Agile AI and Firmware Management in IoT: DevOps for Low-Power Microcontroller-based Platforms

Mohammad AlSelek, Jose M. Alcaraz-Calero, Qi Wang
School of Computing, Engineering & Physical Sciences
University of the West of Scotland, United Kingdom
{mohammad.alselek, jose.alcaraz-calero, qi.wang}@uws.ac.uk

Abstract—In this paper, we introduce a Development and Operations (DevOps)-based IoT system tailored for the dynamic management of firmware and AI models across distributed IoT environments. The system offers scalability, resource efficiency, and cost-effectiveness for updating the AI model and firmware on the IoT devices without a need for human intervention. To accomplish this, we have developed a continuous integration and continuous deployment (CI/CD) pipelines that operate across multiple platforms, leveraging the capabilities of Kubernetes and GitLab-Runner. Moreover, the system is specifically designed for Microcontroller-based low-power devices capable of running tiny AI models. The new deployment is sent to the IoT devices despite their location to start interacting with the surrounding environment and perform predictions regarding its application. Through empirical experiments, we demonstrate the system effectiveness with promising results in terms of scalability, resource utilization, and deployment efficiency.

Keywords—IoT, AI, DevOps, Kubernetes, GitLab, Microservices, Microcontroller, Micropython.

I. INTRODUCTION

The integration of Artificial Intelligence (AI) applications with Internet of Things (IoT) devices is rapidly gaining momentum. Projections indicate a staggering rise in the global count of IoT devices, expected to reach nearly 30 billion by 2030 [1], alongside significant annual growth in the AI market size, projected at 15.83% between 2024 and 2030 [2].

There are ongoing efforts within the domain of DevOps to automate the implementation and delivery of AI models, while initiatives in DevOps for IoT focus on automating the deployment of microservices across distributed IoT networks, leveraging technologies such as Kubernetes [3], Docker [4], Helm [5], and others. This approach offers a promising solution to dynamically updating multiple AI models and firmware within distributed cloud-native ecosystems by integrating DevOps practices for AI. It can enhance cost-effectiveness and operational efficiency compared to traditional methodologies [6]. However, agile AI and firmware management in IoT devices remain insufficiently explored.

In this paper, we present a new system utilizing CI/CD pipelines powered by GitLab-runners. These pipelines automate essential processes, including building, testing, packaging, releasing images, and deploying final applications into a distributed IoT cluster. The system comprises four GitLab repositories: Firmware Development, Data Acquisition, AI Training, and AI Optimization. Notably, the firmware development repository can independently run its pipeline. During the AI model update process on an IoT device, data collection begins with the IoT device, transmitting data to the AI training repository for dataset curation. Subsequently, the AI optimization pipeline refines the trained AI model, preparing it for direct deployment to the target IoT device.

The proposed system has a structure based on a Kubernetes cluster, where a master manages workers installed on mini-PCs, i.e. Raspberry Pi 4 Model B. This setup handles tasks such as collecting data, deploying AI models, and updating firmware. Workers handle updating IoT devices with data and AI models using RESTful API, while firmware updates use RESTful API along with Over-The-Air (OTA) mechanism. IoT devices are Pycom FiPy motes.

The proposed system can easily scale to work with multiple IoT devices in different places. It also provides feedback to developers through logs, making it easier to fix problems quickly and avoid deployment failures. The proposed system offers several advantages and contributions, including:

- Cost-effective software lifecycle: Applying DevOps practices for continuous AI models and firmware updates.
- Highly adaptable environment: The system allows multiple IoT devices to be deployed in various locations.
- Optimized resource utilization: The system enables dynamic scaling of deployments, efficiently allocating resources based on demand.
- Distributed OTA Firmware Updates: Leveraging Kubernetes workers to facilitate the deployment of OTA firmware updates on IoT devices.
- Distributed AI Model Updates: Utilizing Kubernetes workers for the deployment of AI model updates within a distributed IoT ecosystem.

The rest of the paper is structured as follows: Section II outlines the architecture and components of the proposed system. Section III details the system’s development. Section IV presents research findings and discusses implications, while Section V offers concluding remarks and suggests potential future research directions.
II. PROPOSED DEVOPS-BASED IOT SYSTEM FOR CONTINUOUS AI MODEL AND FIRMWARE UPDATES

In this research, we introduce an IoT-centric system empowered by advanced DevOps practices and modern software technologies to automate the firmware and AI model updates into resource-constrained devices. The system’s architecture revolves around a central server, acting as a hub for developers to design, create, and initiate CI/CD pipelines, ensuring efficient and automated updates. Additionally, the system incorporates a network of Raspberry Pis forming a Kubernetes cluster, crucial for managing and orchestrating AI models and firmware across distributed IoT devices.

The system architecture, depicted in Fig. 1, consists of four main components: two virtual machines (VMs) within the central server, namely the GitLab VM and the Kubernetes Master VM, along with Kubernetes Workers and IoT Devices. The GitLab VM serves as the host for all our repositories. We have developed three distinct CI/CD pipelines, each containing multiple jobs for execution. For instance, when the firmware pipeline is activated, the initial job compiles the firmware using the ESP-IDF and Micropython frameworks. Subsequently, the binary firmware file is incorporated into a docker container in the container registry during the second job. Finally, the last job deploys this container onto the target IoT device. Executing the CI/CD pipeline across various platforms is facilitated by GitLab-Runner, which needs to be installed and configured on all machines involved in the pipeline. The same logic can be applied within the other pipelines.

On the other hand, the Kubernetes master VM manages the deployments to target Raspberry Pi devices. As illustrated in Fig. 1, there are various deployments for Firmware updates, AI model updates, and Data collection. Depending on the CI/CD pipeline in operation and utilizing a specific GitLab-runner for this machine, a corresponding deployment is applied according to the configuration in the deployment file. This file contains numerous parameters, such as the deployment name, image URL, container name, and port. The worker node where the deployment will be implemented can be specified in the deployment file. Otherwise, it will be automatically selected based on Kubernetes’ internal load balancer.

After the deployment is executed, it signifies that an application is active and functioning on the worker node. Each application is essentially a Python script with the Flask library imported to create a RESTful API, which listens on a specific port. In the case of firmware deployment, we have devised an endpoint where the firmware file is stored and can be requested. Similarly, for AI model deployment, the application includes an endpoint that can be accessed by IoT devices to request an AI model update.

Furthermore, we have chosen the Pycom FiPy development board as the primary IoT device in our system, harnessing the capabilities of the ESP32 Microcontroller Unit (MCU). Tailored for IoT, the ESP32 MCU excels in low-power operation, aligning perfectly with the energy-efficient demands of our Kubernetes cluster. Additionally, we have selected the Pycom FiPy development board for its exceptional wireless communication capabilities, offering various connectivity.
options such as Wi-Fi, Bluetooth, and LoRaWAN.

For example, we have configured the FiPy to connect to a Wi-Fi router to request firmware updates using the requests MicroPython library. Once the updated firmware is saved in the FiPy’s memory, the previous version can be replaced through OTA updates, ensuring that devices are consistently equipped with the latest features and security enhancements. Moreover, the FiPy actively checks for AI model updates. Once received and loaded into its memory, this resource-constrained IoT device becomes capable of performing inferences and generating predictions specific to the designated use case. This systematic approach ensures that the IoT devices are consistently outfitted with the most recent firmware and AI model versions, enabling them to effectively execute inference tasks despite the resource limitations inherent in such devices.

It is crucial to note that Kubernetes deployments consist of a specific number of replicas, referred to as pods or containers. In essence, each pod represents a container housing our application, which can be deployed on a worker node selected by the master. The scalability of these pods, meaning their ability to be increased or decreased in number, is significant. This scalability enhances availability and performance, especially concerning the resources available on the worker node, as elaborated in the subsequent section.

III. DevOps-based IoT System Development

This section presents the key components and repositories within the proposed system, aiming to smoothly integrate, continuously improve, and efficiently manage AI models on IoT devices. Each component plays a vital role in the development and deployment process, contributing to the system’s robustness and scalability.

A. Network Design

Our system emphasizes portability and scalability. In Fig. 2, the network setup is illustrated, featuring a router for internet access. GitLab and Kubernetes master VMs are configured with bridged network interfaces, connecting to the router. Moreover, Raspberry Pi devices, serving as Kubernetes worker nodes, link to the router via Ethernet for internet access and communication with the Kubernetes master. These Raspberry Pi devices are also outfitted with Wi-Fi interfaces, connecting to an access point for communication with end FiPy IoT devices.

B. Hardware Setup

The hardware setup for our DevOps-based IoT system consists of an Ubuntu 20.04 virtual machine for GitLab, equipped with 8 CPUs and 18 GB of memory, and a Kubernetes master with 3 CPUs and 3 GB of RAM. We employ various IoT devices, including the Pycom FiPy for lightweight AI tasks and the Raspberry Pi 4 Model B for more demanding computational tasks. Communication between IoT devices and cloud services is facilitated through HTTP, ensuring efficient data transmission and updates.

C. Software Setup

Our software setup revolves around key components, including GitLab, Kubernetes, and GitLab-Runner, crucial for enabling CI/CD pipelines in our DevOps-based IoT system.

We deployed GitLab version 16.1.0, leveraging its robust features for version control and CI/CD automation. The installation process involved setting up GitLab on a dedicated Ubuntu 20.04 virtual machine. Following installation, we configured GitLab to enable CI/CD pipelines, ensuring automated testing and deployment of IoT applications.

For IoT application management, we deployed Kubernetes v1.21.0 using kubeadm. We set up a master node on an Ubuntu 20.04 VM and integrated Raspberry Pi 4 Model B devices as worker nodes. Kubernetes organizes containers into clusters, overseen by essential components: API Server, Scheduler, Controller Manager, and etcd for data storage. Within these Kubernetes clusters, the fundamental unit of deployment is the Pod. Pods host containerized applications and are allocated a unique IP address. The Kubelet, a critical component of Kubernetes, is responsible for overseeing container execution within Pods on each node. It ensures that containers are launched and managed correctly, maintaining the desired state of applications. Moreover, the Kube Proxy is another essential element that aids in network routing within Kubernetes clusters. It directs incoming traffic to the appropriate containers.

We configured GitLab-Runner v16.3.1 to run CI/CD pipelines in both GitLab and Kubernetes environments. Instances were deployed on both GitLab and Kubernetes master VMs, along with additional ones on Kubernetes worker nodes, ensuring efficient resource utilization and scalable CI/CD operations.

D. Firmware Update Process

The Firmware Update Process is crucial in the IoT ecosystem, focusing on developing reliable and resource-
efficient firmware to integrate AI applications into resource-constrained IoT devices. This involves iterative steps such as code composition, testing, and deployment, to ensure smooth operation across diverse hardware configurations and meet stringent performance criteria. For detailed firmware development processes, refer to [7], which explains MicroPython firmware with TensorFlow capabilities for AI model execution.

Algorithm 1 outlines the approach for updating firmware on end IoT devices, integrated with the CI/CD pipeline.

Algorithm 1 Firmware Update Methodology

1. Fetch latest firmware updates ($F_{\text{latest}}$)
2. Build firmware image: $F_{\text{build}} \leftarrow \text{BuildFirmware}(F_{\text{latest}})$
3. Store firmware artifacts in registry: $F_{\text{artifacts}} \leftarrow \text{DeployCluster}(F_{\text{artifacts}})$
4. Deploy firmware to Kubernetes cluster: $D_{\text{cluster}} \leftarrow \text{DeployFirmware}(F_{\text{artifacts}})$
5. Initiate firmware deployment on IoT devices: $D_{\text{deployment}} \leftarrow \text{StartDeployment}(F_{\text{artifacts}})$
6. Choose worker node for firmware pull
7. Transfer firmware to FiPy via HTTP REST API
8. Update firmware via OTA updates
9. Restart FiPy

This methodology ensures a systematic firmware update approach. It begins by fetching the latest firmware updates from the version control system (GitLab) and triggering the pipeline for automated build. Then, the resulting firmware artifacts are stored in the container registry and deployed to the Kubernetes cluster. The deployment process is initiated on end IoT devices through the Kubernetes worker node. Leveraging Over-the-Air updates and HTTP REST API, the new firmware replaces the existing one, ensuring successful completion of the update process. Finally, the FiPy device undergoes a restart to apply the updated firmware.

E. Data Acquisition Process

The Data Acquisition Process focuses on collecting, preprocessing, and transmitting data sourced from IoT devices, crucial for AI training procedures. As depicted in the system architecture diagram (Figure 1), the data acquisition repository adopts a comprehensive approach, encompassing three distinct types of data collectors: gesture recognition, aquaponics, and CO2 concentration prediction.

Notably, while the gesture recognition and CO2 datasets have been curated and validated previously in [7] and [8] respectively, highlighting the role of existing research in providing essential datasets for AI model training, challenges persist in acquiring data for the aquaponics domain, as outlined in our previous work [9]. The intricacies of this domain necessitate a nuanced approach, considering diverse environmental conditions for comprehensive data collection.

Regarding CI/CD integration, gesture recognition, CO2 concentration prediction, or aquaponics data collectors can be developed and committed to GitLab. These collectors are dockerized and stored within the container registry. The deployment process is initiated through the REST API to apply the requested data collector to the end devices. Once running on the FiPy, the data collector gathers specific data to form the dataset. This automated approach facilitates the deployment process onto targeted IoT devices.

F. AI Training Process

The AI Training Process is dedicated to creating and refining machine learning models tailored for deployment on IoT devices. Commencing with the dataset collated by the data collector, the training process embarks on various phases to ultimately yield the TensorFlow AI model. For instance, the CO2 concentration prediction model utilizes a long short-term memory (LSTM) architecture [8].

However, it is imperative to note that the resultant model may not be inherently suitable for execution on microcontroller-based IoT devices due to resource constraints and architectural limitations. Hence, there arises a need to delve into the realm of AI model optimization, as expounded in the subsequent section.

G. AI Model Update Process

The AI Model Update Process involves the creation and refinement of specific AI models optimized for deployment on IoT devices. This includes designing lightweight model architectures, optimizing model parameters, and evaluating performance metrics under resource-constrained environments.

Our AI model update methodology is outlined in Listing 1:

```
# Input: Latest TensorFlow model (M_latest)
# Output: Updated deployed model (M_deployed)

def AIModelUpdate(TensorflowModel):
    M_latest = FetchLatestModel(TensorflowModel)
    # Step 1:
    M_pruned = PruneModel(M_latest)
    # Step 2:
    M_tflite = ConvertLiteModel(M_pruned)
    # Step 3:
    M_c = ConvertCMModel(M_tflite)
    # Step 4:
    M_deployed = DeployModel(M_c)
    return M_deployed

# Call the function with the latest model
M_deployed = AIModelUpdate(TensorflowModel)
```

Listing 1: AI Model Update Methodology

This methodology involves several key steps:

1) Input and Output Declaration: The algorithm begins by taking the latest TensorFlow model, referred to as $M_{\text{latest}}$, as input. It then produces the updated deployed model, denoted as $M_{\text{deployed}}$, as output.

2) Fetch Latest Model Updates: This step involves retrieving the most recent updates of the TensorFlow model using the FetchLatestModel function.
TABLE I: AI model deployment average time taken per job.

<table>
<thead>
<tr>
<th>Job id</th>
<th>Job name</th>
<th>Average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Clear previous deployment</td>
<td>1.48</td>
</tr>
<tr>
<td>2</td>
<td>Build AI model docker image</td>
<td>338.24</td>
</tr>
<tr>
<td>3</td>
<td>Deploy into Kubernetes worker</td>
<td>31.41</td>
</tr>
<tr>
<td>4</td>
<td>Worker to FiPy</td>
<td>5.05</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>376.18</strong></td>
</tr>
</tbody>
</table>

3) Triggering AI Model CI/CD Pipeline for Model Optimization: The CI/CD pipeline for model optimization is initiated in this step. It comprises the following sub-steps:

**Step 1: Prune the Model:** The model pruning process entails removing unnecessary weights or connections to reduce its size. To achieve this, the PruneModel function is invoked.

**Step 2: Convert the Pruned Model to TFLite Format:** The model undergoes conversion to the TFLite format using the ConvertLiteModel function.

**Step 3: Convert the TFLite Model to a C Array Format:** The TFLite model is further converted to a C array format using the ConvertCModel function. The resultant C array model ($M_c$) is MCU-compatible and able to run efficiently on the FiPy board.

**Step 4: Deploy Updated Model:** Finally, the updated model, now in the C array format, is deployed to the target FiPy board utilizing the DeployModel function.

The AI model deployment process mirrors the firmware deployment. Once the newly updated AI model is built and stored, it is deployed to the Kubernetes cluster. Upon successful deployment, a container containing the updated AI model is initiated on the designated Kubernetes worker node. Subsequently, the REST API application, responsible for managing communication with IoT devices, begins the process of updating the AI model on the FiPy device.

Table I summarizes the average time required for each task in the end-to-end process, from the moment the developer does a push of a new AI model into the Git repository until its successful deployment onto IoT devices. The total average deployment time, obtained by summing the durations of all individual tasks, amounts to 376.18 seconds. Notably, the most time-consuming task is building the docker image, primarily due to the download time of the base image (python-alpine) and the subsequent push of the built image to the GitLab container registry. However, the AI model can be updated in approximately 5 seconds when requested from the FiPy board.

**IV. DISCUSSION**

The experiments covered deployment scaling, stress testing, and resource utilization, offering insights into the system’s performance across different workloads and deployment scenarios. Fig. 3 depicts the system’s worker nodes alongside the FiPy board, indicating our experimental setup.

Deployment scaling experiments, depicted in Fig. 4, examined deployment times for varying numbers of containers (pods) on a single worker node. We observed a nonlinear increase in deployment time as the number of containers increased. For instance, deploying 10 pods averaged 10 seconds, while 100 pods took around 121 seconds. Kubernetes faced challenges deploying over 110 pods efficiently. Therefore, any deployment should not exceed the 110 pods.

Stress tests, illustrated in Fig. 5a, evaluated response times for downloading AI models to FiPy devices under varying request loads. With 1, 10, and 100 concurrent requests, response time increased as request load intensified. Interestingly, we found that the number of pods running on the worker node impacts response time. For example, with 10 concurrent requests, the best average response time (6.7 seconds) occurred when the worker had 2 pods, showcasing an 86% increase in response time (12.45 seconds) with 100 concurrent requests.

To determine the optimal number of pods for Raspberry Pi 4, we conducted resource utilization tests, monitoring CPU and RAM usage under different request loads. Fig. 5b illustrates that as the number of pods increased, CPU and RAM usage also rose, indicating resource saturation under heavy loads. For instance, with 100 pods, CPU usage reached 100%, while RAM usage peaked at 45%.

Depending on the workload, we recommend adjusting the
number of pods accordingly. For light to moderate loads (up to 10 requests), having up to 10 pods yields optimal performance. However, for heavier loads, fewer pods are recommended to ensure resource availability and timely request processing.

A critical aspect is verifying if the downloaded file is an updated version. To address this, we employ hash functions computed within the FiPy and sent along with the request to the worker node. Upon reception, the application compares the hash code with the one generated on the worker node to determine file integrity.

Furthermore, building docker images for IoT devices, especially Raspberry Pis, requires special tools like buildx with architecture parameters to ensure compatibility with the target device’s architecture.

V. CONCLUSION

In this paper, we introduce a DevOps solution for automating AI model and firmware updates across distributed IoT devices. Our approach utilizes DevOps principles to streamline updating low-powered microcontrollers with limited computational resources, addressing a critical need in IoT ecosystems.

Central to our system is the utilization of Kubernetes, which orchestrates containerized applications across distributed environments, ensuring efficient resource allocation and scalability. Through the design, implementation, and deployment of our DevOps-based IoT system, we demonstrate the importance of Kubernetes in enabling continuous updates and optimizing system performance.

Our system boasts scalability, offering flexibility to adjust the number of worker nodes, Pods on the worker nodes, and IoT devices as needed. The development of CI/CD pipelines further enhances efficiency, ensuring that updated models and firmware are readily available for download upon request, minimizing downtime and optimizing system performance.

While our system lays the foundation for automated updates, it sets the stage for further exploration into AI model optimization strategies, scalability enhancements, and innovative approaches to refine model parameters for enabling real-time updates. Future studies could delve into the integration of advanced transfer learning techniques for adaptive model updates in IoT environments.

ACKNOWLEDGEMENT

This work is funded in part by the EU HE project "INCODE: Programming platform for intelligent collaborative deployments over heterogeneous edge-IoT environments" (Grant Agreement Number 101093069), the EU H2020 project "ARCADIAN-IoT: Autonomous Trust, Security and Privacy Management Framework for IoT" (Grant Agreement Number 101020259), and the UKRI EPSRC EP/Y028813/1 "National Edge AI Hub for Real Data: Edge Intelligence for Cyber-disturbances and Data Quality".

REFERENCES