), we predict a fault score f_t using a predicted reconstruction of the next window W_{t+1}, denoted as $\hat{W}_{t+1}$. The fault score f_t is obtained by calculating the deviation between the true window W_{t+1} and its reconstruction $\hat{W}_{t+1}$. 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, W, S, $\hat{W}$, and f.

B. Generative Optimization Networks

We now describe the GON framework for generic data generation and later elucidate its function in fault-detection. In GON, we have a single discriminator model $D(\cdot;\theta)$ that is a differentiable multilayer perceptron with parameters θ. For any input set of G, W and S, the output $D(G, W, S; \theta)$ is a scalar. We denote a generated distribution with p_g, and with $D(G, W, S; \theta)$ the probability that W belongs to the data p_{data} instead of p_g. We train D such that it correctly labels samples from data and p_g. A working representation is shown in Fig. 3. The key idea is divided into three parts.

Training using real samples. For any sample W from the data, to make sure the output of D is maximized, we train it to minimize the cross-entropy loss by descending the stochastic gradient

$$-\nabla_\theta \log\big(D(G, W, S; \theta)\big). \tag{2}$$

Generating new samples. Starting from a random noise sample Z, we aim to increase its probability of coming from the data. To do this, we use D as a surrogate model and maximize the objective $\log\big(D(G, Z, S)\big)$. To do this, starting from a random noise sample, a stochastic gradient ascent based solution executes the following until convergence

$$Z \leftarrow Z + \gamma \nabla_Z \log\big(D(G, Z, S; \theta)\big). \tag{3}$$

Here, γ is a step parameter. The optimization loop converges to Z^* that should have a high $D(G, Z^*, S; \theta)$ probability and represents a reconstruction of the next window. The working intuition is that the surrogate surface is highly non-convex and optimization over diverse noise samples would lead to distinct local optima. In principle, every optimum corresponds to a

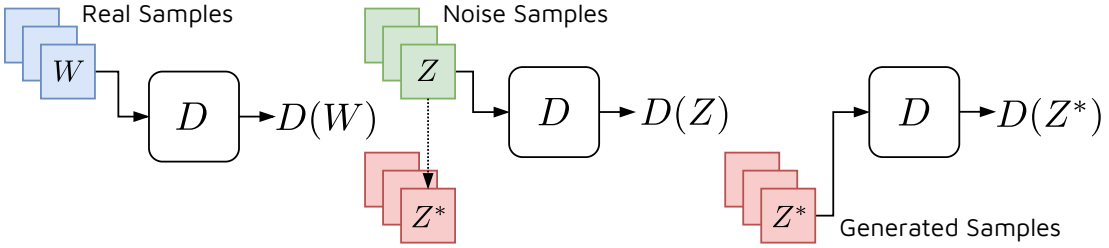


Fig. 3. The GON framework. (left) training using real samples from the data. (middle) generating new samples by gradient optimization starting from random noise samples. (right) training using generated samples.

unique element from the data distribution, allowing the GON model to generate a diverse set of samples. Alternatively, we could stop after each step with probability $D(G, Z, S; \theta)$. A caveat of this approach is that sample generation could be slow as we run optimization in the input space. For some applications this additional overhead might be negligible or acceptable [33], [34].

Training using generated samples. We consider the new samples as self-supervised fake ones [48] not belonging to the data and train D using the cross-entropy loss by descending the gradient

$$-\nabla_\theta \log \big(1 - D(G, Z^*, S; \theta)\big). \tag{4}$$

Informally, as D becomes a better discriminator, minimizing (2) and (4), the generated samples should also be close to the data.

C. Optimizing GONs for fault-detection

The above described GON framework is for generic data generation. However, for the specific case of anomaly/fault detection, we leverage some of its properties to optimize GONs. Consider a GON model that takes inputs the graph topology G, time-series window W and scheduling decision S for each LEI. The output of such a GON needs to be a reconstruction of the next window

$$\hat{W} = D(G, W, S; \theta).$$

A bottleneck of the vanilla framework is that the generation of new samples is slow [1]. To bring down the time to generate new samples, we develop the following optimization.

Window Initialization. Firstly, exploiting the temporal trend of time-series data, instead of starting from a random noise sample Z we can start from the window W and optimize $D(G, Z, S; \theta)$. This is because of the high correlation in the time-series windows of the current and next scheduling intervals in typical edge environments. Thus, we initialize

$$Z = W.$$

Second-Order Optimization. Secondly, second-order optimization methods have been shown to be much faster for diverse optimization setups [49]. So, instead of using a first-order gradient optimization in (3), we can use higher-order variants. Thus, we could use

$$Z \leftarrow Z + \gamma [\nabla_Z^2 \log \big(D(G, Z, S; \theta)\big)]^{-1} \nabla_Z \log \big(D(G, Z, S; \theta)\big). \tag{5}$$

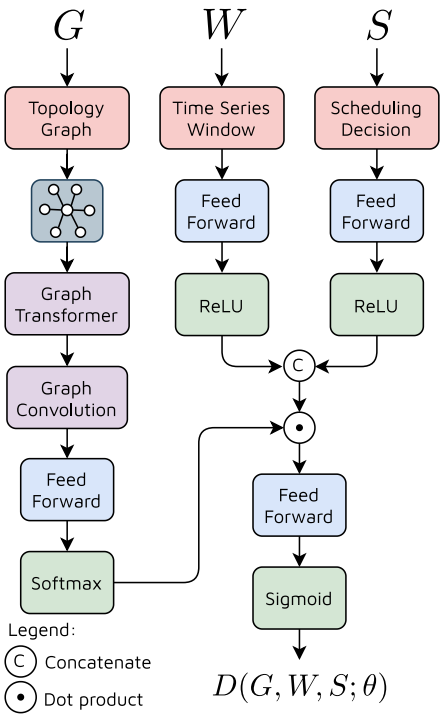


Fig. 4. Neural network used in DRAGON. The three inputs to the model are shown in red. Feed-forward, graph operations and activations are shown in blue, purple and green.

However, computing the inverse of the second-order gradient (Hessian) is expensive [50, § 6.3]. This is alleviated by adapting the Hutchinson's method and using the moving average of the approximate Hessian diagonal to speed up convergence. This is done by using the `AdaHessian` optimizer [49]. We also use warm restarts using cosine annealing [51] to speed up convergence.

GON Network. It is crucial that the D network is able to capture the temporal trends in the time-series data with the scheduling decision to effectively predict a reconstruction of the next time-series window. The model used in our approach is a composite neural network shown in Fig. 4. 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 m (number of hosts in the LEI). We thus get a matrix of scheduling decisions (S) of size $[p \times m]$. Also, the n features of m hosts and p tasks in the form of resource utilization metrics, each state window (W) is encoded as a $[(m+p) \times n \times k]$ tensor. All operations are performed in a factored fashion, *i.e.*, the

neural network operates on S as a batch of p vectors, each having dimension m, and on W as a batch of size $m+p$ tensors of size $n \times k$. To encode inputs W and S, we use feed-forward layers to bring down the dimension size of the input and ReLU activation:

$$\begin{aligned} E^W &= \text{ReLU}(\text{FeedForward}(W)), \\ E^S &= \text{ReLU}(\text{FeedForward}(S)). \end{aligned} \tag{6}$$

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 each 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 h_i as e_i. We encode this graph G using a graph transformer network [52] for scalable computation over the input graph. The motivation behind using a graph transformer network is to allow computation across cross-LEI paths that are defined as paths from a worker in one LEI to another worker in a different LEI. This is done by modifying the input graph by graph-transformer operations that generate a transformed graph with cross-LEI paths as edges. As we run preemptive migrations across LEIs, this facilities prediction of effects on QoS and resource utilization metrics in the next interval when migrating task along the cross-LEI paths. Thus, we transform the input graph G as

$$G_1 = \text{GraphTransformer}(G). \tag{7}$$

We then pass the graph through a graph convolution network [53] to capture the inter host dependencies by convolving feature vectors across the cross-LEI paths in G, *i.e.*, the edges in G_1. Here, the features for host h_i are aggregated over one-step connected neighbors $n(i)$ in the graph over r convolutions, resulting in an embedding e_i^r for each host node in the graph. Here, r denotes the number of convolution operations and is typically kept less than the number of LEIs. Specifically, this results in *graph-to-graph* updates as:

$$\begin{aligned} e_i^0 &= \tanh(W\ e_i + b), \\ e_i^q &= \sum_{j \in n(i)} W^q e_j^{q-1}, \end{aligned} \tag{8}$$

where W^q are the parameters of the graph convolution block in Figure 4. The stacked representation for all hosts is represented as E^H. Now, we pass this representation through a feed-forward layer with softmax activation that allows us to use this as attention weights

$$E_1^H = \text{Softmax}(\text{FeedForward}(E^H)). \tag{9}$$

To generate the final window reconstruction, we use dot-product attention [54] as

$$D(G, W, S; \theta) = \text{Sigmoid}(\text{FeedForward}(E_1^H \cdot [E^W, E^S])). \tag{10}$$

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

Algorithm 1 Minibatch stochastic gradient based training of generative optimization networks. Input is dataset Λ and hyperparameters m and γ. In experiments we use `AdaHessian` optimizer for line 4 and `Adam` for line 5.

Require: Dataset $\Lambda = \{W_0, \ldots, W_T\}$

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

2: Sample minibatch of size m window samples $\{W^{(1)}, \ldots, W^{(m)}\}$ from Λ.

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

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

$$Z \leftarrow Z + \gamma [\nabla_Z^2 \log\big(D(G, Z, S; \theta)\big)]^{-1} \cdot \nabla_Z \log\big(D(G, Z, S; \theta)\big).$$

5: Update the discriminator $D(\cdot; \theta)$ by ascending the stochastic gradient.

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

D. Offline Model Training

A more pragmatic implementation of GON training is summarized in Alg. 1, where we combine the first and last steps in Section IV-B when training D using minibatches. The training procedure runs offline, *i.e.*, before executing any workloads on the federated edge setup, training the neural network in a GPU equipped machine. The input to the training framework is a dataset of time-series windows $\Lambda = \{W_0, \ldots, W_T\}$. We first randomly sample m window samples from Λ (Z^*s in line 2). We also initialize noise samples as the window samples (line 3). We then perform `AdaHessian` optimization on the noise samples to obtain generated windows (line 4). Finally, we train the model using a combined cross-entropy loss of generated ($Z^{*(i)}$) and data samples ($W^{(i)}$), as described in Section IV-B (line 5):

$$L = \log\big(D(G, W^{(i)}, S; \theta)\big) + \log\big(1 - D(G, Z^{*(i)}, S)\big). \tag{11}$$

E. Decentralized Fault Tolerance using GON

The working of the DRAGON framework is summarized in Alg. 2. The algorithm runs in each broker node of the federated edge setup where the neural network is periodically fine-tuned to adapt to changing environment and workload characteristics. We now elucidate the different steps in detail.

Fault Detection. For unsupervised fault detection using the GON framework, for each input datum from a multivariate time-series, we create a reconstruction using a neural network D. Now, for an input window W_t, we create a reconstruction of the next window $\hat{W}_{t+1}$, using a trained model D and the `AdaHessian` based version of (5) (line 5). However, to generate an approximation of the next window, we leverage co-simulation [32] that enables a model to quickly run discrete-event simulations alongside and synchronously with the running computer system. Similar to digital-twins, this allows us to continuously capture the system state and get an estimate of the utilization metrics of a future state of the system, further

Algorithm 2 The DRAGON fault-tolerance algorithm

Require:
Trained GON Model $D(\cdot;\theta)$;
Set of hosts Ω

1: **for** $t = 0$ to T **do**
2: **for each** broker **do**
3: S_t from underlying scheduler
4: $W_{t+1} \leftarrow Sim(W_t, S_t)$
5: $\hat{W}_{t+1} \leftarrow \texttt{AdaHessian}(D(G_t, W_t, S_t; \theta))$
6: $f = \|\mathrm{ReLU}(W_{t+1} - \hat{W}_{t+1})\|$
7: $l = \mathbb{1}(f \geq POT(W_t))$
8: Compile time-series windows of all brokers as W_t^M
9: Faulty hosts $\omega \subseteq \Omega$ that have $l = 1$
10: Randomly initialize migrations from ω as M_t
11: $M_t^* \leftarrow \mathrm{SimulatedAnnealing}(M_t, \text{Fitness = (18)})$
12: **for each** allocation (t, h) in M_t^* **do**
13: **if** t in LEI and h not in LEI **do**
14: Migrate t to LEI corresponding to h
15: $L = \log D(G, W_{t+1}, S; \theta) + \log(1 - D(G, \hat{W}_{t+1}, S))$
16: Fine-tune D using L and `Adam`

allowing proactive fault remediation. Having the scheduling decision S_t (line 3) and the current window W_t, we denote the discrete-event simulation as (line 4)

$$W_{t+1} \leftarrow \mathrm{Sim}(W_t, S_t). \tag{12}$$

To generate the fault score we only consider upward spikes of the true window from the reconstructed window. This is due to the nature of our state data, *i.e*, resource utilization metrics leading to faults only when there is a sudden increase in CPU/RAM/disk/network consumption. The ReLU activation is apt for this as it gives a zero fault score when the true utilization metrics are lower than the predicted ones. Thus, as in line 6

$$f = \|\mathrm{ReLU}(W_{t+1} - \hat{W}_{t+1})\|. \tag{13}$$

Once we have the fault scores, we generate fault labels using the Peak Over Threshold (POT) method [41]. POT chooses the threshold automatically and dynamically, above which an fault score is classified as a true class label. Thus, our fault label (line 7) is calculated as

$$l = \mathbb{1}(f \geq POT(W_t)). \tag{14}$$

Comparing the resource utilization metrics of each host separately, the edge broker generates a fault-label for each edge device in its LEI.

Co-Simulated Fictitious Play. Now that we predict the edge hosts that could possible have fault in the next interval, we now need to decide the tasks that should be preemptively migrated from these nodes. To do this, we again use the co-simulation technique. A co-simulator can also give us the QoS score of the next state for resource utilization metrics of all nodes instead of the LEI, denoted by W_t^M (line 8), and a preemptive migration decision M_t. We denote the co-simulated QoS score and the next window by

$$QoS_{t+1} \leftarrow \mathrm{Sim}^{\mathrm{QoS}}(W_t^M, M_t), \tag{15}$$

$$W_{t+1}^M \leftarrow \mathrm{Sim}(W_t^M, M_t). \tag{16}$$

This score could be a convex combination of multiple metrics like SLO violation rate, energy consumption, response time, etc. However, for this work, we only use response time and energy consumption as these are the most important metrics in edge federations [3], [5], [11], [32]. Thus,

$$\mathrm{QoS}_{t+1} = \alpha \cdot ART_{t+1} - \beta \cdot AEC_{t+1}. \tag{17}$$

Here, ART_{t+1} is the average response time for all tasks leaving the system in interval I_{t+1} normalized by the maximum response time until the current interval. AEC_{t+1} is the average energy consumption of the infrastructure (which includes all edge hosts) in interval I_{t+1} normalized by the maximum power of the hosts.[1] Also, α and β are hyperparameters that can be tuned by the user based on the application requirements. Also, using W_{t+1}^M we detect the number of anomalies in I_{t+1}. We denote this by N_{t+1} and use the normalized value N_{t+1}/H, where H is the total number of hosts in the federated environment (in our case $H = 16$). To keep into account the costly migration overhead, we also include the migration overhead score O_t that is the migration time of all tasks in M_t normalized by the maximum migration time until the current interval. We define a fitness score as

$$\text{Fitness}_{t+1}(W_t^M, M_t) = QoS_{t+1} + \delta_1 \cdot \frac{N_{t+1}}{H} + \delta_2 \cdot O_t, \tag{18}$$

where δ_1 and δ_2 are hyperparamaters set based on user requirements. The motivation behind this fitness score is that when we optimize a preemptive migration decision, we need to make sure that we consider the immediate reward in terms of the QoS score, but also consider the migration overheads and number of faults in the next interval. This entails the minimization of the fitness score given in (18).

Thus, at the start of every interval, each broker node gets the time-series window of other LEIs, combines them to form W_t^M and runs a simulated execution to get the next window and QoS scores. This kind of distributed execution in a multi-agent setup to estimate objective function of a global environment is commonly referred to as "fictitious play" [55].

Preemptive Migration Decision. The co-simulated fictitious-play described earlier is a very fast and lightweight operation compared to an actual execution of a preemptive migration decision. In fact, running a simulation for W_t^M and M_T is so fast that testing for all possible M_t takes $\sim$0.5 seconds second for 16 nodes. This increases linearly, by around 5% for each new node in the system. Still, for scalability, we run simulated annealing on M_t to optimize the fitness score (line 11). A neighbor of a decision M_t is defined by either migrating a task not in M_t to another LEI or migrating a task in M_t to a different LEI than in M_t. Thus, optimizing fitness score from a preemptive migration decision, which randomly selects tasks to be migrated from faulty hosts detected by (14) (line 10), converges to the decision M_t^*. This is the output of the DRAGON model. Note that each broker generates such a decision and migrates the set of tasks in M_t^* that are active in its LEI to the target LEI group (lines 12-14). All brokers altruistically accept

[1]Both AEC and ART are unit-less metrics and belong in the range $[0, 1]$. See [32, § 5.1] for more details.

incoming tasks from other LEIs and consider them new tasks in their group. The fault labels are also combined using logical or, for each interval from t to $t+N-1$ to generate the fault label for DRAGON.

Extending to DRAGON+. As DRAGON just optimizes the fitness score for the next interval, it is myopic in its optimization capacity. To tackle this, we run multiple steps of DRAGON for a long-term optimization strategy and call this DRAGON+. Say in the first iteration DRAGON generated a decision M_t^*. Now, having W_{t+1}^M, we can generate S_{t+1} from the underlying scheduler and run DRAGON again to generate M_{t+1}^*. We continue this for N intervals and combine $\{M_t^*, \ldots, M_{t+N-1}^*\}$. To do this, if task t is allocated to h_i in M_p^* and h_j in M_q^*, then we select $h_{\max(p,q)}$. This is done because a migration to h_i then later to h_j has higher overhead that direct migration to h_j. However, the level of performance uplift depends on the volatility of the workloads and the look-ahead interval count N (see Section V-C).

Model Fine-tuning. Now that we have a true window W_{t+1} and a generated window $\hat{W}_t$, we can fine-tune the model with this new datapoint using the loss

$$L = \log\big(D(G, W_{t+1}, S; \theta)\big) + \log\big(1 - D(G, \hat{W}_{t+1}, S)\big). \tag{19}$$

This allows the model to dynamically adapt in non-stationary environments (lines 15-16).

To summarize, the novel contribution of this work is two-fold. First, we develop a decentralized fault-detection model based on the GON framework that uses significantly lower number of parameters than previous generative models to detect faults in each LEI independently. Second, we develop a synchronization strategy using co-simulated fictitious play to jointly decide of preemptive migrations that aim at avoiding faults arising in the system. Thus, we run migrations proactively, *i.e.*, predict faults using GONs and simulations, to avoid faulty states. For crash-like states, we restart tasks.

V. Experiments

We compare the DRAGON and DRAGON+ methods against three heuristic baselines: PBFM [15], IoTEF [39], TBAFT [14], and three DL baselines: AWGG [40], TopoMAD [17] and StepGAN [22] (more details in Section II). As TopoMAD and AWGG are only fault-detection methods, we supplement them with the multi-objective integer linear programming based preemptive migration scheme for fault-recovery from another baseline, PBFM. 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 [56] library.

A. Evaluation Setup

Our evaluation setup consists of 16 Raspberry Pi 4B nodes, 8 with 4GB RAM and another 8 with 8GB RAM each. This allows the setup to have heterogeneous nodes with different memory capacities. We consider the three federated configurations presented in Section IV-A. In configuration 1, we have 8 brokers with 4GB and other 8 with 8GB RAM. In configuration 2, one LEI has only 4GB nodes, second LEI has only 8GB nodes, the other two have two 4GB and two 8GB nodes with one with 4GB node as broker and other with 8GB node as its broker. In configuration 3, the LEI with 8 nodes has all 8GB nodes. These configurations and distribution of nodes was kept to maximize heterogeneity and consider diverse federated setups. 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 node using the model described in [57].

To generate the tasks in our system, we use the *AIoTBench* applications [37]. AIoTBench is a 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.[2] This benchmark has been used in our experiments due to its volatile utilization characteristics and heterogeneous resource requirements. The benchmark consists of 50,000 images from the COCO dataset as workloads [58]. To evaluate the proposed method in a controlled environment, we abstract out the users and IoT layers described in Section IV-A and use a discrete probability distribution to realize tasks as container instances. Thus, at the start of each scheduling interval, we create new tasks from a Poisson distribution with rate $\lambda = 1.2$, sampled uniformly from the three applications. The Poisson distribution is a natural choice for a bag-of-tasks workload model, common in edge environments [59], [60]. Our tasks are executed using Docker containers. We run all experiments for 100 scheduling intervals, with each interval being 300 seconds 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

Fault Detection. To evaluate fault-detection efficacy, we utilize commonly used metrics including accuracy, precision, recall, area under the receiver operating characteristic curve (AUROC) and F1 score. We also consider the ratio of F1 score with the memory consumption of the model (F1/GB) to get an estimate of the detection performance normalized by the memory footprint.

Fault Diagnosis. For fault-diagnosis (*i.e.*, the prediction of the specific hosts that have faults), we use the prediction of fault scores and labels for each host independently. We use two popular metrics for comparison: (1) HitRate@100% is the measure of how many ground truth dimensions have been included in the top candidates predicted by the model [61], (2) Normalized Discounted Cumulative Gain NDCG@100%) [62].

QoS. For QoS, we measure the energy consumption of the federated setup, average response time and SLO violation

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

TABLE I
DATASET STATISTICS

Dataset	Train	Test	Dimensions	Anomalies (%)
FTSAD-1	600	5000	1	12.88
FTSAD-25	574	1700	25	32.23
FTSAD-55	2158	2264	55	13.69

rate of completed tasks, Jain's fairness index. We consider the relative definition of SLO (as in [32]) 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 number of task migrations, average time per migration. We also compare overheads in terms of scheduling time and memory consumption of all models.

C. Implementation and Training Details

Extending COSCO. To implement and evaluate the proposed methods, we need a framework that we can use to deploy containerized workloads on a federated edge computing environment. One such framework is COSCO [32]. It enables the development and deployment of integrated edge-cloud environments with structured communication and platform independent execution of applications. The resource management and task initiation is undertaken on edge nodes in the broker layer. The framework uses HTTP RESTful APIs for communication and seamlessly integrates a `Flask` based web-environment to deploy and manage containers in a distributed setup [63].

It uses the Checkpoint/Restore In Userspace (CRIU) [64] tool for container migration. All sharing of resource utilization characteristics across brokers uses the `rsync` utility. We extend the `Framework` class to allow decentralized decision making. For synchronization of outputs and execution of workloads, we utilize the HTTP Notification API.

Fog Time Series Anomaly Detection (FTSAD) datasets. The FTSAD is a suite of datasets collected on a Raspberry Pi 4B compute cluster [1]. The cluster consists of five 4GB versions and five 8GB versions. We ran the DeFog benchmark applications [36], specifically the Yolo, PocketSphinx and Aeneas workloads. We use a random scheduler and execute these tasks as Docker containers. We create faults of the type: CPU overload, RAM contention, Disk attack and DDOS attack [44]. In a CPU overload attack, a simple CPU hogging application is executed that create contention of the compute resources. In RAM contention attack, a program is run that performs continuous memory read/write operations. In disk attack, we run the IOZone benchmark that consumes a large portion of the disk bandwidth. In DDOS attack, we perform several invalid HTTP server connection requests causing network bandwidth contention. We generate faults using a Poisson distribution with rate $\lambda_f = 5$, sampled uniformly at random from the attack set. More details are given in [44].

For the 10-node cluster, we collect traces of resource utilizations including CPU, RAM, disk and network. Each datapoint is collected at the interval of 10 seconds. We truncate

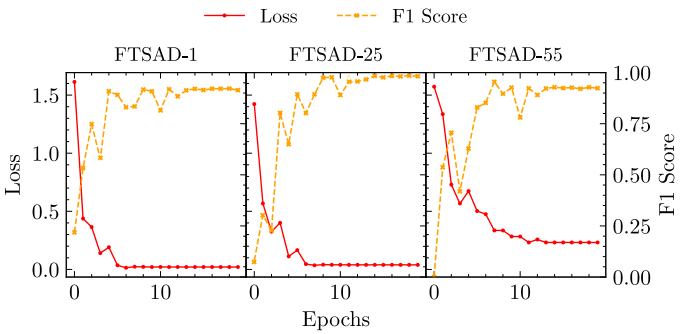


Fig. 5. Training the GON model.

the dataset to have only 1, 25 or 55 dimensions and call these FTSAD-1/25/55 datasets. For FTSAD-1, we use the CPU utilization of one of the cluster nodes. For FTSAD-25, we use the CPU and network read bandwidth utilization traces for all 10 nodes and RAM utilization for the five 4GB nodes (as 8GB nodes rarely have memory contentions). For FTSAD-55, we use the CPU utilization, disk read, disk write, network read, network write bandwidths for all nodes and RAM utilization of the five 4GB nodes. The overall traces are collected for 737 minutes, giving 4422 datapoints. For the FTSAD-55 dataset, we split them at the 2158-*th* interval at which we started saving the anomaly labels as well, giving a test set size of 2264 datapoints. The FTSAD-1/25 datasets were manually curated to have low anomaly frequency in FTSAD-1 and high frequency in FTSAD-25. This allowed us to create a collection of datasets with diverse fault characteristics and dataset sizes. We summarize the characteristics of the three datasets in Table I

Model Training. We train the GON network using Alg. 1 on the three FTSAD datasets. All model training is performed on a separate server with this configuration: Intel i7 10700k CPU, 16GB RAM, Nvidia RTX 3060 and Windows 11 OS. As the model is agnostic to the dimension size of the multivariate time series, our model can be trained for any of these datasets without changing the network size. For our experiments, we use the model trained on FTSAD-55 as it covers the highest number of parameters. The time-series windows and the scheduling decisions were taken directly from the datasets, whereas the graph topology was taken from the three configurations. For each configuration, we train a separate model. For training we randomly split the dataset into 80% training and 20% testing data. We use learning rate of 10^{-4} with weight decay of 10^{-5}. We use `Adam` optimizer for optimizing the loss function. The training curves showing loss and F1 scores on the test set are shown in Fig. 5. We use early stopping criterion for convergence.

Hyperparameter Selection. All baseline models have been developed by us for the COSCO framework. For fault-labeling, we use the following POT parameters, $\text{coefficient} = 10^{-4}$ for all data sets, low quantile is 0.07. These are selected as per [61]. No baseline method provides the flexibility of changing the memory footprint of the trained model. In heuristic models, we use the same parameters as mentioned in

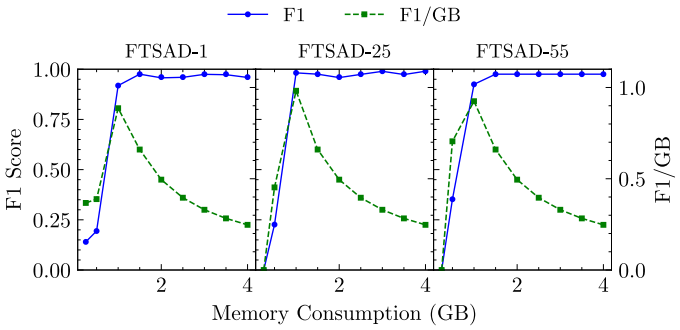


Fig. 6. F1 and F1/GB of DRAGON with memory consumption on the FTSAD-1/25/55 datasets.

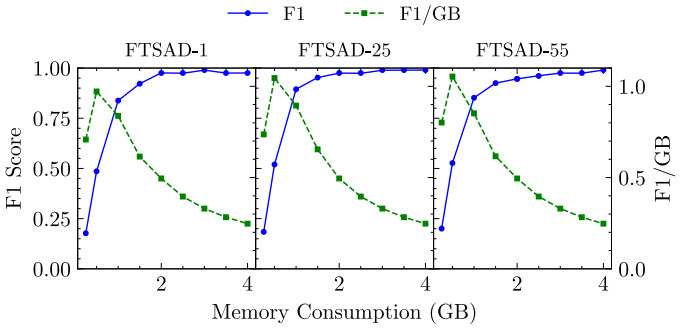


Fig. 7. F1 and F1/GB of DRAGON+ with memory consumption on the FTSAD-1/25/55 datasets.

their respective papers. However, for AWGG, TopoMAD and StepGAN we can also control the number of parameters by employing model compression or GAN slimming [28], [29]. For GON, we only change the number of layers in the feed-forward networks in Fig. 4 (keeping a fixed layer size of 128 nodes). We choose the model memory footprint based on grid search so to maximize F1/GB. The motivation behind this metric is as follows. Increasing memory footprint leads to a monotonic increase in the anomaly detection performance (see Fig. 6 for DRAGON and Fig. 7 for DRAGON+). However, when running online fault detection on edge devices, several other applications may be running alongside. Having the maximum possible F1 score would impact negatively the performance of the running applications due to memory contentions. However, having a low memory footprint model but with poor F1 score would be ineffective in detecting anomalies for apt remediation. Thus, a careful balance needs to be maintained to prevent memory contentions and provide high detection. This can be seen by Fig. 6 that shows the F1 and F1/GB scores of DRAGON model on the FTSAD datasets. For the models with memory consumption of $\sim$1GB from DRAGON and $\sim$0.5GB for DRAGON+, the F1/GB is maximum and hence have been used in our experiments. Similarly, we determine the compression in AWGG, TopoMAD and the slimming factor in StepGAN.

For next window generation we use `AdaHessian` optimizer with `CosineAnnealing` learning rate decay with warm restarts every 10 epochs. We use learning rate (γ in Alg. 1) of 10^{-3} and window size ($K = 10$ in (1) for DRAGON and $K = 5$ for DRAGON+). These values were selected using grid-search to optimize F1 score. Other parameters chosen using grid search include: minibatch size m as 10 to optimize F1 score, $\alpha = \beta = 0.5$ in (15) as per prior work [32], [60], $\delta_1 = \delta_2 = 0.5$ in (18) and number of iterations $N = 5$ in DRAGON+ to minimize SLO violation rates.

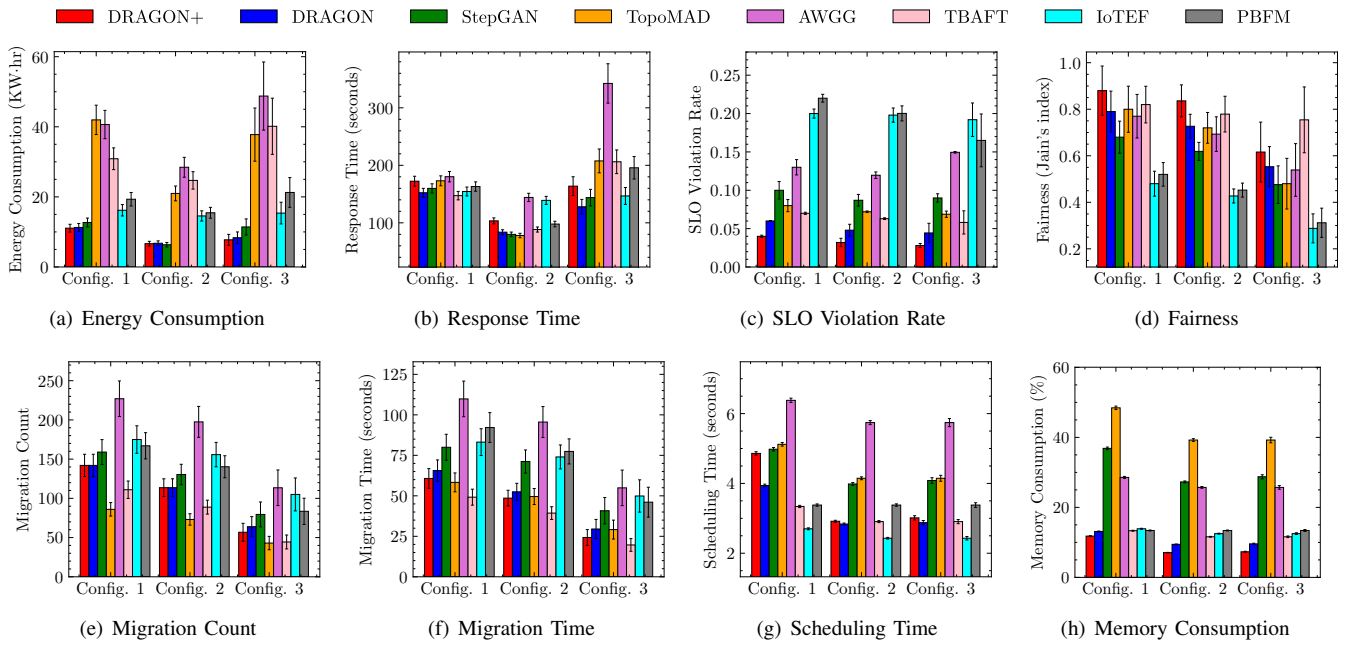
D. Results

Comparison of Fault-Detection Performance. The detection and diagnosis scores for the FTSAD datasets are presented in Table II. We do not present results on AIoTBench as it does not have fault labels. We use the *point-adjust* technique to determine whether a fault prediction is true or false. Specifically, faulty observations usually occur in the form of contiguous anomaly segments; thus, if at least one observation of an faulty time-series window is correctly detected, all the other observations of the segment are also considered as correct. The approach was first presented in [27]. Compared to the baselines, the DRAGON and DRAGON+ models improve upon the accuracy, precision and recall by up to 17.28%-11.87%-14.14% and 18.32%-12.50%-14.34% respectively. Among the baselines, the StepGAN baseline has the highest average F1 score of 0.9040. The DRAGON and DRAGON+ models gives a higher average F1 scores of 0.9418 and 0.9453, *i.e.*, 4.18% and 4.56% higher than StepGAN respectively. In terms of AUROC, the TopoMAD baseline has the highest value of 0.9416. The proposed models outperform this score by 3.56%-4.08%, getting 0.9751 and 0.9800 scores. When comparing F1/GB, the deep learning methods have the lowest scores. Among the baselines, the TBAFT model has the highest average F1/GB score of 0.8252. DRAGON and DRAGON+ improve upon this by 17.94%-21.34%. Compared to the deep learning baselines, DRAGON and DRAGON+ improve the F1/GB by up to 305.03% and 316.68% respectively. In terms of diagnosis scores, StepGAN is the best baseline with HR of 0.7160 and NDCG of 0.7772. Compared to these scores, DRAGON and DRAGON give 5.01%-6.54% and 5.69%-8.95% improvement for HR and NDCG respectively. These results show that having only a single discriminator network allows us to significantly reduce the network's number of parameters and give a performance boost compared to having two such networks in StepGAN.

Comparison of QoS Metrics. We now present the QoS scores of all models for the three federated configurations in Fig. 8. Fig. 9 presents the average response time and SLO violation rates for each application in the AIoTBench workloads. To test DRAGON and DRAGON+, we use a GON network trained on the FTSAD-55 dataset and the scheduling decisions from the state-of-the-art GOBI scheduler [32]. Training time of the GON network on this dataset is 46.23 seconds. Among the baselines, StepGAN has the lowest energy consumption of 10.15 KW-hr averaged over the three configurations. The DRAGON+ model has the lowest energy consumption of 8.48 KW-hr, 16.40% lower than StepGAN. Second lowest is DRAGON with 8.79 KW-hr, 13.39% lower than StepGAN (Fig. 8(a)). We see similar trends

TABLE II
COMPARISON OF DETECTION AND DIAGNOSIS SCORES WITH STANDARD DEVIATIONS. RESULTS REPORTED ON THE TEST SET WITH 80:20 TRAIN-TEST SPLIT.

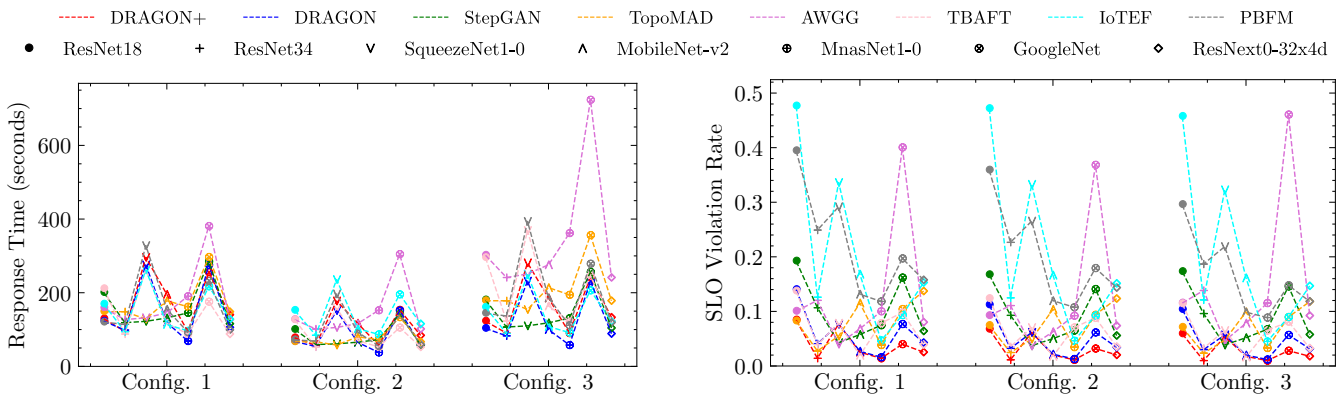
Method	Detection						Diagnosis	
	Accuracy	Precision	Recall	AUROC	F1	F1/GB	HR@100	NDCG@100
				FTSAD-1				
PBFM	0.8526±0.0039	0.7362±0.0015	0.8412±0.0075	0.8713±0.0070	0.7852±0.0031	0.7325±0.0062	0.5461±0.0048	0.4716±0.0031
IoTEF	0.8713±0.0014	0.7814±0.0049	0.8721±0.0032	0.8827±0.0048	0.8243±0.0010	0.7418±0.0001	0.4958±0.0049	0.5125±0.0019
TBAFT	0.8967±0.0088	0.8012±0.0068	0.8516±0.0037	0.8561±0.0054	0.8257±0.0000	0.7737±0.0055	0.5112±0.0022	0.6019±0.0042
AWGG	0.7981±0.0039	0.7613±0.0039	0.8913±0.0055	0.8824±0.0056	0.8212±0.0024	0.4599±0.0006	0.5827±0.0044	0.6110±0.0010
TopoMAD	0.8924±0.0083	0.8385±0.0072	0.7123±0.0055	0.9012±0.0033	0.7703±0.0036	0.2095±0.0020	0.5918±0.0028	0.6512±0.0044
StepGAN	0.8987±0.0058	0.8622±0.0045	0.8345±0.0011	0.9066±0.0062	0.8481±0.0052	0.3420±0.0017	0.5946±0.0037	0.6622±0.0000
DRAGON	0.9101±0.0023	0.9524±0.0020	0.8872±0.0019	0.9418±0.0066	0.9186±0.0063	0.9494±0.0056	0.6124±0.0060	0.7013±0.0060
DRAGON+	0.9201±0.0020	0.9601±0.0037	0.8901±0.0003	0.9539±0.0056	0.9238±0.0061	0.9786±0.0070	0.6200±0.0016	0.7221±0.0025
				FTSAD-25				
PBFM	0.6712±0.0060	0.9173±0.0008	0.7235±0.0054	0.5571±0.0022	0.8090±0.0048	0.7546±0.0057	0.6110±0.0054	0.7701±0.0048
IoTEF	0.9513±0.0012	0.9413±0.0009	0.9828±0.0026	0.8712±0.0040	0.9616±0.0008	0.8654±0.0040	0.6918±0.0039	0.8371±0.0032
TBAFT	0.9351±0.0061	0.9547±0.0046	0.9561±0.0064	0.8517±0.0033	0.9554±0.0058	0.8952±0.0041	0.6713±0.0004	0.8127±0.0070
AWGG	0.9013±0.0089	0.9326±0.0092	0.8913±0.0022	0.8091±0.0079	0.9114±0.0013	0.5104±0.0017	0.6542±0.0026	0.7817±0.0062
TopoMAD	0.9123±0.0019	0.9497±0.0075	0.9781±0.0002	0.9882±0.0065	0.9637±0.0050	0.2622±0.0009	0.7474±0.0011	0.8720±0.0057
StepGAN	0.9233±0.0022	0.9378±0.0056	0.9669±0.0028	0.9866±0.0022	0.9521±0.0018	0.3839±0.0023	0.7512±0.0041	0.8654±0.0010
DRAGON	0.9781±0.0040	0.9665±0.0075	0.9978±0.0082	0.9918±0.0048	0.9819±0.0008	1.0148±0.0014	0.7981±0.0034	0.9017±0.0053
DRAGON+	0.9812±0.0058	0.9665±0.0010	0.9998±0.0051	0.9934±0.0005	0.9829±0.0007	1.0412±0.0062	0.8013±0.0055	0.9172±0.0003
				FTSAD-55				
PBFM	0.8912±0.0066	0.8721±0.0048	0.9163±0.0020	0.6029±0.0006	0.8937±0.0061	0.8336±0.0053	0.7612±0.0052	0.7410±0.0051
IoTEF	0.8888±0.0011	0.9038±0.0068	0.9424±0.0060	0.9011±0.0078	0.9227±0.0044	0.8304±0.0076	0.7812±0.0025	0.7717±0.0032
TBAFT	0.7812±0.0034	0.7673±0.0062	0.9801±0.0065	0.9635±0.0033	0.8607±0.0035	0.8065±0.0080	0.7911±0.0064	0.6526±0.0063
AWGG	0.8964±0.0029	0.9054±0.0076	0.9121±0.0069	0.9251±0.0019	0.9087±0.0046	0.5089±0.0046	0.8013±0.0026	0.7915±0.0070
TopoMAD	0.9350±0.0024	0.9105±0.0001	0.9212±0.0009	0.9355±0.0001	0.9158±0.0031	0.2491±0.0002	0.8129±0.0051	0.7998±0.0001
StepGAN	0.9361±0.0010	0.9128±0.0017	0.9107±0.0031	0.9078±0.0007	0.9117±0.0085	0.3676±0.0011	0.8022±0.0052	0.8041±0.0060
DRAGON	0.9440±0.0019	0.9038±0.0005	0.9469±0.0047	0.9916±0.0048	0.9248±0.0046	0.9558±0.0039	0.8452±0.0082	0.8612±0.0042
DRAGON+	0.9561±0.0094	0.9120±0.0068	0.9469±0.0081	0.9927±0.0077	0.9291±0.0080	0.9842±0.0012	0.8671±0.0040	0.9013±0.0079



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

(e) Migration Count (f) Migration Time (g) Scheduling Time (h) Memory Consumption

Fig. 8. Comparison of QoS parameters of DRAGON against baselines. DRAGON and DRAGON+ trained on complete FTSAD-55 data and tested on AIoTBench based workloads.



(a) Response Time

(b) SLO Violation Rate

Fig. 9. Comparison of QoS parameters of DRAGON against baselines for each application type in AIoTBench workloads.

for response time. The AWGG model struggles in the third configuration with heterogeneous resource capacities across LEIs due to the drop in the performance of reconstructive clustering. DRAGON and DRAGON+ have average response times comparable to the best baseline for each configuration. Comparing the energy consumption and response times across configurations, we observe that they are lower for the second configuration compared to the first. This is primarily because the second configuration has only 4 running instances of the fault-tolerance model due to four brokers. The third configuration has the highest energy consumption and response time due to the increasing difficulty in fault-tolerance in heterogeneous federated setups (Figs. 8(a) and 8(b)). The SLO violation rates remain mostly consistent across configurations, except for AWGG that has a higher violation rate for the third configuration due to higher response times (Figs. 8(c) and 9(b)). Among all baselines, TBAFT has the lowest average SLO violation rate of 6.37%, where DRAGON and DRAGON+ improve upon this by 20.25% and 47.72%, giving violation rates of 5.08% and 3.33% respectively (Fig. 8(c)). In terms of fairness, DRAGON+ has the highest average fairness index of 0.78 (Fig. 8(d)). The fairness drops from the first to third configurations due to more disparate LEI resources. In terms of migration count and migration time, AWGG is the most aggressive in terms of migrations (Figs. 8(e) and 8(f)). On the other hand, TopoMAD is the most parsimonious. DRAGON and DRAGON+ trade off between the fault-tolerance of migrations and their overheads. In terms of scheduling time, the heuristic solutions have the lowest times with IoTEF having only 2.52 seconds on an average. The deep learning methods have much higher scheduling times. However, for the second and third configurations, deep learning baselines have scheduling times consistent with the first configuration. On the other hand, DRAGON and DRAGON+ have lower scheduling times than all deep learning baselines. This is due to the dot-product attention in the GON network and the higher-order optimization, allowing the model to scale with the dimension size of the time-series window and the number of edge devices in an LEI (Fig. 8(g)). Memory consumption of the DRAGON and DRAGON+ models is the lowest, being only 10.71% and 8.72%. It is lower for DRAGON+ due to the lower parameter size GON network compared to DRAGON (see Section V-C and Fig. 8(h)).

VI. Conclusions

We have presented a decentralized fault-tolerance model, DRAGON, that utilizes fault predictions from GON and runs co-simulated fictitious play to converge to preemptive migration decisions. DRAGON leverages a novel memory-efficient generative model called generative optimization networks (GONs). Unlike previously proposed GAN models, GON uses a single neural network to both discriminate and generate samples. This comes with the advantage of significant savings in terms of memory footprint. Unlike prior work aiming at reducing the parameter size of popular generative models such as GANs, this framework is suitable for memory-constrained systems like edge devices. We extend DRAGON to DRAGON+ where we perform a multi-step optimization of a fitness score. The proposed models give up to 4.56% higher F1 scores compared to the best baseline StepGAN. In a federated Raspberry Pi based edge setup with real-world AIoTBench workloads, DRAGON and DRAGON+ are able to outperform baselines by giving up to 74%, 63% and 82% better energy consumption, response time and service deadline violations, respectively. These improvement are primarily due to advances like window initialization, second-order optimization and graph topology based attention.

In the future, the model could be adapted to make it more robust against non-altruistic entities in the system.

Software Availability

The code and relevant training scripts will be made publicly available on GitHub under BSD-3 licence upon publication.

Acknowledgments

Shreshth Tuli is supported by the President's Ph.D. Scholarship at the Imperial College London.

REFERENCES

[1] S. Tuli, S. Tuli *et al.*, "Generative optimization networks for memory efficient data generation," *Advances in Neural Information Processing Systems, Workshop on ML for Systems*, 2021.

[2] B. Rochwerger, D. Breitgand *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.

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

[4] S. S. Gill, S. Tuli *et al.*, "Transformative effects of IoT, Blockchain and Artificial Intelligence on cloud computing: Evolution, vision, trends and open challenges," *Internet of Things*, vol. 8, pp. 100–118, 2019.

[5] X. Cao, G. Tang *et al.*, "Edge federation: Towards an integrated service provisioning model," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1116–1129, 2020.

[6] E. Zeydan, E. Bastug *et al.*, "Big data caching for networking: Moving from cloud to edge," *IEEE Communications Magazine*, vol. 54, no. 9, pp. 36–42, 2016.

[7] B. Sharma, L. Sharma *et al.*, "Anomaly detection techniques using deep learning in IoT: A survey," in *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)*. IEEE, 2019, pp. 146–149.

[8] W. Zhang, Z. Zhang *et al.*, "Masm: A multiple-algorithm service model for energy-delay optimization in edge artificial intelligence," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 7, pp. 4216–4224, 2019.

[9] A. Harish, S. Jhawar *et al.*, "Implementing machine learning on edge devices with limited working memory," in *Inventive Communication and Computational Technologies*. Springer, 2020, pp. 1255–1261.

[10] T. Hao, Y. Huang *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.

[11] 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.

[12] S. Mukherjee and S. Sidhanta, "Amas: Adaptive auto-scaling on the edge," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 618–621.

[13] P. A. Costa, X. Bai *et al.*, "Medusa: An efficient cloud fault-tolerant mapreduce," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 443–452.

[14] A. Rawat, R. Sushil *et al.*, "A new adaptive fault tolerant framework in the cloud," *IETE Journal of Research*, pp. 1–13, 2021.

[15] B. Ray, A. Saha *et al.*, "Proactive fault-tolerance technique to enhance reliability of cloud service in cloud federation environment," *IEEE Transactions on Cloud Computing (Early Access)*, pp. 1–1, 2020.

[16] Y. Gan, S. Dev *et al.*, "Sage: Leveraging ML To Diagnose Unpredictable Performance in Cloud Microservices," *ML for Computer Architecture and Systems*, 2020.

[17] Z. He, P. Chen *et al.*, "A spatiotemporal deep learning approach for unsupervised anomaly detection in cloud systems," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[18] A. Khanna, A. Sah *et al.*, "Intelligent mobile edge computing: A deep learning based approach," in *International Conference on Advances in Computing and Data Sciences*. Springer, 2020, pp. 107–116.

[19] D. Li, D. Chen *et al.*, "MAD-GAN: Multivariate anomaly detection for time series data with generative adversarial networks," in *International Conference on Artificial Neural Networks*. Springer, 2019, pp. 703–716.

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

[21] 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.

[22] Y. Feng, Z. Liu *et al.*, "Make the Rocket Intelligent at IoT Edge: Stepwise GAN for Anomaly Detection of LRE with Multi-source Fusion," *IEEE Internet of Things Journal*, vol. 9, no. 4, pp. 3135–3149, 2022.

[23] J. Grover and R. M. Garimella, "Reliable and fault-tolerant iot-edge architecture," in *2018 IEEE sensors*. IEEE, 2018, pp. 1–4.

[24] M. Mudassar, Y. Zhai *et al.*, "A decentralized latency-aware task allocation and group formation approach with fault tolerance for iot applications," *IEEE Access*, vol. 8, pp. 49 212–49 223, 2020.

[25] A. Javed, K. Heljanko *et al.*, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *2018 IEEE 4th world forum on internet of things (WF-IoT)*. IEEE, 2018, pp. 813–818.

[26] Q. Wang and H. Mu, "Privacy-preserving and lightweight selective aggregation with fault-tolerance for edge computing-enhanced iot," *Sensors*, vol. 21, no. 16, p. 5369, 2021.

[27] J. Audibert, P. Michiardi *et al.*, "USAD: Unsupervised anomaly detection on multivariate time series," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3395–3404.

[28] A. Polino, R. Pascanu *et al.*, "Model compression via distillation and quantization," in *6th International Conference on Learning Representations*, 2018.

[29] H. Wang, S. Gui *et al.*, "GAN Slimming: All-in-One GAN Compression by A Unified Optimization Framework," in *European Conference on Computer Vision*. Springer, 2020, pp. 54–73.

[30] P. Li, H. Seferoglu *et al.*, "Model-distributed dnn training for memory-constrained edge computing devices," in *2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 2021, pp. 1–6.

[31] I. Goodfellow, J. Pouget-Abadie *et al.*, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[32] S. Tuli, S. R. Poojara *et al.*, "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.

[33] A. Goli, O. Hajihassani *et al.*, "Migrating from monolithic to serverless: A fintech case study," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 20–25.

[34] J. Huang, C. Samplawski *et al.*, "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*, 2020, pp. 1–12.

[35] C. Engelmann, G. R. Vallee *et al.*, "Proactive fault tolerance using preemptive migration," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2009, pp. 252–257.

[36] J. McChesney, N. Wang *et al.*, "DeFog: fog computing benchmarks," in *The 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 47–58.

[37] C. Luo, F. Zhang *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.

[38] S. Bagchi, M.-B. Siddiqui *et al.*, "Dependability in edge computing," *Communications of the ACM*, vol. 63, no. 1, pp. 58–66, 2019.

[39] A. Javed, J. Robert *et al.*, "IoTEF: A federated edge-cloud architecture for fault-tolerant IoT applications," *Journal of Grid Computing*, pp. 1–24, 2020.

[40] X. Hu, Y. Li *et al.*, "A novel two-stage unsupervised fault recognition framework combining feature extraction and fuzzy clustering for collaborative AIoT," *IEEE Transactions on Industrial Informatics*, 2021.

[41] A. Siffer, P.-A. Fouque *et al.*, "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.

[42] 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.

[43] A. S. Tanenbaum and A. S. Woodhull, *Operating systems: design and implementation*. Prentice Hall Englewood Cliffs, 1997, vol. 68.

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

[45] X. Vasilakos, W. Featherstone *et al.*, "Towards Zero Downtime Edge Application Mobility for Ultra-Low Latency 5G Streaming," in *2020 IEEE Cloud Summit*. IEEE, 2020, pp. 25–32.

[46] 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.

[47] G. Liu, K. J. Shih *et al.*, "Partial convolution based padding," *arXiv preprint arXiv:1811.11718*, 2018.

[48] K. Chen, Y. Chen *et al.*, "Self-supervised adversarial training," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 2218–2222.

[49] Z. Yao, A. Gholami *et al.*, "Adahessian: An adaptive second order optimizer for machine learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 12, 2021, pp. 10 665–10 673.

[50] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for optimization*. MIT Press, 2019.
[51] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations*, 2018.
[52] S. Yun, M. Jeong *et al.*, "Graph transformer networks," *Advances in Neural Information Processing Systems*, vol. 32, pp. 11 983–11 993, 2019.
[53] S. Zhang, H. Tong *et al.*, "Graph convolutional networks: a comprehensive review," *Computational Social Networks*, vol. 6, no. 1, pp. 1–23, 2019.
[54] J. Chorowski, D. Bahdanau *et al.*, "Attention-based models for speech recognition," in *Advances in Neural Information Processing Systems*, 2015.
[55] C. Eksin and A. Ribeiro, "Distributed fictitious play for multiagent systems in uncertain environments," *IEEE Transactions on Automatic Control*, vol. 63, no. 4, pp. 1177–1184, 2017.
[56] A. Paszke, S. Gross *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, pp. 8026–8037, 2019.
[57] K. Gilly, S. Alcaraz *et al.*, "Modelling edge computing in urban mobility simulation scenarios," in *2020 IFIP Networking Conference (Networking)*. IEEE, 2020, pp. 539–543.
[58] T.-Y. Lin, M. Maire *et al.*, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.
[59] Y. Mao, J. Zhang *et al.*, "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.
[60] D. Basu, X. Wang *et al.*, "Learn-as-you-go with megh: Efficient live migration of virtual machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1786–1801, 2019.
[61] Y. Su, Y. Zhao *et al.*, "Robust anomaly detection for multivariate time series through stochastic recurrent neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2828–2837.
[62] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of ir techniques," *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 4, pp. 422–446, 2002.
[63] M. Grinberg, *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.
[64] R. S. Venkatesh, T. Smejkal *et al.*, "Fast in-memory criu for docker containers," in *The International Symposium on Memory Systems*, 2019, pp. 53–65.

Shreshth Tuli is a President's Ph.D. Scholar at the Department of Computing, Imperial College London, UK. Prior to this he was an undergraduate student at the Department of Computer Science and Engineering at Indian Institute of Technology - Delhi, India. He has worked as a visiting research fellow at the CLOUDS Laboratory, School of Computing and Information Systems, the University of Melbourne, Australia. He is a national level Kishore Vaigyanik Protsahan Yojana (KVPY) scholarship holder from the Government of India for excellence in science and innovation. His research interests include Internet of Things (IoT), Fog Computing and Deep Learning. For further information, visit `https://shreshthtuli.github.io/`.

Giuliano Casale joined the Department of Computing at Imperial College London in 2010, where he is currently a Reader. Previously, he worked as a research scientist and consultant in the capacity planning industry. He teaches and does research in performance engineering and cloud computing, topics on which he has published more than 100 refereed papers. He has served on the technical program committee of over 80 conferences and workshops and as co-chair for several conferences in the area of performance and reliability engineering, such as ACM SIGMETRICS/Performance and IEEE/IFIP DSN. His research work has received multiple awards, recently the best paper award at ACM SIGMETRICS. He serves on the editorial boards of IEEE TNSM and ACM TOMPECS and as current chair of ACM SIGMETRICS.

Nicholas R. Jennings is the Vice-Provost for Research and Enterprise and Professor of Artificial Intelligence at Imperial College London. He is an internationally-recognised authority in the areas of AI, autonomous systems, cyber-security and agent-based computing. He is a member of the UK government's AI Council, the governing body of the Engineering and Physical Sciences Research Council, the Monaco Digital Advisory Council, and chair of the Royal Academy of Engineering's Policy Committee. Before Imperial, he was the UK's first Regius Professor of Computer Science (a post bestowed by the monarch to recognise exceptionally high quality research) and the UK Government's first Chief Scientific Advisor for National Security.