*electronics*

*Article*

# Automatic Failure Recovery for Container-Based IoT Edge Applications

**Kolade Olorunnife *, Kevin Lee * and Jonathan Kua ***

School of Information Technology, Deakin University, Geelong, VIC 3220, Australia
* Correspondence: kolorunnife@deakin.edu.au (K.O.); kevin.lee@deakin.edu.au (K.L.); jonathan.kua@deakin.edu.au (J.K.)

**Abstract:** Recent years have seen the rapid adoption of Internet of Things (IoT) technologies, where billions of physical devices are interconnected to provide data sensing, computing and actuating capabilities. IoT-based systems have been extensively deployed across various sectors, such as smart homes, smart cities, smart transport, smart logistics and so forth. Newer paradigms such as edge computing are developed to facilitate computation and data intelligence to be performed closer to IoT devices, hence reducing latency for time-sensitive tasks. However, IoT applications are increasingly being deployed in remote and difficult to reach areas for edge computing scenarios. These deployment locations make upgrading application and dealing with software failures difficult. IoT applications are also increasingly being deployed as containers which offer increased remote management ability but are more complex to configure. This paper proposes an approach for effectively managing, updating and re-configuring container-based IoT software as efficiently, scalably and reliably as possible with minimal downtime upon the detection of software failures. The approach is evaluated using docker container-based IoT application deployments in an edge computing scenario.

**Keywords:** Internet of Things (IoT); edge computing; failure recovery

check for updates

**Citation:** Olorunnife, K.; Lee, K.; Kua, J. Automatic Failure Recovery for Container-Based IoT Edge Applications. *Electronics* **2021**, *10*, 3047. https://doi.org/10.3390/electronics10233047

Academic Editor: George Angelos Papadopoulos

Received: 9 November 2021
Accepted: 2 December 2021
Published: 6 December 2021

## 1. Introduction

The past decade has seen the rapid development and adoption of the Internet of Things (IoT). IoT refers to an ecosystem where billions of physical devices/objects are equipped with communication, sensing, computing and actuating capabilities [1,2]. In 2021, there is an estimated 12.3 billion of active IoT endpoints and it is forecast that the number of active IoT endpoints will reach 27 billion in 2025 [3]. IoT-based systems have been extensively deployed across many industries and sectors that impact our everyday lives, ranging from smart homes, smart cities, smart transport and smart logistics. Business opportunities and new markets abound in IoT, with new IoT applications and use cases constantly evolving. However, IoT applications are increasingly complex, commonly with a large number of end devices with many sensors and actuators and computing spread across end-nodes, edge and cloud locations. There has been substantial effort in designing architectures and frameworks to support good IoT application design [4,5]. An effective and efficient IoT system requires optimal architectural, communication and computational design. The heterogeneity of device hardware, software, communication protocols has made achieving the performance requirements of specific IoT services challenging [6–8]. Service requirements for IoT systems have motivated the development of new computational paradigms such as edge and fog computing to facilitate IoT data computation and analysis, bringing data intelligence and decision-making processes closer to IoT sensors and actuators, thus improving their performance by reducing service latency [9–12]. This is particularly important for time-sensitive tasks, such as those in autonomous vehicles, manufacturing and transport industries, where minute delays in services can have serious safety consequences [13].

There are significant trade-off considerations when deciding on the computational paradigm for IoT systems [14]. Cloud computing offers higher reliability, availability and capabilities, while edge computing offers low-latency services. However, like all complex computer systems, edge-based IoT deployments can suffer from failures [15]. This is potentially more problematic for IoT deployments due to the nature of these often being in an embedded or hard to reach physical locations. IoT devices are usually deployed at scale and has cheaper chipsets and hardware, which make them more prone to faults. It is challenging to manage faults, especially when these devices are deployed across a large area in physically hard-to-reach environments. Solving this issue can be through fault-tolerance [16,17] attempting to prevent errors in the first place, or through failure recovery techniques that attempt to recover from problems [18,19]. In addition, IoT applications are increasingly deploying software on cloud services which have no physical access making fault-tolerance necessary and difficult [20]. A potential solution to this problem is to use virtualisation technologies for resource optimisation in heterogeneous service-oriented IoT applications [21]. Redundancy is another common approach which has been extensively studied, particularly providing fail-overs in the IoT sensing, routing and control processes [22]. In addition to fault-tolerance, approaches using anomaly detection and self-healing techniques have also been explored for building more resilient IoT and Cyber-Physical Systems (CPS), addressing the various risks posed by threats at the physical, network and control layers [23].

Among the many fault-tolerance approaches studied and proposed, container-based approaches are still lacking. The aim of this paper is to investigate the feasibility of automatically detecting failures in container-based IoT edge applications. Specially, this paper investigate if this technique be used in scenarios with IoT software deployed in embedded or hard to reach scenarios, with no or difficult physical access. The proposed approach can automatically diagnose faults with IoT devices and gateways by monitoring the output of IoT applications. When faults are discovered, the proposed approach will reconfigure and redeploy container-based deployments. The experimental evaluation analyses the impact of error rate, redeployment time and packed size on the recovery time for the IoT application.

The contributions of the paper are (i). An evaluation of approaches for failure recovery for IoT applications (ii). A proposed framework for enabling failure recovery for IoT applications (iii). Experiments to evaluate the flexibility, resilience and scalability of the proposed approach.

The structure paper is as follows. Section 2 provides background of edge computing and failure recovery for IoT applications. Section 3 proposes the architecture of a framework for automatic failure recovery of IoT applications. Section 4 describes the experimental setup for this paper and Section 5 presents results of the evaluation of the framework. Section 6 concludes the paper and discuss future work.

## 2. Background

This section presents some background information on edge computing, IoT software management and failure recovery in IoT.

### *2.1. Edge Computing*

With billions of IoT sensors and devices generating a large amount of data and exchanging communication and control messages across complex networks, there arise a need for more efficient computational paradigm, rather than merely relying on cloud computing infrastructure. Edge computing is designed and proposed to address the complex challenges resulting from the large amount of data generated by IoT systems, such as resource congestion, expensive computation, long service delays which negatively impact the performance of IoT services. Edge computing aims to be a distributed infrastructure and perform data computation and analysis closer to the sensors that collect the data and actuators that act upon the decisions [9]. This significantly reduce service response time and is particularly important for IoT applications that requires real-time or time-sensitive services [13].

Put simply, an edge device is a physical piece of hardware that is a bridge between two given networks. In IoT, an edge device would commonly receive data from end devices which include sensors such as temperature sensors, moisture sensors or radio-frequency identification (RFID) scanners. The data from the sensors would be passed onto a edge device that then forwards the data to the cloud or minor data processing occurs at the edge which is then forwarded to the cloud [24].

Figure 1 illustrates a general IoT and edge-based architecture. On the left there are various sensors connected to a singular edge device which in this case is an embedded device. The edge device in Figure 1 is connected to an actuator which could possibly flip on a light switch or turn on an air conditioning unit. The edge device will forward the sensor data to an internet gateway. The internet gateway will pass this data onto an application server which will then handle the processing of this data or store it. The relationship between an edge device to edge gateway can be many to many. We can have many edge devices connect to a singular edge gateway or have many edge devices to connect to one of many edge gateways. Processing of data can happen at any stage of this architecture [25].

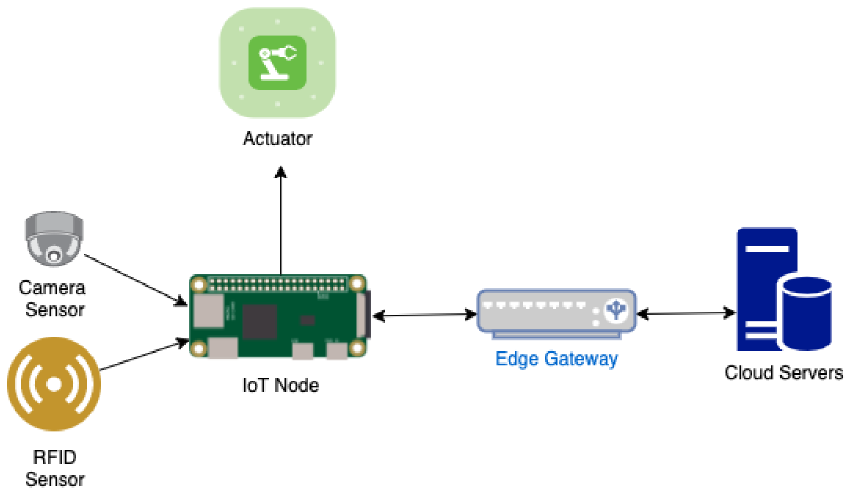

**Figure 1.** Edge Computing Architecture.

The increase in computational power for embedded devices allows for complex data processing on the edge devices before they even reach the cloud service [26]. Fog computing is where the majority of the processing is done at the gateway rather than the edge or the cloud. For IoT applications, there are strong benefits of using edge computing or fog computing. One such benefit is that latency for time-sensitive IoT applications is minimised due to the fact that the processing occurs at the edge rather than transferring data to and from the cloud through a gateway. Lower latency allows for real-time applications to become a forefront within the consumer, industrial and commercial space. In edge architectures, it is also possible that the end device and edge device work together to perform edge computing in order to make informed decisions or trigger an action. In edge architectures, real-time operations become a more feasible option because of lower latency due to the fact that no communication with a cloud service is needed. The sensor(s) embedded within the edge device can process the data as they are collected and then act upon this information without minimal impact to quality of service [9].

To understand the reliability and failure characteristics of IoT application deployments its useful to discuss the architecture. IoT can be thought of as an evolution of traditional sensor networks; however, there is an inherent and growing need for resources to e.g., process video. The traditional solution to this has been to use Cloud Computing resources. This has distinct advantages, such as access to almost unlimited cheap resources, but distinct disadvantages such as high-latency and a lack of control. If there is a failure of communication to the cloud resources or a failure of the cloud computing resource itself, there is often no ability to recover from this. Edge computing offers a compromise, with the introduction of powerful resources at the edge of the network, often in full or partial control of the application developer. In an edge computing architecture, reliability of the IoT application can potentially be better than just using a cloud service as the system does not need to rely entirely on the cloud service for it to fully function. In particular, communications are much simpler for IoT node to edge node, than that of to the Cloud Computing resources which will have many hops.

However, edge computing reaps several benefits as previously mentioned. They still suffer from a common issue, namely, network instability, which plagues any system or application that requires a steady connection to the internet or some form of secondary device. A device network stability has a correlation to consumed power. The greater the amount of consumed power, the higher the chance of instability possibly due to heavy computation on the edge device for processing a large volume of data, which is why segmenting and de-identifying the sensor data for privacy reasons to then send to the gateway for pre-processing is a new challenge.

The need for fault tolerant IoT systems and application have been on the steady incline and also the need for fault tolerant within IoT has been made apparent. The need for fault tolerant IoT systems is due to the possibility of intermittent long distance network connectivity problems, malicious harming of edge devices, or harsh environments physically affecting the devices performance [27]. As IoT systems become increasingly large, the effectiveness of having software frameworks automatically manage the different components within an IoT application is needed [28].

### *2.2. IoT Software Management*

Recently, software defined network (SDN) technologies have been considered as a dominant solution for managing IoT network architecture. Dang et al. propose incorporating SDN-based technologies with over the air (OTA) programming to design a systematic framework which allows for remotely reprogramming heterogeneous IoT devices. This framework was designed for dynamic adaptability and scalability within IoT applications [29].

Soft-WSN is a software defined wireless sensor network architecture in an effort to support application aware service provisioning for Internet of Things systems. Soft-WSN proposed architecture involves a software controller, which includes two management procedures, device management and network management. Device management allows users to control their devices within the network [30]. The network configurations is controlled by the network management policy, which can be modified in run time to deal with dynamic requirements of IoT applications.

UbiFlow, is a software defined IoT system for mobility management in urban heterogeneous networks. UbiFlow adopts multiple controllers to divide software defined networks into different partitions and achieve distributed control of IoT data flows. UbiFlow mostly pertains to scalability control, fault tolerance and load balancing. The UbiFlow controller differentiates scheduling based on per device requirements. Thus, it can present an overall network status view and optimize the selection of access points within the heterogeneous networks to satisfy IoT flow requests, and guaranteeing network performance for each IoT device [31].

### *2.3. Fault Tolerance in IoT*

Providing fault tolerance support to Internet of Things systems is an open field, with many implementations utilising various technologies like artificial intelligence, reactive approaches and algorithmic approaches [32].

A plug-able micro services framework for fault tolerance in IoT devices can be used to separate the work flow for detecting faults. The first micro service utilises complex event processing for real time reactive fault tolerance detection, whereas the second micro service uses cloud-based machine learning to detect fault patterns early and is a proactive strategy for fault tolerance. The reactive fault tolerance that uses complex event processing only initiates recovery protocols upon the detection of an error. This sort of strategy is only effective for systems that have a low latency connection to the faulty device. Whereas the proactive strategy that uses machine learning initiates recovery protocols before errors occur using predictive technologies. The main process behind a proactive strategy is to temporally disable or isolate IoT devices that will cause an error or harmfully impact the system before it occurs [33].

A mobile agent-based methodology can be utilised to build a fault-tolerant hierarchical IoT-cloud architecture that can survive the faults that occur at the edge level. In the proposed architecture, the cloud is distributed across four separate levels which are cloud, fog, mist and dew. The distribution is based on the processing power and distance from the edge IoT devices. This makes for a reliable system by redirecting the application onto an alternate server when faults occur within any level of the system [34].

Whereas, utilising a bio-inspired particle multi-swarm optimization routing algorithm to ensure connections between IoT devices remain in a stable condition is also a feasible methodology for fault tolerant IoT networks. The multi-swarm strategy determines the optimal directions for selecting the multipath route while exchanging messages from any positions within the network [35].

In deployment scenarios where wireless technologies are used, such as those in WSNs, virtualisation technologies for resource optimisation can be used to assist heterogeneous service-oriented IoT applications [21]. There are many different redundancy and fail-over strategies across the IoT stack and ecosystem. The paper in [22] provides a comprehensive survey, particularly in the IoT sensing, routing and control processes [22]. Another paper [23] presents a comprehensive roadmap for achieving resilient IoT and CPS-based systems, with techniques encompassing anomaly detection and self-healing techniques to combat various risks and threats posed by internal/external agents across the physical, network and control layers.

### *2.4. Theories, Metrics and Measurements for System Reliability*

Today's technological landscape requires high system availability and reliability. System downtime, failures, or glitches can result in significant revenue loss and more critically, compromising the safety of systems. Hence, measurement metrics for system reliability are used by companies to detect, track and manage system failures/downtime. Some commonly used metrics are Mean Time Before/Between Failure (MTBF), Mean Time to Recovery/Repair (MTTR), Mean Time to Failure (MTTF) and Mean Time to Acknowledge (MTTA). These metrics allow the monitoring and management of incidents, including tracking how often a particular failure occurs and how quickly can the system recover from such failure. For a more detailed explanation and derivation of these metrics, we refer the readers to [36].

There are many approaches presented in the literature to minimise system failures and manage incidents more effectively. These approaches span multiple industry applications, with most techniques focusing on optimising the MTBF. However, there is currently limited work in applying these techniques to IoT and edge computing. For example, Engelhardt et al. [37] investigated the MTBF for repairable systems by considering the reciprocal of the intensity function and the mean waiting time until the next failure; Kimura et al. [38] looked at MTBF from an applied software reliability perspective by analysing software reliability growth models as described by non-homogenous Poisson process; in two separate works, Michlin et al. [39,40] performed sequential MTBF testing on two systems and compared their performance; Glynn et al. [41] proposed a technique for efficient estimation of MTBF in non-Markovian models of highly dependable systems; Zagrirnyak et al. [42] discussed the use of neuronets in reliability models of electric machines for forecasting the failure of the main structural units (also based on MTBF); Suresh et al. [43] unconventionally applied MTBF as a subjective video quality metric, which makes for an interesting evaluation of MTBF in other application areas other than literal system failures.

Reliability curve is also an important metric that is used widely across many applications and industries [44]. Variants of reliability curves have been recently applied to IoT, edge computing and Mobile Edge Computing (MEC) in the advent of innovations in 5G. For example, Rusdhi et al. [45] performed an extensive system reliability evaluation on several small-cell heterogeneous cellular networks topologies and considered useful redundancy region, MTTF, link importance measure and system/link uncertainties as metrics to manage failure incidents; Liu et al. [46] propose a MEC-based framework that incorporates the reliability aspects of MEC in addition to the latency and energy consumption (by formulating these requirements as a joint optimisation problem); Chen-Feng Liu et al. [47,48] proposed two (related by separate) MEC network designs that also considered the latency and reliability constraints of mission-critical applications (in addition to average queue lengths and queue delays, by using Lyapunov stochastic optimization and extreme value theory); Han et al. [49] proposed a context-aware decentralised authentication MEC architecture for authentication and authorisation to achieve an optimal balance between operational costs and reliability.

Link importance measure is another important aspect for measuring system reliability. There are several techniques that considered link importance measure in the context of IoT and edge computing. For example, Silva et al. [50] developed a suite of tools to measure and detect link failures in IoT networks by measuring the reliability, availability and criticality of the devices; Benson et al. [51] proposed a resilient SDN-based middleware for data exchange in IoT edge-cloud systems, which dynamically monitors IoT network data and periodically sends multi-cast time-critical alerts to ensure the availability of system resources; Qiu et al. [52] proposed a robust IoT framework based on Greedy Model with Small World (GMSW), that determines the importance of different network nodes and communication links and allows the system to quickly recover using “small world properties” in the event of system failures; Kwon et al. [53] presented a failure prediction model using a Support Vector Machine (SVM) for iterative feature selection in Industrial IoT (IIoT) environments which calculates the relevance between the large amount of data generated by IIoT sensors and predict when the system is more likely to experience downtime, Dinh et al. [54] explores the use of Network Function Virtualisation (NFV) for efficient resource placements to manage hardware and software failures when deploying service chains in IoT Fog-Cloud networks.

## 3. Proposed Framework for IoT Failure Recovery

The aim of this paper is to propose a failure recovery framework for IoT applications. This framework focuses on the problem of failures in IoT applications that are deployed on end nodes and edge nodes that utilise container deployment techniques. This is a relatively new potential problem, as it is only recently that end nodes and edge nodes have the resources to use containerization. The proposed framework assumes that failures occur in IoT deployments, due to corruption or configuration in the end node or edge gateway [55]. The framework will passively monitor communications from the IoT node and gateway to detect potential failures. On discovery of a potential failure, the framework will deploy known good applications in containers. The general aim of the framework is to minimise downtime due to application failure.

Figure 2 presents the overall architecture of the proposed framework which would be bolted on to an existing IoT edge-computing deployment. The deployment controller is a monitoring agent which can send action requests to any IoT gateway or device. An action request is one of two things, either a reconfiguration request to reconfigure one to many IoT gateways or devices, or a redeployment request in which the code-base for the IoT gateways or devices can be updated. The following describes how each device of the IoT-edge deployment interacts with the proposed framework.

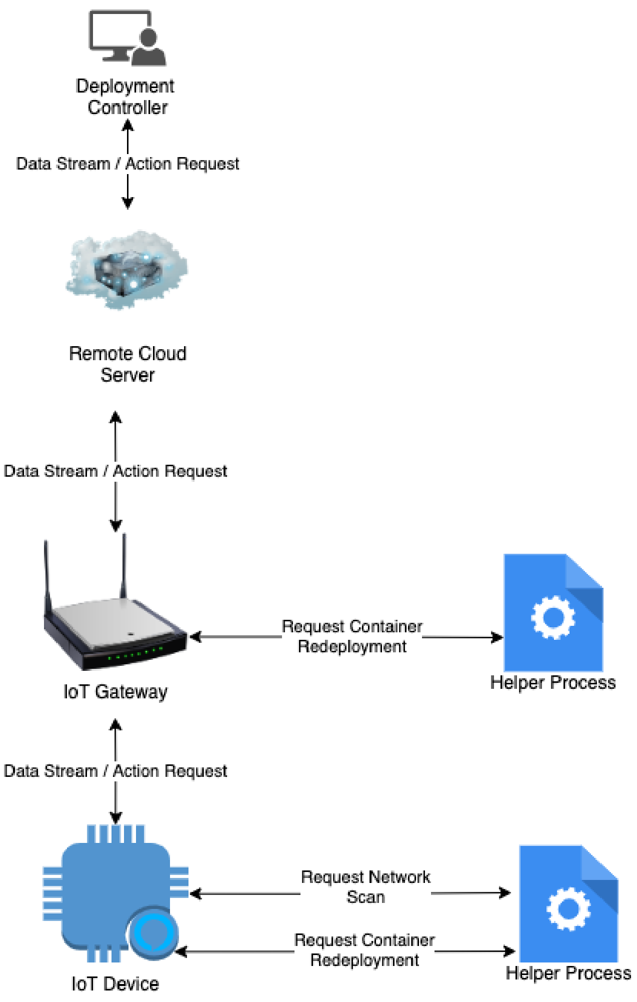
Deployment Controller
Data Stream / Action Request
Remote Cloud Server
Data Stream / Action Request
Request Container Redeployment
IoT Gateway
Helper Process
Data Stream / Action Request
Request Network Scan
Request Container Redeployment
IoT Device
Helper Process

**Figure 2.** Proposed framework for IoT failure recovery.

The IoT Device receives action requests from the gateway. If the action request is a redeployment request, the helper process is notified and proceeds to handle managing, rebuilding, restarting and deleting of containers and images for a seamlessly migration to the new version. The IoT Device can also request from the helper process to perform a network scan for any active gateways that are connected to the same network as the IoT device. The IoT device sends the collected data to the gateway.

The IoT Gateway receives action requests from the cloud server and will route the request to their corresponding target devices or ignore the request entirely if it is required to execute it. The IoT gateway will communicate with the helper process if the gateway is the target for the redeployment request. When the helper process is notified it will proceed to handle managing, rebuilding, restarting and deleting of containers and images for a seamlessly migration to the newer version. The IoT gateway receives the data stream and forwards it to the cloud server.

The Cloud Server receives action request from the deployment controller and will propagate the request to all connected gateways. The cloud server receives the data stream from the IoT gateway and forwards it to the deployment controller.

The Deployment Controller receives the response of an action request completion or failure from the cloud server which is sent from the IoT devices through the IoT gateway. The deployment controller initiates of action requests and receive the data stream from the cloud server and display them.

IoT-edge Deployments are described through *YAML* configuration files. Redeployment requests must have the *build.yml* file at the root of the update folder. The updated folder is located at /update of the current working directory of the deployment controller. The structure of the build file is illustrated in Figure 3.

```
target:
group: "*"
type: iot-gateway

actions:
- DELETE cache.json
- OVERWRITE Dockerfile
- OVERWRITE device-helper.js
- OVERWRITE gateway.js
- MKDIR unit-tests
```

**Figure 3.** Example build.yml.

The build file is split into two configuration sections. The target section indicates to the IoT gateways whether this build package is for a gateway or IoT device(s). The first path within the target section, target.group is the identifier of the device to update. Specifying the * key, like in the example, will address all devices of target.type to update. The second path of target within the target section, target.type specifies if the build package should be deployed on the IoT gateway or IoT device.

The second configuration section is the actions section which will tell either the IoT device or gateway what actions to perform to unpack the build. There are only 4 action commands that are usable within the build file actions configuration section. OVERWRITE (filename) will create a new file or overwrite the existing file, MKDIR (dir) will create a directory if it does not exist, DELETE (filename) will remove a file, finally REBUILD will rebuild the device's image. The REBUILD action will not run any subsequent actions and should always be at the bottom of the actions list.

The build.yml refers to a small number of files, which allow the building and rebuilding of the node software. The first of these, *cache.json* is a JSON file on the IoT gateways that store the time data was last received from an IoT device. This allows the IoT gateway to know when data was last sent even if the IoT gateway restarts. *Dockerfile* is the docker configuration file that describes how the image should be built.

*device-helper.js* is a JavaScript file that executes outside the docker container environment and is solely responsible for interacting and modifying the behaviour of the docker containers when requested to do so. The device-helper primary role is to delete and rebuild images to the new specifications, attempt to start the new image and if a failure occurs attempt failure recovery protocols. The device-helper script is always in continuous communication with the active running docker container.

*gateway.js* is the main JavaScript file that runs within the docker container and is responsible for keeping track of all the currently connected IoT devices and automatically removing any disconnected IoT device from the list. *gateway.js* has the critical responsibility of relaying data and requests to the required devices at any given point in time.

## 4. Experiment Setup

To evaluate the effectiveness and robustness of the proposed framework, a series of experiments have been performed with varying configurations. A testbed configuration was setup using common hardware platforms widely used for IoT nodes and gateways. The proposed architecture as described in Figure 2 using several Raspberry Pi small board computers. Raspberry Pis are commonly used in IoT testing and deployment due to their versatility. Their computational capabilities are typically suitable for most IoT deployment scenarios.

The experimental testbed setup consists of (i) a laptop as the deployment controller, (ii) a Raspberry Pi 4 as an IoT node and a (iii) Raspberry 3B+ as an edge node. The Raspberry Pi 4 Model B consists of a 4 GB LPDDR4-3200 SDRAM, Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC at 1.5 GHz. The Raspberry Pi 3 Model B+ consists of 1 GB LPDDR2 SDRAM Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC at 1.4 GHz. The laptop consists of an Intel i7-8565U at 2.00 GHz, 16 GB RAM, Geforce GTX 1050 GPU. Figure 4 illustrates the different devices in the experimental setup.

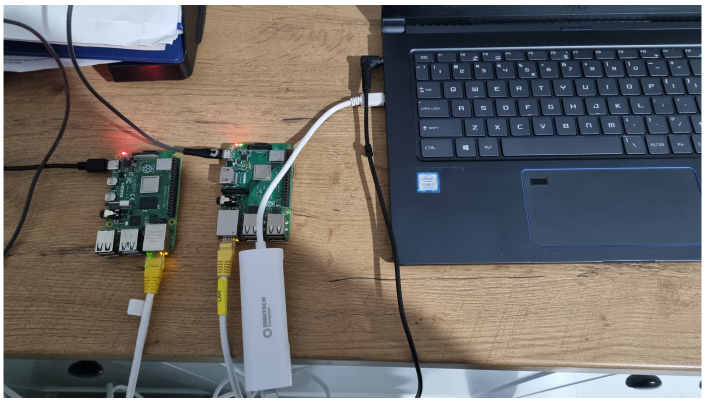

**Figure 4.** Experiment setup with IoT devices/nodes (RPi 4B), IoT gateway (RPi 3B+), deployment controller (laptop).

There are multiple stages of the IoT redeployment process. The deployment controller will first read the *build.yml* file and send it to the IoT gateway. Upon sending the *build.yml* file, the deployment controller will open a stream pipe to the IoT gateway and start streaming a large data payload. While the segmented data are being received at the IoT gateway, the *build.yml* is stored in memory and the large payload for the image rebuild process is written to the Raspberry Pi's disk space. After the streaming process has completed, the IoT gateway will again read the *build.yml* file and verify if the redeployment request must be forwarded to an IoT devices or the gateway should apply the update to itself.

If the redeployment request must be forwarded to an IoT device, the IoT gateway will open a direct stream pipe to the IoT device and proceed to send the *build.yml* first and then stream the payload that previously saved to the IoT gateway's disk. When each bit of the segmented stream data is received at the IoT device, the IoT device will write the file to the disk on the Raspberry Pi. After the stream process is complete on the IoT device, the IoT device will automatically incorporate the large payload to as a part of the build process for the new image.

The IoT device build process starts with the helper process as shown in Figure 2. It will first terminate the active container, rebuild the new Docker image with the included payload, run the new Docker image and when successfully starts delete the previous image and record the elapsed time for that process of building a new image and then starting the new container and forward the results back to the IoT gateway.

## 5. Experimental Evaluation

This section presents an experimental evaluation of the proposed framework by investigating five distinct scenarios using the same testbed setup described in Section 4: (i) IoT device redeployment; (ii) IoT gateway redeployment; (iii) IoT device sensor fault detection; (iv) IoT device redeployment failure detection; and (v) IoT gateway redeployment failure detection.

### *5.1. Scenario 1: IoT Device Redeployment*

In this scenario, the aim is to calculate the time required for a redeployment request to be sent from the deployment controller and for the IoT device to successfully fulfill the request and send a response back to the corresponding IoT gateway. The data collected from this experiment will be used as a base value to determine on average how long the deployment controller should wait with no response from the IoT devices. IoT devices that exceed this threshold will be considered to have possibly encountered an error and further diagnostics on that device may be required.

Table 1 contains the results of 3 consecutive redeployment requests.

**Table 1.** IoT device redeployment duration.

| **Test Results** | |
|---|---|
| **Test Number** | **Elapsed Time (secs)** |
| 1 | 56.6 |
| 2 | 102.3 |
| 3 | 61.8 |
| Average | 73.6 |

The *Test Number* refers to the number of that test and *Elapsed Time* is the overall duration for the following actions to occur.

The redeployment processes/steps across multiple IoT devices are described as follows (presented in Figure 5):

1. The deployment controller sends a redeployment request for an IoT device to the Cloud server.
2. The Cloud server forwards the request to the IoT gateways.
3. The IoT gateway check what IoT devices to forward the request to.
4. The IoT device receives and completes the redeployment request.
5. The IoT device sends the response of success or error message back to the IoT gateway.
6. The IoT gateway then passes the message back to the cloud server and which then passes it to the deployment controller and the timer is stopped and time taken is displayed.

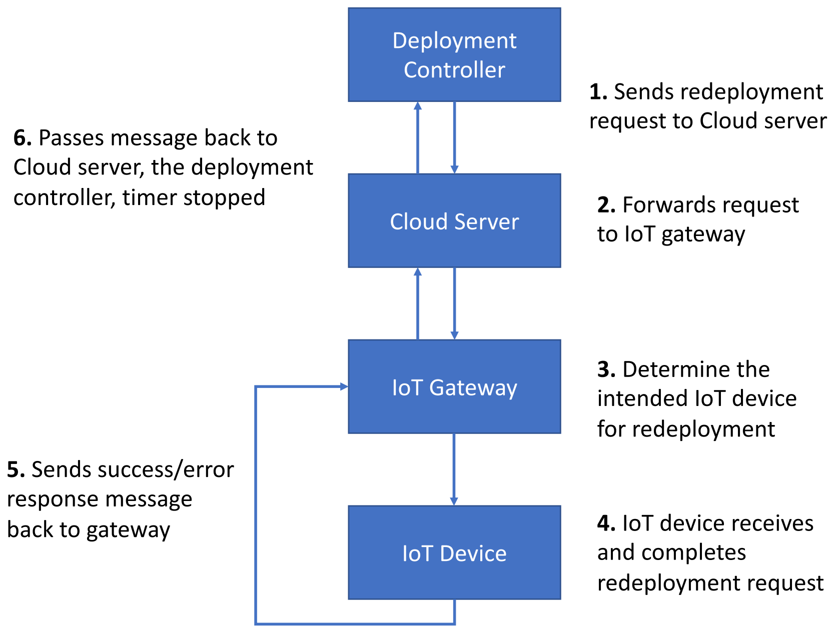
Deployment Controller
1. Sends redeployment request to Cloud server
6. Passes message back to Cloud server, the deployment controller, timer stopped
Cloud Server
2. Forwards request to IoT gateway
IoT Gateway
3. Determine the intended IoT device for redeployment
5. Sends success/error response message back to gateway
IoT Device
4. IoT device receives and completes redeployment request

**Figure 5.** Scenario 1: Redeployment process across multiple IoT device(s).

Scenario 1 explores the average time in seconds for a redeployment request to be successfully fulfilled. The results from this experiment can be used as a benchmark to determine a time frame window for how long the deployment controller should wait for requests from the IoT device or devices. If the response time exceeds the time frame, the system will assume that the device was unable to migrate to the new version and run further diagnostics as to determine if the device is fully offline or requires another redeployment request.

Further exploration of Scenario 1 led to varying the size of the redeployment package by 50 MB chunks and simultaneously recording the time data as inactive in the system while the IoT devices migrate to the new build.

Figure 6 displays a linear trend that a larger build package sent across the network will increase the time it takes for the redeployment request to be completed. At a 50 MB redeployment package, it takes approximately 150 s for a successfully redeploy and a 100 MB redeployment package on average requires 250 s. The time difference between each 50 MB increase in the redeployment package size is about 100 s.

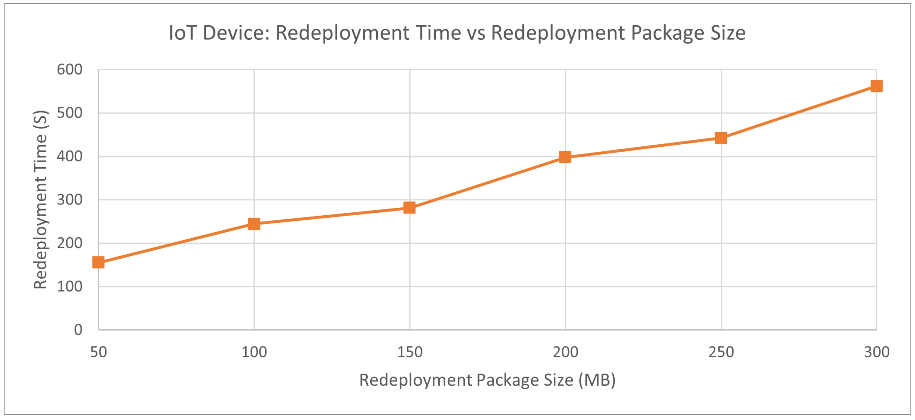

**Figure 6.** Redeployment Time vs. Redeployment Package Size.

Figure 7 displays a relatively linear trend where the larger the redeployment package, the greater time, no data are sent from the IoT devices to the IoT gateways. The time difference between each 100 MB redeployment package size is greater than 40 s, whereas the jump between 50 MB is relatively stagnant.

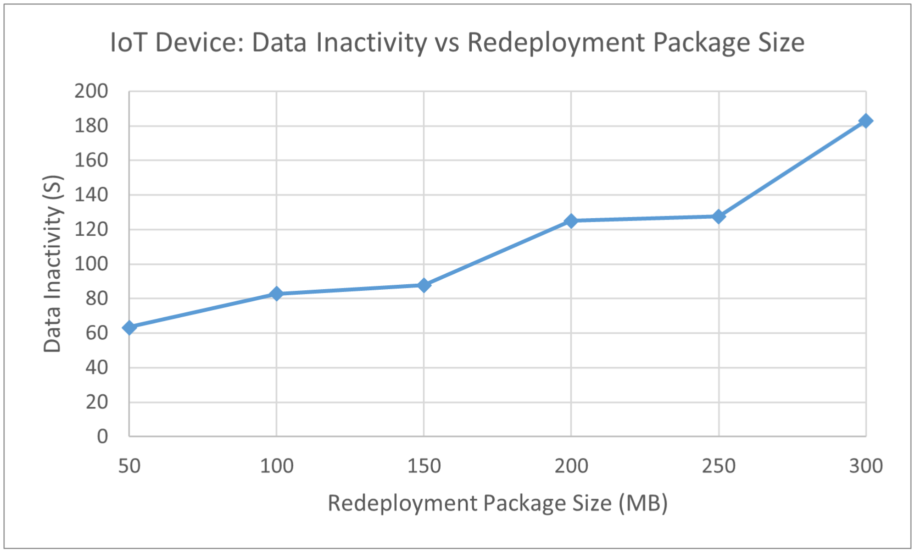

**Figure 7.** Data Inactivity vs. Redeployment Package Size.

Although Scenario 1 gives a benchmark for the average time it would take for a redeployment request to be successfully fulfilled, it does not take into account when a IoT device does successfully migrate to the new version but the time it took exceeded the response timeout window. In this scenario, the deployment controller assumes something is wrong with the IoT device and will resend the redeployment request which in effect will cause the already updated IoT device to forcefully update again. Scenario 1 results do not take into account build packages that are large or very small, which can greatly vary the average time required for the completion of a redeployment request.

### *5.2. Scenario 2: IoT Gateway Redeployment*

Scenario 2 explores the average time in seconds for a redeployment request to be successfully fulfilled. The results from this experiment can be used as a benchmark to determine a time frame window for how long the deployment controller should wait for requests from the IoT gateway(s). If the response time exceeds the time frame, the system will assume that the gateway was unable to migrate to the new version and run further diagnostics as to determine if the gateway is fully offline or requires another redeployment request.

**Table 2.** IoT gateway redeployment times.

| **Test Results** | |
|---|---|
| **Test Number** | **Elapsed Tim (secs)** |
| 1 | 28.39 |
| 2 | 92.27 |
| 3 | 112.8 |
| Average | 77.6 |

Table 2 contains the results of 3 consecutive gateway redeployment requests. The *Test Number* refers to the number of that test and *Elapsed Time* is the overall duration for the following actions to occur.

The redeployment processes/steps across multiple IoT gateways are described as follows (presented in Figure 8):

1. The deployment controller sends a redeployment request for an IoT device to the Cloud server.
2. The Cloud server forwards the request to the IoT gateways.
3. The IoT gateway(s) check if the request is intended for them.
4. The IoT gateway(s) complete the redeployment request.
5. The IoT gateway(s) send the response whether success or error message back to the Cloud server.
6. The Cloud server passes it to the deployment controller and the timer is stopped and time taken is displayed.

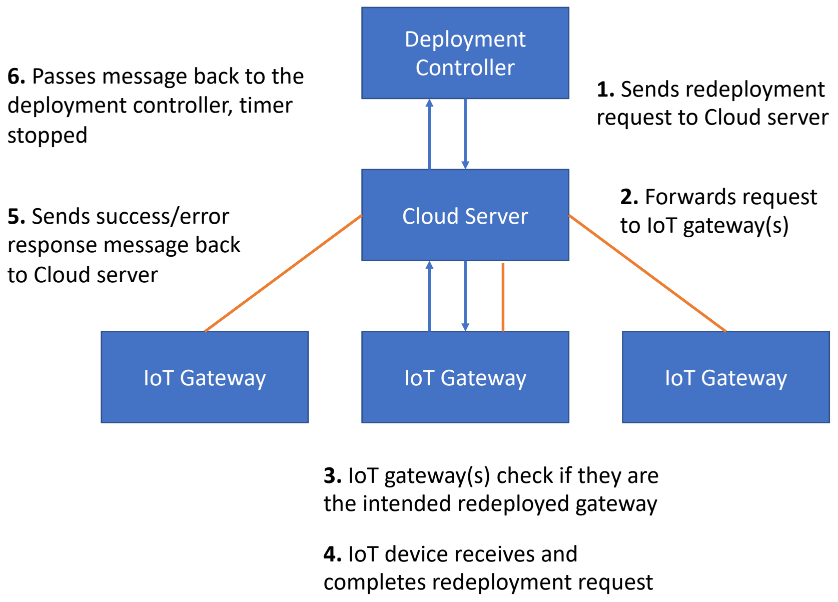
Deployment Controller
6. Passes message back to the deployment controller, timer stopped
1. Sends redeployment request to Cloud server
Cloud Server
5. Sends success/error response message back to Cloud server
2. Forwards request to IoT gateway(s)
IoT Gateway
IoT Gateway
IoT Gateway
3. IoT gateway(s) check if they are the intended redeployed gateway
4. IoT device receives and completes redeployment request

**Figure 8.** Scenario 2: Redeployment process across multiple IoT gateway(s).

Scenario 2 explores the average time in seconds for a redeployment request to be successfully fulfilled. The results from this experiment can be used as a benchmark to determine a time frame window for how long the deployment controller should wait for requests from the IoT gateway or gateways. If the response time exceeds the time frame, the system will assume that the gateway was unable to migrate to the new version and run further diagnostics as to determine if the gateway is fully offline or requires another redeployment request.

Further exploration of Scenario 2 lead to varying the size of the redeployment package by 50 MB chunks and simultaneously recording the time data as inactive in the system while the IoT devices migrate to the new build.

Figure 9 displays a very linear trend that a larger build package sent across the network will increase the time it takes for the redeployment request to be fulfilled. At a 50 MB redeployment package, it takes approximately 110 s for a successful redeploy and a 100 MB redeployment package on average requires 185 s. The time difference between each 50 MB increase in the redeployment package size is approximately 80 to 90 s.

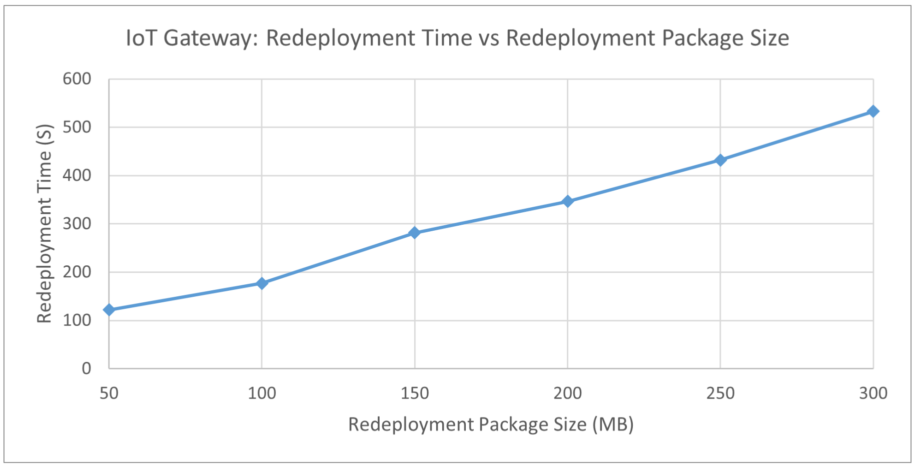

**Figure 9.** Redeployment Time vs. Redeployment Package Size.

Figure 10 displays a linear trend where a larger build package that is sent across the network increases the time no data will be received from the IoT devices. The time difference between each 100 MB redeployment package size is not consistent by a certain amount, but starts to sharply increase when the redeployment package is above 200 MB.

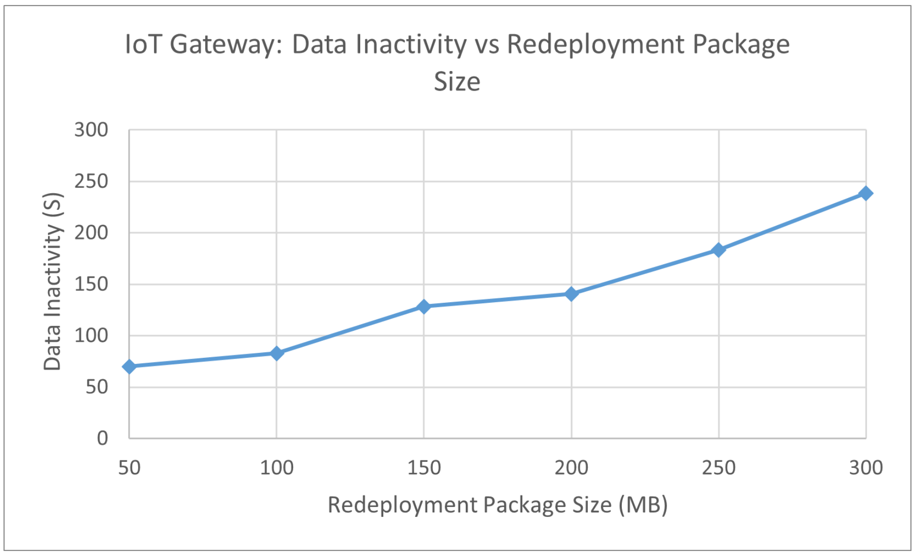

**Figure 10.** Data Inactivity vs. Redeployment Package Size.

Although Scenario 2 gives a benchmark for the average time it would take for a redeployment request to be successfully fulfilled, it does not take into account when a IoT gateway does successfully migrate to the new version and the time it took exceeded the response timeout window. In this scenario the admin panel assumes something is wrong with the IoT gateway and will resend the redeployment request which in effect will cause the already updated IoT gateway to forcefully update again. Scenario 2 results do not take into account build packages that are large or very small which can greatly vary the average time required for the completion of a redeployment request.

### *5.3. Scenario 3: IoT Device Sensor Fault Detection*

Scenario 3 aims to test how long it takes a gateway to detect that an IoT device has stopped sending sensor data and how long the IoT device spends to recover and reinitialise data sending to the IoT gateway. For Scenario 3, each message sent from the IoT device had a 60 percent chance of failing and each request sent from the IoT gateway to the IoT device to recover had a 40 percent chance of success.

Scenario 3 explores the average time in seconds for a IoT device to recover from a sensor fault. The results from this experiment can be used as a benchmark to determine a time frame window for how long on average an IoT device will take to recover one of its critical functions.

Exploration of Scenario 3 lead to varying the frequency at which data are sent from 1 to 10 s and recording the elapsed time for the IoT gateway to detect an error and request for the IoT device to self-heal. The IoT gateway was configured to check for errors every 5 s.

Figure 11 is a demonstration of the time increasing for the IoT device to recover from sensor failure when the time between messages is increased with 10 s having the largest spread of varying recovery times. The 5 s interval for Figure 11 contains the smallest spread due to the fact that messages are being sent every 5 s to the IoT gateway and the IoT gateway itself was configured to check for problems with the IoT device every 5 s. There is a clear distinction that increasing the frequency at which data are sent allows for a much quicker recovery time with a smaller spread.

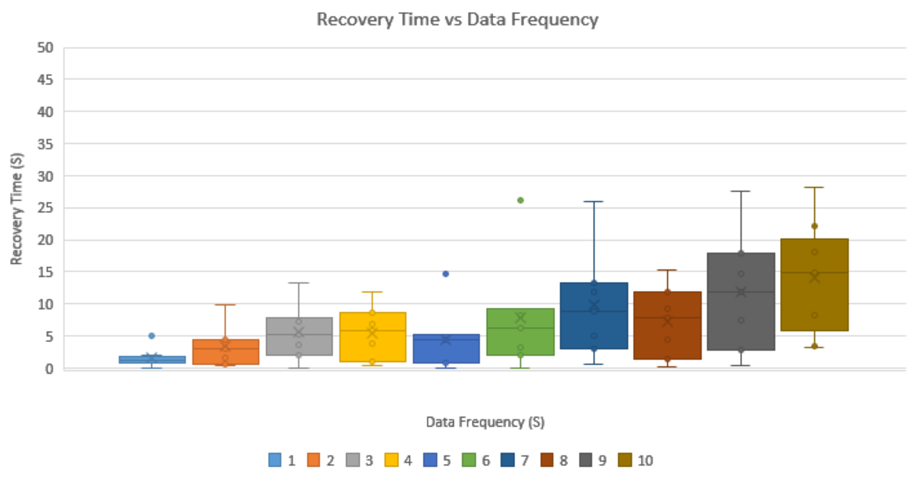

**Figure 11.** Recovery time vs. Data Frequency.

Although, Scenario 3 explores the average time in seconds for a IoT device to recover from a sensor fault. It is not a true measurement of what would happen in the real world as no physical sensors were attached and forced to fail and then recover. The sensor failures in this experiment are purely software-based and there is a fixed chance for sensor failure and recovery. Due to the software constraints imposed to test sensor recovery the results can only be regarded as a theoretical time constraint.

### *5.4. Scenario 4: IoT Device Redeployment Failure Detection*

Scenario 4 explores the average time in seconds for a redeployment request to successfully recover after encountering failure on the IoT devices. The results from this experiment can be used as a guide to determine on average number of attempts until the redeployment is successful and as well as the elapsed time from failure detection to recovery.

Exploration of Scenario 4 led to varying the chance of and triggering an error during a redeployment request and measuring the elapsed time and attempts that is required for the system to recover and update the IoT device.

Redeployment time is the overall duration for the following actions to occur: deployment controller sends redeployment request to the IoT gateway. IoT gateways verifies the IoT devices to forward the request to. IoT device receives and complete the redeployment request. IoT device sends success of failure message back to the IoT gateway. IoT gateway checks if the redeployment request was successful and if it has failed it will request for the IoT device to perform the redeployment request again. Upon success of the redeployment request the IoT gateway will pass the message back to the deployment controller and the timer is stopped and time taken is displayed.

Figure 12 shows that the redeploy time is not greatly affected when the chance of a build error occurring is lower than 30 percent. The redeployment time begins to have a major increase from 60 percent chance of build error.

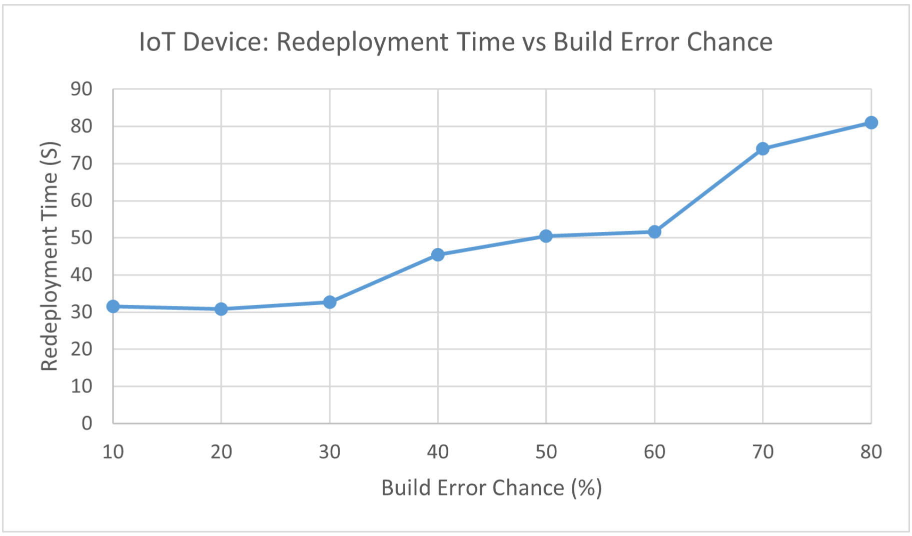

**Figure 12.** Redeployment Time vs. Build Error Chance.

Scenario 4 does not take into account when the IoT device attempting to update is stuck in an infinite loop due to attempting to start the program that has errors. Currently, the experiment will try as long as it needs to until the IoT device has updated which can result in an extremely high number of attempts.

### *5.5. Scenario 5: IoT Gateway Redeployment Failure Detection*

Scenario 5 explores the average time in seconds for a redeployment request to successfully recover after encountering failure on the IoT gateways. The results from this experiment can be used as a guide to determine on average number of attempts until the redeployment is successful and as well as the elapsed time from failure detection to recovery.

Exploration of Scenario 5 lead to varying the chance of an triggering an error during a redeployment request and measuring the elapsed time and attempts that is required for the system to recover and update the IoT device.

Redeployment time is the overall duration for the following actions to occur: deployment controller sends redeployment request to the IoT gateway. IoT gateways verifies if it is required to execute the redeployment request. IoT gateway completes the redeployment request. IoT gateway helper process (See Figure 2) sends success of failure message back to the IoT gateway. Iot gateway checks if the redeployment request was successful and if it has failed it will request for the IoT gateway helper process to perform the redeployment request again. Upon success of the redeployment request the IoT gateway will pass the message back to the deployment controller and the timer is stopped and time taken is displayed.

Figure 13 shows that the redeploy time is not greatly affected when the chance of a build error occurring is lower than 30 percent. The redeployment time begins to have a major increase from 40 percent chance of build error and does not slow down. Results indicate that minimising the build error as much as possible for the gateway will ensure quality of service for end users will not be tarnished.

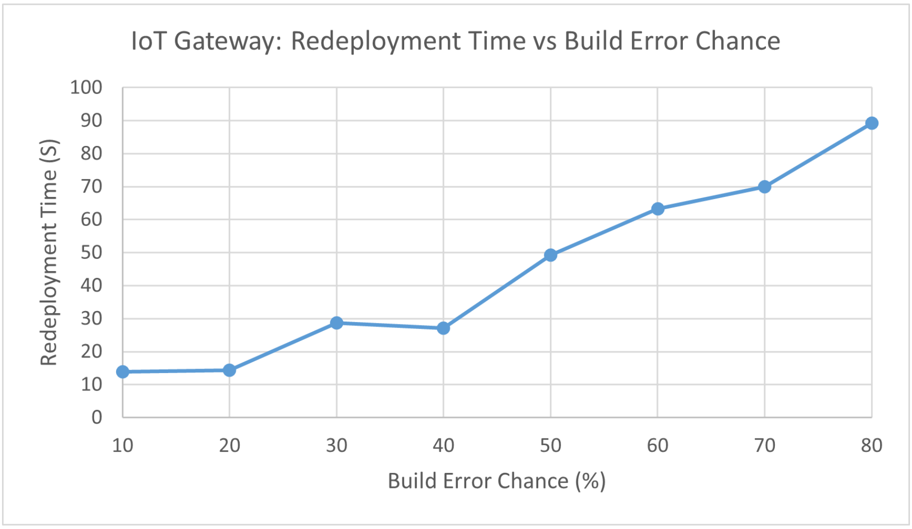

**Figure 13.** Redeployment Time vs. Build Error Chance.

Scenario 5 does not take into account when an IoT gateway attempting to update is stuck in an infinite loop due to attempting to start the program that has errors. Currently the experiment will try as long as it needs to until the IoT gateway has updated, which can result in an extremely high number of attempts.

## 6. Conclusions and Future Work

The focus of this paper is to add a layer of failure recovery to deployed container-based IoT edge applications. A framework was proposed the monitors deployed IoT applications and detects if either the IoT end node or IoT gateway has potentially failed. Once potential failures have been detected, the deployment controller will rebuild the application and deploy to containers in the effected node. The aim is to minimise downtime due to potential failures. This paper evaluated an implementation of this approach through a series of experiments testing different configurations for viability.

This paper argues that low latency IoT systems have a significantly higher fault detection and recovery time due to the system being able to verify more frequently if a device has yet to send a message for a specified number of seconds. The paper argues that decreasing the chance of build errors is critical in ensuring a that the quality of service remains consistent even when the devices require new redeployed software. It has demonstrated the design and implementation of a framework that automatically detects faults and attempts to automatically recover and as well as contains functionality to automatically reconfigure and redeploy software to all or targeted IoT devices or gateways. The proposed framework can be used to evaluate the occurrence of errors with an multi tiered system and the average theoretical recovery time for when an IoT device or gateway is down due to faults.

The work in this paper can be expanded on by implementing and refactoring the framework to be suitable to run on low powered devices. This paper had a primary focus on varying many factors to see the significant changes and impacts that could be seen if implemented on a real physical system. However, the paper uses higher tier technologies such as Docker, Raspberry Pis and real-time socket streams with the intended purpose to simulate the behaviour of low powered IoT devices and IoT gateways and the failures that can occur. The paper still presents a general overview of how faults can affect any given multi-tiered IoT application. The proposed framework for automatic failure recovery and the simulation of a multi tiered IoT system can be improved by rate-limiting in the network connection and processing power to closely mimic the behaviour of a lower powered device and what behaviours can occur when faults arise within these systems.

**Author Contributions:** Conceptualization, K.O., K.L. and J.K.; methodology, K.O., K.L. and J.K.; software, K.O.; validation, K.O., K.L. and J.K.; investigation, K.O.; resources, K.L.; data curation, K.O.; writing—original draft preparation, K.O.; writing—review and editing, K.O., K.L. and J.K.; visualization, K.O.; supervision, K.L. and J.K.; project administration, K.L. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Li, S.; Da Xu, L.; Zhao, S. The internet of things: A survey. *Inf. Syst. Front.* **2015**, *17*, 243–259. [CrossRef]
2. Atzori, L.; Iera, A.; Morabito, G. The internet of things: A survey. *Comput. Netw.* **2010**, *54*, 2787–2805. [CrossRef]
3. IoT Analytics, State of IoT 2021: Number of Connected IoT Devices Growing 9% to 12.3 Billion Globally, Cellular IoT Now Surpassing 2 Billion. Available online: https://iot-analytics.com/number-connected-iot-devices/ (accessed on 21 November 2021).
4. Wang, W.; Lee, K.; Murray, D. A global generic architecture for the future Internet of Things. *Serv. Oriented Comput. Appl.* **2017**, *11*, 329–344. [CrossRef]
5. Wang, W.; Lee, K.; Murray, D. Building a generic architecture for the Internet of Things. In Proceedings of the 2013 IEEE Eighth International Conference on Intelligent Sensors, Sensor Networks and Information Processing, Melbourne, VIC, Australia, 2–5 April 2013; pp. 333–338.
6. Al-Fuqaha, A.; Guizani, M.; Mohammadi, M.; Aledhari, M.; Ayyash, M. Internet of things: A survey on enabling technologies, protocols and applications. *IEEE Commun. Surv. Tutor.* **2015**, *17*, 2347–2376. [CrossRef]
7. Ai, Y.; Peng, M.; Zhang, K. Edge computing technologies for Internet of Things: A primer. *Digit. Commun. Netw.* **2018**, *4*, 77–86. [CrossRef]
8. Salman, O.; Elhajj, I.; Kayssi, A.; Chehab, A. Edge computing enabling the Internet of Things. In Proceedings of the 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), Milan, Italy, 14–16 December 2015; pp. 603–608.
9. Yu, W.; Liang, F.; He, X.; Hatcher, W.G.; Lu, C.; Lin, J.; Yang, X. A Survey on the Edge Computing for the Internet of Things. *IEEE Access* **2018**, *6*, 6900–6919. [CrossRef]
10. Kua, J.; Armitage, G.; Branch, P. A Survey of Rate Adaptation Techniques for Dynamic Adaptive Streaming Over HTTP. *IEEE Commun. Surv. Tutor.* **2017**, *19*, 1842–1866. [CrossRef]
11. Kua, J.; Nguyen, S.H.; Armitage, G.; Branch, P. Using Active Queue Management to Assist IoT Application Flows in Home Broadband Networks. *IEEE Internet Things J.* **2017**, *4*, 1399–1407. [CrossRef]
12. Kua, J.; Armitage, G.; Branch, P.; But, J. Adaptive Chunklets and AQM for Higher-Performance Content Streaming. *ACM Trans. Multimed. Comput. Commun. Appl.* **2019**, *15*, 115. [CrossRef]
13. Pan, J.; McElhannon, J. Future edge cloud and edge computing for internet of things applications. *IEEE Internet Things J.* **2017**, *5*, 439–449. [CrossRef]
14. Premsankar, G.; Di Francesco, M.; Taleb, T. Edge computing for the Internet of Things: A case study. *IEEE Internet Things J.* **2018**, *5*, 1275–1284. [CrossRef]
15. Solaiman, E.; Ranjan, R.; Jayaraman, P.P.; Mitra, K. Monitoring internet of things application ecosystems for failure. *IT Prof.* **2016**, *18*, 8–11. [CrossRef]
16. Terry, D. Toward a new approach to IoT fault tolerance. *Computer* **2016**, *49*, 80–83. [CrossRef]
17. Moghaddam, M.T.; Muccini, H. Fault-tolerant iot. In Proceedings of the International Workshop on Software Engineering for Resilient Systems, Naples, Italy, 17 September 2019; pp. 67–84.
18. Nishiguchi, Y.; Yano, A.; Ohtani, T.; Matsukura, R.; Kakuta, J. IoT fault management platform with device virtualization. In Proceedings of the 2018 IEEE 4th World Forum on Internet of Things (WF-IoT), Singapore, 5–8 February 2018; pp. 257–262.

19. Kodeswaran, P.A.; Kokku, R.; Sen, S.; Srivatsa, M. Idea: A system for efficient failure management in smart iot environments. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications and Services, Singapore, 26–30 June 2016; pp. 43–56.
20. Di Modica, G.; Gulino, S.; Tomarchio, O. IoT fault management in cloud/fog environments. In Proceedings of the 9th International Conference on the Internet of Things, Bilbao Spain, 22–25 October 2019; pp. 1–4.
21. Kaiwartya, O.; Abdullah, A.H.; Cao, Y.; Lloret, J.; Kumar, S.; Shah, R.R.; Prasad, M.; Prakash, S. Virtualization in wireless sensor networks: Fault tolerant embedding for internet of things. *IEEE Internet Things J.* **2017**, *5*, 571–580. [CrossRef]
22. Rullo, A.; Serra, E.; Lobo, J. Redundancy as a measure of fault-tolerance for the Internet of Things: A review. In *Policy-Based Autonomic Data Governance*; Springer: Cham, Switzerland, 2019; pp. 202–226.
23. Ratasich, D.; Khalid, F.; Geissler, F.; Grosu, R.; Shafique, M.; Bartocci, E. A roadmap toward the resilient internet of things for cyber-physical systems. *IEEE Access* **2019**, *7*, 13260–13283. [CrossRef]
24. Pahl, C.; Ioini, N.E.; Helmer, S.; Lee, B. An architecture pattern for trusted orchestration in IoT edge clouds. In Proceedings of the 2018 Third International Conference on Fog and Mobile Edge Computing (FMEC), Barcelona, Spain, 23–26 April 2018; pp. 63–70. [CrossRef]
25. Ahuja, S.P.; Wheeler, N. Architecture of fog-enabled and cloud-enhanced internet of things applications. *Int. J. Cloud Appl. Comput. (IJCAC)* **2020**, *10*, 1–10. [CrossRef]
26. Hassan, N.; Gillani, S.; Ahmed, E.; Yaqoob, I.; Imran, M. The Role of Edge Computing in Internet of Things. *IEEE Commun. Mag.* **2018**, *56*, 110–115. [CrossRef]
27. Javed, A.; Heljanko, K.; Buda, A.; Främling, K. CEFIoT: A fault-tolerant IoT architecture for edge and cloud. In Proceedings of the 2018 IEEE 4th World Forum on Internet of Things (WF-IoT), Singapore, 5–8 February 2018; pp. 813–818. [CrossRef]
28. Silva, J.D.C.; Rodrigues, J.J.P.C.; Saleem, K.; Kozlov, S.A.; Rabêlo, R.A.L. M4DN.IoT-A Networks and Devices Management Platform for Internet of Things. *IEEE Access* **2019**, *7*, 53305–53313. [CrossRef]
29. Dang, H.; Quan, L. SD-IoTR: An SDN-based Internet of Things reprogramming framework. *IET Netw.* **2020**, *9*, 305–314.
30. Bera, S.; Misra, S.; Roy, S.K.; Obaidat, M.S. Soft-WSN: Software-Defined WSN Management System for IoT Applications. *IEEE Syst. J.* **2018**, *12*, 2074–2081. [CrossRef]
31. Wu, D.; Arkhipov, D.I.; Asmare, E.; Qin, Z.; McCann, J.A. UbiFlow: Mobility management in urban-scale software defined IoT. In Proceedings of the 2015 IEEE Conference on Computer Communications (INFOCOM), Hong Kong, China, 26 April–1 May 2015; pp. 208–216.
32. Medjek, F.; Tandjaoui, D.; Djedjig, N.; Romdhani, I. Fault-tolerant AI-driven Intrusion Detection System for the Internet of Things. *Int. J. Crit. Infrastruct. Prot.* **2021**, *34*, 100436. [CrossRef]
33. Power, A.; Kotonya, G. A microservices architecture for reactive and proactive fault tolerance in iot systems. In Proceedings of the 2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM), Chania, Greece, 12–15 June 2018; pp. 588–599.
34. Grover, J.; Garimella, R.M. Reliable and Fault-Tolerant IoT-Edge Architecture. In Proceedings of the 2018 IEEE SENSORS, New Delhi, India, 28–31 October 2018; pp. 1–4. [CrossRef]
35. Hasan, M.; Al-Turjman, F. Optimizing Multipath Routing with Guaranteed Fault Tolerance in Internet of Things. *IEEE Sens. J.* **2017**, *17*, 6463–6473. [CrossRef]
36. MTBF, MTTR, MTTA and MTTF: Understanding a Few of the Most Common Incident Metrics. Available online: https://www.atlassian.com/incident-management/kpis/common-metrics (accessed on 24 November 2021).
37. Engelhardt, M.; Bain, L.J. On the mean time between failures for repairable systems. *IEEE Trans. Reliab.* **1986**, *35*, 419–422. [CrossRef]
38. Kimura, M.; Yamada, S.; Osaki, S. Statistical software reliability prediction and its applicability based on mean time between failures. *Math. Comput. Model.* **1995**, *22*, 149–155. [CrossRef]
39. Michlin, Y.H.; Grabarnik, G.Y. Sequential testing for comparison of the mean time between failures for two systems. *IEEE Trans. Reliab.* **2007**, *56*, 321–331. [CrossRef]
40. Michlin, Y.H.; Grabarnik, G.Y.; Leshchenko, E. Comparison of the mean time between failures for two systems under short tests. *IEEE Trans. Reliab.* **2009**, *58*, 589–596. [CrossRef]
41. Glynn, P.W.; Heidelberger, P.; Nicola, V.F.; Shahabuddin, P. Efficient estimation of the mean time between failures in non-regenerative dependability models. In Proceedings of the 25th conference on Winter Simulation, Los Angeles, CA, USA, 12–15 December 1993; pp. 311–316.
42. Zagirnyak, M.; Prus, V. Use of neuronets in problems of forecasting the reliability of electric machines with a high degree of mean time between failures. *Prz. Elektrotechniczny (Electr. Rev.)* **2016**, *92*, 132–135. [CrossRef]
43. Suresh, N.; Jayant, N. 'Mean time between failures': A subjectively meaningful video quality metric. In Proceedings of the 2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings, Toulouse, France, 14–19 May 2006; Volume 2.
44. Duane, J. Learning curve approach to reliability monitoring. *IEEE Trans. Aerosp.* **1964**, *2*, 563–566. [CrossRef]
45. Rushdi, A.M.A.; Hassan, A.K.; Moinuddin, M. System reliability analysis of small-cell deployment in heterogeneous cellular networks. *Telecommun. Syst.* **2020**, *73*, 371–381. [CrossRef]

46. Liu, J.; Zhang, Q. Offloading schemes in mobile edge computing for ultra-reliable low latency communications. *IEEE Access* **2018**, *6*, 12825–12837. [CrossRef]
47. Liu, C.F.; Bennis, M.; Poor, H.V. Latency and reliability-aware task offloading and resource allocation for mobile edge computing. In Proceedings of the 2017 IEEE Globecom Workshops (GC Wkshps), Singapore, 4–8 December 2017; pp. 1–7.
48. Liu, C.F.; Bennis, M.; Debbah, M.; Poor, H.V. Dynamic task offloading and resource allocation for ultra-reliable low-latency edge computing. *IEEE Trans. Commun.* **2019**, *67*, 4132–4150. [CrossRef]
49. Han, B.; Wong, S.; Mannweiler, C.; Crippa, M.R.; Schotten, H.D. Context-awareness enhances 5G multi-access edge computing reliability. *IEEE Access* **2019**, *7*, 21290–21299. [CrossRef]
50. Silva, I.; Leandro, R.; Macedo, D.; Guedes, L.A. A dependability evaluation tool for the Internet of Things. *Comput. Electr. Eng.* **2013**, *39*, 2005–2018. [CrossRef]
51. Benson, K.E.; Wang, G.; Venkatasubramanian, N.; Kim, Y.J. Ride: A resilient IoT data exchange middleware leveraging SDN and edge cloud resources. In Proceedings of the 2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI), Orlando, FL, USA, 17–20 April 2018; pp. 72–83.
52. Qiu, T.; Luo, D.; Xia, F.; Deonauth, N.; Si, W.; Tolba, A. A greedy model with small world for improving the robustness of heterogeneous Internet of Things. *Comput. Netw.* **2016**, *101*, 127–143. [CrossRef]
53. Kwon, J.H.; Kim, E.J. Failure Prediction Model Using Iterative Feature Selection for Industrial Internet of Things. *Symmetry* **2020**, *12*, 454. [CrossRef]
54. Dinh, N.T.; Kim, Y. An efficient availability guaranteed deployment scheme for IoT service chains over fog-core cloud networks. *Sensors* **2018**, *18*, 3970. [CrossRef] [PubMed]
55. Makhshari, A.; Mesbah, A. IoT bugs and development challenges. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 22–30 May 2021; pp. 460–472.