information

Article

Cyber Security on the Edge: Efficient Enabling of Machine Learning on IoT Devices

Swati Kumari [1,2,†], Vatsal Tulshyan [1,†] and Hitesh Tewari [1,*]

[1] School of Computer Science & Statistics, Trinity College Dublin, D02 PN40 Dublin, Ireland; kumaris@tcd.ie or swati.kumari@thapar.edu (S.K.); tulshyav@tcd.ie (V.T.)
[2] Thapar Institute of Engineering & Technology, Patiala 147004, Punjab, India
* Correspondence: hitesh.tewari@tcd.ie; Tel.: +353-872122008
† These authors contributed equally to this work.

Abstract: Due to rising cyber threats, IoT devices' security vulnerabilities are expanding. However, these devices cannot run complicated security algorithms locally due to hardware restrictions. Data must be transferred to cloud nodes for processing, giving attackers an entry point. This research investigates distributed computing on the edge, using AI-enabled IoT devices and container orchestration tools to process data in real time at the network edge. The purpose is to identify and mitigate DDoS assaults while minimizing CPU usage to improve security. It compares typical IoT devices with and without AI-enabled chips, container orchestration, and assesses their performance in running machine learning models with different cluster settings. The proposed architecture aims to empower IoT devices to process data locally, minimizing the reliance on cloud transmission and bolstering security in IoT environments. The results correlate with the update in the architecture. With the addition of AI-enabled IoT device and container orchestration, there is a difference of 60% between the new architecture and traditional architecture where only Raspberry Pi were being used.

Keywords: IoT; cyber threats; distributed computing; AI-enabled chips; container orchestration; DDoS attacks

Citation: Kumari, S.; Tulshyan, V.; Tewari, H. Cyber Security on the Edge: Efficient Enabling of Machine Learning on IoT Devices. *Information* **2024**, *15*, 126. https://doi.org/10.3390/info15030126

Academic Editors: Krzysztof Szczypiorski and Daniel Paczesny

Received: 5 January 2024
Revised: 9 February 2024
Accepted: 19 February 2024
Published: 23 February 2024

1. Introduction

1.1. Background

Since cyber security threats are rising, securing the Internet of Things (IoT) is crucial to daily life. Since IoT hardware cannot manage sophisticated workloads, they cannot run complex security risk detection algorithms on themselves, making them vulnerable to high-level security attacks. Instead, they must send data to cloud nodes to process inputs, leaving a loophole for attackers to steal data midway. However, if they could run those models, they could process their real-time packets or inputs. Security is still important when embedded technology advances and becomes robust enough to create apps on it to make life easier. Security adaptation cannot keep up with rapid technology change. Growing IoT devices have caused exponential growth. Imagine its economic impact on global markets. Distributed computing on the edge is desired to offload activities and improve resource planning and IoT device use due to cloud reliability, latency, and privacy problems [1].

Most workloads are in the cloud, making data transit vulnerable to hackers. Fitness trackers broadcast important data like heart rate and blood pressure to the cloud, which might be compromised during transmission [2,3]. To circumvent these issues, it is necessary to compute close to IoT devices to scale edge computing and make them capable of processing large amounts of data.

Container orchestration is essential for IoT application management and deployment. Developers may automate container deployment, scaling, and administration to optimize IoT device resources. The dynamic and resource-constrained nature of IoT settings requires

load balancing, auto-scaling, and fault tolerance, which container orchestration provides. IoT service dependability and availability are improved, updates and maintenance are simplified, and developers can focus on developing creative, robust solutions that respond to changing conditions in real time [4].

The methodology involves setting up IoT devices with static IP addresses, building a machine learning model for DDoS detection, and setting up traditional devices (e.g., Raspberry Pi) in a distributed computing environment with an AI-embedded chip IoT device and a container orchestration tool. A Raspberry Pi-only distributed computing cluster with and without container orchestration tools would be compared to this cluster. All clusters will be evaluated by running ML models and comparing their performance and hardware metrics (CPU and Memory). This research aims to find whether scaling up traditional IoT devices with AI-enabled chips can help with running complex models and whether lightweight orchestration tools could help the resource utilisation on IoT devices and help them to run machine learning and apply them to cyber security improvement.

Figure 1 shows the proposed state-of-the-art architecture in this research. It combines AI-enabled IoT devices with other IoT devices to form a cluster. This design would enable near-edge computation by sending IoT packets to the AI IoT device for processing to detect anomalies. The cloud would preserve these anomalies for future improvements. The figure's network cloud is any usable network interface such as WiFi or swap networks. Due to the AI IoT device's sole purpose of processing and communicating with the cloud, it remains a secure route to transmit information. With the help of container orchestration tools and AI IoT Devices in a networking architecture, the IoT devices will be able to run machine learning models efficiently on the edge to tackle attacks like DDoS by reducing CPU utilization.

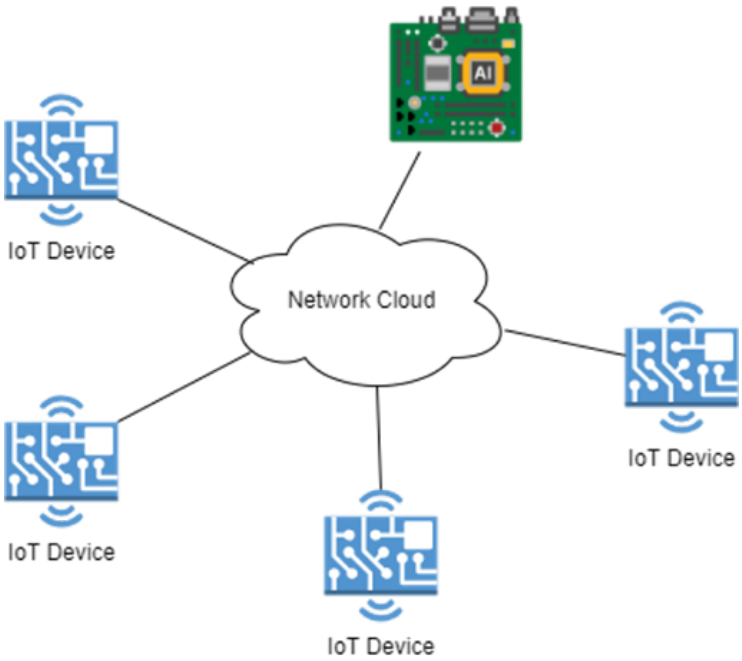


Figure 1. General Architecture.

1.2. Research Contribution

This research aims to scale up the performance of IoT devices to support machine learning models and perform security risk detections on top of them with the help of AI-enabled IoT device and container orchestration like Microk8s. This research accomplished its goals by employing the following methods:

1. Use of AI-enabled architecture in the cluster;
2. Use of machine learning models for the detection of DDos and other attacks;

3. Use of container orchestration tool like Microk8s;
4. Use of different forms of architecture with AI-enabled IoT devices or simple IoT devices like Raspberry Pi;
5. Use of Docker images for microservices;
6. An overall decrease of 60% in CPU utilisation from the traditional architecture of Raspberry Pi to Microk8s architecture with Nvidia is achieved;
7. Container orchestration-as-a-solution: it provides a way to efficiently manage the autoscaling of the resources if required and did a great job with job scheduling.

2. Related Work

2.1. Cyber Security as a Challenge to IoT Devices

IoT devices are being used in many fields. This covers manufacturing, logistical, medical, military, etc. Research predicts 100 billion devices will be in use by 2025 [5]. Considering the predicted amount of devices [5], it is clear that attackers are targeting the IoT. The attackers would compromise selected nodes in the architecture to access the source code and infect the remainder of the deployments, including cloud resources. IoT deployments have violated local laws in earlier studies. Current IoT security methods often rely on manual intervention for maintenance and updates, leading to a lag in protection and an inability to learn and adapt to evolving threats. The unity and integrity of IoT, which spans terminals, networks, and service platforms, necessitate security solutions capable of effectively handling massive amounts of complex data [6].

Traditional data transfer to central servers is still popular. The data are usually sent over MQTT or HTTP. Previous research has shown transmission problems with MQTT and HTTP protocols [7,8]. Attackers find transfer scenarios to be the entryway to their target information. They would use man-in-the-middle attacks to infect routers that could sneak malicious packets to the main servers. The attackers also turned IoT devices in the large-scale infrastructure into botnets by planting malicious malware. After that, these devices send simultaneous requests to primary servers, causing a Distributed Denial of Service on cloud servers. If the main servers cannot receive data from legitimate IoT devices, incorrect or timed-out responses during data transmission cause data overload. In such a scenario, the data start collecting and rendering the legitimate devices to shut down operations [9].

2.2. Solutions Implemented to Solve Security Vulnerabilities

Malicious attacks have taken many forms. Attackers know that IoT security breaches take time to detect. Due to hardware issues, botnet and malware detection in the IoT environment might be difficult. However, edge computing with machine learning models on IoT devices has yielded novel solutions. Edge computing is a computing method in which the workload is divided amongst nodes to form a distributed architecture. It is an important area in research for IoT as this minimises the bandwidth and response time in an IoT environment. Moreover, it reduces the burden of a centralized server [10]. There have been detection methods which have been developed to be used in an edge computing scenario.

Myneni et al. [11] suggest "SmartDefense", a two-stage DDoS detection solution that utilizes deep learning algorithms and operates on the provider edge (PE) and consumer edge (CE). The PE and CE refer to the routers which are close to consumers and Internet service providers (ISPs). Edge computing helps SmartDefense detect and stop DDoS attacks near their source, reducing bandwidth waste and provider edge delay. The method also uses a botnet detection engine to detect and halt bot traffic. SmartDefense improves DDoS detection accuracy and reduces ISP overhead, according to the study. By detecting botnet devices and mitigating over 90% of DDoS traffic coming from the consumer edge, it can cut DDoS traffic by up to 51.95%.

Bhardwaj et al. [12] suggest a method called ShadowNet that makes use of computational capabilities at the network's edge to speed up a defence against IoT-DDoS attacks. The edge tier, which includes fog computing gateways and mobile edge computing (MEC)

access points, can manage IoT devices' massive Internet traffic using edge services. As IoT packets pass through gateways, edge functions build lightweight information profiles that are quickly acquired and sent to ShadowNet web services. IoT-DDoS attacks are initially prevented by ShadowNet's analysis of edge node data. It detects IoT-DDoS attacks 10 times faster than victim-based methods. It can also react to an attack proactively, stopping up to 82% of the malicious traffic from getting to the target and harming the Internet infrastructure.

Mirzai et al. [13] discovered the benefits of building a dynamic user-level scheduler to perform real-time updates of machine learning models and analysed the performance of the hardware of Nvidia Jetson Nano and Raspberry Pi. The scheduler allows local parallel model retraining on the IoT device without stopping the IDS, eliminating cloud resources. The Nvidia unit could retrain models while detecting anomalies, and the scheduler outperformed the baseline almost often, even with retraining! The Raspberry Pi unit underperformed due to its hardware's architecture. The experiments showed that the dynamic user-level scheduler improves the system's throughput, which reduces the attack detection time, and dynamically allocates resources based on attack suspicion. The results show that the suggested technique improves IDS performance for lightweight and data-driven ML algorithms for IoT.

2.3. Container Orchestration Applied to IoT Devices

Google's Borg was turned into Kubernetes using Go programming for Docker and Docker containers for orchestration. Kubernetes expanded on Google's Borg, which inspired its development. Later, Kubernetes was viewed as a strong edge computing solution, which led to the creation of lightweight container orchestration solutions like k3s, Microk8s, and others because Kubernetes required intensive hardware specs that IoT platforms lack. Performance of lightweight container orchestration technologies has been studied. All research is from the last two to three years.

The researchers [14] established a Kubernetes cluster using readily available Raspberry Pi devices. The cluster's capacity to handle IoT components as gateways for sensors and actuators was tested utilizing synchronous and asynchronous connection situations. Linux containers like Docker can be used to quickly deploy microservices on near-end and far-end devices. Container technologies isolate processes and hardware within an operating system. Due to its fast deployment, communication management, high availability, and controlled updates, Kubernetes may be a solution for IoT. The paper draws attention to the paucity of research on deploying Kubernetes in the IoT setting and the need for additional research to analyze its applicability and limitations. Performance testing, evaluating response times, request rates, and cluster stability revealed the architecture's pros and cons [15].

Lightweight solutions have been developed in recent years: Minikube, Microk8s, k3s, k0s, KubeEdge, and Microshift. In several tests, the authors tested tiny clusters of lightweight k8s distributions under high workloads. These insights can assist researchers in locating the lightweight k8s distribution for their use case. The researchers have basically found the resource consumption on idle for different k8s distributions, finding the resource consumption when the pods are being created, deleted, updated or being read and when the pods are assigned intensive workloads. Azure VMs, not IoT devices, powered the research. It would have been ideal to collaborate with IoT devices near to production. The researchers acknowledge the above shortcoming and suggest replacing artificial benchmark situations with production workloads and using black-box measures instead of white-box techniques. If the controller node has at least 1–2 GB of RAM and worker nodes have even less hardware, all lightweight k8s distributions perform well on low-end single-board computers, according to the study.

2.4. Mitigation to DDoS Attacks on IoT Devices

While the detection work has been seen to be a work in progress, mitigation mechanisms after a DDoS attack happen are also being given due attention. Blockchain with

the help of smart contracts is one interesting way of implementing this. Although, the solution at large is not localised to the IoT devices but seems to be making the solution work through the cloud.

Hayat et al. [16] explore IoT DDoS defences, including smart contracts and blockchain-based methods. Multiple-level DDoS (ML-DDoS) is being developed to protect IoT devices and compute servers against DDoS attacks using blockchain. It prevents devices turning into bots and increases security with gas restriction blockchain, device blacklisting, and authentication. Blockchain technology is also called a public ledger with unchangeable records. It facilitates peer agreements in asset management, finance, and other fields via consensus methods. The decentralized and open-source Ethereum blockchain is promoted for protocol-based transactional protocols between parties. Comparison of ML-DDoS performance in the presence of bot-based scenarios shows that it outperforms other cutting-edge methods including PUF, IoT-DDoS, IoT-botnet, collaborative-DDoS, and deep learning-DDoS in terms of throughput, latency, and CPU use.

2.5. Research Gap

Table 1 highlights the issues of cyber security in IoT. Hardware complexity, linear cloud complexity increase that does not match IoT's exponential development, data manipulation by attackers, infrastructure botnets, and DDoS or Man in the Middle attacks on servers were mentioned. In the literature, deploying detection methods on routers for edge computing still leaves devices vulnerable because they are not secured. Additionally, infected devices are blocked from accessing the main servers. The detection model deployment should treat the device instead of rendering it useless. An IoT device with AI capabilities, including hardware configuration for machine learning algorithms, was proposed in [6]. In [13]. Nvidia Jetson Nano, an AI IoT Device, was shown to have significant potential above Raspberry Pi. Even IoT research has advanced with container orchestration which is shown in [14,15]. These studies suggest that edge computing, container orchestration, AI IoT devices, and IoT devices in general can be combined to achieve more. This study combines various concepts to work cohesively.

Table 1. Literature review.

Ref.	Work Carried Out	Challenges
[6]	Attribution analysis of IoT Security Threats Security Threats in different layers of OSI model Discussion of Feasibility Analysis of AI in IoT Security AI solution for IoT security threats	Linear Growth of Cloud Computing does not match the exponential growth of IoT Devices. Computational Complexity for IoT Devices Data Manipulation done by attackers
[11]	A two-stage DDoS detection solution that utilizes deep learning algorithms and operates on the provider edge (PE) and consumer edge (CE).	Solution is not based on IoT devices but rather on routers.
[17]	ShadowNet deployed at the network's edge to speed up defence against IoT-DDoS attacks.	Solution deployed on gateways
[14]	Exploring different Kubernetes distributions on rasp- berry pi clusters. Exploring pros and cons	Did not perform a load test on the cluster
[18]	Distributed platform over IoT devices so as to make them support GAN as a solution to Intrusion. Reduced dependency on Central cloud systems	The research did not present the CPU utilisation and Memory utilisation on the devices.
[15]	Analysis of different lightweight k8s distributions	Implemented it on Azure VMs.
[16]	Ethereum based blockchain Signature-based authenticity test of devices	Eliminates the affected botnet devices
[12]	Kubernetes deployed on Raspberry Pi. Test conducted using synchronous and asynchronous scenarios to handle IoT components	The test could have also involved testing on machine learning paradigm as well.

3. Methodology

The CICDDoSDataset2019 has been used for training the machine learning model. The dataset is shown in Table 2. This dataset has 2 Million entries and 80 columns after preprocessing. The preprocessing involved replacing spaces with no spaces in every column wherever applicable, dropping columns such as FlowID, SourceIP, DestinationIP, Timestamp, SimillarHTTP, SourcePort, and DestinationPort. Since the numeric data were in text format so those were converted into the numeric format. There are 2,180,000 samples for DDoS packets and 1542 samples are benign samples. This indicates a heavy imbalance between the classes [19].

Table 2. DDoS dataset.

S.No.	Flow Duration	TotalFwd Packet	..	Similar HTTP	Inbound	Label
1	9141643	85894	..	0	1	DrDoS_LDAP
2	1	2	..	0	1	DrDoS_LDAP
3	2	2	..	0	1	DrDoS_LDAP
4	1	2	..	0	1	DrDoS_LDAP
5	2	2	..	0	1	DrDoS_LDAP
6	2	2	..	0	1	DrDoS_LDAP
7	2	2	..	0	1	DrDoS_LDAP
..	..	..	..	..	..	..
2181536	1	2	..	0	1	DrDoS_LDAP
2181537	50	2	..	0	1	DrDoS_LDAP
2181538	1	2	..	0	1	DrDoS_LDAP
2181539	1	2	..	0	1	DrDoS_LDAP
2181540	1	2	..	0	1	DrDoS_LDAP
2181541	1	2	..	0	1	DrDoS_LDAP
2181542	2	2	..	0	1	DrDoS_LDAP

3.1. DDoS Detection Model

Three Machine learning models were trained for the detection of DDoS.

1. Dummy Classifier: A dummy classifier is a simple and baseline machine learning model used for comparison and benchmarking purposes. It is typically employed when dealing with imbalanced datasets or as a reference to assess the performance of more sophisticated models. The dummy classifier makes predictions based on predefined rules and does not learn from the data. The dummy classifier with the strategy of predicting the most frequent class is utilised for the baseline. The most frequent class classifier always predicts the class that appears most frequently in the training data. This is useful when dealing with imbalanced datasets where one class is significantly more prevalent than others [20].
2. Logistic Regression: Logistic regression is a popular and widely used classification algorithm in machine learning. Despite its name, it is used for binary classification tasks, where the output variable has two classes (e.g., "Yes" or "No", "True" or "False"). The logistic regression model calculates the probability that an input data point belongs to a particular class. It does this by transforming its output through the logistic function (also known as the sigmoid function). The logistic function maps any real-valued number to a value between 0 and 1, which can be interpreted as a probability [21].

3. Deep Learning Model: The deep learning model has three layers to it. The first layer is the input layer. It has got the input shape of 80 neurons which outputs a shape of (None, 128). A dropout regularisation enables this layer to handle overfitting of the data. This is followed by another dense layer which also has a dropout regularisation. In the end, the model has an output layer with one neuron giving a probability as an output due to the sigmoid function being used in the last layer.

Performance Metrics

1. F1 score: It combines the precision and recall scores of a model. It is usually more useful than accuracy, especially if you have an uneven class distribution. The equation is provided in Equation (1).

$$F1Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{1}$$

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{2}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{3}$$

3.2. Technology Used

3.2.1. Python

Python Programming Language has been used throughout the complete research. There are some important frameworks used as part of this research.

1. Flask: This is a framework defined under the Python language for hosting REST APIs. The programmer can define the business logic which interacts with the front end. It provides an interface between the end user and databases.
2. Tensorflow (TF): This is a package built by Google for deep learning-based workloads. This package comes along with Keras as shown in Figure 2. Keras is an Application Programming Interface which allows the programmer to build layers for Artificial Neural Networks [22].

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               10240

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 64)                8256

 dropout_1 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 1)                 65

=================================================================
Total params: 18,561
Trainable params: 18,561
Non-trainable params: 0
_________________________________________________________________
```

Figure 2. Model definition of Keras model.

3.2.2. Microk8s

Microk8s has been developed by canonical and is being provided as a snap package as part of Ubuntu. It controls containerized services at the edge [23]. The previous researches have mentioned that the Microk8s requires atleast 4 GB of RAM usage. The Microk8s initially ships with the basics of k8s for example the controller, API server, dqlite storage, scheduler, and so on. Additionally, there are more add-ons that can be enabled by running "microk8s enable service name". The name of the service can be Prometheus, DNS, metallb, helm, metrics-server, Kubernetes-dashboard, and so on [15].

3.3. Components

This study used four Raspberry Pis with 4 GB RAM each and one Nvidia Jetson Nano with 4 GB RAM as part of the device infrastructure. A total of 4 MicroSD cards with 16 GB capacity were used for Raspberry Pi and a 64 GB MicroSD card was used for Nvidia Jetson Nano. Ubuntu Server 18.04 was installed on the Raspberry and Nvidia jetpack on the Nvidia Jetson Nano. The devices were connected with LAN wires and a switch as a medium between them. A WiFi adapter was also used with Nvidia Jetson Nano because Jetson does not come with an onboard WiFi module. The switch model is Netgear JGS524E ProSafe Plus 24-Port Gigabit Ethernet Switch.

3.3.1. Nvidia Jetson Nano

A robust single-board computer made specifically for AI and robotics applications is the Nvidia Jetson Nano. The Nvidia Tegra X1 processor, which houses a quad-core ARM Cortex-A57 CPU and a 128-core Nvidia Maxwell GPU, powers the Jetson Nano [24]. This GPU is especially well suited for jobs requiring parallel processing, making it the best choice for effectively executing deep learning models and AI workloads. There are other variations of the Jetson Nano available, but the most typical model comes with 4 GB of LPDDR4 RAM, a microSD card slot for storage, Gigabit Ethernet, and USB 3.0 ports for fast data transfer. Its capabilities for robotics and computer vision are increased with the addition of GPIO ports, a MIPI CSI camera connection, and Display Serial Interface (DSI) for interacting with cameras and displays. The device is displayed in Figure 3.

Figure 3. Nvidia Jetson Nano.

3.3.2. Raspberry Pi

The fourth version of the successful and adaptable single-board computer created by the Raspberry Pi Foundation is known as the Raspberry Pi 4. It became available in June 2019. It has a quad-core ARM Cortex-A72 CPU that runs up to 1.5 GHz, giving it a substantial performance advantage over earlier iterations. The RAM is 4 GB for the device being used in this project. The device is displayed in Figure 4 [25].

Figure 4. Raspberry Pi.

3.3.3. Netgear Switch

Netgear JGS524E ProSafe Plus 24-Port Gigabit Ethernet Switch has got 24 ports on it. A gigabit connection delivers up to 2000 Mbps of dedicated, non-blocking bandwidth per port. It is a plug-and-play type of device. The device is shown in Figure 5.

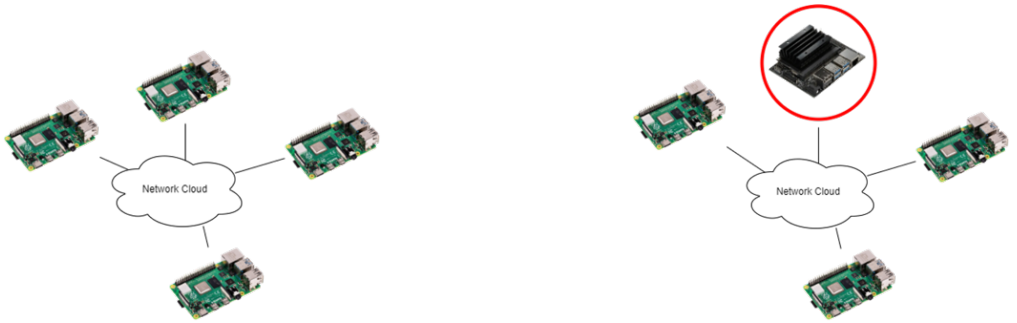
Figure 5. Netgear switch.

3.4. Networking Architectures

The architecture figures that you would see below have a cloud named Network Cloud. The network cloud in general means any sort of networking interface can be used, which could be a switch, WiFi network, or maybe a private 5G network. For this study, a switch and WiFi have been used to network the device together.

3.4.1. Traditional Architecture and Nvidia Architecture

Figure 6a shows an architecture which has been used widely. This kind of architecture is used in a fashion to interact with the real world and collects data from the sensors connected to the Raspberry Pi. The devices like Raspberry Pi transport the data to the cloud for processing and running detection models. When this architecture is used, the computational workloads become a problem.



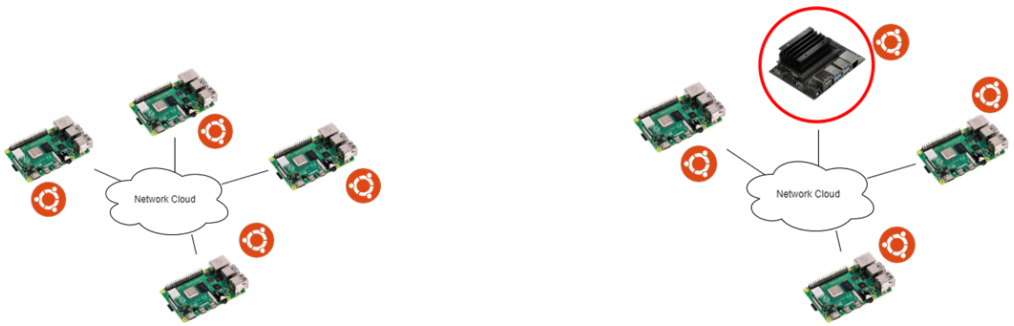
(**a**) Validation of Architecture (**b**) Nvidia Raspberry Pi Architecture

Figure 6. Traditional architecture and Nvidia architecture.

As introduced previously, in the general architecture utilise the AI-enabled IoT Device and replace one of the nodes in the Raspberry Pi architecture with an AI-IoT device. In this situation, the architecture is introduced with an Nvidia Jetson Nano which is responsible for running various types of security breach detection models and transporting anomaly data to the cloud for further inspection. This way the amount of data stored in the cloud decreases by a major margin and goes light on storage as well. So, the only data which are stored are the anomalies which allow better analysis and deploy recovery automation solutions. The architecture is shown in Figure 6b.

3.4.2. Raspberry Pi Microk8s Archtitecture and Nvdida Raspberry Pi Microk8s Architecture

The architecture introduces Microk8s as a resource management and container orchestration tool. This architecture with Microk8s has been previously seen in various research. Although, these researchers have not performed major load testing on the cluster for example running a machine learning model on the architecture. This study finds the CPU utilisation in the case of load testing related to running machine learning. The architecture is shown in Figure 7a.



(**a**) Raspberry Pi Microk8s Archtitecture (**b**) Nvdida Raspberry Pi Microk8s Architecture

Figure 7. Raspberry Pi Microk8s archtitecture and Nvdida Raspberry Pi Microk8s architecture.

Nvdida Raspberry Pi Microk8s architecture is introduced as the state-of-the-art architecture which includes the capabilities provided by the GPU provided in the Nvidia Jetson Nano as well as resource management tools like Microk8s. Together, these both provide exceptional abilities of efficient scheduling of memory and keeping track of microservices on the network as well as running the detection models with comparatively less CPU utilisation as compared to other network elements. The architecture is shown in Figure 7b.

3.5. Validation of Hypothesis

3.5.1. Machine Learning Validation

When it comes to Machine Learning model performance, it is expected that the model is able to predict classes distinctively without any major misclassification issue. In such scenarios, the F1 score associated with the classes should be substantial enough to make a feasible prediction to not fail when running on production systems. As we have seen earlier, the F1 score is the weighted harmonic mean of Precision and Recall, the changes in the F1 score indicate an overall improvement as compared to the previous model. The Dummy classifier indicates the baseline performance that needs to be outperformed by other models. This study explores the logistic regression model and the neural network for the detection models. These models would be expected to perform better than the baseline, and the state-of-the-art neural network to perform better than Logistic Regression.

3.5.2. Architecture Improvement Validation

For validation, it is expected that the traditional architecture will perform the poorest in terms of CPU utilisation and memory because of the hardware specifications of Raspberry Pi being used in the architecture. Therefore, the performance obtained in this architecture becomes the baseline which needs to be beaten by other architectures. Next,

the Nvidia-enabled architecture is expected to utilise 20% less CPU as compared to traditional architecture. The Raspberry Pi architecture with Microk8s is expected to perform 10% less than the Nvidia enabled architecture because it is running an efficient container orchestration tool. In the end, the state-of-the-art architecture where the Nvidia device and Microk8s are together is expected to perform at least 60% less than the traditional architecture.This is demonstrated in Figure 8.

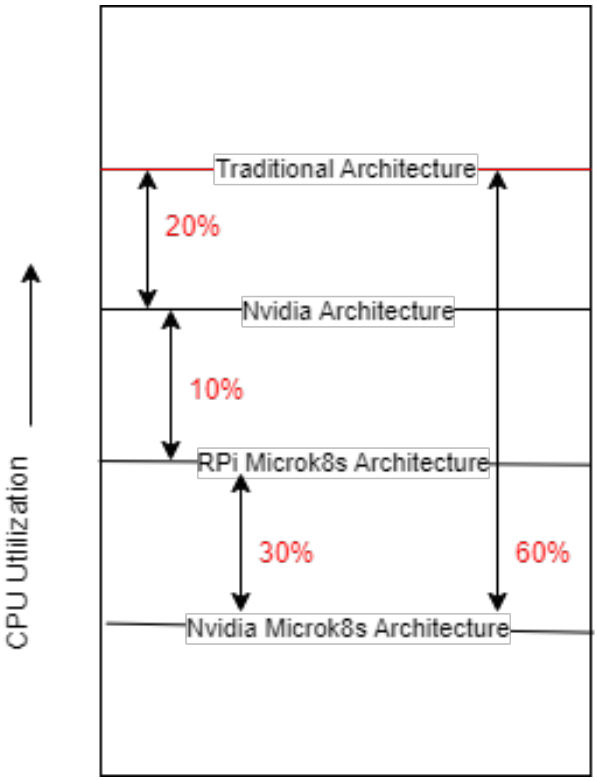


Figure 8. Validation of architecture.

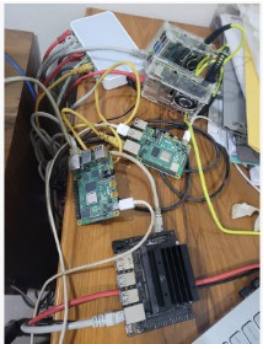
4. Design and Implementation

4.1. Cluster Setup

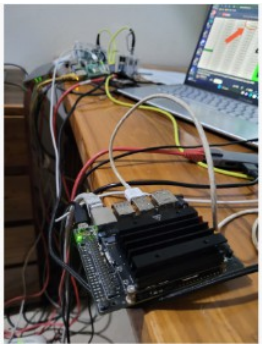
Two network structures were used for this research.

Raspberry Pi Cluster and Nvidia-Raspberry Pi Cluster

In the cluster shown in Figure 9a, four Raspberry Pis were used and one of them was made the master to handle operations in the Kubernetes environment. In a non-Kubernetes environment, the device was used to host the message queue Flask server and run the machine learning model.



(**a**) Raspberry Pi Cluster



(**b**) Nvidia-Raspberry Pi Cluster

Figure 9. Raspberry Pi Cluster and Nvidia-Raspberry Pi Cluster.

In the cluster shown in Figure 9b, three Raspberry Pis and one Nvidia Jetson Nano were used and one of the Raspberry Pis was made the master to handle operations in the Kubernetes environment. In a non-Kubernetes environment, the Nvidia device was used to host the message queue Flask server and run the machine learning model.

4.2. Docker

Docker [26] is a containerisation service which is used to build microservices. For all the microservices, i.e., the Flask server for the message queue, the ML model, and the data sending micro-services were converted into respective Docker images. These microservices all have Docker files in their directory. These Docker files can build using "docker build -t < name of the build >". Once the build is complete, the Docker image needs to be pushed to the Docker hub. An account on the Docker hub is required and three Docker hub repositories are required. For every repository, the Docker image built previously needs to be pushed. For doing so, first the image needs to be tagged against the path of the repository location on the Docker hub. For example, "docker tag < name of user >/ < name of online repository >:< tag > < name of the build >". Once this is completed, the "docker push < name of user >/< name of online repository >:<tag>" command needs to be executed. An example of building and pushing a Docker image of a Machine learning microservice to the Docker hub is shown in Figure 10. Before pushing the Docker image to the Docker hub, it is required to first authorize your push by login into your ID associated with the Docker repository. One last step before pushing the image is to associate the name of the Docker hub repository with the name of the local Docker build. This is achieved by the "docker tag" command.

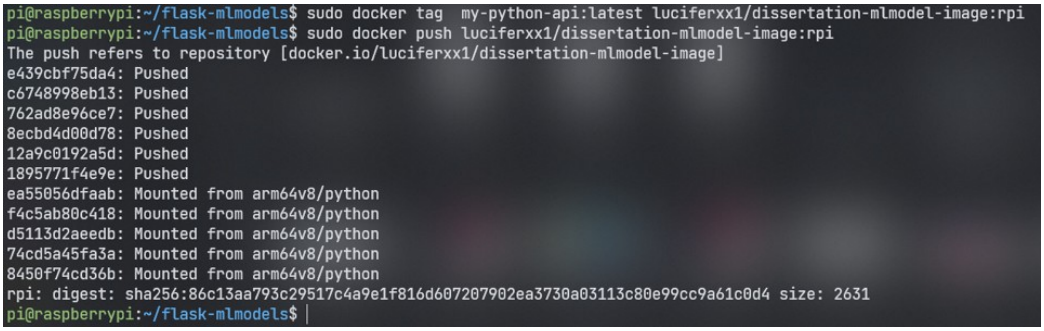


Figure 10. Docker tag and push.

4.3. Kubernetes Terminologies

Kubernetes is an open-source container orchestration platform used for automating the deployment, scaling, and management of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes provides a powerful and flexible framework for managing containerized applications across a cluster of machines [27].

4.3.1. Pods

The smallest deployable compute units that Kubernetes allows you to construct and control are called pods. A collection of one or more containers, with common storage and network resources, and a specification for how to execute the containers, is referred to as a "pod" (as in a pod of whales or peas). The components of a pod are always co-located, co-scheduled, and executed in the same environment. A pod includes one or more closely connected application containers and serves as a representation of an application-specific "logical host". Apps running on the same physical or virtual system are comparable to cloud apps running on the same logical host in non-cloud scenarios [28].

4.3.2. Deployment

A deployment is a crucial object in Kubernetes that controls the scaling and deployment of a collection of identical pods. One of the higher-level abstractions offered by Kubernetes to make managing containerized apps easier is this one. Using a deployment, you may provide the container image, the number of replicas (identical pods), and the update method to describe the intended state of your application [29].

4.3.3. Namespaces

A method for separating groupings of resources inside the same cluster is provided by namespaces. Within a namespace, but not between namespaces, resource names must be distinctive. Only namespace-based scoping is applicable to cluster-wide items, such as Storage Class, Nodes, and Persistent Volumes, not namespace-based objects (such as Deployments, Services, etc.) [30].

4.4. Design Flow

A visual representation of the workflow of the initial design is shown in Figure 11. The idea initially for the workflow about traditional implementation was to implement a complete DDoS happening environment where a real test on the devices could be done. The design goes as the record of performance metrics would be commenced to be written into a file followed by starting the Flask server on the master node. Once these steps are done, the worker nodes would be administered a SYN Flood attack from the Kali virtual machine running on the laptop. Simultaneously, a Docker container of cicflowmeter would be started.

CICFLowmeter 4.0 is a software which allows the network-related data stored in a PCAP format to be transformed into the CSV format of the network logs based on the physical interfaces of the device. This CSV data would be sent to the Message queue running on the Flask server. Once these messages are received, the machine learning script would be executed in a cronjob timed 1 min which would first check for availability of the logs, if there are any, it will start fetching until the message queue is empty. This keeps on looping until the Flask server is stopped or crashes due to some reason. Once the Flask server is stopped, the logs can be stopped from recording and stored.

The design described above could not be followed because CICFlowmeter is an archive library which has outdated dependencies. Even though those dependencies were configured, the program was not able to function as needed. The other option available was to use a Docker container but that did not work as well since the container would get crashed due to Java dependencies issues. So, the only way to use CICFlowmeter was to directly generate a PCAP file on a Windows computer and then utilise the CSV file generated which was not feasible in this research's use case because the cicflowmeter workflow needs to be automated.

Message Queue Implementation

The intuition behind having a message queue is to store the logs for the machine learning model to fetch from. For a message queue (MQ), a First In First Out (FIFO) implementation is required. A normal queue has a fixed size for the array. But for this use case of the message queue, the dynamic length of an array is required which can extend based on logs being inserted from worker nodes. Therefore, the Doubly Linked List becomes an eventually important choice as the list can be traversed bidirectionally and insertions and deletions can be performed from both the ends. Due to initial design issues, the CICFlowmeter was skipped. The dataset provided above sends data to the message queue. The message queue Flask server is executed after hardware performance metrics are recorded. After the Flask server gets requests, the script to send data is executed, and the message queue stores the data until the ML model script demands it. The ML model script checks if the message queue is active and then processes data from it. After using message queue data, the ML model scripts stop and run on a 1-min cronjob time. The above is presented in a visual form in the Figure 12.

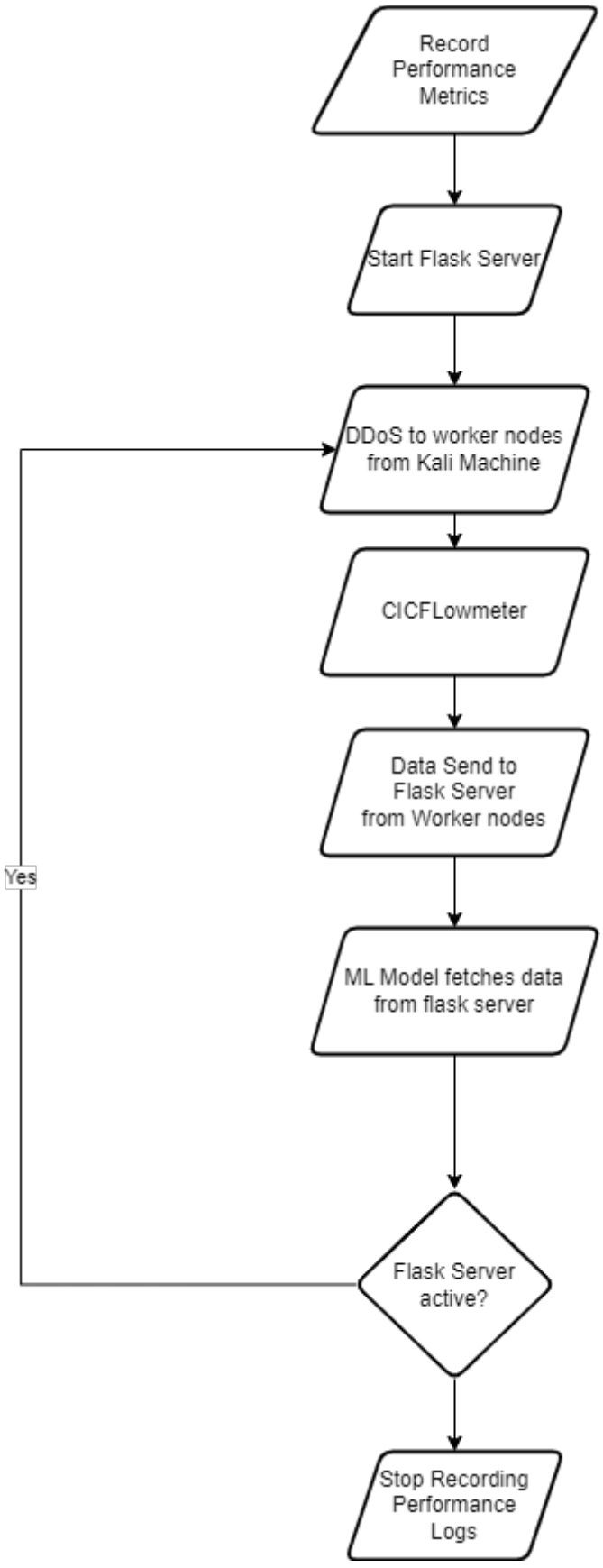


Figure 11. Initial design flow.

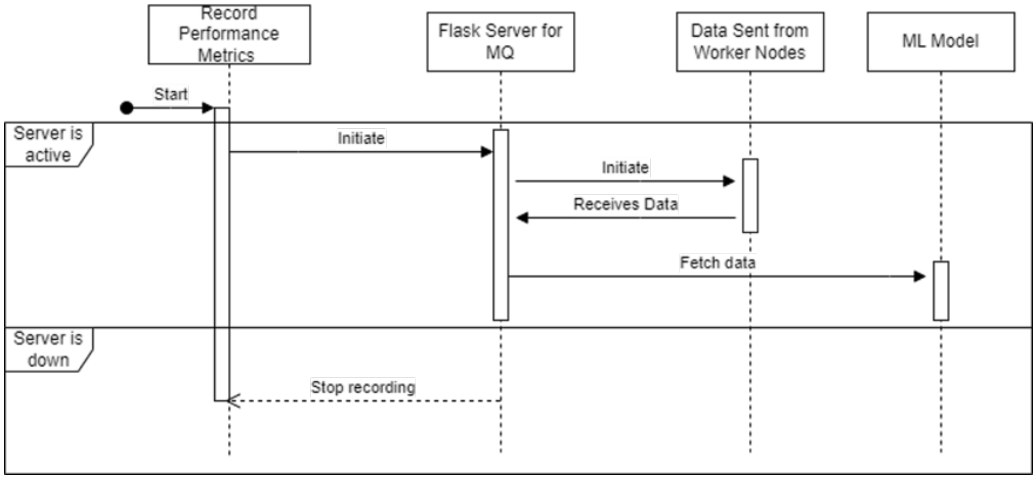


Figure 12. Workflow without Kubernetes.

For the Kubernetes cluster, different devices need to be first formed into a cluster. For that, one of the nodes is decided as a master node. The /etc/hosts file is updated with the hostname of the other devices and their respective static IP addresses. It is necessary that all the devices are connected to the internet and same access point. Once this is completed, the "microk8s add-node" run on the master node, which provides a connection string which needs to be executed on the other devices so that they know which is the master node. After this, additional services need to be enabled on the cluster. The services are as follows:

1. DNS (Domain Name Service): This is required for the pods to resolve the flask-app service internally when the machine learning/data-sending script is executed.
2. Metallb (Metal Load Balancer): This load balancer is required to load balance if there are multiple pods for flask service running.
3. Ingress.
4. GPU: This only applies to Nvidia-based hardware and given that the device should have an Nvidia GPU. The Microk8s detect the GPU and execute the tasks which require GPU if the deployment is launched.
5. Metrics-server: This is required for recording the performance metrics of the Kubernetes cluster. This service enquires about the utilisation of CPU and memory for every pod running in a particular namespace.

After finishing the instructions, start the pods. Before Flask deployment, the performance metrics recording script is run. Flask-app service must be checked after starting the flask pod. To verify the status of the flask-app service, use "kubectl get services". Data-send deployment can begin after the pod is up. When data-send pods are active, Flask receives data via internal domain name resolution. To verify requests, use "kubectl logs $<$ name of flask pod $>$". The name of the Flask pod may be obtained with "kubectl get pods | grep flask". After seeing the requests, the deployment file's cronjob option can perform and clock the machine learning model deployment every minute. Requests appear in Flask server logs after execution. This completes the flow shown in Figure 13. The monitor script can be stopped and the performance metrics can be noted.

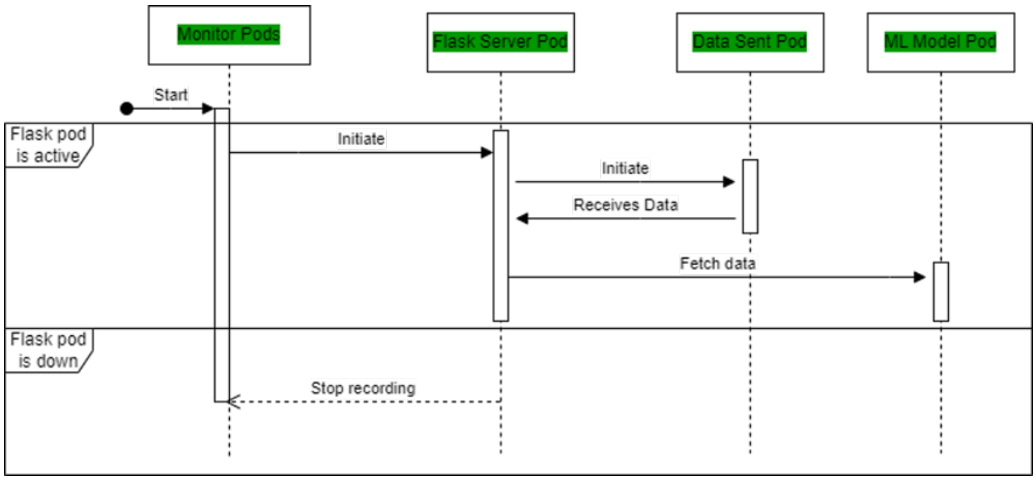


Figure 13. Kubernetes workflow diagram.

4.5. Implementation of Non-Microk8s Architecture

4.5.1. Performance Metrics

Figure 14 shows the execution of the command to start recording the performance parameters of CPU and Memory. For this, 'while true' is used to execute a never-ending loop until the script is stopped. echo "%CPU %MEM ARGS $(date)" is printed at the top of every output which is printed into the file.

```
pi@raspberrypi:~/logs$ echo "This is the terminal for Monitoring the CPU And Memory utilization"
This is the terminal for Monitoring the CPU And Memory utilization
pi@raspberrypi:~/logs$ while true; do (echo "%CPU %MEM ARGS $(date)" && ps -e -o pcpu,pmem,args --sort=pcpu | cut -d" " -f1-5
 | tail) >> ps.log; sleep 5; done
```

Figure 14. Performance metrics collection.

The provided Linux shell command is a pipeline that extracts and logs information about processes on the system with CPU usage. It begins by executing the ps command with the -e flag, which selects all processes running on the system. The -o pcpu,pmem,args flag customizes the output format, displaying CPU usage percentage, memory usage percentage, and the command with its arguments. The results are then sorted based on CPU usage using the –sort=pcpu flag. The cut command is used with the -d flag to extract the first five columns (CPU usage, memory usage, and command) from each line. The 'tail' command displays the last few lines of the sorted output, representing processes with the highest CPU usage. Finally, the » ps.log command appends the selected output to a file named "ps.log", providing users with a log of processes' CPU usage for analysis and future reference. And sleep 5 is used for recording entries every 5 s. This script can be stopped once the Flask server is killed or stopped. In the Flask framework, Figure 15 displays the transfer of information to the REST API server. The "Success" print presents the successful request to the REST API server.

```
pi2@raspberrypi2:~/csv-data-send$ python3 data-send.py
Success
Success
Success
Success
Success
```

Figure 15. Data sent execution in Flask framework.

4.5.2. Machine Learning Execution

Figure 16 displays the execution of the Machine Learning model on the Raspberry Pi platform. At the end of the verbose, the prediction is displayed. The implementation

of the Nvidia enabled architecture remains the same as mentioned for the Raspberry Pi. The machine learning model and the Flask script runs on the Nvidia device. In Figure 17, the Nvidia Tegra X1 is detected as a GPU because Tegra X1 contains the CPU and GPU inside it. The CPU's clock speed is displayed as 0.91 GHz. The memory of the device is displayed as 3.86 GB.

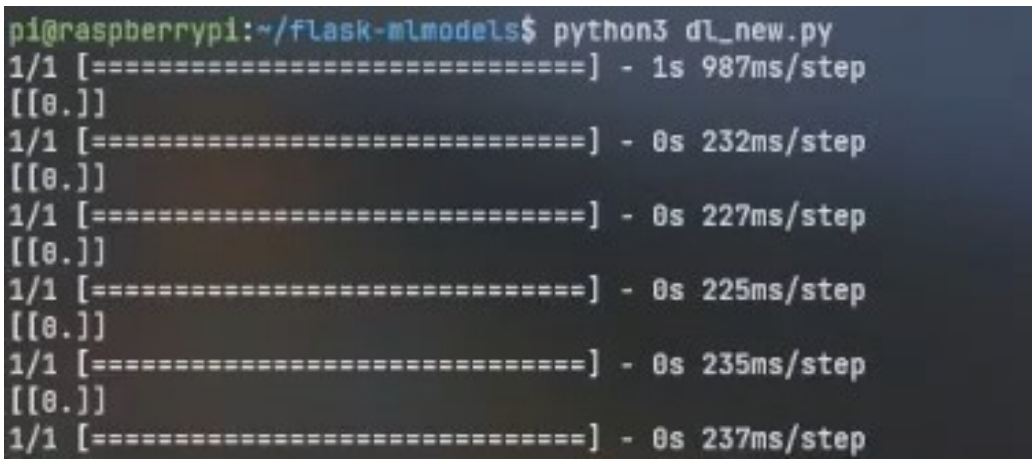


Figure 16. Machine learning script execution.

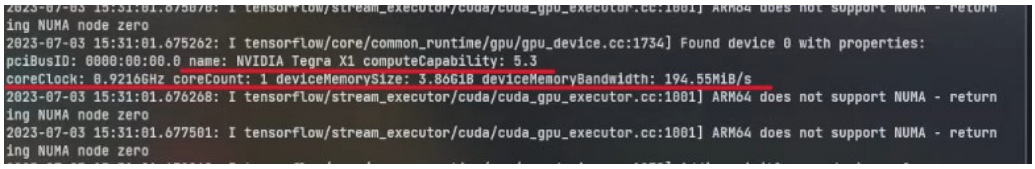


Figure 17. Nvidia machine learning script execution.

4.5.3. Reading Performance File

In Figure 18, 'cat' is used for reading the file of performance metrics. The output of this command is piped together with a word selection command called "grep". Grep is used to select those lines where the word occurs. For example, it shows the line where the word dl_new.py occurs.

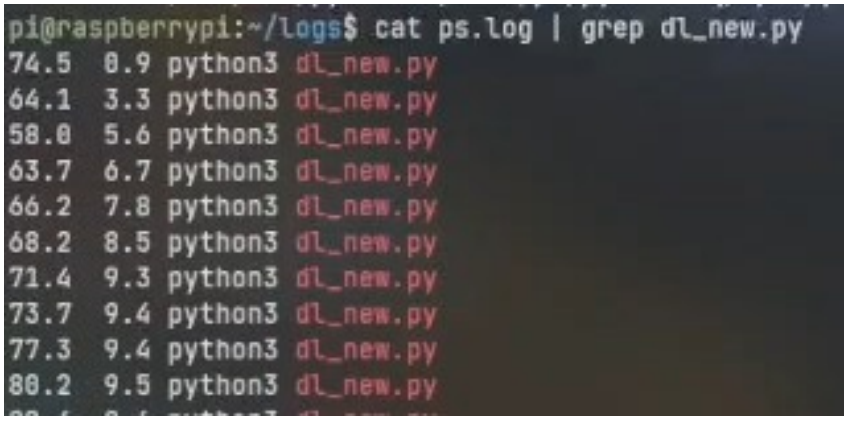


Figure 18. Viewing the performance metrics.

4.6. Implementation of Microk8s Architecture

In Figure 19, the nodes running in a cluster are enlisted. The image displays all the Raspberry Pis online at the moment when the command "kubectl get no" is executed. Figure 20 presents the execution of the deployment file called "flask-app-dep.yml".

```
pi2@raspberrypi2:~$ kubectl get no
NAME           STATUS   ROLES    AGE   VERSION
raspberrypi3   Ready    <none>   47m   v1.22.17-3+ac17f367a0fe12
raspberrypi1   Ready    <none>   46m   v1.22.17-3+ac17f367a0fe12
raspberrypi2   Ready    <none>   56m   v1.22.17-3+ac17f367a0fe12
raspberrypi    Ready    <none>   44m   v1.22.17-3+ac17f367a0fe12
```

Figure 19. "kubectl get no" command.

```
pi2@raspberrypi2:~/kubernetes-iot$ kubectl apply -f flask-app-dep.yml
deployment.apps/flask-app created
service/flask-app-service created
pi2@raspberrypi2:~/kubernetes-iot$
```

Figure 20. Flask app deployment.

Performance Metrics

For performance metrics to be recorded, the metrics-server add-on needs to be enabled on Microk8s. With this enabled, a shell script is used to get the output of the command "microk8s kubectl top pods" to receive the information of CPU and Memory utilisation into a file. This information gets recorded every 10 s while the script is running. Figure 21 shows the output of the command. In the output, the pulling image signifies the cluster is telling the Docker hub to fetch the Docker image while the statement above it resembles the successful image pull and the master node assigning the job to a node automatically if not specified otherwise.

```
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-4b297 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             False
  ContainersReady   False
  PodScheduled      True
Volumes:
  kube-api-access-4b297:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  37s   default-scheduler  Successfully assigned default/flask-app-54b868c566-lknhs to raspberrypi1
  Normal  Pulling    33s   kubelet            Pulling image "luciferxx1/flask-app-python:rpi"
```

Figure 21. Output of "kubectl describe po" command.

Figure 22 displays the output of the command "kubectl get pods". This command is followed by a "-o" flag with the keyword "wide" which displays additional information such as the node, IP address, Nominated Node and Readiness Gate. The image signifies the Flask pod in the creation stage and the pod is running on the raspberrypi1 node.

Figure 23 displays the IP addresses on which the Flask app can be accessed. Kubernetes allows domain name resolution which could be used by other pods to resolve "flask-app-service" into the corresponding internal cluster IP addresses.

```
pi2@raspberrypi2:~/kubernetes-iot$ kubectl get pods -o wide
NAME                   READY   STATUS              RESTARTS   AGE   IP       NODE          NOMINATED NODE   READINESS GATES
flask-app-54b868c566-lknhs   0/1     ContainerCreating   0          25s   <none>   raspberrypi1   <none>           <none>
```

Figure 22. "kubectl get pods" command.

```
pi2@raspberrypi2:~/kubernetes-iot$ kubectl get services
NAME                TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
kubernetes          ClusterIP      10.152.183.1    <none>          443/TCP          58m
flask-app-service   LoadBalancer   10.152.183.11   192.168.120.7   5001:30823/TCP   14s
```

Figure 23. "kubectl get services" command.

Therefore, no hard coding is required for the IP addresses in the scripts of the data-sending and machine-learning model. The external IP address is the IP address which is not the IP address of any node inside the cluster but a different address altogether which allows the "flask-service" to be accessed from the external world. Since this address does not correspond to any node, it provides extensive security to the nodes as they are hidden from the external world and thus remain protected from attacks. Also, the external IP address is provided by a load balancer called "metallb". This load balances the requests across the pods running the Flask app deployment.

Figure 24 displays the container logs of the pod running the Flask API microservice. It shows the Flask server is up and running. The 127.0.0.1:5001 is the local host API deployment which can be accessed from the pod itself while the 10.1.245.3:5001 is the IP address for the internal resolution to the other pods for accessing the service if required. Figure 25 shows the execution of the "csv-data-dep.yml". This YML file holds the name of the Docker image which needs to be sourced from the Docker hub. This deployment is responsible for sending the information to the Flask deployment.

```
pi2@raspberrypi2:~/kubernetes-iot$ kubectl logs flask-app-54b868c566-lknhs
 * Serving Flask app 'api'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://10.1.245.3:5001
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 920-895-473
```

Figure 24. Flask logs after pod deployment.

```
pi2@raspberrypi2:~/kubernetes-iot$ kubectl apply -f csv-data-dep.yml
deployment.apps/python-app-csv-data-send created
pi2@raspberrypi2:~/kubernetes-iot$
```

Figure 25. CSV data send deployment.

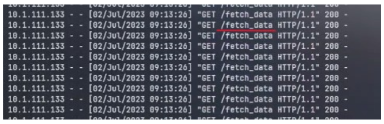
Figure 26 shows the logs of the Flask pod after the csv data-sending pods use the Flask server's endpoint to send the data. Once the request is served by the Flask, it logs into the output shown. Figure 27 shows the execution of the "ml-model-dep.yml". This YML file holds the name of the Docker image which needs to be sourced from the Docker hub. This deployment is responsible for retrieving the data from the Flask server for running the machine learning model.

```
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
192.168.120.1 - - [02/Jul/2023 08:41:14] "POST /insert_data HTTP/1.1" 200 -
```

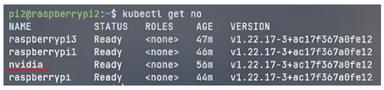
Figure 26. Insertion of logs after CSV pod deployment.

Figure 27. Machine learning pod deployment.

Figure 28a shows the fetch_data API endpoint used in the logs of Flask pod. It signifies that the Machine learning pod made requests to fetch the data. The implementation of the Nvidia Microk8s remains the same as mentioned above in the Raspberry Pi implementation of Microk8s but the only thing that changes is the static allocation of the machine learning pod on the Nvidia device. Therefore, this pod cannot be scheduled on any other node other than Nvidia. So, any number of pods which are scheduled will be handled by the Nvidia device. Figure 28b shows the Nvidia as a node in the Kubernetes cluster.



(**a**) Fetchdata API after ML pod deployment



(**b**) "kubectl get no" for Nvidia

Figure 28. Pod deployment.

Figure 29 shows the different pods running different micro-service but if observed for the machine learning pod, the pod is scheduled to run the Nvidia device, thereby utilizing the power of the GPU. The Nvidia deployment is different from the one used for Raspberry Pi. It includes the node Selector in the deployment script which points to the Nvidia hostname. So, the master node will look for the node with the hostname "nvidia" and launch the pod over that node.

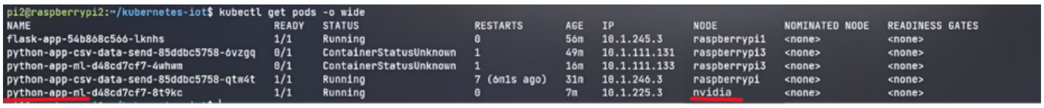


Figure 29. kubectl get pods (Nvidia machine learning).

4.7. Challenges

1. Microk8sSnap package manager has many Microk8s variants. This requires choosing the proper version for the use case. Initial use of the current version caused issues with the Kubelet API server and connecting devices through Microk8s. These flaws were also in the lower version. The author consulted the Microk8s community to locate a version that supported Prometheus, Grafana, and GPU add-ons. Despite version difficulties, all versions allowed the addition of devices and capabilities of Kubernetes provided through the Microk8s. The version suggested by them was 1.22.
2. Nvidia Jetson Nano Setup Finding the appropriate operating system was initially difficult due to the variety of versions and their features, some of which were outdated and unsupported by hardware. Operating system images were available for prior Nvidia chipsets. Nvidia forums and community helped choose the correct image. Nvidia crashed when a 16 GB Micro SD card was used because the image size was 14 GB after installation. Nvidia could no longer use the SD card for utility software. This was carried out later with a 64 GB card.
3. TFLite Model on Nvidia Jetson The TFLite model dependencies could not run on Nvidia Jetson due to the Tensorflow version being very different from other IoT device versions. This is because of the hardware architecture of Nvidia. Therefore, the H5 model is used because it is able to run on both platforms and allows a fair comparison between the traditional Raspberry Pi and Nvidia-enabled architecture.
4. Prometheus and Grafana These services are offered as add-ons to the Microk8s (version 1.22+). They are responsible for capturing resource utilisation of every pod in each namespace or node in the cluster. Prometheus and Grafana were initially tested on the

cluster. Still, the resource utilisation by the Prometheus namespace was high which was crashing the Microk8s service on the nodes when the other deployments were being initiated. Therefore, it is suggested to have at least 8 GB of RAM on the node's hardware.

5. Evaluation

5.1. Machine Learning

The classes which are classified as 0 are the DDoS Packets while 1 indicates the normal packets.

5.1.1. Dummy Classifier Performance

In Figure 30a, The F1 score for class 0 (DDoS) is predicted as 1.00 because it has samples with the highest frequency. This is supported by the reason that the strategy for the dummy classifier was set as the most frequent. The F1 score of class 1 is 0 because it has fewer samples as compared to class 0.

5.1.2. Logistic Regression Performance

In Figure 30b, The F1 score for the benign class improves more than when it was previously 0 in the baseline model. There is an improvement over there.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	428260
1	0.00	0.00	0.00	319
accuracy			1.00	428579
macro avg	0.50	0.50	0.50	428579
weighted avg	1.00	1.00	1.00	428579

(**a**) Dummy Classifier Performance

	precision	recall	f1-score	support
0	1.00	1.00	1.00	428260
1	0.92	0.92	0.92	319
accuracy			1.00	428579
macro avg	0.96	0.96	0.96	428579
weighted avg	1.00	1.00	1.00	428579

(**b**) Logistic Regression Performance

Figure 30. Dummy Classifier and Logistic Regression Performance.

5.1.3. Neural Network Performance

In Neural Network, a regularized architecture has been defined as shown in Figure 31. There is a 3% improvement as compared to the F1 score of class 1 in Logistic Regression Performance.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	428260
1	0.92	0.98	0.95	319
accuracy			1.00	428579
macro avg	0.96	0.99	0.98	428579
weighted avg	1.00	1.00	1.00	428579

Figure 31. Neural Network Performance.

Given all of this above architecture, the prediction for class 0 is an absolute 100 because of the biased samples in the dataset. Even if the undersampling is carried out, the relative samples for class 1 would still be undermined. Therefore, an improvement in such scenarios is difficult in this kind of unbalanced situation. Although, the model can be improved when the real data come into the IoT devices by retraining the network in real-time. Apart from the DDoS attacks, the proposed architecture can also run detections for example, the Man in the Middle attacks and anomaly detection. Abdelkader et al. [31] have conducted MITM detection on the IoT devices. Using the same methodology, the models can be launched in the proposed architecture as well. Ibrahim et al. [32] show that the Anomaly detection can be done on the network traffic and associated fog layers. Therefore, these examples explain that the machine learning models are classification problems mostly and the DDoS model used in the current study is similar to that. Therefore, the resilience noticed with the current results should be similar whenever other cyber threat detection runs on the proposed architecture.

5.2. Architecture

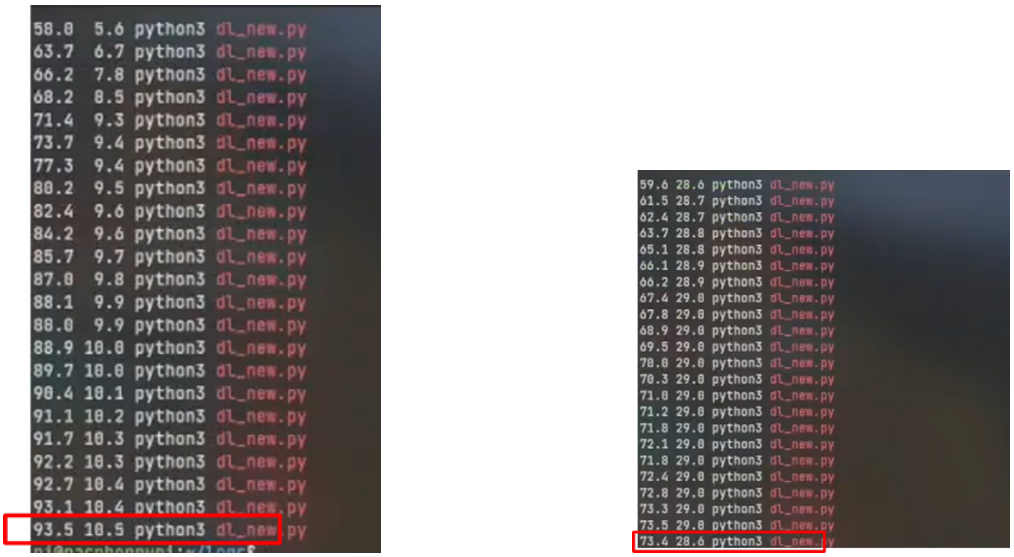
In Table 3, the CPU and memory utilisation is displayed for all the architectures obtained from the experiments. The data in the table are sourced from the experiments conducted and the evidence has been provided below in the figures. Looking at the table, it is observable that the Non-Microk8s architecture has the highest CPU utilisation compared to the others. The CPU utilisation drops down to 73% when the Nvidia-enabled architecture is brought in. There is definitely an improvement over there which showcases one of the reasons where the GPU assists the CPU in running the machine learning models in an efficient manner. Having looked at the memory utilisation, it is increasing because Nvidia uses a Graphic-based User Interface of Ubuntu which also runs other System operations as well whereas the memory utilisation is less because of the server image of Ubuntu being used.

Discussing the Microk8s architectures, the Raspberry Pi architecture performs better in terms of both CPU and Memory utilisation as compared to non-microk8s architecture and, interestingly, 10% less than the Nvidia Non-Microk8s architecture. Microk8s does a good job of managing the resources. While looking at the state-of-the-art architecture where the Nvidia and Microk8s have been used, a CPU utilisation of 31.5% is noticed which is 60% of the traditional architecture.

Table 3. CPU and memory usage comparison.

	Non-Microk8s (H5 Model)		**Microk8s (TFLite Model)**	
	Rpi (Traditional)	Nvidia-Rpi	Rpi	Nvidia-Rpi
CPU (%)	93.5	73.4	64.2	31.5
Memory (%)	10.5 (420 MB)	28.6 (1120 MB)	0.0007 (3 MB)	0.01 (73 MB)

Figure 32a, displays the CPU utilisation in the leftmost column while the column beside it displays the memory utilisation of the Raspberry Pi traditional architecture. The 'dl_new.py' is the file which holds the logic to fetch the data from the message queue and run the machine learning model once the data are available in the memory. The image, therefore, shows the continuous CPU consumed with time and a constant rate of increase.



(**a**) Raspberry Pi traditional architecture performance (**b**) Nvidia Enabled Architecture Performance

Figure 32. Raspberry Pi Traditional & Nvidia Enabled Architecture Performance.

Figure 32b presents the CPU utilisation in the leftmost column while the column beside it displays the memory utilisation of the Nvidia-enabled architecture. The Nvidia-enabled

architecture starts with a similar amount of CPU utilisation when the script initiates but over time the difference between traditional and this architecture grows larger. In the end, the final CPU utilisation is observed to be 73.4%.

Figures 33 and 34 presents the name of the pods, the CPU utilisation and memory utilisation for the Raspberry Pi Microk8s architecture & Nvidia-Raspberry Pi Microk8s Architecture respectively. In the red box, the machine learning pod can be seen with the CPU utilisation and Memory utilisation beside it. The CPU utilisation is 64% & 31.5% respectively.

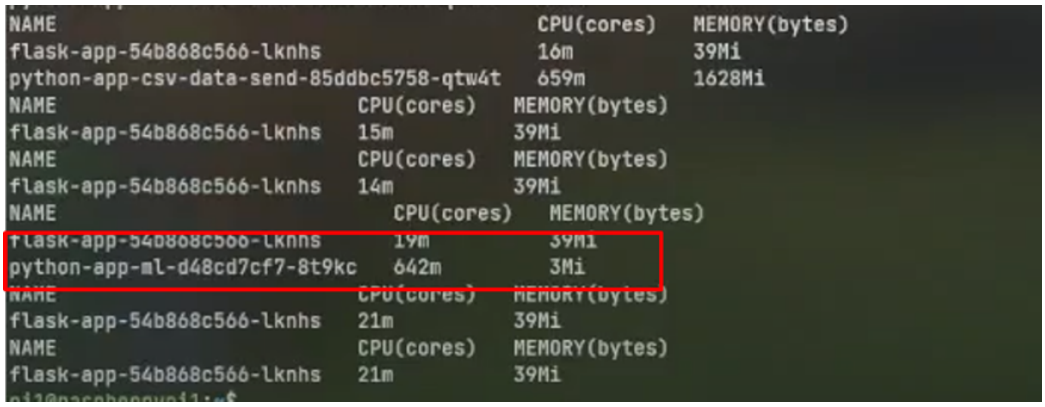


Figure 33. Raspberry Pi Microk8s performance.

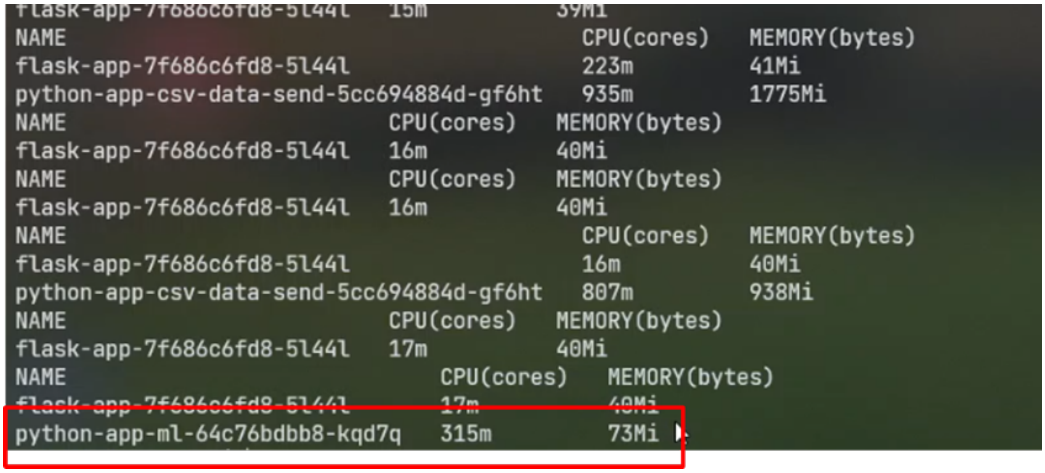


Figure 34. Nvidia Raspberry Pi Microk8s performance.

5.2.1. Energy Consumption

Going by the energy formula, $Energy\ (Wh) = Power\ (W) * Time\ (h)$, the energy consumed by the Nvidia Jetson Nano can be calculated. For three raspberry Pis sending the logs to the machine learning model running on Nvidia Jetson Nano, having the Jetson nano configured to use the maximum available power and also considering the Kubernetes running on the Jetson, the maximum power taken by the Jetson at any particular time is 10 W. We only considered the window when the machine learning model is executing. The time taken was approximately 1 min for all the device's logs to be transferred and processed. In that particular case, the energy consumed is 0.016 h * 10 W = 0.16 Wh. Therefore, on increasing the devices in the edge network, the number of logs transferred to the Nvidia Jetson Nano also increases. As said before, the maximum power is 10 W for operating the nvidia Jetson nano. The energy consumed would be proportional to the total time taken by the machine learning model to be executed. The total power consumed by any one of the Raspberry Pis in the network depends on the role of the Pi device along with transferring the logs. The power consumption can vary in a real world scenario. In our particular case, it was designed to transfer network logs to the Nvidia Jetson Nano which involved running a python script and Kubernetes running on the device. The execution time of the python

script was 15 s (0.25 min). The power consumed at any particular instant was reported to be 6 W. Therefore, the energy consumed by the device during this particular operation was 6 * (0.25/60) = 0.025 Wh and for 3 devices, it is 0.075 Wh. Thus, the total the energy consumed during the machine learning model execution was approximately 0.235 Wh.

5.2.2. Scalability and Latency

The scalability is measured based on the number of IoT devices (non-AI) which can be added to the network without having the Nvidia Jetson Nano (AI IoT Device) being overwhelmed in terms of CPU utilization. In the current architecture where there were 3 raspberry pi devices, 31.5% of utilization is reported. When we increase the number of devices, the operation of the Nvidia Jetson Nano is not affected much. It can run multiple neural networks together and is able to handle concurrent requests from different nodes. The CPU utilization increases to 40% when the number of devices are more than 10. Whereas latency is between 10ms to 15ms on varying loads and with an increasing number of devices. Latency involved in this particular experiment would be ideal to be observed for the logs being transferred between the raspberry Pi and the Nvidia Jetson Nano. Theoretically speaking, the latency is said to be the time difference of when the data packet leaves the source device and at the time when the data packet was received at the destination device. In this experiment, the devices were connected through LAN cables over a Gigabit switch. Given this kind of network setup between the devices, the latency between the devices would be low as compared to a wide area network setup. For example, the latency in transporting the logs between the Raspberry Pi and cloud would definitely be higher than the setup in this experiment. While looking at the metrics of the experiment, the latency at any particular time when the packets were transferred were close to 10 ms. This is in the case of three devices transferring the logs to the flask server. This would be more or less the same on increasing the devices since the physical memory is utilized on the Nvidia Jetson Nano to store the incoming logs into a message queue. A comparative analysis of latency with the existing state-of-the-art is given in Table 4.

Table 4. Comparative analysis of latency.

Ref	[33]	[34]	[35]	[36]	[37]	[38]	Proposed
Latency	18 ms	400 ms	2 s	2 s	2 s–5 s	25–50 ms	10–15 ms

5.2.3. Efficiency, Security and Cost Effectiveness

A comparative analysis of efficiency, security and cost effectiveness of the proposed model with the existing state-of-the-art is given in Table 5.

Table 5. Efficiency, security, and cost effectiveness.

Ref.	Efficiency	Security	Cost Effective
[17]	The study in this research evolved around raspberry Pis and how different light versions of Kubernetes perform on the cluster of raspberry Pis. There was no load test to understand the efficiency of the architecture.	Kubernetes is being used in the architecture of this paper but it is not leveraging any security benefits from the Kubernetes.	It's just using Raspberry Pis with Kubernetes running on them but 20 of them for the research makes the experiment expensive.
[15]	This research aimed to test the lightweight Kubernetes solutions for IoT devices. The idea was to perform a formal comparison between them. Efficiency was looked at the best distribution which can be used.	The study utilized Azure VMs to test their architecture. The security associated with the cloud resources is better than doing so on the IoT devices.	Although cloud resources are better than IoT devices, they come along with costs. They prove to be way more costly if not configured properly.

Table 5. *Cont.*

Ref.	Efficiency	Security	Cost Effective
Our	When it comes to efficiency, the Kubernetes deployment mechanism and distribution of workloads is state-of-the-art. It distributes the workloads efficiently with the distribution of processing between different pods. While considering use of AI IoT devices, the performance of hardware given its GPU provides enhanced efficiency.	In terms of security, all of the data processing is performed on the AI-IoT device, which is not exposed to the external world. They function in their own private subnet of the edge network. Kubernetes ensures the segregation of the network packets are maintained between the devices from the external world.	The AI-IoT device such as the Nvidia Jetson Nano are a bit expensive as compared to the costs of the Raspberry pi but the abilities they provide as seen in this particular research showcases their effectiveness for the price. Google's Coral device can also be an alternative which can be used along with raspberry Pis.

5.3. Real-World Applications

There are potential applications of such research in the industry as well. For example, when smart cities are considered, there are traffic lights being used at every traffic junction and their functioning is being scaled up using CCTV cameras and computer vision. To keep their functionality working properly without any security breaches to avoid traffic jams, the general architecture shown before is where computer vision can run directly at the junction without transferring the data to the nearby data centre. Meghana et al. [39] developed a method in which aRaspberry Pi along with a Pi Camera is used to capture events at a traffic junction and image processing of the data happens on the device. This research showcases that these kind of jobs are viable but the problem comes down to the overwhelming of the Pi devices. With the proposed architecture in this research, it would be a game changer since the job of image processing would be done by AI IoT devices which are capable of running multiple neural network operations at once. All the image streams from these Raspberry Pis can be provided to AI IoT devices and hence more operations can be done at once. The practical challenges associated with using the architecture is that the devices would be used outside on the junction which comes along with weather challenges and devices are supposed to operate within a desired temperature range. Especially using Nvidia devices, they are fragile to varying temperatures and would affect its performance. Regular maintenance is a minor one but it has to be carried out regularly given the device's physical condition.

Another example is the application in the field of medical industry, where hospitals in Ireland for instance are providing wearable devices to patients to record their health metrics while they are being transferred to another location in the hospital or a situation where they cannot be monitored with specialized equipment. The protection of data that the wearable stores should be prioritized. In such cases, a local deployment of a general architecture could be useful. There can be much more instances where the use of such an architecture will be of great utility. In this particular research [40], the idea seems similar to what is being discussed here. The paper presents the idea of the wearable IoT devices in the sphere of hospitals and how their communications are modelled to transmit data to the cloud. This paper does not discuss any cyber challenges associated with this transmission but what seems obvious is that there are multiple layers of communications involved until it reaches the central servers. The proposed architecture overcomes the challenge of going through multiple layers of communication until it reaches the cloud servers. All the processing is offloaded on the AI IoT device while storage of data can be carried out locally at the hospitals. Practical Challenges associated with this model would be implementing redundancy and failover mechanisms that become essential to prevent data loss or system downtime. In case, let us say, the hospital decides to keep data in the cloud, outages may disrupt the communication between wearable devices and central systems. The major challenge of all would be to ensure that systems remain robust even when hospital is dealing with large volume of patients.

6. Conclusions

This research investigates distributed computing on the edge, using AI-enabled IoT devices and container orchestration tools to handle data in real time. Security is improved by identifying and mitigating threats like DDoS attacks while minimizing CPU usage. It compares typical IoT devices with and without AI-enabled chips, container orchestration, and machine learning model performance in different cluster settings. By enabling IoT devices to process data locally, the suggested design seeks to reduce reliance on cloud transmission and improve IoT environment security. Results align with architecture update. The following statements are concluded after looking at the results:

1. Comparison Of Raspberry Pi to Microk8s Raspberry Pi: Reduction of almost 30% (CPU) and 99% (Memory).
2. Comparison Of Nvidia—Raspberry Pi to Microk8s Nvidia—Raspberry Pi: Reduction of almost 42% (CPU) and 98% (Memory).
3. Comparison of Raspberry Pi and Nvidia-Raspberry Pi: Reduction of almost 21% (CPU) and Increase of 62.5% (Memory).
4. Comparison of Microk8s Architecture—Raspberry Pi vs. Nvidia-Raspberry Pi: Reduction of almost 32.5% (CPU) & Increase of 95% (Memory).
5. An overall decrease of 60% in CPU utilisation from the traditional architecture of Rasp berry Pi to Microk8s architecture with Nvidia.
6. Container Orchestration-as-a-solution: It managed resources efficiently, auto-scaled when needed, and scheduled jobs well. Since the solutions are not directly distributed to the host, malware implantation can be stopped by restarting another pod for that deployment, making it security-reliable.

Future Work

1. Use of Google Coral Device: The Google gadget has a Tensor Processing Unit. Eventually, this gadget and Raspberry Pi can be compared to Nvidia Jetson Nano for performance analysis. This can be clustered or standalone.
2. Realtime updating of Machine Learning Model: When data become available on the platform, cloud updating of the machine learning model affects the model's real-time performance reliability. If this could be conducted on the edge with CPU optimization in mind, imagine what IoT devices could achieve with merely data coming in and the model being updated in real time without sending it to the cloud.
3. Adversarial AI for Malware Detection: Attackers must escape IoT devices after planting botnets to avoid leaving cyber fingerprints. Adversarial AI that predicts evasion and avoids malware plants. This would preserve the attackers' fingerprints and make tracking malware to its source easier. Generative Adversarial Network implementation in malware detection is also receiving interest.

Looking at the above results and future work, there is tremendous potential which can be unlocked with the introduction of Kubernetes and AI-enabled IoT devices to the existing network of IoT devices. This will yield better product line-ups for industries in this business. With the scale at which IoT devices grow, these solutions will be able to keep security growing as well.

Author Contributions: Conceptualization, S.K. and H.T.; methodology, S.K.; software, V.T.; validation, S.K., H.T. and V.T.; formal analysis, S.K.; investigation, H.T.; resources, S.K.; data curation, V.T.; writing—original draft preparation, S.K.; writing—review and editing, S.K and H.T.; visualization, V.T.; supervision, H.T.; project administration, H.T.; funding acquisition, H.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research was conducted with the financial support of Ripple.com under their University Blockchain Research Initiative (UBRI) and the Science Foundation Ireland at ADAPT, the SFI Research Centre for AI-Driven Digital Content Technology at Trinity College Dublin [13/RC/2106_P2].

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Merenda, M.; Porcaro, C.; Iero, D. Edge machine learning for ai-enabled iot devices: A review. *Sensors* **2020**, *20*, 2533. [CrossRef] [PubMed]
2. Ghosh, A.; Chakraborty, D.; Law, A. Artificial intelligence in Internet of things. *CAAI Trans. Intell. Technol.* **2018**, *3*, 208–218. [CrossRef]
3. Covi, E.; Donati, E.; Liang, X.; Kappel, D.; Heidari, H.; Payvand, M.; Wang, W. Adaptive extreme edge computing for wearable devices. *Front. Neurosci.* **2021**, *15*, 611300. [CrossRef] [PubMed]
4. Fayos-Jordan, R.; Felici-Castell, S.; Segura-Garcia, J.; Lopez-Ballester, J.; Cobos, M. Performance comparison of container orchestration platforms with low cost devices in the fog, assisting Internet of Things applications. *J. Netw. Comput. Appl.* **2020**, *169*, 102788. [CrossRef]
5. Taylor, R.; Baron, D.; Schmidt, D. The world in 2025-predictions for the next ten years. In Proceedings of the 2015 10th International Microsystems, Packaging, Assembly and Circuits Technology Conference (IMPACT), Taipei, Taiwan, 21–23 October 2015; IEEE: Piscataway, NJ, USA , 2015; pp. 192–195.
6. Wu, H.; Han, H.; Wang, X.; Sun, S. Research on artificial intelligence enhancing internet of things security: A survey. *IEEE Access* **2020**, *8*, 153826–153848. [CrossRef]
7. Shakdher, A.; Agrawal, S.; Yang, B. Security vulnerabilities in consumer iot applications. In Proceedings of the 2019 IEEE 5th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS), Washington, DC, USA, 27–29 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1–6.
8. Dinculeană, D.; Cheng, X. Vulnerabilities and limitations of MQTT protocol used between IoT devices. *Appl. Sci.* **2019**, *9*, 848. [CrossRef]
9. Pokhrel, S.; Abbas, R.; Aryal, B. IoT security: Botnet detection in IoT using machine learning. *arXiv* **2021**, arXiv:2104.02231.
10. Alrowaily, M.; Lu, Z. Secure edge computing in IoT systems: Review and case studies. In Proceedings of the 2018 IEEE/ACM Symposium on Edge Computing (SEC), Seattle, WA, USA, 25–27 October 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 440–444.
11. SmartDefense: A distributed deep defense against DDoS attacks with edge computing. *Comput. Netw.* **2022**, *209*, 108874. [CrossRef]
12. Bhardwaj, K.; Miranda, J.C.; Gavrilovska, A. Towards {IoT-DDoS} Prevention Using Edge Computing. In Proceedings of the USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18), Boston, MA, USA, 11–13 July 2018.
13. Mirzai, A.; Coban, A.Z.; Almgren, M.; Aoudi, W.; Bertilsson, T. Scheduling to the Rescue; Improving ML-Based Intrusion Detection for IoT. In Proceedings of the 16th European Workshop on System Security, Rome, Italy, 8 May 2023; pp. 44–50.
14. Beltrão, A.C.; de França, B.B.N.; Travassos, G.H. Performance Evaluation of Kubernetes as Deployment Platform for IoT Devices. In Proceedings of the Ibero-American Conference on Software Engineering, Curitiba, Brazil, 4–8 May 2020.
15. Koziolek, H.; Eskandani, N. Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift. In Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering, Coimbra, Portugal, 15–19 April 2023; pp. 17–29.
16. Hayat, R.F.; Aurangzeb, S.; Aleem, M.; Srivastava, G.; Lin, J.C.W. ML-DDoS: A blockchain-based multilevel DDoS mitigation mechanism for IoT environments. *IEEE Trans. Eng. Manag.* **2022**, 1–14. [CrossRef]
17. Todorov, M.H. Deploying Different Lightweight Kubernetes on Raspberry Pi Cluster. In Proceedings of the 2022 30th National Conference with International Participation (TELECOM), Sofia, Bulgaria, 27–28 October 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 1–4.
18. Ferdowsi, A.; Saad, W. Generative adversarial networks for distributed intrusion detection in the internet of things. In Proceedings of the 2019 IEEE Global Communications Conference (GLOBECOM), Waikoloa, HI, USA, 9–13 December 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1–6.
19. Elsayed, M.S.; Le-Khac, N.A.; Dev, S.; Jurcut, A.D. Ddosnet: A deep-learning model for detecting network attacks. In Proceedings of the 2020 IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM), Cork, Ireland, 31 August–3 September 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 391–396.
20. Kannavara, R.; Gressel, G.; Fagbemi, D.; Chow, R. A Machine Learning Approach to SDL. In Proceedings of the 2017 IEEE Cybersecurity Development (SecDev), Cambridge, MA, USA, 24–26 September 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 10–15.
21. Bapat, R.; Mandya, A.; Liu, X.; Abraham, B.; Brown, D.E.; Kang, H.; Veeraraghavan, M. Identifying malicious botnet traffic using logistic regression. In Proceedings of the 2018 Systems and Information Engineering Design Symposium (SIEDS), Charlottesville, VA, USA, 27 April 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 266–271.
22. Ma, L.; Chai, Y.; Cui, L.; Ma, D.; Fu, Y.; Xiao, A. A deep learning-based DDoS detection framework for Internet of Things. In Proceedings of the ICC 2020—2020 IEEE International Conference on Communications (ICC), Dublin, Ireland, 7–11 June 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–6.

23. Debauche, O.; Mahmoudi, S.; Guttadauria, A. A new edge computing architecture for IoT and multimedia data management. *Information* **2022**, *13*, 89. [CrossRef]
24. Süzen, A.A.; Duman, B.; Şen, B. Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn. In Proceedings of the 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), Ankara, Turkey, 26–28 June 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–5.
25. Gizinski, T.; Cao, X. Design, Implementation and Performance of an Edge Computing Prototype Using Raspberry Pis. In Proceedings of the 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 26–29 January 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 0592–0601.
26. Docker Docs. Docker Overview. 2023. Available online: https://docs.docker.com/get-started/overview/ (accessed on 25 July 2023).
27. Redhat. "What is kuberentes?". 2023. Available online: https://www.redhat.com/en/topics/containers/what-is-kubernetes (accessed on 25 July 2023).
28. Kubernetes. "Pods". 2023. Available online: https://kubernetes.io/docs/concepts/workloads/pods/ (accessed on 25 July 2023).
29. Kubernetes. "Deployment". 2023. Available online: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (accessed on 25 July 2023).
30. Kubernetes. "Namespaces". 2023. Available online: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ (accessed on 25 July 2023).
31. Lahmadi, A.; Duque, A.; Heraief, N.; Francq, J. MitM attack detection in BLE networks using reconstruction and classification machine learning techniques. In Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Bilbao, Spain, 14–18 September 2020; Springer: Cham, Switzerland, 2020; pp. 149–164.
32. Alrashdi, I.; Alqazzaz, A.; Aloufi, E.; Alharthi, R.; Zohdy, M.; Ming, H. Ad-iot: Anomaly detection of iot cyberattacks in smart city using machine learning. In Proceedings of the 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 7–9 January 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 0305–0310.
33. Atutxa, A.; Franco, D.; Sasiain, J.; Astorga, J.; Jacob, E. Achieving low latency communications in smart industrial networks with programmable data planes. *Sensors* **2021**, *21*, 5199. [CrossRef] [PubMed]
34. Ferrari, P.; Sisinni, E.; Brandão, D.; Rocha, M. Evaluation of communication latency in industrial IoT applications. In Proceedings of the 2017 IEEE International Workshop on Measurement and Networking (M&N), Naples, Italy, 27–29 September 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 1–6.
35. Cui, L.; Xu, C.; Yang, S.; Huang, J.Z.; Li, J.; Wang, X.; Ming, Z.; Lu, N. Joint optimization of energy consumption and latency in mobile edge computing for Internet of Things. *IEEE Internet Things J.* **2018**, *6*, 4791–4803. [CrossRef]
36. Azari, A.; Stefanović, Č.; Popovski, P.; Cavdar, C. On the latency-energy performance of NB-IoT systems in providing wide-area IoT connectivity. *IEEE Trans. Green Commun. Netw.* **2019**, *4*, 57–68. [CrossRef]
37. Javed, A.; Malhi, A.; Kinnunen, T.; Främling, K. Scalable IoT platform for heterogeneous devices in smart environments. *IEEE Access* **2020**, *8*, 211973–211985. [CrossRef]
38. Badiger, S.; Baheti, S.; Simmhan, Y. Violet: A large-scale virtual environment for internet of things. In Proceedings of the Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, 27–31 August 2018; Proceedings 24; Springer: Cham, Switzerland, 2018; pp. 309–324.
39. Meghana, V.; Anisha, B.; Kumar, P.R. IOT based Smart Traffic Signal Violation Monitoring System using Edge Computing. In Proceedings of the 2021 2nd Global Conference for Advancement in Technology (GCAT), Bangalore, India, 1–3 October 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1–5.
40. Surantha, N.; Atmaja, P.; Wicaksono, M. A review of wearable internet-of-things device for healthcare. *Procedia Comput. Sci.* **2021**, *179*, 936–943. [CrossRef]