IoT Management with Container Orchestration

Cristian Figueroa
Department of Systems and Computer Eng.
Carleton University
Ottawa, Canada
cristianfigueroa @cmail.carleton.ca

Timothy Knowles
Department of Systems and Computer Eng.
Carleton University
Ottawa, Canada
timothyknowles @cmail.carleton.ca

Vineet Kukreja
Department of Systems and Computer Eng.
Carleton University
Ottawa,Canada
vineetkukjera@cmail.carleton.ca

Chung-Horng Lung
Department of Systems and Computer Eng.
Carleton University
Ottawa,Canada
chlung@sce.carleton.ca

IOT Availability
Container Orchestration
MQTT

2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB) | 979-8-3503-3386-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICEIB57887.2023.10170261

***Abstract*—Internet of Things (IoT) systems manage microcontroller devices through cloud infrastructure. The modern cloud infrastructure is composed of containerized applications that facilitate internal and external communications. Containers have been used in recent years in a wide range of applications. Containerization of applications and services allows improved modularity for traditional monolithic applications. Combined with IoT, containerization allows efficient allocation, fast execution, and deployment of hardware resources. Docker is the most popular and open-source containerization tool currently used in the field. However, the need to manage these various containers warranted the advent of container orchestration technologies to facilitate the deployment, status monitoring, and scaling of containerized applications. In this study, the container management engine Kubernetes was implemented through the Google Cloud Platform to create an IoT infrastructure composed of containers. The infrastructure used the Message Queuing Telemetry Transport (MQTT) protocol for machine-to-machine (M2M) communications. The design of the IoT infrastructure also involved system architecture trade-off analysis of architectural alternatives. The process improves both functional and non-functional requirements such as reliability, maintainability, and scalability. The implemented IoT infrastructure in potential applications and reliable for failure scenarios had containers and container orchestration in the Google Kubernetes Engine. The orchestration infrastructure facilitates efficient and effective management of IoT devices.**

Keywords—IoT, container, container management, Kubernetes, Docker, MQTT, Raspberry Pi

I. Introduction

In recent years, the Internet of Things (IoT) has become a significant technology. IoT is the system of network-enabled devices performing various functions and communications. Devices such as kitchen and home appliances, thermostats, and baby monitors can be connected to the Internet via embedded chips to generate notifications and data which are directed to central servers. In addition, devices receive messages through the Internet for a specific action or a change to a configuration. As the number of IoT devices continues to increase, how to efficiently manage these devices and the traffic is important.

This study is conducted to explore how to create an infrastructure that can efficiently manage IoT devices. Several factors are considered, including the communication protocol used between servers and IoT devices for one-time connection, e.g., HTTP or a longer-term connection such as MQTT. The design of the server-side applications to send/receive data to/from IoT devices is also investigated for how to make the server meet non-functional requirements. We present an approach for the management of IoT devices using a cloud-hosted and containerized application cluster. The cluster is managed using a container orchestrator. The cluster and its contents as well as the container orchestrator are referred to as the cloud infrastructure throughout this paper.

A "Smart Home" is an application considered with this system to demonstrate how it is applied to a realistic scenario. The Smart Home implies the existence of "smart" devices which are defined as Internet-enabled devices capable of performing tasks manually or automatically. The Smart Home also has sensory devices to monitor the environment and collect data. These smart and sensory devices exemplify the IoT devices that the cloud infrastructure manages. The components of the design are intended to support the container orchestration and management infrastructure (such as Kubernetes), whereas other components support the Smart Home application such as the GUI.

Kubernetes facilitates backup or recovery from failures. In addition to the availability feature, the following non-functional requirements are outlined for the cloud infrastructure based on the objective of this study.

1) *Reliability*: The infrastructure can expect the same behavior any time during operation.
2) *Maintainability*: The infrastructure should allow for changes or updates to be made to it.
3) *Scalability*: This infrastructure should be scalable to larger number of requests or replicas in Kubernetes.

The rest of this article is organized as follows. Section II describes the background and related work. Section III discusses the system architecture and design. Section IV presents the implementation of the system. Section V covers experimental results. Finally, section VI concludes this study.

II. Background and Related Works

This section highlights background information and related work, including containers, MQTT, the Google Cloud platform, and Raspberry Pi.

A. Containers and Container Management

A container is a virtual environment that runs within a host machine with the hardware and software resources of the host machine, e.g., the operating system and CPU [1]. The contents of a container consist of an application with the minimum dependencies needed to run such as libraries, configuration files, and other binaries [2]. Because these containers are bundled to run, it is easy to deploy to other

machines. Containers are simpler and require less overhead than traditional virtual machines. Docker [5] is a common container platform.

To manage the cluster of containers, an orchestrator is needed. Kubernetes [4] provides automated deployment, scaling, and maintenance of containers in a cluster [1]. Kubernetes clusters are composed of master and worker nodes. Masters nodes handle the deployment and fault recovery and manage the overall state of the system and execute the 'work'. Worker nodes contain pods. Each pod hosts an instance of a container. An important feature of Kubernetes is the "self-healing" to restart and move containers to different nodes and to make them reliable preventing a node failure [3]. Kubernetes and Docker containers are used together. Kubernetes has a command line tool called kubectl which allows commands to be run against a cluster and deploy applications [6]. Cluster configurations are tweaked using a deployment that describes the desired state of the cluster that Kubernetes maintains [7]. In addition, a *service* exposes network applications within the cluster to external traffic [8].

B. Message Queuing Telemetry Transport (MQTT)

MQTT is a publish/subscribe message protocol used between devices [9]. One of the main benefits of MQTT is low bandwidth usage for having many devices on a local area network communicating with a cloud provider. The protocol employs a broker which behaves as an intermediate, relaying data sent between subscribers and publishers. The system is organized through topics where the broker distributes a publisher's message to all clients that are subscribed to a topic. The protocol relies on TCP and is initiated with an acknowledged/received exchange.

C. Google Cloud Platform (GCP)

GCP is used as the Cloud Computing service in this study. GCP provides tools for the development and deployment of the applications that make up the cloud infrastructure with various tools. The tools used in this study include GKE (Google Kubernetes Engine) and kubectl, SQL Database, IoT Core, and Pub/Sub.

D. Raspberry Pi (RPi) Microcomputer and SenseHat Add-On

The RPi is a single-board microcomputer that is frequently used for computing development. The microcomputer integrates auxiliaries via GPIO, connects to networks, and has common functionalities of computers. The SenseHat is an attachment to the RPi that connects through GPIO. SenseHat acquires environmental data including temperature, pressure, and humidity through sensors.

III. System Architecture and Design

This section describes the system architecture and design after conducting critical trade-off analyses [10−12] for five design alternatives. Due to the space limit, only the final design is presented in Fig. 1. The first section (a) is contained within a Local Area Network (LAN). Sections (a), (b), and (c) show three IoT devices surrounding a central Gateway. The IoT devices provide functions such as the collection of sensory data and the activation of a light and a fan. In this case, the RPI with its connected components represents the IoT devices. The design details of the central Gateway are beyond the scope of this study, and the Gateway acts as a communication medium between the IoT devices and the cloud infrastructure. A cluster of Docker containers on the Gateway collects data from a sensory device and forwards it to the cloud through HTTP. The MQTT broker operating on the Gateway is a part of the "MQTT bridge" between the LAN smart devices and cloud infrastructure. This MQTT bridge is used as a communication data path between IoT devices and cloud infrastructure.

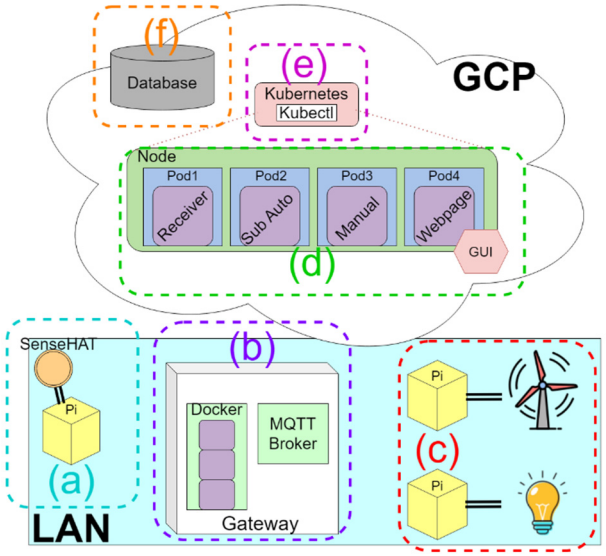

Fig. 1. System architecture.

The cloud infrastructure is composed of several components and is managed by Kubernetes as shown in Fig. 1(d). The *Receiver* handles requests from the Gateway containing sensory data. This data is then stored in a database (Fig. 1(f)) that is hosted by GCP. The *Sub Auto* container (referred to later as the Automatic Configuration application) coordinates the automatic control of smart devices. The *Manual* container handles the manual control of smart devices. Finally, the *Webpage* container hosts a web-based GUI for user interaction with IoT devices. The cluster is managed by Kubernetes (Fig. 1(e)). kubectl is the command line tool used to interact with and configure the cluster.

The cloud infrastructure in Fig. 1 resides under GCP which allows the hosting and deployment of Kubernetes clusters. Many non-functional requirements outlined in Section I are handled through GCP and Kubernetes features. Figure 2 shows the Kubernetes cluster design. The design uses multiple Kubernetes worker nodes for system availability since Kubernetes moves pods to different (idle) nodes in the case of an active node failure. The design uses four containerized applications that reside within the active worker node. Each of the four containers is encompassed within a pod, and the proposed design uses multiple pods per container. This ensures system reliability since Kubernetes monitors pod health and replaces faulty pods. Through kubectl, pods can be killed and restarted while the cluster remains active. This allows the maintainability of the infrastructure, since any changes can be made to applications and then applied to the cluster with a pod restart.

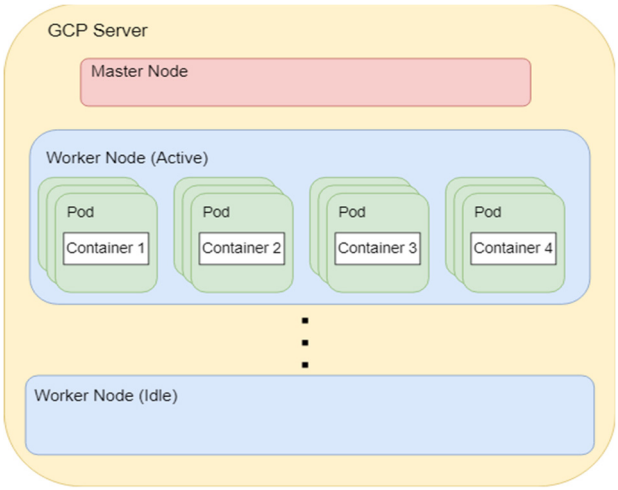

Fig. 2. Kubernetes cluster design.

IV. System Implementation

This section describes the implementation of proof-of-concept applications.

A. Application Descriptions

In the design (Fig. 1), the sensory device is built in a SenseHAT connected to the RPi via the GPIO pins. This allows the RPi to monitor the ambient temperature, pressure, and humidity. The RPi is configured to send the environmental data to the Gateway through UDP (User Datagram Protocol) unicast messages [13]. This sensory device also publishes sensory data through the MQTT bridge described later.

The two "smart devices" are a fan and a light for proof-of-concept connected to RPi. Both the fan and light are set at 5 V and wired to their respective RPis to be controlled via the cloud infrastructure. The light and fan had to be wired through a bipolar junction transistor (BJT) and a diode on a 5 V power line, as a direct connection. Figure 3 shows a schematic view of the devices. These two devices communicate with the Gateway via Wi-Fi. A script is constantly running on both devices to receive instructions from the cloud and trigger the light or fan on or off.

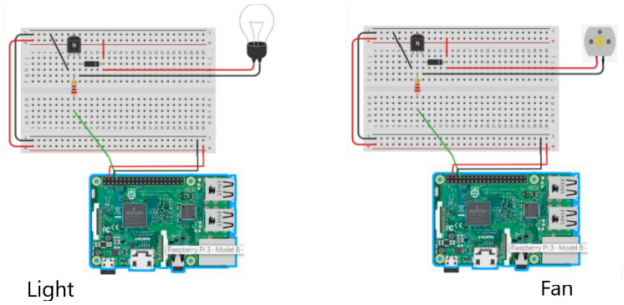

Fig. 3. Wiring diagram of two "Smart Devices".

Figure 1(b) shows the Gateway hosting Docker containers and an MQTT broker. The Docker containers hosted applications to forward sensory data received from the RPi / SenseHat to the cloud infrastructure. For the MQTT bridge implementation, GCP's *IoT Core* and *Pub Sub* tools are used. Within the *IoT Core,* a registry is created, and within this registry, a telemetry topic is created to serve as the median for publishing and subscribing. Figure 4 displays the main data paths through the MQTT bridge.

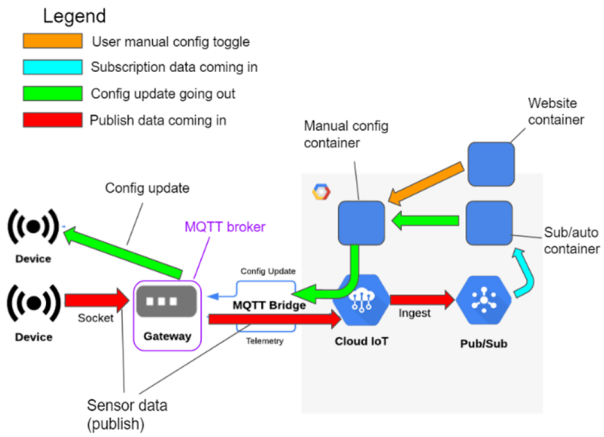

Fig. 4. MQTT bridge datapaths.

The red arrows indicate the incoming device data being published to the topic in the registry. The telemetry data is then sent via the MQTT bridge, ingested, and 'published'. The *Automatic Configuration* application receives incoming data through a subscription as indicated by the cyan arrow. The bridge also offers a path for the cloud applications to change the status of the LAN smart devices through configuration updates. Either the *Manual Configuration* application or the *Automatic Configuration* application issues a configuration update using the path in the green arrow. The orange arrow indicates the data flow should a user manually toggle on/off the smart device via the GUI.

In the implementation of the cloud infrastructure (Fig. 1(d), (e), and (f)), the *Receiver* application listens for HTTP POST requests from the Gateway containing sensory data and inserts that data into the *Database*. This application is implemented in the form of a PHP script. PHP is chosen for this application for its ease of interaction with databases using a PHP Data Object (PDO). PDOs are an object-oriented API that provides an abstraction layer with many databases including SQL *Databases* [9]. The *Database* stores information about IoT devices. This is implemented in the form of a MySQL *Database*. Each entry in the database has a date, a time, a type of device, and a 'value' depending on the device type. For sensory devices, the value stored is sensory data such as a temperature reading. For smart devices, the data stored is a device status such as 'ON', 'OFF' or "AUTO". Entries in the database also have an entry ID. This ID is automatically incremented with each new entry added. This allows the data to be easily queried in reverse chronological order from the entry ID. Historical data collected from IoT devices can be easily queried later.

The purpose of the *Webpage* application is to allow users to interact with Smart Home IoT devices through a GUI hosted on a website. This application is implemented in the form of a PHP script acting as a backend application. The GUI is implemented using simple HTML and CSS for styling. The landing page allows a user to enter a Home ID and select a Smart Home device type. Once the user enters the Home ID and Device Type, the user can see additional information about the device. What the GUI displays depends on the Device Type specified. If the Device Type is temperature, pressure, or humidity, the GUI shows a table filled with historical sensory data. If the Device Type is Light or Fan, the GUI shows the latest status (either ON or OFF), and gives the user 3 options: Turn ON, Turn OFF, or AUTO. The AUTO option signifies that the device turns on or off based on

predefined cases. After selecting any of the above options, an HTTP POST request is sent to the *Receiver* application containing the new device status. This way, the latest device status is always available in the *Database*.

The *Manual Configuration* application handles user requests for turning smart devices on or off manually. The *Manual Configuration* application is designed as a web application similar to the Cloud *Receiver* and the *Webpage*. The application is implemented using a Python script for ease of use with Google Cloud libraries to interact with IoT Core. Since this application receives HTTP traffic, Flask is used, which is a Python library for hosting web servers [14]. The *Manual Configuration* application works by waiting for HTTP POST requests which contain the device ID and the status of the device. The device ID is the one registered to the device in the *IoT Core*. To change the device status, a configuration update is issued through the MQTT bridge. Also, the *Database* is updated with the new status of the device. Figure 5 shows the sequence from the top-level component perspective. The user first submits the "config" which means they select a new device status (either "ON" or "OFF") using the GUI. A request is sent to the *Manual Configuration* application which forwards the request to the Cloud *Receiver* to update the database with the device's new status. Afterward, the configuration update is received by the MQTT broker who forwards the configuration to the RPi. Finally, the RPi changes the GPIO value for the external component.

The *Automatic Configuration* application is responsible for turning smart devices on or off automatically based on predefined cases. Similar to the *Manual Configuration* application, this application is implemented using a Python script due to dependencies with Google Cloud libraries. The application operates by determining whether a smart device status is set to "AUTO" (by a user) by querying the database for device status entries. If a device is set to auto, it subscribes to the sensory IoT device data topic to read the latest telemetry (sensory) data being pushed by the sensory device. If the sensory data surpasses a certain threshold, e.g., 25 °C, the smart device is turned either on or off depending on its previous state. The idea is simply to automatically trigger one device based on information coming from a different device. In the context of a Smart Home, this turns a fan on if the room temperature reaches a certain number. The threshold used is arbitrary and only used to exemplify the functionality.

Figure 6 shows the *Automatic Configuration* application querying the *Database*. Once the device status is set to "AUTO", it begins to query the topic subscription for the latest telemetry data coming from the IoT device. It compares the sensory data against a predefined threshold and decides whether to turn the device on or off. It then sends the updated configuration to the MQTT broker which forwards the message to the RPi controlling which in turn changes the GPIO value for the external component.

B. Containerization and Cluster Configuration

The containerization process is generally the same for all applications. The main difference is the configuration that is used for each application. The applications need to have their respective environments set up to run. Using Docker is simple since official Docker images such as Python-based images can be pulled into the container environment. This allows any Python script to run within the container. In addition, any necessary modules/packages can be imported. For example, the 'pdo-my-sql' extension can be installed in the container environment which allows the use of the aforementioned PDOs within a PHP script. All of the above configurations are specified within a Dockerfile. Once the Dockerfile is written for each application, the Docker image is built and tagged using the "docker build" and "docker tag" commands. Finally, the image is pushed to the Google Container Registry (GCR). Once the image is in the GCR, it can be accessed by Kubernetes which deploys the containers.

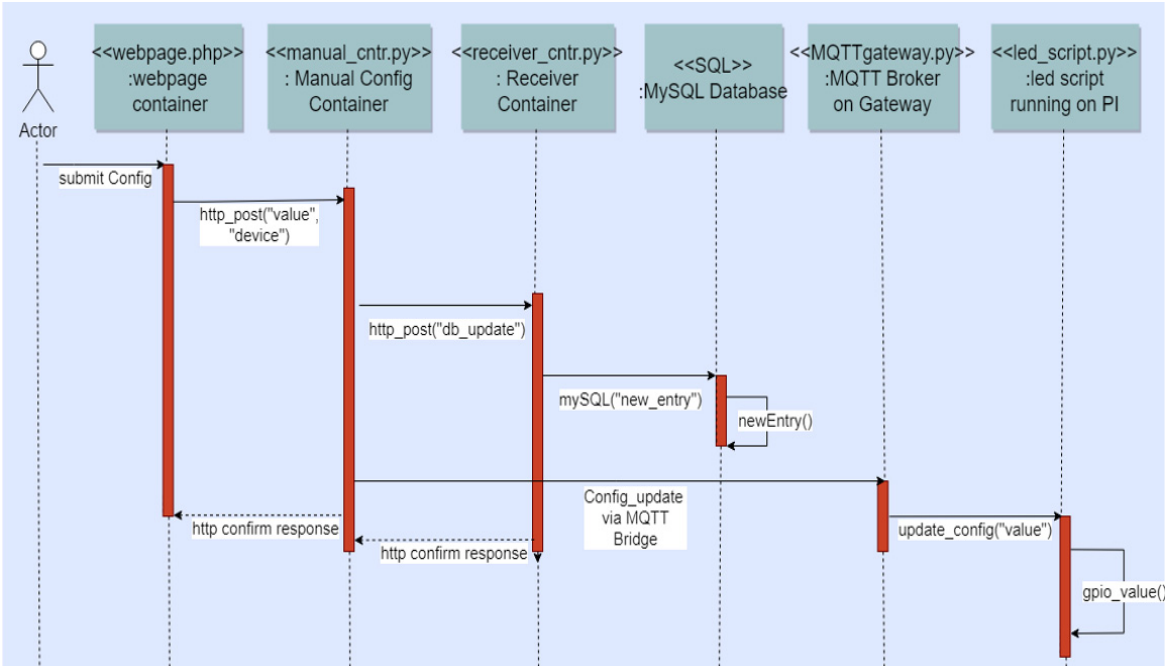

Fig. 5. Sequence flow of manual triggering of the IoT devices.

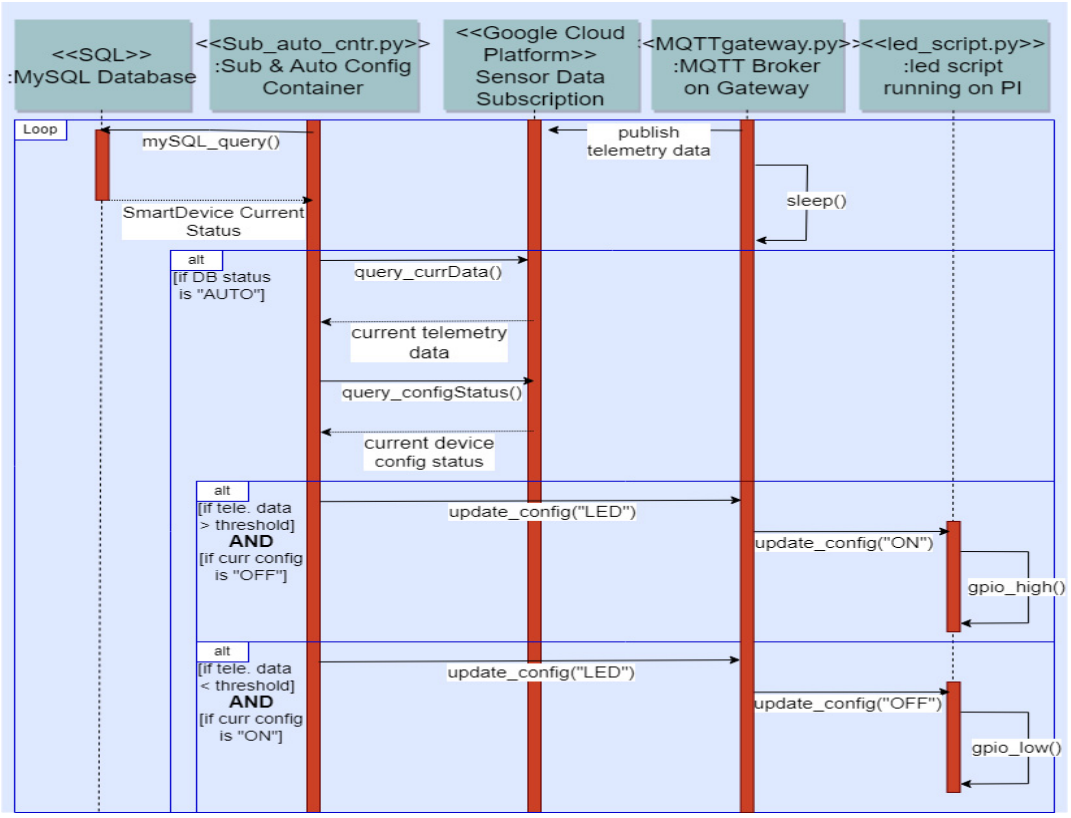

Fig. 6. Sequence flow of automatic handling of the IoT devices

Kubernetes manages the containerized Cloud applications. The cloud infrastructure is deployed through a Kubernetes cluster. When creating the cluster, various parameters are chosen such as the location of the cluster nodes and the number of nodes. Three nodes are chosen during testing. Once the cluster is created, deployments and services are made. Both deployments and services are specified within a YAML file. The deployment file specifies four deployments for each of the cloud applications. Within the deployment, the number of Pods 'replicas' is specified. During testing, 3 replicas are made for each of the four applications. In addition, the 'image' is specified which contains a link to the pre-made Docker image in the GCR. The service file is made to specify three services for each of the applications that receive HTTP traffic. Within each service, an IP address needs to be specified to access the application. These IP addresses were allocated through GCP.

For non-functional requirements evaluation, we consider *reliability*, *scalability*, and *maintainability*. Since the cluster state depends on the deployment configuration, any changes to the deployment result in changes to the cluster state. In testing *reliability*, the deployment file is manually tweaked and then removed/reapplied to the cluster. Kubernetes automatically corrects the state of the cluster to account for any changes to the deployment. *Scalability* is tested by increasing the number of replicas per Pod. In theory, this attribute is supported by Kubernetes in various reports. More experiments are demanded for further verification. For *maintainability*, once a change is made to an application, the Docker image is recompiled, and the deployment is reapplied. Since the image link is specified within the deployment, Kubernetes pulls the updated Docker image when redeploying the application. This is demonstrated by changing stylings for the Webpage application, generating a Docker image, and redeploying the application while the cluster was still live.

V. Experimental Results

Overall, the infrastructure performed its intended function of managing IoT devices. The smart devices (light and fan) are operated manually via the GUI (*Webpage*) or set to operate automatically depending on changes in the environment. The automatic feature is shown by setting a temperature threshold, where any temperature above it turns on the fan. Afterward, the SenseHat RPi software application is used as a test stub to send dummy data. Once the temperature rises above the temperature threshold, the fan turns on. Similar results are obtained for the light. For sensory devices, sensory data is successfully forwarded to the cloud infrastructure and stored or retrieved in the database through the GUI. The entire cloud infrastructure is deployed at once with all applications running simultaneously. All containers are hosted and maintained within the cluster by Kubernetes.

VI. Conclusion

The system performs well to achieve the initial objective. Software architecture trade-off analysis [10,12] or sensitivity analysis [11] are essential and effective in supporting and achieving this objective. Containerization of applications combined with Kubernetes management enables an efficient deployment of a system. Using MQTT as a communication protocol proved to be effective for the management objective. Certain parts of the system such as the Gateway to cloud sensory data forwarding through HTTP are adjusted to use MQTT. The design needs to be explored for further benefits such as the scalability of the cloud infrastructure in response to an increasing number of IoT devices. The system can also be made available by strategically configuring the geographical location of Kubernetes worker nodes in the

cluster. In addition, MQTT is advantageous for low power devices which could include battery or solar powered IoT devices.

REFERENCES

[1] A. Javed, *Container-based IoT Sensor Node on Raspberry Pi and the Kubernetes Cluster Framework*, M.S. Science in Technology Thesis, School of Science, Aalto Univ., Espoo Finland, 2016.

[2] R. Muddinagiri, S. Ambavane, and S. Bayas, "Self-Hosted Kubernetes: Deploying Docker Containers Locally with Minikube", *Proc. of International Conf. on Innovative Trends & Advances in Eng. and Tech.*, 2019, pp. 239–243.

[3] J. Shah and D. Dubaria, "Building Modern Clouds: Using Docker, Kubernetes & Google Cloud Platform", *Proc. of IEEE 9th Annual Computing and Communication Workshop and Conf.*, 2019, pp. 184–189.

[4] Kubernetes, https://kubernetes.io/, Accessed: March, 2023.

[5] Docker, https://www.docker.com/, Accessed: March, 2023.

[6] Install Tools, https://kubernetes.io/docs/tasks/tools/, Accessed: March 2023.

[7] Deployment Kunbernetes, https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, Accessed: March 2023.

[8] Service Kunbernetes, https://kubernetes.io/docs/concepts/services-networking/service/, Accessed: March 2023.

[9] T. Jaffey, "MQTT and CoAP, IoT Protocols," Eclipse Foundation, //www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php, Accessed: March 2023].

[10] "Architecture Tradeoff Analysis Method", Carnegie Mellon Software Engineering Institute, Accessed: March 2023.

[11] C.-H. Lung and K. Kalaichelvan, "An Approach to Software Architecture Sensitivity Analysis", *International Journal of Software Eng. and Knowledge Eng*., vol. 10, no. 1, 2000, pp. 97–114.

[12] L. Bass, P. Clement, and R. Kazman, *Software Architecture in Practice*, 4th Edition, O'Reilly, Aug. 2021.

[13] R. A. Nathi and D. Sutar, "Secured and Lightweight Communication Scheme on UDP for Low End IoT Devices", *Proc. of IEEE International Conf. on Advanced Networks and Telecommunications Systems (ANTS)*, 2019, pp. 1–6.

[14] R. Connolly and R. Hoar, *Fundamentals of Web Development*. Pearson, 2018.