Delft University of Technology
Master of Science Thesis in Embedded Systems

Validating Hue Bridge firmware with Device Virtualization and Kubernetes

Siyuan Fang

Networked Systems

TU Delft Delft University of Technology

Validating Hue Bridge firmware with Device Virtualization and Kubernetes

Master of Science Thesis in Embedded Systems

Embedded Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Siyuan Fang

2023-06-18

Author
Siyuan Fang

Title
Validating Hue Bridge firmware with Device Virtualization and Kubernetes

MSc Presentation Date

2023-06-28

Graduation Committee

dr. Marco Antonio Zúñiga Zamalloa (chairman)	Delft University of Technology
dr. Guohao Lan	Delft University of Technology
ir. Dmitry Korolev	Signify

Abstract

In recent years, with the rapid expansion of IoT (Internet of Things) devices, more and more research and commercial projects have focused on various application areas of IoT. Signify, as a leading player in the smart home industry, has been deeply involved in this field for many years, particularly focusing on smart lighting for smart homes and providing consumers with a whole-house smart lighting solution (Hue System). The Hue System consists of several IoT devices, such as the Hue Bridge, Hue Bulbs, and Hue Accessories. This research paper specifically targets the IoT field and aims to reduce the firmware update cycle of the Hue Bridge.

The Hue Bridge, which serves as a central device in the Philips Hue internet-connected lighting system, connects users with other Hue devices. However, the Hue Bridge faces the challenge of inefficient firmware updates, which require validation engineers to wait for 6-8 weeks to ensure firmware reliability. To address this issue, this paper proposes a virtual system solution that improves the virtualization procedure of the Hue Bridge devices and utilizes Kubernetes for large-scale deployment to accelerate the generation of diagnostic data. Furthermore, a Use Case Model is established based on users' daily data, and a model based on Frequent Pattern Mining is applied to simulate users' daily behaviors in the Kubernetes Deployment.

To validate our virtual system, we designed validation experiments from multiple perspectives, including validation of the use case model and automated feedback. Our validation results demonstrate that this system enables more efficient and convenient acquisition of automated feedback (issues/bugs), while significantly enhancing the generation of diagnostic data in the firmware update cycle. Moreover, it offers advantages such as high availability, convenience, and cost-effectiveness in deployment. This research provides valuable references and insights for firmware update-related studies in the IoT domain.

"Everything that can be automated will be automated."
—— Robert Cannon

Preface

Embedded systems and the Internet of Things (IoT) are the driving forces behind my choice to pursue a Master's degree in Embedded Systems at Delft University of Technology. Fueled by my interest in smart homes, I have constantly challenged and pushed myself by engaging in various embedded systems-related projects. It is my great honor to have been invited by Signify to participate in the embedded development of the Hue Bridge during my graduate thesis.

At the very begining, I would like to express my gratitude to the many individuals who have supported me throughout this year. Firstly, I am most grateful to my industry mentor, Dmitry Korolev, and my academic mentor, Prof. Marco, for their patient guidance on a weekly basis. Without their assistance and encouragement, each step of my research would have been incredibly challenging. It is through their support that I have been able to achieve the expected project outcomes. Similarly, I would like to thank every member of the Signify IoT team who patiently answered my questions and provided invaluable technical assistance, allowing me to acquire in-depth knowledge of the Hue System.

I would also like to express my gratitude to Dmitry and Sonny, who attended my thesis meetings every Tuesday. They attentively listened to my progress reports and consistently provided accurate corrections and valuable suggestions. Their presence made me feel that I was not alone in my exploration. Prof. Marco was my professor during my first year of study, and I was deeply impressed by his lectures. As a result, I proactively approached him to be my thesis supervisor. Throughout the guidance of my thesis, Prof. Marco has always been prompt in responding to my inquiries and has consistently guided me in the most logical and correct direction.

I would also like to thank Prof. Guohao for serving as a Committee Member. Thank you for your time!

Finally, I would like to express my gratitude to myself for not giving up. The journey from Delft to Eindhoven for my thesis was destined to be a solitary one, and I am thankful that I persevered and did not give up.

Siyuan Fang

Eindhoven, The Netherlands
21st June 2023

Contents

Chapter 1

Introduction

With the rapid expansion of IoT (Internet of Things) devices and the continuous development of IoT technology, more and more IoT devices are being widely used in various industries, ranging from smart homes to smart cities, from industrial automation to smart agriculture, and from remote medical care to smart lighting. IoT devices work together and developers regularly update the device firmware according to user demands to build reliable device networks for these application scenarios. In these application scenarios, the accuracy and stability of IoT device firmware are particularly important for the reliability of device networks. Therefore, the R&D teams of devices typically test and verify firmware during every release to enhance user experience and discover and solve potential firmware issues.

However, due to the diversity of hardware and software environments in IoT devices, coupled with the usual lack of coordination or sync between software and hardware development, firmware testing and verification for IoT devices can be exceptionally complex and time-consuming, resulting in a significantly extended release cycle. Therefore, improving the efficiency of firmware testing and verification to accelerate the release cycle has become an important research topic in the IoT field.

This paper focuses on the field of smart lighting and the device we investigate is the Hue Bridge in the Philips Hue system. Philips Hue is a smart lighting solution for home scenarios proposed by Signify (formerly known as Philips Lighting), consisting of five main components: Hue Light, Hue Bridge, Hue Accessories, Hue App, and Hue Cloud. The Hue Bridge is the heart of the Philips Hue system, connecting users' devices to smart lamps and providing a Zigbee gateway service and rich automated lighting control for the entire home[28].

1.1 Hue System Introduction

This project is completed in Signify. Signify is a technology company specializing in the field of lighting. Signify was formerly known as Philips Lighting, and in 2018, Philips Lighting changed its name to Signify. As a pioneer in the lighting industry, Signify has rich industry experience and patented technologies in LED, smart lighting control, bulb design, and other fields, and is committed

to providing people with sustainable, energy-efficient, environmentally friendly, and intelligent lighting experiences.

Signify provides innovative lighting solutions covering the consumer sector, commercial lighting sector, and professional lighting sector. As a lighting system designed specifically for consumers by Signify, the Hue system provides a connected and intelligent lighting ecosystem for consumers based on the Zigbee network protocol.

The Hue System focuses on consumer home lighting and provides the consumer with a wide range of lighting products, lamps, light strips, floor lamps and many other products. The Hue system consists of five core components, In Figure 1.1 shown the relationship of the components:

- **Hue Bulb:** A hue bulb is a connected LED bulb. This means a hue bulb is always "on" or in "standby"
- **Hue Bridge:** Hue bridge is a smart gateway that serves as the central device in the lighting system, it connects users with other Hue devices. The Hue Bridge also serves as a central hub for your smart lighting setup, providing a more comprehensive smart home experience.
- **Hue Accessories:** Accessories include sensors, switches, smart buttons, etc.
- **Hue App:** The user interface of the system is the Hue app. Via the Hue app users can extend the system with new hue compatible lights and devices:
- **Hue Cloud:** Hue Cloud enables users to control their devices when they are not at home.

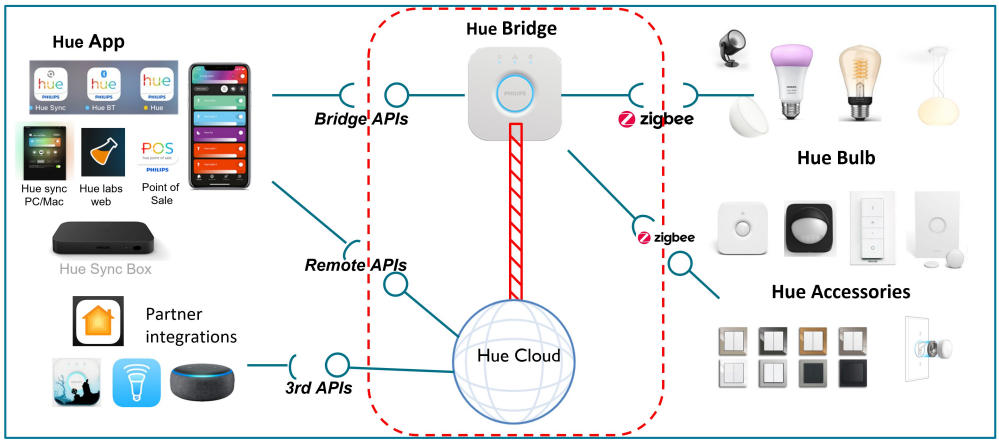


Figure 1.1: **Hue system overview**

Hue System is an smart home lighting solution that offers a wide range of features and applications. The following are the main applications and functions of the Hue system:

1. **Light Control:** The Philips Hue system allows users to control lights, adjust brightness, color temperature, and colors through the Hue app or switches. Users can control their lights anytime, anywhere, create custom scenes and presets to meet different needs and create desired lighting atmospheres.

2. **Color Lighting Effects:** Hue bulbs support 16 million colors, allowing users to choose different colors according to personal preferences and needs, creating unique lighting effects. For example, users can choose soft warm white for relaxation and reading or vibrant colored lights for parties and entertainment[28].

3. **Rule Engine:** The Hue system can be integrated with other smart devices, creating a more intelligent home experience. For example, users can set the lights to automatically turn on when the motion sensor detects someone at the door, or have the lights flash when receiving a phone call.

4. **Automation Schedules:** With the automation feature provided by the Hue Bridge, the Hue system can automatically adjust the lighting based on sunrise and sunset times. This eliminates the need for manual operation as the system adapts to the changing lighting needs throughout the day[27].

5. **Remote Control:** Users can remotely control their lights over the internet, regardless of their geographic location. This means that even when users are not at home, they can ensure that the lights are in the desired state, enhancing convenience and security[28].

6. **Voice Control:** The Hue system is compatible with third-party voice assistants such as Amazon Alexa, Google Assistant, and Apple Siri. Users can use voice commands to control all the lights in their homes, adding convenience to their lighting control experience[28].

Among these components, there is an important device that serves as the central device in the lighting system, it connects users and devices. It connects the whole system like a bridge, so we call it Hue Bridge.The Hue Bridge let users control their devices through applications. Signify chose Zigbee as their default communication protocol to connect to the lighting devices. The Hue Bridge also allows partner integrations, users can also use Alexa, and HomeKit to control their Hue devices. In this paper, the Hue Bridge is classified as an IoT device and will be the focus of our research.

1.2 Motivation

Philips Hue is an Internet-connected lighting system designed to transform the way users experience light inside their homes. It is one of the leading and largest home IoT products in the world. I belong to the IoT team of the Hue Bridge, which is responsible for the IoT services based on the bridge and cloud components. These services serve as the engine of the connected lighting system, improving security, latency, and availability for home-cloud connectivity worldwide. Our team ensures that the lighting system can be updated and provides an analytics and diagnostics infrastructure.

Signify follows the common firmware release cycle, which typically consists of several phases including planning, development, testing, deployment, and maintenance. Each phase involves specific activities and deliverables aimed at ensuring that the firmware is of high quality, meets customer requirements, and is delivered on time and within budget. Effective management of the firmware release cycle is essential for successful firmware development projects. Each development team completes the process of developing, testing, delivering, and releasing the new firmware for Hue Bridge in accordance with the prescribed steps and specifications.

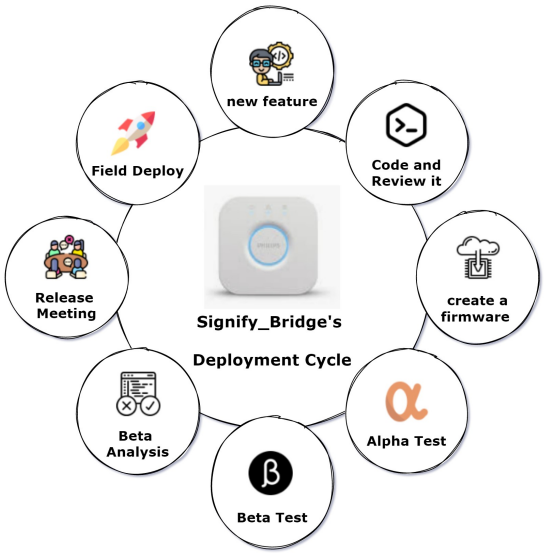


Figure 1.2: **Hue Bridge's firmware release cycle**

However, as the system's features continue to increase, the development team has found that the deployment efficiency of Hue Bridge cannot keep up with the growing user demand. The current deployment cycles for bridge firmware take around 6-8 weeks, which is too long to receive useful data/feedback from the field regarding reliability and availability, making it difficult to initiate improvements based on this data/feedback.

At the beginning of Chapter 1, we discussed the efficiency issues commonly encountered in firmware verification and updates in the field of IoT. These issues also exist in the development process of Hue Bridge devices. Due to the longer update cycles, the efficiency of firmware updates and the release schedule of new features for bridge devices are affected. In the rapidly changing IoT market, efficient and concise firmware release cycles are crucial to meet growing user demand and maintain the product's market competitiveness.

Therefore, Signify proposes to analyze the existing bridge device release cycles to find ways to improve update efficiency. This is also the research direction and core objective of this thesis. The Goals of this proposal are described as follows:

- Reduce feedback cycle time for IoT components from weeks to hours

- Improve the deployment architecture of Hue Bridge and make it more portable

- The deployment of firmware should be fast enough to make it part of the CI/CD(continuous integration and continuous delivery/continuous deployment[30]) pipeline

1.3 Contributions

The main contribution of this research is to propose a method for using virtualization technology to shorten the firmware release cycle of Hue Bridge devices. Specifically, we use Docker technology to create a virtual Bridge device and simulate user behavior and instructions using a use case model. Additionally, we use Kubernetes technology to deploy virtual containers on a large scale to supplement or replace beta test devices during the firmware release cycle. Our method can significantly reduce the time and resources required for the beta test phase in the firmware release cycle, thereby improving the efficiency and quality of software development. Our contributions include the following:

- Proposing a method for shortening the firmware release cycle of Hue Bridge devices based on virtualization technology. Compared to traditional beta test devices, our virtualization method can more flexibly simulate user behavior and instructions, thereby accurately testing the functionality and performance of the firmware.

- Using a use case model to simulate user behavior and instructions, which better simulates real-world usage scenarios and improves the accuracy and reliability of testing.

- Using Kubernetes technology to deploy virtual containers on a large scale, enabling the testing of multiple virtual Bridge devices simultaneously, thereby improving testing efficiency and coverage.

- Through experimentation, we demonstrate that our method significantly reduces the time and resources required for the beta test phase in the firmware release cycle, thereby improving the efficiency and quality of firmware development.

In conclusion, our research proposes an innovative method that can significantly shorten the time and resources required for the beta test phase in the firmware development cycle of Hue Bridge devices, thereby improving the efficiency and quality of firmware development. Our approach has significant reference value for manufacturers of household appliances in the IoT field, especially those with fixed firmware update cycles.

1.4 Project roadmap

We formulated five main stages for the . The following figure shows the Roadmap of this project.

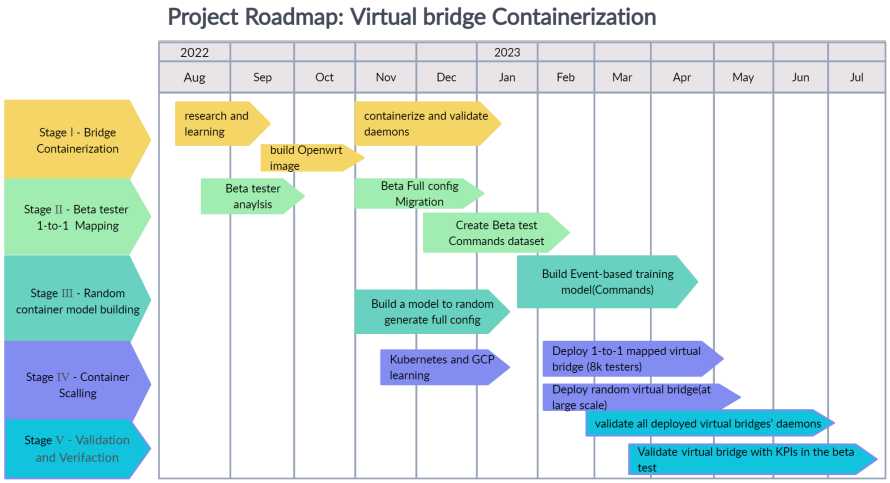


Figure 1.3: **Project roadmap**

1.5 Thesis Outline

The chapters of the thesis are organized as follows: Chapter 2 introduces the literature related to the work presented in this paper and its methodology. Chapter 3 presents the proposed solution and steps for virtualizing the Hue Bridge. Chapter 4 introduces and applies the Use Case Model and Frequent Pattern Mining algorithm in the Hue system. In Chapter 5, the implementation of the Use Case Model and the large-scale deployment of virtual bridges are discussed, and the validation results of real bridge devices are compared and analyzed. Finally, Chapter 6 discusses the conclusions, existing problems, and suggestions for future work.

Chapter 2

Related work

This chapter is structured as follows: Firstly, we introduce recent research and work related to IoT device firmware release and analyze the methods and shortcomings to improve the IoT device release cycle. Meanwhile, we conducted sufficient research on user behavior analysis of IoT devices in Section 2.2, exploring relevant work on device behavior analysis and simulation in the IoT field.

2.1 Firmware updates

Firmware updating for IoT devices has become a crucial part of maintaining IoT systems in recent years [14]. IoT device updating involves modifying or updating firmware, configuration, or other relevant information on IoT devices to maintain system operation, functional integrity, security, and compatibility. However, updating firmware for IoT devices is not an easy task due to various challenges and difficulties, as defined in [14, 20], such as update time, managing a large number of distributed devices, security risks, version control, and dependency management. These challenges affect the user experience and device stability at various levels. In Section 1.2, we mention the challenges faced by the development team of the Hue Bridge: as the deployment cycle for Hue Bridge firmware accumulates with functionality and firmware patches, it becomes longer and significantly affects the timely handling of user feedback or bugs, greatly increasing the time and resource costs of firmware updating. Therefore, the challenge faced by this study belongs to the "Update time" challenge defined in [14], focusing on device update efficiency and update cycle.

In recent years, many related works have gradually focused on and attempted to address these challenges to improve firmware update efficiency and update cycles. These works mainly fall into three directions: introducing the DevOps (a combination of software development and operations) methodology, blockchain technology, and device containerization technology.

2.1.1 DevOps

DevOps methodology is the most important code development model in software development teams in recent years, greatly improving team collaboration

efficiency and efficiently handling exceptions and bugs after software/firmware deployment, and is suitable for the development of most software programs and hardware firmware. Before DevOps, software teams proposed the Waterfall process to address the lack of review and trade-off considerations in the development workflow. The Waterfall process originated from the manufacturing field and is a linear, sequential process where all project development stages, such as software design, development and testing, are completed once within the waterfall model[22]. However, the Waterfall model has a drawback in that the project value is only realized at the end of the process, leading to many projects failing to persevere until the end. As a result, teams gradually transitioned from the waterfall process to agile development. Compared to the rigid and fixed process of the Waterfall model, agile development focuses on responding to changes[1]. It addresses the issues in Waterfall by abandoning complete project planning and dynamically adapting to uncertainty. Agile development has been around for over 20 years, and in recent years, the introduction of DevOps has further improved the agile development process. The two development method are very similar, with the main difference lying in the acquisition of feedback after software deployment. Agile development obtains feedback from clients/users, while DevOps introduces the operational phase after deployment to receive feedback internally[1]. Therefore, DevOps has a faster release cycle and quicker feedback. For development teams, all members participate in the process, where development is operations and operations is development[32].

[21, 18] focus on the low-power distributed mode in IoT devices, where IoT devices typically return sensing data, and firmware functions are mainly to monitor the environment and send data. DevOps methodology is applied to device monitoring and fault detection in [18], where the author proposes the "fast and continuous monitoring feedback of system availability" activity (F&CF availability) and embeds it in infrastructure (IoT sensing devices) in the form of components. DevOps teams can quickly deploy the latest firmware and monitor availability anomalies and faults caused by firmware in real-time. The benefits of using DevOps in IoT systems are summarized in [20], where 11 papers classify the benefits of applying DevOps to IoT systems as "DevOps framework proposals for IoT," which solves the problem of phased pipelines in IoT development. In addition, some literature attributes the benefits to the combination of cloud development technology or containerization technology with DevOps methodology, greatly improving code development efficiency.

2.1.2 Blockchain

Blockchain technology has been widely used in firmware updates of IoT devices in recent years. In [26], the authors explored how to solve the scalability challenges of firmware updates through a blockchain-based firmware update architecture, achieving efficient and secure firmware updates for IoT devices. [34] proposed a blockchain-based method for updating IoT device firmware, focusing on the security and integrity of device updates and avoiding the risks and single points of failure brought by centralized management. To address the vulnerabilities of firmware updates in previous blockchain-based IoT device firmware update solutions, such as system consensus attacks, centralized firmware storage, and large storage space requirements, [34] defined firmware update vulnerabilities and proposed a protection mechanism based on blockchain technology and

a distributed network storage mechanism, verifying the practical feasibility of blockchain technology in improving system security and firmware transmission integrity through comparison with existing work.

2.1.3 Virtualization

Virtualization is a mechanism to share hardware resources(Kernel, CPU, memory) among virtual machine instances on the same machine[23]. Before Containerization was proposed, development teams wanting to virtualize IoT devices typically had to use techniques such as full virtualization, hybrid virtualization, or paravirtualization[23]. These virtualization techniques provided developers with an environment to run applications in virtual machines. However, to emphasize the isolation between virtual machines, these virtualization technologies allocated a separate kernel for each virtual machine, making deployment and management complex and slow. Additionally, deploying a large number of virtual machines would be costly, and hardware resources would not be efficiently utilized. The emergence of containerization technologies addresses these issues.

Containerization can be seen as a form of operating system-level virtualization, where containers share resources (kernel) to provide high availability and portable deployment. Unlike traditional virtual machines, containers do not require a separate operating system instance for each container, resulting in faster startup times, lower resource overhead, and easier management. Containerization provides a lightweight and efficient approach to virtualization, allowing multiple containers to run on a single host machine and utilize hardware resources more effectively[23].

With the rapid development of container technology, more and more IoT devices are beginning to attempt Containerization to improve convenience and deployment efficiency [15]. Compared with the first two technological solutions, Containerization will detach from the IoT device hardware layer and virtualize the device hardware to solve the testing and verification difficulties that existed before firmware release, realizing the efficiency of testing and verifying firmware. Regarding updating IoT devices, researchers have proposed a new solution. Among them, [5, 15] discussed the feasibility and advantages of using containerization technology and Kubernetes container orchestration engine to update IoT devices. They believe that using container technology and container orchestration can easily update software and firmware on devices, while also improving system stability and maintainability. The combination of cloud computing and IoT device virtualization is a common IoT device solution after many cloud computing vendors have lowered service fees [5]. With the rapid progress of container technology, IoT device virtualization is gradually upgrading from management program-based virtualization, such as Virtual Machine Monitor (VMM), to container-based virtualization. Container-based virtualization is more lightweight and efficient, supporting Single Board Computer (SBC) devices and Microcontroller Unit (MCU), as well as Linux Container Virtualization (LVC) layer. LVC is an operating system-level virtualization paradigm that allows the same operating system to share the kernel in the user space and can have many different instances [5]. Container virtualization can significantly improve efficiency when dealing with small-scale devices, but it becomes more challenging when facing a certain scale of IoT devices. In [31], the authors introduced Kubernetes technology and applied it to a publish/subscribe-based

IoT system to explore the impact of Kubernetes on the efficiency and scalability of the system. Kubernetes is a container orchestration management technology that can automate the management and operation of large-scale virtualization containers.

2.1.4 Methodology

In addition to the three techniques mentioned above, there are also some latest technologies applied to address the efficiency issues of firmware upgrades in the IoT field. For instance, Celesti et al. explored the adaptability and resilience of software updates in large-scale IoT devices in [5]. They proposed a swarm intelligence-based update scheme that can achieve adaptive update strategies according to the device status and network load, thus improving the success rate and efficiency of updates.

Now let's take a look back at the issues faced by the Signify development team in firmware updates and explore the feasibility of applying these technologies to the Hue System. Instead of a direct comparative analysis of the strengths and weaknesses of these three technologies in IoT systems, we are more focused on their respective emphases when applied to IoT devices.

Firstly, DevOps focuses on quickly addressing faults and bugs in IoT firmware, aiming to improve the efficiency of problem feedback and code development for the development team. Currently, DevOps is being applied in various teams within the Signify development team. However, while DevOps emphasizes post-deployment operations and rapid feedback compared to the waterfall model, it does not necessarily reduce the time taken for firmware updates. As mentioned in Section 1.2, the development team needs to wait for users to use the Hue System to generate sufficient analysis and diagnostic data. The role of DevOps in this context is mainly in task allocation and operations, and it does not directly accelerate the generation of user data or improve validation efficiency. Therefore, the application of DevOps in this project does not directly address the problem.

Blockchain has the characteristics of distribution and high availability. The feasibility of using Blockchain to enhance system security and firmware transmission integrity in IoT systems has been verified in [34]. Similarly, containerization and Kubernetes also have the advantages of distribution and high availability. The main difference between these two lies in the fact that Blockchain enhances deployment security and distributed deployment efficiency by improving the security and high availability of communication transmission, focusing on firmware security. On the other hand, containerization optimizes the process of firmware deployment cycle by virtualizing IoT devices, focusing on improving the efficiency of firmware updates. Through containerization, development teams can deploy the latest firmware in containerized IoT devices in the cloud or locally before releasing firmware updates. When combined with Kubernetes, automated management and diagnosis of the feedback and user data on the latest firmware in containers can be achieved. Based on our analysis of the project requirements, we have chosen a containerization solution (Docker+Kubernetes) that focuses more on update efficiency rather than deployment security. Therefore, the research question we need to investigate in this thesis is whether containerization can truly be applied to the Hue Bridge to improve firmware deployment efficiency.

2.2 IoT device user behavior analysis

It is mentioned earlier that Signify faces challenges in firmware release efficiency, which can be addressed with Containerization technologies. However, simply virtualizing the devices is not sufficient to accelerate the generation of user data. We also need to simulate user behavior. In Section 1.2, we introduced the firmware release cycle consisting of several phases, including planning, development, testing, deployment, and maintenance. During the testing phase, developers usually design automated tests for various components to test new features. The purpose of these tests is to identify firmware issues before releasing the latest firmware to customers and address them based on the diagnostic data obtained from the tests. The basic logic behind these automated tests is to simulate a large volume of machine-readable instructions representing user actions with IoT devices, enabling the identification of issues prior to firmware deployment. Therefore, a complete solution requires a combination of analyzing/simulating user behavior with respect to the Hue Bridge and containerizing the Bridge devices.

With the geometric growth of IoT device data and the popularity of AI algorithms, analysis of IoT device user behavior has gradually become a focus of device manufacturers in recent years. In our search for relevant work, we found that many works use data mining or machine learning techniques to analyze perception-type data generated by sensor-type IoT devices and establish data models for them [17, 33]. However, there are few published results for user interaction-type IoT devices, especially highly integrated gateway-type IoT devices like the Hue Bridge, which are usually developed by IoT companies with tens of millions of users (such as Cisco, TP-Link, Xiaomi, etc.). Analysis of user behavior for these devices is often not shared. Therefore, we need to start from some existing work that is relatively close and find a suitable solution for the Hue Bridge.

In order to provide consumers with a better control experience, many commercial IoT devices are accompanied by mobile applications (Apps). In [29], the authors invented a tool called "IoT App Privacy Inspector" to identify and distinguish user interaction-type data packets in communication data between the App and the device, and automatically infer the corresponding user behavior of the data packet from network data generated by user interaction behaviors (such as login, device on/off, input of identity information, verification code, etc.). Although the ultimate goal of this work is to improve the privacy of IoT device users, it does not analyze specific control-type user behaviors, but it does provide some ideas that can be explored. Compared to traditional algorithms that analyze and predict perception-type data generated by users, this work uses state data of IoT devices such as network traffic to infer user behavior in reverse. For the Hue Smart Lighting System, the state data collected is the status data returned by the lights when the user controls home lighting, which help us infer the user's behavior habits when using the Hue System.

In terms of algorithm selection, we chose frequent pattern mining algorithm, where a frequent pattern is a pattern which occurs in comparatively more transactions [10]. In Chapter 4, we explained in detail the reasons for choosing frequent pattern mining instead of machine learning or deep learning algorithms. Frequent pattern mining is a popular data mining technique [13, 6], commonly used in many real-life applications such as market basket analysis, clustering,

series analysis, decision making, object mining, etc [10]. After the authors of [2] proposed the Apriori and AprioriHybrid algorithms in 1994, more and more researchers began to pay attention to the field of frequent pattern mining and proposed many more efficient algorithms, among which the most common three algorithms are Apriori, Eclat, and FP-Growth [13]. Apriori algorithm is the most classic and general frequent pattern mining algorithm, but it has scalability issues and exhausts computer memory much faster than other algorithms [13]. Eclat and FP-Growth have similar performance on itemset and transaction size indicators, but Eclat is based on a depth-first search algorithm, while FP-Growth is a tree-based algorithm designed by Han et al. in 2000 [12]. The performance and output results of frequent pattern mining algorithms are closely related to the type of dataset and data preprocessing. In Chapter 4, we will experimentally demonstrate the feasibility of applying frequent pattern mining algorithms to Hue Bridge user behavior analysis by applying common algorithms to Hue Bridge user state data.

In this chapter, we conducted a literature review on two research directions in the field of IoT: firmware updates and user behavior. We analyzed the problems encountered by Signify Firmware update and their corresponding solutions. A more detailed examination of the Hue System and the firmware update cycle will be presented in Chapter 3.

Chapter 3

Hue Bridge Device Virtualization

The focus of this work is the efficiency issue in the Hue Bridge firmware update. Therefore, let us narrow down our scope to the Bridge device and its Firmware Update cycle. In Chapter 2, we analyzed the solutions proposed by scholars in the IoT field for improving the efficiency and security of the firmware cycle. We concluded that virtualization technology of IoT devices is theoretically most suitable for the Hue Bridge. However, to return from theory to reality, we need to answer a question: if we design a virtualized Hue Bridge, how can it improve efficiency in firmware release? To answer this question, this chapter first presents the existing methodology and firmware update cycle of the Hue System to help readers further understand the Hue Bridge. Secondly, we introduce the process of virtualizing the Hue Bridge and conduct functional verification of the virtual bridge.

3.1 Methodology

In Section 1.1, we introduced the composition of the Hue System and emphasized the central role of the Hue Bridge device in the Hue System. In Section 1.2, we introduced the motivation of this project, which mentioned that "the current deployment cycles for bridge firmware take around 6-8 weeks, which is too long to receive useful data/feedback from the field regarding reliability and availability." At this point, readers may have a general framework for the concepts of the Hue Bridge and its Firmware Update Cycle, but may still have some incomplete understanding of the details of these concepts. Therefore, in this section, we will help readers understand and analyze the software and hardware composition of the Hue Bridge, and how the Firmware Update Cycle of the Hue Bridge works as an IoT device. We will also explain why the development team needs to spend so much time updating the firmware. Finally, based on Signify's previous virtualization projects, we will seek inspiration and propose our containerization solution.

3.1.1 Hue Bridge

Hue System is a Home lighting system[28]. Why does it need a device like the Hue Bridge (Shown in Figure 3.1)? Compared to traditional lighting systems, Hue System has rich home automation functions (such as setting schedules for turning on/off lights, setting rules for automatically turning on lights when motion sensors detect users, etc.). In addition, compared to traditional wired communication protocols, Hue System supports Zigbee/BLE wireless communication protocols. The Hue Bridge is the fundamental driving force that empowers these intelligent experiences of Hue System[28].

Figure 3.1: **The Hue Bridge**

The main functions of the Hue Bridge can be summarized as follows:

- Offering features such as automation, scene control, schedules, dynamic light effects
- Providing whole-home connectivity, enabling user to connect their lighting-related devices and processing user control messages.
- Serve an easy way to use internet API for every connected device proxied behind the bridge.
- Devices in the smart home are always connected, user can control their devices even if they are not in the home
- Guarantee user data security and privacy, providing local Zigbee network services

To virtualize IoT devices, we first need to understand how the internal software and hardware of the device are constructed. Let's analyze the composition of the Hue Bridge from a developer's perspective. Table 3.1 lists some technical parameters of the Hue Bridge device. The Hue Bridge is based on OpenWrt Operating system, which is an open-source Linux-based operating system commonly used for IoT embedded device development. The Hue Bridge processor selected is the NXP Semiconductors i.MX 6SoloLite processor, which integrates Zigbee/Ethernet/Wi-Fi/BLE/MQTT communication interfaces, providing the

Table 3.1: **The Hue Bridge: Device Information**

Device name	Hue Bridge
Operating System	OpenWrt
Protocols	Zigbee/Ethernet/Wi-Fi/BLE/MQTT
Functional process	Daemon per function

Hue Bridge with rich network functionality. Based on OpenWrt and the interfaces provided by NXP, the developers of the Hue Bridge have implemented rich functional development such as automation, scene control, schedules, and dynamic light effects.

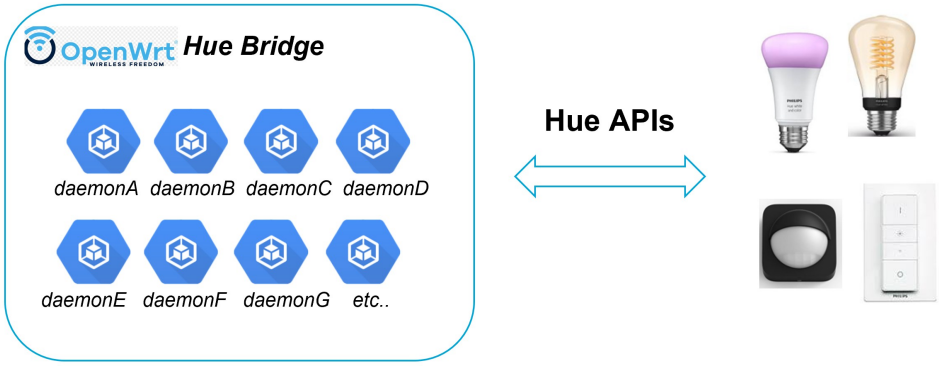


Figure 3.2: **The Hue Bridge: Software Architecture**

Figure 3.2 shows the top-level architecture of the Hue Bridge software. Within the bridge, each Bridge function runs continuously 24/7 in the form of a daemon (a background program). For example, the Fluent-bit (open source) daemon allows for real-time forwarding and collection of diagnostic messages or logs from Hue Bridge devices, processes them, and delivers them to different backends[9]. These daemons serve as the core software components of the Hue Bridge. Hue Bridge provides a public interface called Hue-API, which enables developers to use the functionality of the bridge and interact with Hue System resources by calling the Hue-API.

3.1.2 Firmware Update Cycle

In this section, we will introduce another core concept: the Firmware Update Cycle in Hue System. In the development process of the Hue Bridge, the Firmware Update follows the standardized steps of planning, development, testing, deployment, and maintenance, in which the planning and maintenance stages require flexible time planning according to actual situations (there is a lot of uncertainty). Therefore, this project will focus on the fixed period of development-testing-deployment Release Cycle. In the DevOps development of the Hue Bridge, each formal firmware update corresponds to a PI Planning (Program Increment Planning, with a total duration of eight weeks)[25].

Each Program Increment (PI) is composed of four firmware Release Can-

didates (RC), which have slight differences but are all part of a larger version update. Typically, Signify selects two of these RCs for deployment in beta testing. In Figure 3.3, we demonstrate the timeline for the first RC selected for beta deployment in a PI, which begins on the first day of Sprint 1 and ends on the first Wednesday of Sprint 3, totaling 23 working days.

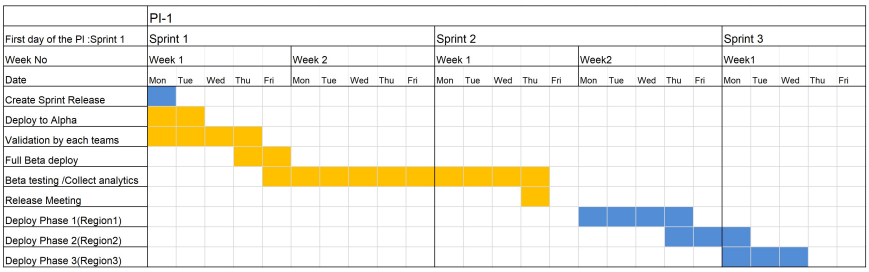


Figure 3.3: **The Hue Bridge: Firmware Update Cycle (The focused stages are highlighted)**

After clarifying the timeline concept, we will divide the steps development-testing-deployment into nine specific stages:

- **Create Sprint Release:** First, developers will package the firmware that needs to be updated, create firmware with a specific version number, and create the first Sprint Release corresponding to that version number.
- **Deploy to Alpha:** The alpha test is a deployment test within Signify, where around 50 dev-bridges will be deployed with the firmware.
- **Validation by each team:** Each development team will conduct the first round of validation on the daemon they are responsible for based on the results of the Alpha test.
- **Full Beta deploy:** The beta test is a deployment test within the end-users (users who are willing to participate in the beta test), where around 8k bridges will be deployed with the firmware.
- **Beta testing/Collect analytics:** Each development team will collect and analyze user diagnostic data generated during the beta test period.
- **Release Meeting:** Validation engineers and team leaders will summarize and troubleshoot the beta test analysis, generate a beta report, and discuss whether the firmware can pass the test at the release meeting.
- **Deploy Phase 1 (Region1):** The firmware approved by the meeting will be deployed in stages to all bridge devices in Region1.
- **Deploy Phase 2 (Region2):** The firmware approved by the meeting will be deployed in stages to all bridge devices in Region2.
- **Deploy Phase 3 (Region3):** The firmware approved by the meeting will be deployed in stages to all bridge devices in Region3.

After communicating with the responsible person of the Update cycle and validation engineers, we clarified the actual situation of the nine steps. In terms of timeline, the days of Create Sprint Release and Release Meeting are fixed and cannot be further optimized, as the team needs to develop a plan and discuss it in the meeting. After firmware review and validation at the Release Meeting, the three stages of deploying firmware to the field (a total of nine working days) are being planned to be merged to shorten the time, which is expected to reduce by 1-2 days. None of these steps can be participated in by this thesis, and the three stages that can be optimized are Deploy to Alpha, Full Beta deploy, Beta testing/Collect analytics. The deployment and testing process for Alpha/Beta are: Engineers deploy firmware Release Candidates (RC) to Alpha/Beta testers' bridges, and these Bridges collect analysis and diagnostic data after RC deployment and forward it to the Hue cloud database. Although this data is sent to the cloud in real-time, Validation Engineers need to wait for a period of time (Alpha 4 days, Beta 10 days) to collect enough analysis and diagnostic data, and generate the Validation Report for each team based on the analysis results before the Release Meeting.

Based on the analysis of the Release Cycle above, we can focus the research scope of this project from the complete Firmware Update Cycle to the Deploy and Test stages of Alpha/Beta, which is highlighted in figure 3.3. At the same time, based on the analysis of the pain points of the Firmware Update Cycle in Section 1.2 and the literature review on improving Firmware Update Cycle efficiency in Chapter 2, we propose the first and most important research question of this project:

To shortening the firmware Update Cycle, can we partially eliminate the role of Alpha/Beta testers in the firmware update cycle by gathering diagnostic data and feedback from virtual bridge?

Virtual bridges can continuously execute instructions through automated programs without waiting for users to generate enough diagnostic data by using the Hue System. We will explore and validate the feasibility of virtualization technology in this project in subsequent chapters.

3.1.3 Virtualization Solution Design

In the early stages of the project, we conducted a comprehensive literature review in the IoT field to explore related work. At the same time, we also did a lot of reading and code learning from Signify's previous projects. After identifying virtualization as the direction to pursue, we immediately began searching for previous attempts by engineers to virtualize the Bridge. After reviewing these projects, we finally selected one that best matched our needs, which we call the Hue System Emulator (HSE). As its name suggests, HSE is not only designed to simulate the Hue Bridge device but also to simulate the complete Hue System, including Hue Bridge, Zigbee devices (lights, sensors, accessories), etc. The dashboard of HSE is shown in the Figure 3.4, which demonstrates that HSE can simulate various types of Hue devices and support developers to control these devices. HSE is an emulator project based on the Ubuntu system and C# (a Programming Language), which can be used for both virtual machines and embedded devices. Many teams and individual developers have already been using HSE to test the performance of new devices and the stability of new features. Additionally, while simulating the Hue Bridge, HSE embeds

Figure 3.4: **Hue System Emulator: Device Dashboard**

some of the Hue Bridge daemons, which can simulate many of the commonly used functions of the Hue Bridge, despite being implemented in different syntax and based on a different platform.

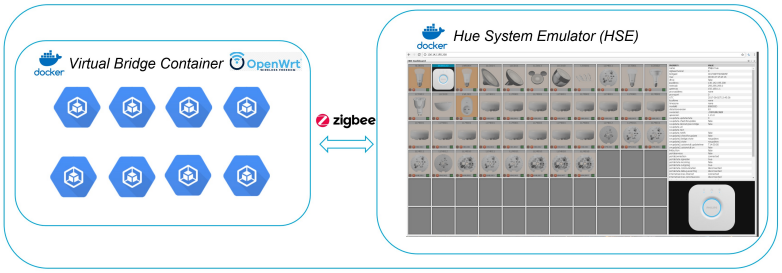
We aim to eliminate the role of beta testers in the firmware update cycle through the use of virtual bridge devices. To achieve this, the virtual bridge needs to meet the basic requirement of being functionally similar to a real device. After a period of evaluating and testing with HSE, we found that although it provides the core drivers of the Hue Bridge (such as various daemons) and a similar experience to the common features of the bridge (such as light control, scene switching, and schedule adding), there are several issues:

1. **System Compatibility:** HSE is based on the Ubuntu system, and many dependencies and programs are based on the Ubuntu underlying logic. However, the Hue Bridge is based on Openwrt, which leads to significant deviations in the underlying code for the entire functionality.

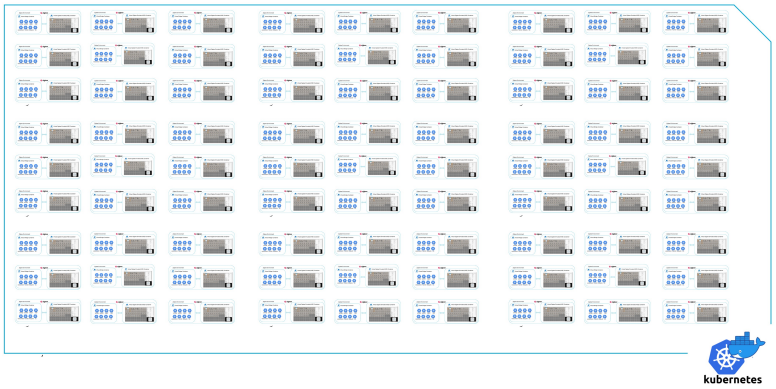
2. **Dameons insufficient:** HSE includes some daemons, but many of the daemons responsible for communicating with the cloud are not integrated, which prevents us from directly obtaining diagnostic data from the cloud.

3. **Portable Issue:** The overall size of the HSE project is not portable, making it difficult to deploy and manage a large scale of virtual devices.

After sorting out the issues mentioned above, we immediately thought of the solution proposed by [15, 31] in Section 2.1, which uses Docker and Kubernetes technology to improve the efficiency of firmware updates. Although these works are completely different from our actual situation, their advantages can to some extent avoid the shortcomings of the HSE. Docker, as the latest virtualization technology, has the greatest advantage of being portable and system compatible (users can specify the system at will). Kubernetes (automating deployment, scaling of containerized applications[36]), is most suitable for the deployment and management of large-scale virtual devices. The containerized (Docker) bridge device will contain all the Daemons embedded in the current



(a) **Architecture: Containerized Hue System**



(b) **Architecture: Kubernetes Scaling**

Figure 3.5: **Virtualization Solution based on Docker and Kubernetes.**

firmware, while Docker provides rich network services and container management methods.

Therefore, we propose a virtualization solution based on Kubernetes, while retaining the advantages of Dockerize Hue Bridge and HSE simulating Zigbee devices, to build a complete virtual Hue System. The architecture of the solution is shown in Figure 3.5, where in Figure 3.5(a), the containerized HSE is responsible for providing virtual Zigbee devices (lights, sensors), and the virtual bridge container is responsible for stable operation of various daemons. The connection between the two containers will be established by the daemon responsible for Zigbee communication, and completing device authorization and pairing (simulating the user's search for surrounding devices). In Figure 3.5(b), we use Kubernetes technology to deploy the containers on a large scale, and ensure the uniqueness of each container by allocating different user configurations during the deployment process. Besides, Docker can pull and load OpenWrt OS (Open Source Image) as the base image, allowing us to quickly adapt to changes even if the Hue Bridge changes its OS version or is ported to other OS.

3.2 Advancements in the Dockerize Hue Bridge

In this section, we will introduce the specific steps to improve the Dockerize Hue Bridge. In the field of containerization, there are two core keywords, one is Container and the other is Image. A container is a standard unit of software

that packages up code and all its dependencies [7]. A docker image is a executable package of software that includes dependencies files needed: code, uboot environment, packages, tools, libraries and configs[7]. In short, the image is the package of the application, and the container is the object after the image is instantiated. Therefore, we need to create an image for the software part of Hue Bridge and run the container based on the image. We create the image through DockerFile (Document that used to contains commands to assemble an image[8]), each commands in the Dockerfile corresponds to a mirror layer, and the mirror layer relies on a series of underlying technologies, such as filesystems, union mounts, copy-on-write, etc[8].

Our system has made the following improvements and advancements to Hue Bridge virtualization, which are divided into five aspects:

1. **Establishing Hue cloud connectivity**

   Previous virtualized Hue Systems simulated the software functionality of the Hue Bridge but lacked communication with the Hue Cloud. During the Beta Test, diagnostic data generated during user product usage will be sent to the Hue Cloud in real-time as messages. These diagnostic data are crucial user data during the validation stage, enabling engineers to identify and promptly address firmware issues. Therefore, we establish communication between the virtual bridge and the Hue Cloud using the MQTT protocol, allowing diagnostic data generated after firmware deployment on the virtual bridge to be sent to the Hue Cloud via the MQTT protocol.

2. **Create a Dockerfile and rebuild the dockerize bridge image**

   Based on the previous work, we create a new Dockerfile and specify Scratch (empty image) as the base image with the FROM keyword. Scratch is a commonly used base image[7]. Each command in the Dockerfile corresponds to a mirror layer. Therefore, scratch will serve as the base image layer, and the subsequent image layers will be accumulated based on this layer. First, we add the Openwrt image produced in step 1. Secondly, we add the remaining image layers required by the Hue Bridge in the Dockfile, including the environmental variable configuration component of the bridge device, the initialization configuration component of the bridge device when it starts, the uboot of the bridge device, the software dependencies and libraries of various bridge components, some binary files required for initialization, and the preinit execution script, etc. When everything is ready, use the *docker build* $-t$ command to build the image and generate the image. It should be noted that each image needs to add a tag during creation, and the tag corresponds to the version number of the current Hue Bridge packaged firmware.

3. **Setup a Private Docker Registry and push the image**

   We need to upload the image so that any authorized device can use the image. The place where the image is stored is called the Docker Registry, which is divided into Private Registry and Public Registry. In order to ensure the security and privacy of the project, we use a Private Registry and set SSL Certificate and Login Credentials. The benefit of this setting is that only designated users with sufficient privileges can obtain our virtual bridge image.

4. **Migrate bridge configuration file to the virtual bridge container**

   After completing the third step, we can pull the image from the registry and create a container instance. Before running the virtual bridge, we need to perform an additional initialization step. In the Hue Bridge, besides the initial configuration file (containing device ID, device type, initial token, etc.), we refer to the user-specific configuration file as $Full\_Config$, which includes the device configurations (lights/sensors) added by the user during the use of the Hue Bridge, lighting area configuration, automation rules/schedules configuration, network configuration, third-party device connection configuration, etc. If the bridge device has a complete $Full\_Config$, it means that the device has been configured and officially used by the user, rather than being in a factory-formatted state as a brand new bridge device.

   In Section 3.1, we propose to use containerization of the bridge to simulate the solution of beta testers' bridge devices during the beta test. Therefore, the containerized virtual bridge containers need to migrate beta testers' $Full\_Config$ to restore the beta testers' test environment and lay the foundation for the subsequent project verification and large-scale deployment.

5. **Validate daemons running in the virtual bridge container**

   Next, we formally start the virtual bridge container. When starting, it is important to allocate and expose specific published ports (such as HTTP 80:80) for the container. Through port mapping, ports inside the container can access services outside Docker (such as connecting to cloud services). By using the *docker run* and $docker\ exec - it$ commands, we start the container and enter the internal Linux (OpenWrt) environment of the container. In this environment, we can execute Linux commands to complete the verification work. First, we need to verify that Docker has assigned a Docker network to the container and established the correct port mapping.

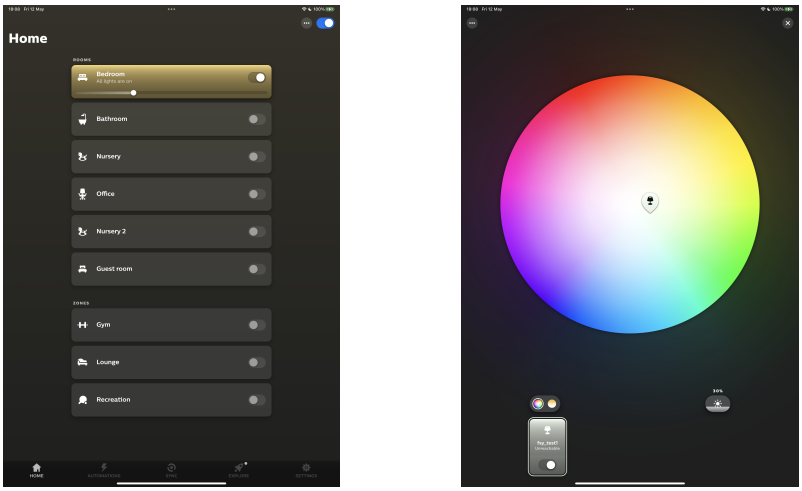
   Secondly, we need to sequentially verify that the daemons responsible for each team in the Hue Bridge are configured correctly and running stably. In the user's Hue Bridge, daemons may reset, which can lead to complex reset situations (due to improper user behavior, firmware updates, etc.). Therefore, after starting the container, we need to ensure that our containerized bridge does not have any daemon that will automatically reset or crash. Considering data security, we anonymously demonstrate the functions of the four daemons and the corresponding verification methods of the daemons in Table 3.2 (annonmymized, for reference only). According to our testing and validation, all Daemons, except for the one responsible for firmware updates, are functioning properly after our adjustments and fixes. As each different image corresponds to a different firmware version, we do not need the Daemon responsible for updates to update the firmware.

Table 3.2: **Containerization Bridge: Daemon Validation Example (anonymized)**

Daemon Name	Function	Validating Bash Commands	Supposed Response
Daemon A	Provide Mosquitto MQTT protocol	log -e mosquitto	the mosquitto configuration was successfully written
Daemon B	Create link between Cloud server and Bridge	[[-f <urls.json>]] && echo "Urls file exists!"	Urls file exists!
Daemon C	Manage Zigbee network	cat /root/hse/emulator.log	Emulator connection established
Daemon D	Process and forward logging and metrics	log -e logging	logging has been forwarded

6. **Validate virtual bridge's functions via Hue App**

   Finally, we place the virtual bridge container in a public network and use a mobile device in the same network environment to open the Hue App and search for the IP address corresponding to the virtual bridge using the "Find device using IP address" function. As shown in Figure 3.6, the Hue App successfully connects to the virtual bridge, and we can add real Zigbee devices to the Hue System. As shown in the figure, the added light devices can be controlled for brightness, color temperature, scene, etc.



(a) **Virtual Bridge: Group Control** (b) **Virtual Bridge: Lights Control**

Figure 3.6: **Virtual Bridge Functional Validation via Hue App**

Following the containerization procedure, we have successfully containerized the Hue Bridge device and verified the Daemons. In the following chapters, we will explore how to use the containerized Hue Bridge to complete the tasks of test bridges in the Alpha/Beta stages as much as possible.

Chapter 4

Use Case Model

In the bridge virtualization process, we want to simulate the bridge functionality as realistically as possible and to simulate the behaviour patterns and habits of Hue users to the maximum extent possible. In the previous chapters, we have made much progress in simulating the functionality of the Hue Bridge, but the simulation of user behaviour is another challenging problem. In Section 2.2, we indicated that the main reason for the slow update cycle of the Hue Bridge firmware is that the diagnostic data generated by user behavior needs to wait for a certain period of time. The advantage of virtualized containers is that we can run automated tests or perform simulated user behavior without interruption and with high frequency.Therefore, high-frequency simulation of bridge user behavior may be able to solve the efficiency problem in firmware update.

The analysis and simulation of user behaviour of IoT devices are very important for designing and developing new features of IoT devices. By studying the user behaviour of IoT devices, we can gain insight into user habits and behaviour patterns. It helps to improve the user experience and satisfaction with IoT devices, thus enhancing the market competitiveness of products.

This section will focus on the analysis and simulation methods of Hue Bridge user behaviour and propose Use Case Model to build a complete virtual bridge system based on containerization technology.

4.1 Hue Bridge user behaviours

This project aims to accelerate the generation and collection of analytics and diagnostic data by containerizing the bridge device to complement the role of beta testers in the Release Cycle. The primary prerequisite for the bridge devices to generate this data is that the beta testers interact sufficiently with the bridge devices during the beta phase.

We can divide the interactions between IoT devices and users into perceptual and control behaviours. Perceptual behaviour is when the IoT device collects data from the external environment and transmits it to the user for observation and analysis, for example, user monitor the air quality and temperature in home. Control behaviours are those where the user controls the external environment through the IoT device, for example. user use switch to turn on the light.

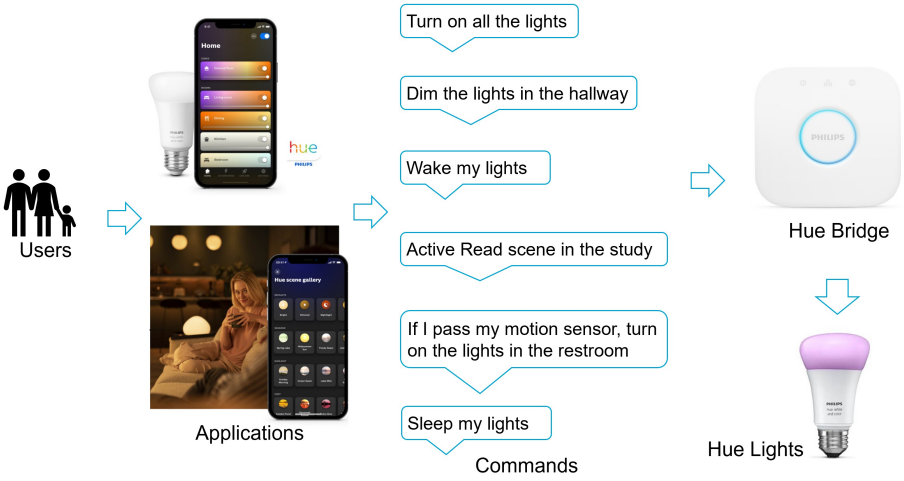


Figure 4.1: **Common User behaviours in Hue System**

In the Hue System, user behaviour is mainly control behaviour (light control), supplemented by perception behaviour. Control behaviour refers to using the Hue Bridge to control the home luminaire, such as switching on and off the lights, adjusting the brightness and color temperature, setting the lighting scenes, etc. Sensing behaviour refers to the ability of Hue Bridge to sense. It provides feedback on the home environment by interconnecting with other sensors and smart devices, such as adding a rule engine to the bedroom by linking with the motion sensor to automatically turn on a pre-set lighting scheme when entering the bedroom.

Users can send control commands to the Hue Bridge via the Hue App or Hue Accessories (Smart Button, Dimmer Switch). The Hue Bridge is a hub that will mobilize the back-end programs to process these commands and ultimately achieve the desired lighting effect.

4.2 Method

After sorting out the behavioural categories and behavioural logic of Hue Bridge users, we need to find a suitable method for modelling user behaviour for Hue Bridge that meets the following requirements:

- A model that provides credible support for subsequent simulations of user behaviour, maximising the commands sequence of users using the bridge device.
- Intuitively reflect the user's interaction with the Signify smart device
- The ability to link to bridge's full_config to provide device and scenario information about user behaviour

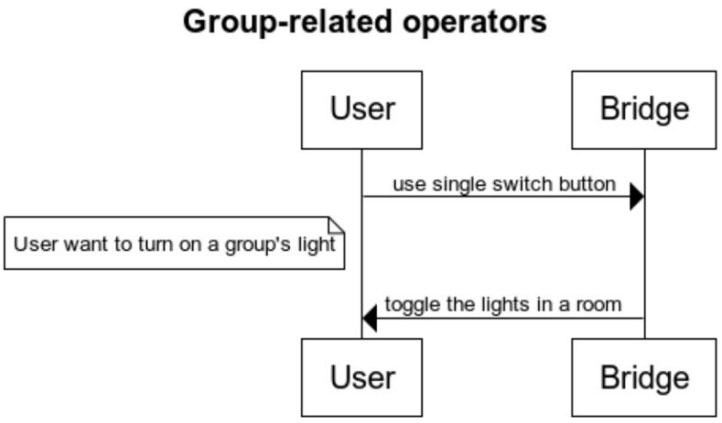


Figure 4.2: **Hue System: Use Case Example via XML**

4.2.1 Use Case

This project synthesises a large body of academic work on user behaviour modelling for IoT devices, combining the characteristics of the bridge device line as a primary control class with a secondary perception class of behaviour, and chooses to use the Use Case approach to investigate the user behaviour of the Hue Bridge. The use case approach is widely used to analyse system requirements and user interactions in software development projects. In this study, we use the use case approach to identify the critical scenarios and tasks involved in using the Hue Bridge product by users.

First, look at user use cases in Signify's intelligent world. As a typical example, when the user comes home, and it is dark, the user finds his Hue dimmer switch, presses the control button on the switch, and the Hue Birdge receives the command and turns on the lights in the whole living room via Zigbee. This use case is widespread and fundamental, and the interaction's logic is drawn in UML in Figure 4.2. The Hue system is very rich in functionality, and the use case is the best way to simulate the natural behaviour of a user for our virtual bridge.

We designed four steps for analyzing use cases by combining the product characteristics of Hue Bridge and the bridge functional design document:

Identifying roles: The Use case analysis begins by identifying the interaction roles, i.e. the users and other devices or objects that interact with the Hue Bridge product. We identified three personas: the end user, the Hue Bridge device and the Hue application.

Identifying scenarios and defining use cases: We generalised the interaction scenarios for Hue Bridge users. This definition of scenarios also divides the use cases of Hue Bridge into categories based on scenarios. We identified five use case scenarios: lighting control, scene creation, rules engine and lighting ambience. (Lights, Groups, Rules, Schedules, Scenes). We will also define use cases in each scenario based on the results of the user behaviour analysis.

Describe the use case: For each use case in the previous steps, describe the purpose and actions that the user performs when interacting with Hue Bridge. For example, in the lighting control use case, the user would turn the lights on

or off, adjust the brightness or change the colour.

Analyse user behaviour: Analyse past user behaviour/data to find patterns and preferences in past data about how users use the Hue bridge. This step will validate and refine the user use cases and uncover user behaviour that humans cannot analyse, summarising the user's habits of using the Hue Bridge product.

4.2.2 Frequent Pattern Mining

The first three steps in the Use Case method are manual analysis and summary, while the fourth step, analyzing user behavior, requires the use of past data to identify patterns and validate and refine the results of the use case analysis. In this section, we describe the method used to analyze and summarize past user behavior data for Hue Bridge.

The user data collected by Signify does not directly include user control commands or HTTP request records. The bridge user data best suited for analyzing user behavior is the State_message data, which collects device status change values generated by users while using the Hue Lighting System, such as the on/off status of lights, light brightness and color temperature, and motion sensor presence status. We chose State_message as our primary dataset for analyzing user behavior because the data recorded in State_message are the most common state information for users when customizing their smart home systems using the app or accessories. By tracking changes in state information, we can intuitively determine users' specific behavior.

Table 4.1: **State_message Example: 10 Messages in Lights sub-tables**

Source	**hue**	**sat**	**xy**	**ct**	**colormode**	**bri**	**on**
zigbee	59086	232	[0.4925, 0.2309]	428	xy		
zigbee	14956	140	[0.4571, 0.4097]	366	ct		
system							FALSE
homekit						175	TRUE
rules	8418	140	[0.1467, 0.2153]	366	xy		
zigbee						99	
clip							FALSE
zigbee				366	ct	254	TRUE
clip	62622	200	[0.5584, 0.2904]	500	hs		

After identifying the dataset, we needed to find suitable algorithms to analyze the State_message dataset. In the IoT field, the main research directions involve using pattern learning or machine learning models to classify and predict perceptual behavior data generated by users[17, 13, 29]. Both pattern learning and machine learning seem suitable for our project, but each has its advantages and disadvantages.We compared pattern mining and machine learning as two different approaches for modeling use cases from incremental messages. Pattern mining is a rule-based approach that relies on extracting patterns from incremental messages and using them to create rules for identifying use cases. This method has a simple algorithmic logic and does not require a large amount of labeled data. However, it may not be able to handle complex patterns and may

require manual translation to refine rules. On the other hand, machine learning is a data-driven approach that relies on training models on labeled data to identify use cases. This method can handle complex patterns and can learn from a large amount of data. However, it requires a large amount of labeled data for training, and the results often have situations that cannot be explained or described by humans.

The State_message dataset mainly consists of status values for lights and sensors (such as motion sensors), an example of State_message is shown in Table 4.1. If we use machine learning to study user behavior, we would be more concerned with predicting users' preferences and habits regarding these status values (for example, predicting the brightness and color temperature values set by users). However, for Hue Bridge, which focuses on control-based user behavior, we are more interested in analyzing the behavior of users changing status information itself, rather than the distribution and prediction of these status values. Therefore, we chose to use pattern mining (also known as frequent pattern mining) as our method for analyzing user behavior. In frequent pattern mining, frequent patterns typically refer to itemsets/subsets that appear frequently in the dataset and have a frequency higher than a set threshold. By analyzing and mining these frequent patterns or frequent itemsets, we can identify the association rules and frequency of all items in the dataset.

Frequent pattern mining is a technique used to find frequently occurring patterns in a dataset. These patterns can help us understand the relationships between State_message data and identify meaningful features. The main objective of frequent pattern mining is to discover itemsets that occur frequently, which can be used to build association rules and help us understand the correlation between data. In this process, we need to calculate metrics of frequent itemsets to determine their statistical significance. The metrics for the various designs are evaluated using the following:

1. **Itemset:**
   Set of items X. A State_message D contains an itemset X if $X \subseteq D$. Each itemset X is associated with a set of State_messages $D_X = \{D \in M \mid D \supseteq X\}$ which is the set of State_messages which contain the itemset X, M is a set of n State_messages $M = \{D_1, D_2, \ldots, D_n\}$ [35]

2. **Association Rule:**
   Frequent itemsets that appear together frequently

3. **Consequent & Antecedent:** An antecedent is an itemset that first found in the data and Consequent is an itemset that can be found after we found the antecedent.

4. **Support:**
   supp(X) of an itemset X is the ratio of transactions in which an itemset appears to the total number of State_messages[35]

$$\text{support}(X) = \frac{|\{D \in M; X \subseteq D\}|}{|M|} \tag{4.1}$$

5. **Confidence:**
   The confidence of a rule $X \to Y$ is the probability of seeing the consequent

in a State_message given that it also contains the antecedent[24].

$$\text{confidence } (X \to Y) = \frac{\text{support}(X \to Y)}{\text{support}(X)} \tag{4.2}$$

6. **Lift:**
   The lift is a measure of importance of a specific association rule[4].

$$\text{lift } (X \to Y) = \frac{\text{confidence}(X \to Y)}{\text{confidence}(X)} \tag{4.3}$$

4.3 Use Case Modeling Process

To validate the feasibility of applying the Frequent Pattern Mining algorithm mentioned in the Method section to Delta Message, we conducted several progressive comparative experiments to find the optimal model for the Hue Bridge use case. In the experimental section, we first introduced the dataset and preprocessing steps. After preparing the data, experiments were designed based on different combinations of sub-tables in the dataset, and algorithm improvements were proposed to adapt to the user behavior of Hue Bridge as issues were encountered during the experiments.The general steps of the experiments are designed as follows:

Data preprocessing: We will first preprocess the Delta messages, including deduplication, filtering, and conversion, etc. This will help improve data quality and usability and provide support for subsequent analysis and mining.

Pattern mining: We will use the Frequent Pattern Mining technique to mine frequent patterns in Delta messages. Specifically, we will use classic frequent pattern mining algorithms such as the Apriori algorithm to discover user behavior patterns and usage scenarios. This will help us better understand user needs and habits and provide a basis for device optimization and intelligence.

Pattern analysis and validation: We will analyze and validate the mined frequent patterns to ensure that these patterns are true and trustworthy. Specifically, we will use statistical analysis and visualization tools to discover and explain these patterns and verify their degree of conformity with actual usage scenarios.

4.3.1 State_message Dataset and its pre-processing

In Section 4.2.2, we chosen the State_message as the dataset. In this section, we will provide a detailed description of this dataset and perform data preprocessing before model building. State_message collects device state change values generated by users while using the Hue Lighting System, such as the on/off state of lights, the brightness and color temperature of lights, and the presence status of motion sensors. The changes in these state values fully reflect the user's interaction behavior with the Hue Bridge when using the Hue system. By mining the association rules and frequent items of the state values in the dataset, we can verify and improve the Use Case.

Signify used the Cloud SQL tool to collect and store State_message, which means that we can use general SQL syntax to query large-scale data for better analysis and comparison. Considering the size of the dataset, in this section

on the application of frequent pattern mining algorithms, we queried a week's worth of State_message data from Bridge production users as the experimental dataset. In the Implementation section, we will query datasets of one year or more according to actual needs.

In the complete dataset, there are a total of seven sub-tables strongly related to user behavior. These sub-tables record the numerical values and timestamps of devices' state changes in the Hue Bridge system. These sub-tables are:

1. Lights: All the info about the bulbs, eg type of light, current state(on/off, colormode, hue), the brightness level
2. Group: Group can be a zone info, like the id of lights in that zone/ group if all/any lights in the zone are turned on or off
3. Config : Has all the info about the bridge like ip address, model id etc
4. Rules: Integrate sensors and lights, user-defined rule engine, if the condition is triggered, the action specified by the action will be executed, and State_message records the trigger record of user rules over time
5. Schedules: Collection of input to test web app.
6. Scene: has info of what scene/ light recipe the lights are set to
7. Sensor: if bridge is connected to any sensor like motion detection, outdoor sensor or etc, info collected from these sensors is saved in this table.

In Section 4.2, we discussed using pattern learning methods to find patterns in the Hue Bridge device State_message, in order to analyze the Use Cases of bridge users. In order to preprocess the data in practice, we followed the following steps:

Data reading and cleaning: We sequentially read the raw data of each sub-table, and then clean the data, including removing irrelevant columns, removing rows with null values, removing duplicate rows caused by MQTT data transmission problems, etc.

Data transformation: When processing the data, there are some columns of string type(such as the Source columns in the Lights sub-table) that we want to include all their possible values as itemsets for pattern mining. To accomplish it, we transform each possible string value in the target column into a binary boolean value.

Create encoded transaction: In order to standardize the data set format, common frequent pattern mining algorithms use the TransactionEncoder method to encode and format the original data set, in order to obtain a standardized array format suitable for the algorithm. For our data set, on the basis of TransactionEncoder, we use all the column names of each sub-table as frequent items. The status data of each row is converted based on whether there is a value (for all numerical types). Considering the memory efficiency of processing large-scale data, we convert numerical values into binary boolean values based on whether they have values. This way, we transform the input data set into a one-hot encoded boolean array that is standardized for model training.

Data concatenation: In order to mine more detailed and accurate Use Cases, we can arrange and combine the data in the seven sub-tables, and combine them with Hue Bridge's Full_config data to obtain more detailed frequent item information.

Table 4.2: **Data transformation and Encoded Transaction example(T is True, F is False)**

hue	sat	xy	ct	cm	bri	on	zigbee	system	homekit	rules	clip
T	T	T	T	T	F	F	T	F	F	F	F
T	T	T	T	T	F	F	T	F	F	F	F
F	F	F	F	F	F	T	F	T	F	F	F
F	F	F	F	F	T	T	T	F	T	F	F
T	T	T	T	T	F	F	F	F	F	T	F
F	F	F	F	F	T	F	T	F	F	F	F
F	F	F	F	F	F	T	F	F	F	F	T
F	F	F	T	T	T	T	T	F	F	F	F
T	T	T	T	T	F	F	F	F	F	F	T

Table 4.3: **Apriori Algorithm Parameters**

Algorithm	**Apriori**
Objective	Lights sub-table
Mnimum support threshold	0.095
Mnimum lift threshold	3.2
Total amount of Messages	100,000,000

In Table 4.2, we performed data transformation and encoded transaction on 10 messages from Table 4.1, and obtained the following results.

4.3.2 Experiments

Experiment 1: How Apriori Algorithm applies to Delta Messages

The aim of this experiment is to verify whether Frequent Pattern Mining technology can be used to mine frequent patterns in Delta messages, and whether the idea of converting these patterns into use cases is feasible. We applied the Apriori algorithm to the Lights sub-table of the Delta messages to conduct frequent pattern mining experiments and verified the feasibility of converting the mined frequent patterns into use cases.

For this experiment, we selected one week's worth of data from the Lights sub-table, which contains all the information changes of the user's home lighting fixtures (e.g. saturation, on/off, color mode, hue, and brightness level), and is the most fundamental and important sub-table in the Delta messages, making it suitable as the dataset for this experiment. The parameter design of this experiment is shown in Table 4.3. We set the minimum support threshold to 0.195, which means that the frequent itemsets must appear in the dataset with a probability of at least 9.5%. The minimum lift value was set to 3.2, which means that we only consider those itemsets whose association rules have a lift value greater than or equal to 3.2.

During the experiment, we performed data validation on the preprocessed lights sub-table data. After confirming the correctness of the data, we applied the Apriori algorithm to the dataset. According to the Apriori algorithm, the model first scans the entire dataset to count the support of each item and filter out the frequent 1-itemsets. Based on the frequent 1-itemsets[2], candidate

itemsets are constructed and the support of each candidate itemset is counted to filter out frequent itemsets. Then, based on the frequent itemsets, the confidence of each itemset is calculated to filter out association rules that meet the requirements and return the frequent itemsets and association rules that meet the requirements[2].

Table 4.4: **Experiments 1 Results: Top 20 Frequent itemsets and Rules**

antecedents	**consequents**	**support**	**lift**
[zigbee, sat]	[xy]	0.2568	3.89408
[xy]	[zigbee, sat]	0.2568	3.89408
[rules, sat , colormode]	[xy]	0.213	3.87412
[zigbee, xy , colormode]	[sat]	0.2014	3.87411
[clip, sat]	[xy , colormode]	0.1725	3.87411
[clip, xy]	[sat , colormode]	0.172	3.82412
[rules, hue]	[sat]	0.182	3.81111
[rules, hue , xy]	[sat]	0.181	3.81014
[clip, xy , colormode]	[hue , sat]	0.177	3.65432
[clip, hue , sat]	[xy , colormode]	0.154	3.65432
[hue , xy]	[system, sat , colormode]	0.1421	3.423
[zigbee, sat , xy]	[hue , colormode]	0.1431	3.42143
[zigbee, hue]	[xy , sat , colormode]	0.1214	3.4134
[zigbee, sat]	[hue , xy , colormode]	0.1211	3.39132
[rules, xy]	[hue , sat , colormode]	0.1198	3.392315
[sat , ct]	[clip, xy]	0.1198	3.391408
[xy , ct]	[homekit, sat]	0.1052	3.31042
[clip, sat]	[xy , ct]	0.1024	3.31042
[zigbee, xy]	[sat , ct]	0.1011	3.30495
[colormode , sat , ct]	[rules, xy]	0.1010	3.30412

In Table 4.4, the results of the Apriori algorithm for association rules are presented, including antecedents, consequents, support and lift. It can be seen from the results that there are strong associations between hue and sat, hue and xy, and sat and xy, with confidence and lift values close to 1, indicating a strong correlation between these attributes in the dataset. In addition, it can be observed that there are also strong associations between different combinations of attributes, such as hue and sat, hue and ct, hue and colormode, and xy, with lift values higher than 3.2, indicating statistically significant correlations between these attribute combinations. Overall, these association rules suggest that there are strong correlations between these attributes and that they can be used to build data mining models.

During the experiment, we needed to convert the mined frequent patterns into use cases for better understanding and analysis of user behavior. The association rules we discovered in this experiment are combination itemsets of user status information, corresponding to the most direct and single light control operation of the user. Therefore, we can compare them one by one with the light control function area in the Hue App. For example, if we find a frequent pattern with antecedents "clip" and consequents "bri", it corresponds to the user frequently adjusting the brightness of the lights in the Hue App, and its association degree

is the highest among all the patterns.

The experimental results in Table 4.4indicate that frequent pattern mining technology can effectively extract frequent patterns in Delta messages, and these frequent patterns can be transformed into use cases. This provides an effective method for better understanding and analyzing user behavior. Therefore, we conclude that frequent pattern mining technology can be used for frequent pattern mining in Hue Bridge Delta messages, and these frequent patterns can be transformed into use cases.

Experiment 2: How Apriori Algorithm applies to Delta Messages

In Experiment 1, we have verified the feasibility of applying frequent mining algorithms to the data message. In this experiment, we will explore how to improve the completeness of Use Case information. In the previous validation experiments, we applied algorithms to the Lights, which is the most important sub-table, and obtained the relevant rules of the Hue Light status attribute. These rules indicate the user's preference for controlling the light attributes. However, a complete control use case not only needs to know which attributes the user prefers to change but also wants to understand the user's way and goal of controlling the behavior. For example, we already know that users tend to change both the color temperature and brightness of the light. Based on this, we also want to know whether the user performed this operation through the Hue app or the Dimmer switch and which type of light the user controlled specifically. In the Hue system, different types of lights support different status attributes. Some lights only support changing the brightness of the light, while others can change all status attributes such as the color temperature, Hue, and Sat. Therefore, this experiment aims to investigate a solution to improve the control use case of the lights.

Firstly, we need to clarify which information in the bridge device's configuration file can be used to determine the user's way and goal of controlling the light. After consulting the complete configuration documentation of the Hue Bridge, we filtered out multiple database query solutions and finally chose the following scheme:

- **Control method:** Source = Clip or Zigbee
- **Control target:** Light type = Extended color light, Color light, On/Off light, Color temperature light, On/off plug-in unit, Dimmable light
- **Light attributes:** hue, sat, xy, ct, colormode, bri, on

Regarding the control method, we selected the Source data in the delta message and filtered out the data with Source as Clip and Zigbee. Clip stands for "Connected Lighting for Interaction and Presence," which is a component of the Hue system used to provide developers with the authority to control Hue devices using RESTful APIs and WebSockets. For users, Source is usually Clip, which represents that the user used the Hue app or a third-party app to control the smart light. The Clip component plays the role of middleware in this process. On the other hand, if the Source is Zigbee, it means that the delta message is communicated through the Zigbee protocol. This represents that the user used a Zigbee switch supporting the ZigBee Light Link (ZLL) protocol and the ZigBee 3.0 protocol to control the smart light. It should be noted that these Zigbee switches are the ones that users already have before purchasing

Hue products, not the Hue Dimmer switch. The case of controlling the light with the Hue Dimmer switch belongs to the perception use case, corresponding to the rules sub-table in the delta message.

Based on the scheme, we designed the following SQL statement to obtain the dataset required for this experiment from the database:

```
start_date='2023-03-11'
end_date = '2023-04-25'
in_date_range = dt >= start_date and dt < end_date
in_source_range = source IN ('clip', 'zigbee')

SELECT *
  FROM (SELECT bridge_token, source, SPLIT_PART(address, '/', 4)AS id, hue, sat,
↪  xy, ct, colormode, bri, on
    FROM delta_messages_test.lights
    WHERE in_source_range
    AND in_date_range
    AND (hue IS NOT NULL OR sat IS NOT NULL OR xy IS NOT NULL
        OR ct IS NOT NULL OR colormode IS NOT NULL OR bri IS NOT NULL
        OR on IS NOT NULL)
    ) delta_lights
  INNER JOIN (SELECT bridge_token, id, type FROM full_config_test_lights WHERE dt
↪  = end_date ) config_lights
  USING(bridge_token, id)

  LIMIT 20000000
```

By this statement, we obtained integrated data within a certain time range, which includes parameter preferences of Light Control Behavior, specific methods of control behavior, and types of lights. Following the steps of frequent pattern mining defined in Section 4.3, we applied the algorithm to this dataset and obtained the results in Table 4.5.

Table 4.5: **Experiment Results: Frequent itemsets and Use cases**

Support	**Frequent itemsets**	**Use Cases [Translated by bridge experts]**
0.2022	[zigbee, hue, sat, xy, ct, colormode, Extended color light]	Use light switch to control the color(hue, sat, ct) of Extended color light#id
0.1921	[zigbee, bri, Extended color light]	Use light switch to modify the brightness of Extended color light#id
0.0879	[zigbee, bri, Dimmable light]	Use light switch to modify the brightness of Dimmable light#id
0.0774	[zigbee, hue, sat, xy, ct, colormode, bri, Extended color light]	Use light switch to modify the brightness of Extended color light#id
0.0708	[clip, bri, Extended color light]	Use Hue App to modify the brightness of Extended color light#id
0.0574	[zigbee, bri, Color temperature light]	Use light switch to modify the brightness of Color temperature light#id
0.0558	[clip, hue, sat, xy, ct, colormode, Extended color light]	Use light switch to modify the brightness of Extended color light#id
0.0347	[zigbee, ct, colormode, Color temperature light]	Use light switch to modify the brightness of Color temperature light#id
0.0232	[clip, hue, sat, xy, ct, colormode, bri, Extended color light]	Use Hue App to modify the brightness of Extended color light#id
0.0211	[clip, off, Extended color light]	Use Hue App to modify the brightness of Extended color light#id
0.0136	[clip, on, Extended color light]	Use Hue App to modify the brightness of Extended color light#id
0.0125	[zigbee, ct, colormode, bri, Color temperature light]	Use light switch to modify the brightness of Color temperature light#id
0.0112	[zigbee, off, Extended color light]	Use light switch to modify the brightness of Extended color light#id
0.0108	[clip, off, Dimmable light]	Use Hue App to modify the brightness of Dimmable light#id
0.0082	[clip, on, Dimmable light]	Use Hue App to modify the brightness of Dimmable light#id
0.0081	[zigbee, on, Extended color light]	Use light switch to modify the brightness of Extended color light#id
0.008	[clip, off, Color temperature light]	Use Hue App to modify the brightness of Color temperature light#id
0.0068	[zigbee, hue, sat, xy, ct, colormode, bri, on, Extended color light]	Use light switch to modify the brightness of Extended color light#id
0.0065	[zigbee, hue, sat, xy, colormode, Color light]	Use light switch to modify the brightness of Color light#id
0.0059	[clip, bri, Color light]	Use Hue App to modify the brightness of Color temperature light#id
0.0058	[zigbee, off, Color temperature light]	Use light switch to modify the brightness of Color temperature light#id
0.0054	[clip, on, Color temperature light]	Use Hue App to modify the brightness of Color temperature light#id
0.0053	[zigbee, off, Dimmable light]	Use light switch to modify the brightness of Dimmable light#id
0.0051	[zigbee, bri, Color light]	Use light switch to modify the brightness of Color light#id
0.0049	[zigbee, on, Dimmable light]	Use light switch to modify the brightness of Dimmable light#id

Based on the results, we found that compared to the results in Experiment 1, we assigned complete meanings to each mined frequent itemset. With these mined itemsets, experts can easily transform itemsets into exclusive Use Cases of the Hue system. This experiment validated that the proposed approach in the experiment can improve the Light Control type Use Cases and enhance their usability in virtualized containers.

Experiment 3: How Use Case Model applies to other sub-tables

In Experiments 1 and 2, we used the lights sub-table in State_message as the experimental object. In this experiment, we aim to explore the use cases corresponding to the other sub-tables and filter out the final use cases that can be applied in virtual containers and are effective. In addition to providing basic light control, the Hue bridge also provides rich functionality. The data generated by these functions is recorded in various sub-tables, such as the Groups sub-table, which records data on users' control of all lights in a room using the Hue App or Dimmer Switch. The Scene sub-table records the state data of users setting lighting scenes for different areas of the home (such as soft ambient lighting, reading lighting, party mode, etc.). The Sensor sub-table records the status data changes of all sensors used by the user (such as when the Motion Sensor is triggered or when the Dimmer Switch is used).

First, we focus on the Groups sub-table, which has a data structure similar to the Lights Control(Table 4.1). Following the method and steps of experiment 2, we designed the following scheme for the Groups sub-table:

- **Control method:** Source = Clip or Zigbee
- **Control target:** Group Class = Living, room, Bedroom, Hallway, TV, Kitchen, Other, Bathroom, Downstairs, Staircase, Office, Dining, Free, Front, door, Toilet, Home, Kids, bedroom, Upstairs, Garden, Laundry, room, Closet, Garage, Lounge, Storage, Computer, Recreation, Driveway, Man, cave, Terrace, Porch, Nursery, Studi, Carport, Guest, room, Top, floor, Music, Attic, Balcony, Reading, Gym, Barbecue, Pool
- **Light attributes:** all_on, any_on, alert, hue, sat, xy, ct, bri

According to the scheme, we performed data preprocessing, concatenation, and algorithm application on the Groups table. We selected 45 days of Groups Control data from the bridge user (a total of 20 million messages), and the model results are output in descending order of Support, as shown in Table 4.6. (Due to space limitations, we only list some of the mined itemsets in the main text.)

Table 4.6: **Experiment 3 Results: Frequent itemsets and Use cases for Groups sub-table**

Support	Frequent itemsets	Group Control Use Case
0.159552	('zigbee', 'Living room', 'message_all_on', 'message_any_on')	Use the switch to turn on all the lights in the Living Room (group#id)
0.148912	('zigbee', 'Hallway', 'message_all_on', 'message_any_on')	Use the switch to turn on all the lights in the Hallway (group#id)
0.1362	('clip', 'Living room', 'message_all_off', 'message_any_off')	Use the Hue App to turn off all the lights in the Living Room (group#id)
0.134292	('clip', 'Living room', 'message_off')	Use the Hue App to turn off one light in the Living Room (group#id)
0.119	('clip', 'Living room', 'message_all_on', 'message_any_on')	Use the Hue App to turn on all the lights in the Living Room (group#id)
0.08106	('clip', 'Bedroom', 'message_all_off', 'message_any_off')	Use the Hue App to turn on all the lights in the Bedroom (group#id)
0.072052	('clip', 'Bedroom', 'message_off')	Use the Hue App to turn on all the lights in the Bedroom (group#id)
0.069344	('clip', 'Bedroom', 'message_all_on', 'message_any_on')	Use the Hue App to turn on all the lights in the Bedroom (group#id)
0.063184	('clip', 'Living room', 'message_on')	Use the Hue App to turn on one light in the Living room (group#id)
0.06298	('zigbee', 'Bedroom', 'message_all_on', 'message_any_on')	Use the switch to turn on all the lights in the Bedroom (group#id)
0.060428	('zigbee', 'Hallway', 'message_bri')	Use the switch to modify the brightness of all the lights in the Hallway (group#id)
0.0537	('zigbee', 'Other', 'message_all_on', 'message_any_on')	Use the switch to turn on all the lights in other room (group#id)
0.050304	('zigbee', 'Living room', 'message_bri')	Use the switch to modify the brightness of all the lights in the Living Room (group#id)
0.047812	('clip', 'TV', 'message_all_off', 'message_any_off')	Use the Hue App to turn off all the lights in the TV room (group#id)
0.044136	('clip', 'TV', 'message_all_on', 'message_any_on')	Use the Hue App to turn on all the lights in the TV room (group#id)
0.041312	('zigbee', 'Kitchen', 'message_all_on', 'message_any_on')	Use the switch to turn on all the lights in the kitchen (group#id)
0.041172	('zigbee', 'Bathroom', 'message_all_on', 'message_any_on')	Use the switch to turn on any light in the bathroom (group#id)
0.039572	('zigbee', 'Living room', 'message_any_on')	Use the switch to turn on any light in the Living room (group#id)
0.039176	('zigbee', 'Staircase', 'message_all_on', 'message_any_on')	Use the switch to turn on all the lights in the Staircase (group#id)
0.036052	('clip', 'Bedroom', 'message_on')	Use the Hue App to turn on all the lights in the Bedroom (group#id)
0.036024	('clip', 'Living room', 'message_any_off')	Use the Hue App to turn off any light in the Living room (group#id)
0.029988	('clip', 'Living room', 'message_on', 'message_bri')	Use the Hue App to modify the brightness of lights in the Living room (group#id)
0.029752	('clip', 'Living room', 'message_any_on')	Use the Hue App to turn on any light in the Living room (group#id)
0.028816	('clip', 'Kitchen', 'message_all_off', 'message_any_off')	Use the Hue App to turn off all the lights in the Kitchen (group#id)
0.02856	('zigbee', 'Living room', 'message_all_off', 'message_any_off')	Use the switch to turn off all the lights in the Living Room (group#id)

According to the results in Table 4.6, we can see that each itemset corresponds to a source and a group class. For example, the first rule corresponds to the Zigbee protocol and the Living Room group class. These two items make up the control method and target of this use case, which is to use a switch based on the Zigbee protocol to control certain properties (message_all_on, message_bri) in the Living Room. Due to space limitations, only the top 25 itemsets by support are listed in the table. It can be observed that the majority of itemsets correspond to use cases where users control the lights in a room (primarily Living Room and Bedroom) using the app and switch, such as turning the lights on/off, changing the scene, and adjusting brightness. This conclusion is consistent with the user habit of using the Hue Bridge to control the lights in a room/area, and theoretically validates that our model analyzes user behavior habits in the Hue system to some extent. In Chapter **??**, we will conduct experiments deploying the model to a virtual bridge to verify that the model also simulates user behavior habits to some extent in engineering.

The Lights and Groups sub-tables are used by users to control home smart lighting using Hue devices. Therefore, the Use Case results obtained from mining these sub-tables are similar, but differ in the target of control and the difference between individual and group lighting. In this experiment, we selected another important feature of the Hue system, the rule engine, as the result demonstration. Compared with the user-initiated actions in the previous two sub-tables, the rule engine triggers actions passively based on the trigger conditions set by the user in advance. For example, if a user sets a rule that the bedroom light should be turned on when the motion sensor near the bed detects human movement and it is not nighttime, the rule engine will trigger this action if all the conditions are met. In the Hue system, users can set up to eight conditions and actions for each rule. Rules are a commonly used feature in the Hue system. The Rules sub-table in the delta message stores the state records of rules triggered by users, while the rules sub-table in the full config contains detailed configurations of user rules, including conditions and actions.

Based on the above information, we designed the following scheme and applied it to preprocessed and transformed rule data. We collected a total of 372,437 rules data for March 2023, with a total of 43,301,615 triggers. The frequent itemsets obtained by the frequent pattern mining algorithm and the use cases transformed through expert collaboration are shown in Table 4.7.

- **Condition Objects:** Sensor name or Sensor type of each condition = Hue dimmer switch, Hue Smart button, CLIPGenericStatus, Daylight, Hue outdoor light sensor, etc.
- **Condition Event:** Event = buttonevent, presence, any_on, daylight, lightlevel, expectedrotation, Status, dark, flag, etc.
- **Action target:** groups_action, light, sensors, storelightstate, groups_scene, etc.

In this chapter, we designed multiple experiments to analyze the feasibility of frequent pattern mining algorithms on the State_message dataset. Based on the algorithm results, we proposed a Use Case Model to analyze the user behavior of the Hue Bridge. We obtained a list of use cases belonging to Lights control/Group Control/Rules Triggered, which will be applied to the containerized virtual bridge in Chapter 5.

Table 4.7: **Experiment 3 Results: Frequent itemsets and Use cases for Rules sub-table**

Support	Frequent itemsets	Rules Triggered Use Case
0.464697238	[Hue dimmer switch_buttonevent, sensors_state_status]	When the user press the Hue dimmer switch, modify the state of sensor#id
0.022894534	[Hue wall switch module_buttonevent, groups_action_scene]	When the user press the Hue wall switch module, alter the scene of groups#id
0.017454014	[Hue motion sensor_presence, storelightstate]	When the Hue motion sensor detects presence, store the state value of light#id
0.015576444	[Hue Smart button_buttonevent, sensors_state_status]	When the user press the Hue Smart button, modify the state of sensor#id
0.004470688	[CLIPGenericStatus_status, groups_action_scene]	When ClipGneric Sensor been triggered, alter the scene of groups#id
0.00344172	[Hue dimmer switch_buttonevent, Lutron Aurora_expectedrotation, sensors_state_status]	When the user press the Hue dimmer switch and Lutron_Aurora Sensor rotate, modify the state of sensor#id
0.003083947	[ZLLSwitch_buttonevent, sensors_state_status]	When the user press the ZLL switch, modify the state of sensor#id
0.002631556	[Hue outdoor motion sensor_presence, _storelightstate]	When the Hue outdoor motion sensor detects presence, stores the state value of light#id
0.002040195	[Hue motion sensor_presence, groups_action_on]	When the Hue motion sensor detect presence, turn on the lights of group#id
0.001995843	Hue motion sensor_presence, sensors_state_status]	When the Hue motion sensor detect presence, store the state value of light#id
0.001812521	[CLIPGenericStatus_status, sensors_state_status]	When ClipGneric Sensor been triggered, modify the state of sensor#id
0.001510927	[Daylight_daylight, sensors_state_status]	When daylight sensor be triggered, modify the state of sensor#id
0.00114724	[Hue motion sensor_presence, groups_action_scene]	When the Hue motion sensor detect presence, alter the scene of groups#id
0.001141326	[Friends of Hue Switch_buttonevent, groups_action_scene]	When user press the Friends of Hue switch, alter the scene of groups#id
0.000987573	[Hue wall switch module_buttonevent, groups_action_on]	When the user press the button of Hue Wall switch module , turn on the lights of group#id
0.000594318	[Friends of Hue Switch_buttonevent, groups_action_on]	When user press the Friends of Hue switch, turn on the lights of group#id
0.000328205	[Hue wall switch module_buttonevent, Lutron Aurora_expectedrotation, groups_action_scene]	When the user press the Hue wall switch and Lutron_Aurora Sensor rotate, alter the scene of groups#id
0.000322292	[Hue outdoor motion sensor_presence, groups_action_on]	When the Hue outdoor motion sensor detects presence, turn on the lights of group#id
0.000298637	[Hue wall switch module_buttonevent, Lutron Aurora_expectedrotation, groups_action_scene]	When the user press the Hue wall switch and Lutron_Aurora Sensor rotate, alter the scene of groups#id
0.000230631	[Hue motion sensor_presence, sensors_config_on]	When the Hue motion sensor detect presence, turn on the config of sensor#id
0.00021289	[Hue dimmer switch_buttonevent, Hue ambient light sensor_lightlevel, sensors_state_status]	When the user press the dimmer switch and ambient light sensor achieve specific brightness, modify the state of sensor#id
0.000204019	[Hue dimmer switch_buttonevent, groups_action_scene]	When the user press the Hue dimmer switch and Lutron_Aurora Sensor rotate, alter the scene of groups#id
0.000189235	[Hue tap switch_buttonevent, groups_action_on]	When the user press the Hue tap switch, turn on the lights of group#id
0.000177408	[Hue outdoor motion sensor_presence, groups_action_scene]	When the Hue outdoor motion sensor detects presence, alter the scene of groups#id
0.000150797	[CLIPGenericStatus_status, groups_action_on]	When ClipGneric Sensor been triggered, turn on the lights of group#id

Chapter 5

Model Implementation and Kubernetes Deployment

In the field of IoT, device virtualization is a common technique [15] that simulates the interfaces, protocols, components of IoT devices to achieve simulation and testing of IoT device functionality. In Chapter 3, we successfully use Docker to virtualize the Hue Bridge device and verify its daemons, achieving the functional virtualization of Hue Bridge devices. In Chapter 4, we proposed a Use Case Model to analyze the behavior patterns of bridge users and applied frequent pattern mining algorithms to simulate Hue Bridge user behavior.

In this chapter, based on the results of two previous chapters, we use Kubernetes to scale up the virtualized bridge containers on a large scale. Kubernetes is a container cluster management and orchestration system based on Docker [16], developers use Kubernetes for automating software deployment, scaling, and Management [36]. The advantage of container technology is that it can easily deploy and manage applications in different environments, and Kubernetes provides automated management, file mounting, load balancing, high scalability, and other functions for these virtual bridges. Using Kubernetes can help us quickly create and manage a large number of virtualized devices, and provide each device with an independent network environment to better simulate scenarios in the real environment. In our experimental environment, we use the automatic scaling feature of Kubernetes to scale up the virtualized Hue Bridge devices, to simulate a more realistic firmware release scenario for bridge devices.

5.1 Model Implementation

In the bridge device, user commands are implemented through HTTP requests, which is a client-server protocol used for transmitting hypertext content. The Hue Bridge will parse and execute the command, thereby controlling the smart device. In this section, we will transform the Use Case Model from theoretical results into engineering achievements and apply it to the containerized virtual bridge. We also designed the Random Hue Commands Generator to simulate and emulate users' daily behaviors.

5.1.1 Hue Command Architecture

For the convenience of the readers, we define the HTTP Request API that can be parsed and executed by Hue Bridge as Hue Command throughout this paper. This section will introduce the structure of a Hue Command. Users can control smart devices in the Hue system through Hue commands. The architecture of Hue command is shown in Figure 5.1.A Hue Command consists of four parts which are sent as an HTTP packet:

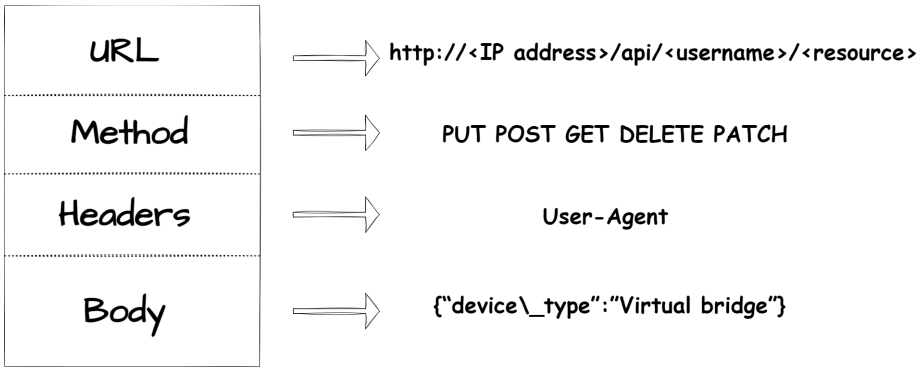


Figure 5.1: **Hue Command Architecture**

Command address: This is the address of the resource(lights/sensors) within the Hue System which the user would like to interact with. In the address, the keyword Username is an ASCII string unique for a registered client.

Command method: This is the method of the HTTP packet which indicates the action to be performed on the resource.

Headers: The HTTP header allow the client to pass additional information about the request to the server

Command body: The command body contains all data required for modification of the resources when using the methods PUT or POST. Command body is formatted in Javascript Object Notation (JSON) format. Using the body specific parameters of the resources can be given new values[27]. In Figure 5.1: "device_type":"Virtual bridge" refers to the device_type parameter of the resource and assigns it a new value of "virtual bridge".

5.1.2 Random Hue Commands Generator

After clarifying the structure of Hue Command, we need to fully apply the Use Case Model to the containerized bridge in order to simulate the daily behaviors of real users (i.e. the control commands sent to Hue Bridge). To achieve this, we have designed the Random Hue Commands Generator based on the Use Case Model, aiming to continuously send Hue Commands to the specified address of the virtual bridge container through an automated program.

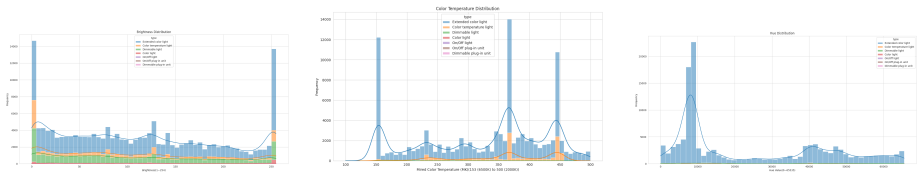
Before designing the generator, we need to preprocess and transform the use case lists obtained in Chapter 4 into machine-readable components of Hue Commands (mainly command body and method). We manually completed this task, combining multiple documents including the Commands Specification document of the Hue system, the bridge system design manual, and the Hue

lamps design document. We also invited several bridge development experts to review our Use Case list and finally obtained the following transformation results.

Table 5.1: **Transformated Results: Use Case ->Hue Commands**

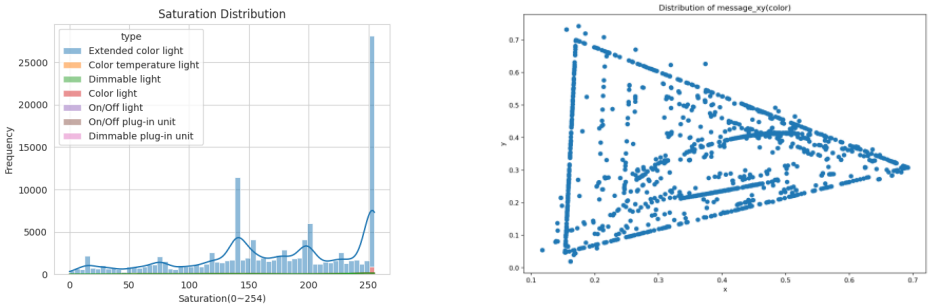
Use Case	**Method**	**Command URL**	**Command body**
Use Accessories to modify the brightness of light#id	PUT	/sensors/#id/state	{"bri": 200}'
Use Accessories to turn off the lights#id	PUT	/sensors/#id/state	{"on": False}'
User set the rule to modify the scene of lights#id	POST	/rules/<n> /light-states/id	{ "on":true,"ct":200}
User set the rule to modify the brightness of lights#id	POST	/rules/#id/	{"bri": 200}'
User set the rule to turn off the lights#id	POST	/rules/#id/	{"on": False}'
Use Accessories to modify the scene and brightness of light#id	PUT	/sensors/#id/state	{"scene":"<scene id>"}
Use Hue apps to modify the brightness of light#id	PUT	/lights/#id/state	{"bri": 200}'
Use Hue apps to modify the scene of light#id	PUT	/lights/#id/state	{"scene":"<scene id>"}
Use Hue apps to turn off the light of light#id	PUT	/lights/#id/state	{"on": False}'
User set the rule to turn on the light#id and modify the scene and brightness of light#id	POST	/rules/#id/	{"on": True, "ct": 200, "bri":200}'

As shown in Table 5.1, we can see that the JSON format values in the body have allocated the upper limit of each parameter as the initial value. Initially, our idea was to randomly select a value for each parameter from its scope when running the automation script, for example, we could randomly select an integer from 1 to 255 for brightness. However, we quickly dismissed this idea because, from the perspective of simulating user behavior rigorously, not only do we need to simulate users' actions, but their preferences for parameters are also equally important.



(a) **bri distribution** (b) **ct distribution** (c) **hue distribution**

Figure 5.2: **Delta Message: Distribution of Parameters (Part A)**



(a) **sat distribution** (b) **xy distribution**

Figure 5.3: **Delta Message: Distribution of Parameters (Part B)**

Therefore, before designing the Random Hue Commands Generator, we used EDA (Exploratory Data Analysis) methods from statistics to analyze the distribution of numerical type parameters (brightness, Hue, Sat, xy, color temperature) in the delta_message dataset. We selected three months of user data, and the results were grouped by resource type, as shown in Figure 5.2and 5.3:

In Figure 5.2and 5.3, we demonstrate the distribution of the status parameters of the Lights sub-table. Based on the distribution data, we generated a normal distribution list for each parameter. These lists record the probability distribution of each parameter. We set boundaries and intervals for each probability distribution and used the *np.random.choice* function from the Numpy library. We input these lists as the probabilities parameter (-p) of the function.

We need to design an automated program that can read a list of Hue commands and send them randomly. The program needs to meet the following requirements:

1. Automatically create users and obtain command addresses and usernames.
2. Select resources (lights/groups/rules) based on the full_config of each bridge, rather than randomly assigning resources.
3. Use the distribution of light state parameters as input and allocate parameters in accordance with the data distribution.
4. Have a protection mechanism to prevent the program from exiting due to accessing invalid resources.
5. Comply with Hue Command restrictions to ensure a success rate of at least 95% for command execution. or meaning of the original text.

According to the program design specifications and requirements, we have designed a multi-threaded Random Hue Commands Generator. The program's design flowchart is shown in Figure 5.4. The program's threads consist of a main thread and three looping threads. Before randomly sending Hue Commands, we need to complete User Authentication and Remoteless Commissioning for the virtual bridge. User Authentication simulates the process of the user authorizing and creating a new user by pressing the Bridge device center button, and returns the current user's username (an ASCII string unique for a registered client) to

the program. We store the current virtual bridge's IP address and username in a Hashmap for later process calling. Remoteless Commissioning is the default commissioning method of the Hue Bridge. After the user completes the Authentication process, the bridge device needs to actively discover connectable devices in the surrounding area. Therefore, before executing the instructions, we also need to make the virtual bridge actively search for and connect to the lights and sensors simulated by HSE. Meanwhile, the main thread will read in the Lights, Groups, and Rules Use Case converted to Hue Commands list from the database.

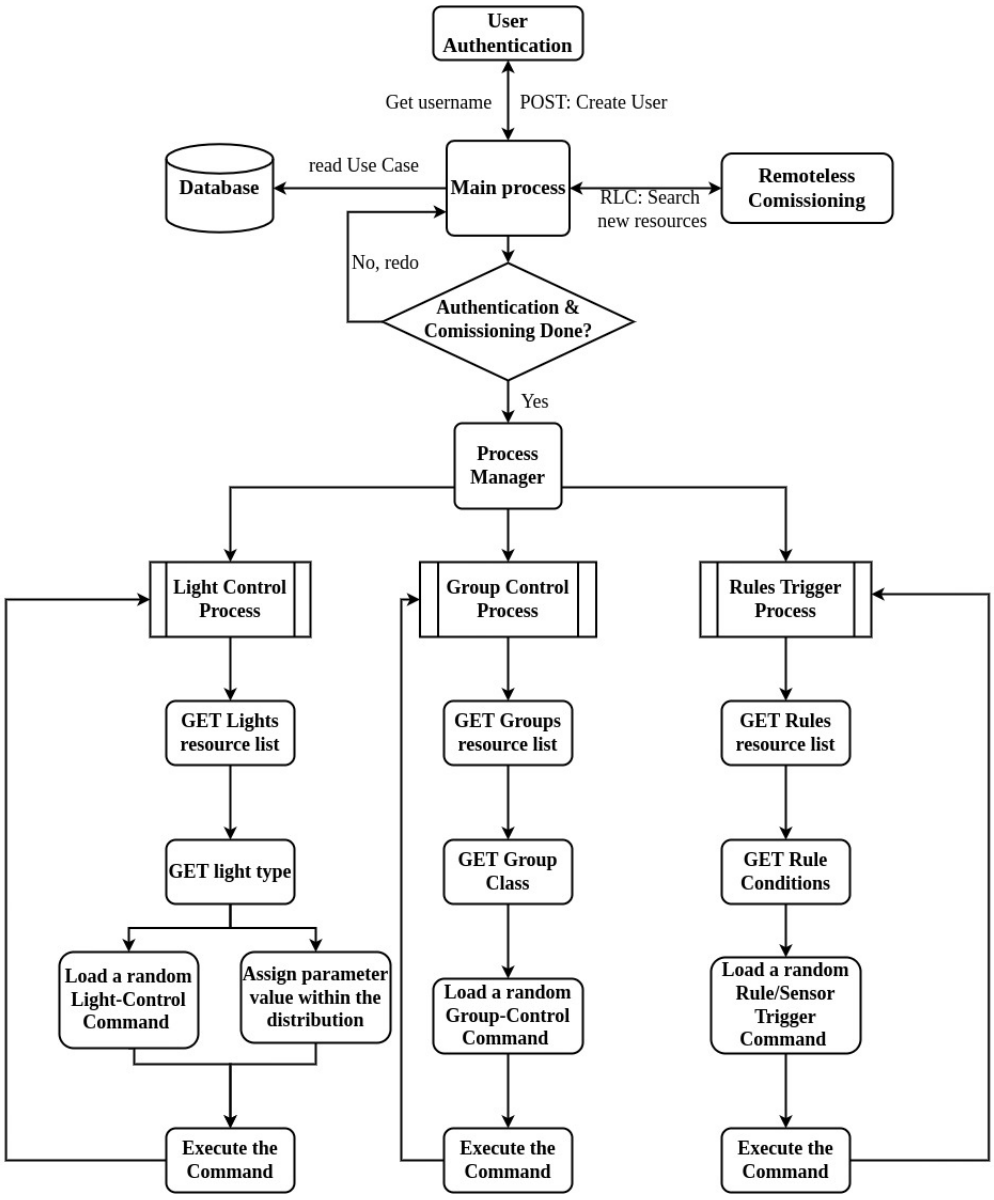


Figure 5.4: **Sequence Diagram: Random Commands Generator**

After completing these tasks, the main thread will check whether the above tasks have been completed. If they are not, the process will restart. If they have been completed, the random command generator thread will be allowed to start executing. The random command generator consists of three threads:

- **Light Control Process**: used to simulate user-controlled light control behavior.
- **Group Control Process:** used to simulate user-controlled behavior of all Lights in a room/Group.
- **Rules Trigger Process:** used to simulate user behavior of controlling lighting devices triggered by sensors or switches. The basic logic of triggering is to simulate the sensor trigger status through Hue Commands to meet the conditions of the bridge full config rule, and finally trigger the rule.

In these three threads, to avoid the program from exiting due to accessing invalid resources and to improve the success rate of command sending, we designed a feedback mechanism. Taking the Light Control Process as an example, we first obtain the light resource of the bridge corresponding to the current IP address through a GET command, which includes the total number of lights, the types of lights, and the status information of the lights. Based on this information, we define the scope of the Commands supported by these resources, and then we select a subset from the Hue Commands Dataset based on the scope. This subset contains all the supported Hue Commands in the scope, unique to the current Virtual bridge. The advantage of doing this is to avoid random command access to invalid resources or attempting to change unsupported parameters while maximizing the individuality and independence of the virtual bridge.

Next, the Light Control Process will randomly select a Light from the subset and randomly select a Hue Command supported by that Light. At the same time, according to the body part of the Command, we randomly generate the parameter values of each attribute from the distributed light attributes in the database as needed (the random process strictly follows the distribution data). Finally, since the Light Control mainly changes the state of the light, the Command Method is mainly PUT. This way, we obtain a complete Light Control command. The thread will randomly send the Hue Command to the specified virtual bridge at a certain frequency, the dashboard of Hue devices is shown in Figure 5.5, We can see that this bridge has changed the status of the Hue Devices it manages through Hue Commands.

It should be noted that, according to the bridge design document's limitations, the frequency of sending Light control commands should be less than 10 commands per second, and the frequency of Group/Rules commands should be less than 1 command per second.

In this section, we apply the generated Use Case model to the containerized bridge and design a random command generator to continuously send commands to the bridge. The generator randomly generates HTTP request commands to simulate the behavior patterns of Hue Bridge users. These commands include Light Control and Sensor Simulation, which can more realistically simulate user operations. Next, we will design large-scale deployment of containers and apply

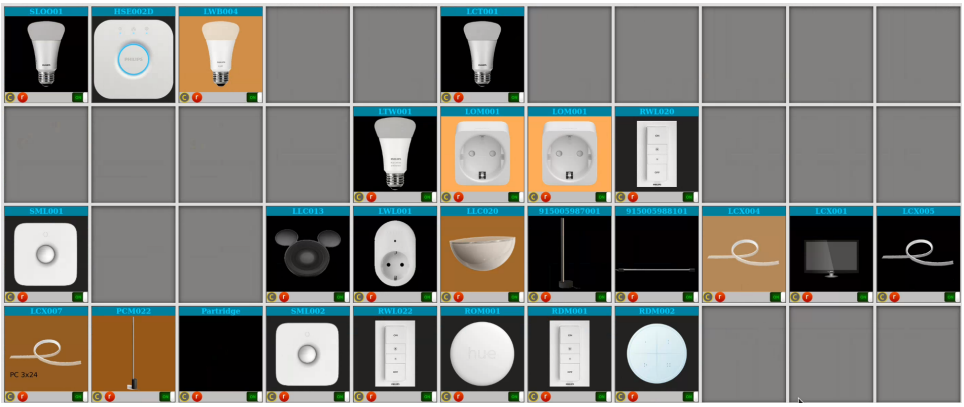


Figure 5.5: **Hue Dashboard: Random Commands Generator Demo**

the random command generator to Kubernetes to achieve batch user behavior simulation for large-scale Virtual Bridges.

5.2 Kubernetes Deployment

In Section 5.1.2, we applied the use case model to the virtual bridge container, combined with the distribution of light state data, to implement a Hue Bridge random control command generator based on HTTP requests. In this section, we will provide a detailed technical introduction to the design of the Kubernetes large-scale deployment and scaling architecture, the architecture of Kubernetes, and the types and functions of each component in the Kubernetes architecture. At the same time, we will test the stability and command success rate of the Use Case Model and random control command generator after Kubernetes (container scaling) deployment to comprehensively verify the feasibility of the use case model in the Hue Bridge. In this chapter, we will also test the stability of the virtual bridge in functional and cloud diagnostic data transmission after deploying it using Kubernetes.

5.2.1 Deployment and Scaling Architecture

Before formal deployment, we first introduce some Kubernetes operation objects involved in this project. We will use these operation objects to construct the project architecture:

- **Pod:** A Pod is a container group consisting of multiple containers, which share the same IP, port, and volume, and is the smallest controllable unit in the deployment aspect of Kubernetes.
- **Service:** A Kubernetes service is a high-level abstraction of a container group's logic, which also provides port-based services to the outside world to provide access policies for the container group [16].
- **StatefulSet:** A StatefulSet is used to manage and scale a set of Pods, and can assign numbers and uniqueness guarantees to these Pods. Compared

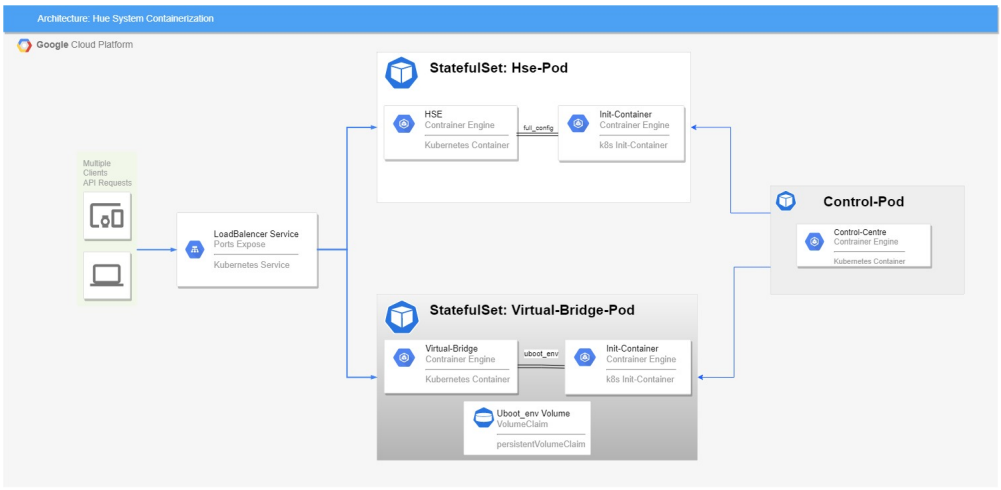


Figure 5.6: **Kubernetes: Deployment Architecture Overview**

with the traditional container orchestration form Deployment, StatefulSet is more suitable for managing stateful applications (applications with unequal relationships between instances and dependencies on external data) of the workload API object.

- **Volume:** A storage mount, used to store data or configuration files that need to be used before the Pod is running.
- **ClusterRole:** Assign specific permissions to specific objects through Role-Based Access Control (RBAC).
- **Init-Container:** The Init container is started before the regular containers in a Pod and is responsible for performing initialization tasks. Init containers support all the features and fields of regular containers, and they automatically exit once their job is done.

The deployment architecture is shown in Figure 5.6, the core logic of the architecture is that the Virtual Bridge and HSE each correspond to a StatefulSet, which are connected through a load-balanced type of service to establish a local network and connect Zigbee communication. At the same time, the Control Pods are responsible for running the random command generator, which has permission to access and execute other pods. Within each pod of the StatefulSet, the Init-container is responsible for distributing configurations and adaptive initialization.

5.2.2 Pod Components and their functions

Table 5.2: **Kubernetes: Components function and replicates**

Name	**Type**	**Function**	**Replicate**
Statefulset Virtual bridge Pod	Stateful set	Generate a containerized Hue Bridge Pod with a specified firmware version and a specified number of replicates. Each Pod will contain a virtual bridge container and an init container.	30
Statefulset HSE Pod	Stateful set	Generate Hue System Emulator (HSE) with specified firmware version and the specified number of replicates. Each pod contains an HSE container and an init container.	30
Control Pod	Pod	As the control center of the entire deployment, it is used to send kubectl commands to pods in batches and start Random Commands Generators.	1
Virtual bridge Service	Service	Responsible for providing LoadBalancer type port mapping services, allowing processes within the container to use designated ports.	1
HSE Service	Service	Responsible for providing LoadBalancer type port mapping services, allowing processes within the container to use designated ports.	1
kubectl control role	Cluster Role	To define access permissions for resources within the cluster scope, including read/write and exec permissions for Pods, to grant Control Pod the permission to send instructions to other pods.	1
kubectl control rolebinding	Cluster Role Binding	Bind the kubectl control role with the ServiceAccount (default).	1
vb claim0	Persistent Volume Claim	The Volume used in Kubernetes to store configuration files, with a capacity of 100Mi.	1

Table 5.2 lists the name, type, function and replicate of all components in the system (amount of replica components[16]).

5.2.3 Deployment Procedures and Strategies

In section 5.2.2, we introduced the components and their functions in a Kubernetes deployment. In this section, we will discuss the deployment process and strategy of Kubernetes. The entire deployment process is divided into three stages: the pre-deployment preparation stage, the component verification stage, and the service running stage.

Pre-deployment preparation stage: In the pre-deployment preparation stage, before starting the containers, we first need to create the resources and tokens required by each component. We write the configuration files (in YAML format) for each Kubernetes component according to the project requirements and Kubernetes orchestration specifications, and choose to use Minikube (a lightweight local Kubernetes cluster) to create each component. The configuration files for each component must meet the following requirements:

1. The uboot configuration mount is shared between the containers in the Statefulset_Virtual_bridge_Pod, and the enable virtual is read and programmed into the assigned uboot_env file before the container init.

2. The full_config configuration file for the bridge is shared between the containers in the Statefulset_HSE_Pod. A command is set up so that the HSE container loads full_config immediately after startup and runs the HSE Zigbee device simulator according to full_config, while waiting for the Virtual Bridge to connect.

3. The Cluster Role grants Control Pod read/write and exec permissions for other Pods, and assigns a token to the Control Pod.

4. The HSE and Virtual Bridge containers pull images with the same firmware version number, and different version numbers will be used for project verification

Components verification stage: After creating the components, we need to verify their functionality and stability. First, we need to verify that the Init-container has completed its initialization work in each Pod and that the configuration files and initial commands have been completed. For other components, we need to check the mapping of each port and the service components to verify that the Control Pod has been authorized to read and write to other pods. The Virtual Bridge and HSE containers need to burn their corresponding uboot and configuration files when they start up. We enter these containers one by one to check whether the configuration files have been successfully loaded and applied to the containers. The Dashboard status of Minikube is shown in Figure 5.7, we can see that all the Pods that need to be deployed have pulled the images, completed the init phase, and successfully run the services in a stable manner.

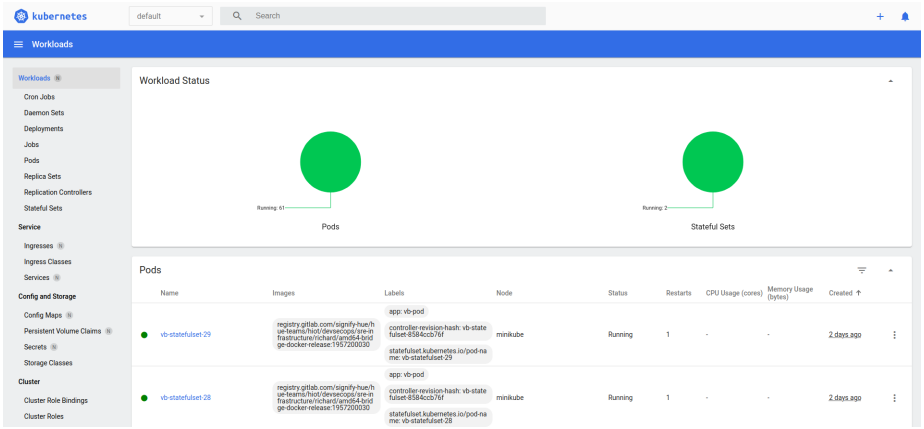


Figure 5.7: **Minikube: Deployment Dashboard Overview**

Service running stage: After everything is ready, we enter the service running stage, where we will officially implement the Use Case Model and Random Hue Commands Generator into the Kubernetes architecture. First, we enter the Control Pod, which contains automation scripts that will send kubectl commands to all Virtual Bridge Pods, with three specific functions:

- First, run the initialization script to assign the IP address of the HSE Pod to each Virtual Bridge in turn and establish communication through the ZigBee daemon to complete the one-to-one pairing of the HSE and Virtual Bridges.
- Start MQTT communication with the cloud. Before simulating user behavior with Hue Commands, we need to establish communication with the cloud database to ensure that the Virtual Bridge's analysis and diagnostic data can be fully synchronized with the cloud. In the Beta stage, this diagnostic data is the core data used to verify whether firmware has reached the release standard. Therefore, we need to ensure that the diagnostic and analysis data generated by the virtual bridge can also be synchronized with the cloud and that the Validation Engineers can use this data for device diagnosis.
- Finally, we start the Random Hue Commands Generator designed in Section 5.1.2. Following the design process, the Generator will create new users for all virtual bridges and complete the RLC (Remoteless Commission) for the new devices, and start three threads to send random Hue Commands at a regular frequency.

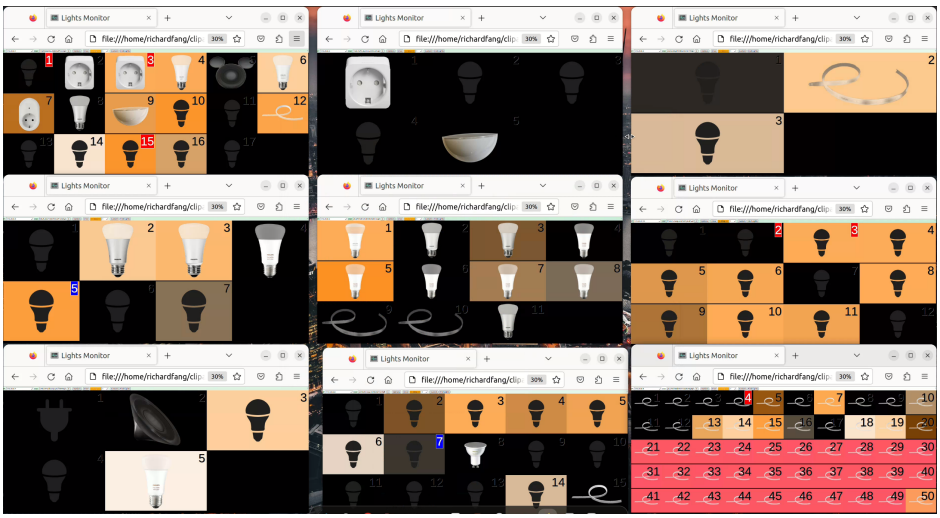


Figure 5.8: **Deployed Kubernetes System Dashboard**

In Figure 5.8, we demonstrate the dashboard of the virtual bridge in the Kubernetes deployed pods. It can be seen that each pod corresponds to a virtual Hue system (virtual bridge + HSE), which changes its status according to the instructions generated by the random command generator, as shown in the demo video of this project. In this section, we designed a random command generator based on the Use Case Model and the architecture for the large-scale deployment of virtual Hue systems using Kubernetes. We also introduced the functions of each component in the architecture. Additionally, we listed the steps for actual large-scale deployment and ultimately achieved deployment, scaling, and user behavior simulation for the containerized Hue System (Bridge).

Chapter 6

System Validation

In Chapter 2, we presented the research direction and research problem of this paper: Whether containerization can truly be applied to the Hue System to improve firmware deployment efficiency and provide valuable feedback as Beta Testers? Based on this question, we explored and designed solutions for the firmware update issue of the Hue Bridge, and now it is time to validate the feasibility of the final solution.

This thesis aims to enhance the efficiency of software deployment for bridge devices and reduce the deployment cycle. After completing the simulation of bridge device functionality and user behavior, as well as the large-scale deployment using Kubernetes, we need to design comparative experiments to validate the feasibility of virtual bridge devices in the software release cycle and the reliability of generating device diagnostic data. Similar to the Hue Bridge used by Beta testers, the virtual bridge will forward the analysis and diagnostic data generated during the operation of virtual bridges to the cloud database after deployment. The analysis data corresponds to the Delta_Message Dataset used to analyze the Use Case Model in Chapter 4, while the diagnostic data corresponds to the Beta Diagnostic Dataset.

The Beta Diagnostic database is used to store software/hardware diagnostic data sent by the Hue Bridge devices to the cloud (forwarded via the MQTT protocol). These diagnostic data include the Reset_reason sub-table, which records daemons reset logs, the Bridge_zigbee_log sub-table, which records Zigbee device logs, and the Iot_conectivity_stats sub-table, which records the communication status of IoT devices. In total, there are 53 sub-tables. Table 6.1 presents the information and examples of the Beta Diagnostic Dataset's sub-tables. The data among the sub-tables are independent and each sub-table represents a specific

Table 6.1: **Diagnostic Dataset illustration**

Name	**Beta Diagnostics**				
Description	Collected Hue Bridge device diagnostic and log data				
Sub-tables	Total 35 Subtables (Independent with each other)				
Example	Resetreason: Daemon Reset Reasons				
	component	reason	bridge id	software version	dt
	Daemon A	Exit Code 1	AE024GH	1957147030	2023/5/20

type of diagnostic data. Some sub-tables record data continuously in real-time, while others record data when events occur. The example in Table 6.1 belongs to the event recording type and records a reset event of Daemon A that occurred on May 20, 2023, with the reset reason being Exit Code 1.

In this chapter, we will divide the validation process into two parts: validation of the Use Case Model and validation of Kubernetes Deployment. In Section 6.1, we aims to validate whether the Use Case Model, when applied to the Kubernetes-based large-scale container deployment, can generate analysis data and use cases similar to that of the Beta testers. In Section 6.2, we aims to validate whether the feedback and diagnostic data generated by the virtual containers, deployed and managed using Kubernetes, exhibit similar behavior to that of the Beta testers. More important, we want to validate whether the virtual system accelerate the generation of user data after firmware deployment.

We referenced ideas and specifications for firmware validation from [11, 3, 19] in our experimental design guidelines.

6.1 Use Case Model Validation

In Chapter 4, we designed the Use Case Model and analyzed the user behavior patterns of the Hue Bridge devices using the frequent pattern mining algorithm. Based on the obtained Use Case Model, we developed the Random Command Generator and applied it to Kubernetes Deployment. In this chapter, we will design experiments to validate the Use Case Model.

Validation Experiment 1: *Will virtual bridges generate similar analysis data (Delta Messages) when the Use Case Model is applied to Kubernetes Deployment?*

First, we need to verify whether the Use Case Model can simulate the user behavior of the Hue Bridge. After deploying virtual bridges using Kubernetes, we can collect the analysis dataset (Delta Messages) from the virtual bridges. We have previously described this dataset in detail, but let's recap. The Delta Messages store the changes in status data of the devices within the Hue system, such as the brightness, switch state, and color temperature of the lights. These status information are generated when users send Hue Commands to the Hue Bridge. Therefore, when we apply the Use Case Model (Random Command Generator) to the virtual bridges, the virtual bridges should also generate and collect Delta Messages. To validate our hypothesis, we have designed the following experiment:

- **Experimental Object:** 30 virtual bridges
- **Experimental Objective:** To validate that virtual bridges can generate analysis data and have consistent data distribution with the beta dataset.
- **Experimental Duration:** 5 days
- **Experimental Tools:** Minikube, Docker, Hue SQL Tool

We proceed with the experiment. On the first day, we prepare specific versions of the Hue Bridge firmware image and HSE image in the private registry. This allows us to pull the images from the private registry when Kubernetes is started. Next, we start the Kubernetes service (Minikube Start) and deploy 30 virtual

bridge pods and the HSE pods following the Kubernetes deployment steps in Chapter 5. We establish Zigbee communication for each virtual bridge. Each virtual bridge has a different configuration file, bridge_id, and associated devices. After checking the status of each daemon and data communication with the cloud, we start the random command generator, and the 30 virtual bridges begin their operation.

Note: This experimental initialization process is applicable to subsequent experiments. The difference lies in the selection of generated data and the analysis methods used. Therefore, the subsequent experiments will not repeat the detailed description.

Table 6.2: **Delta_Message: Config sub-table Example (collected from virtual bridges)**

fw_version	timestamp	bridge_id	address	source	linkbutton
1957200040	2023-05-19T21:31:54	6HFDUI98JH	api/0/config	system	TRUE
1957200040	2023-05-19T18:39:41	678DFJHXC	api/0/config	system	FALSE
1957147030	2023-04-25T12:09:45	9DFNKJC78	api/0/config	system	FALSE
1957147030	2023-05-02T09:16:01	7D9F8GJVS	api/0/config	system	TRUE
1957147030	2023-05-01T12:25:10	89CVJF8SD	api/0/config	system	TRUE

After reaching the designated termination time, we organized and analyzed the experimental data. Based on the data analysis, we discovered that the virtual bridge and the bridge device were consistent and generated Delta_Message data for seven sub-tables. The data types and formats were identical. In Table 6.2, we took the Config sub-table as an example. This example illustrates the link button status data collected from virtual bridges. It indicates that at that moment, we simulated the action of pressing the bridge button to create a new user (True indicates a press, False indicates a release), and this behavior was recorded in its entirety.

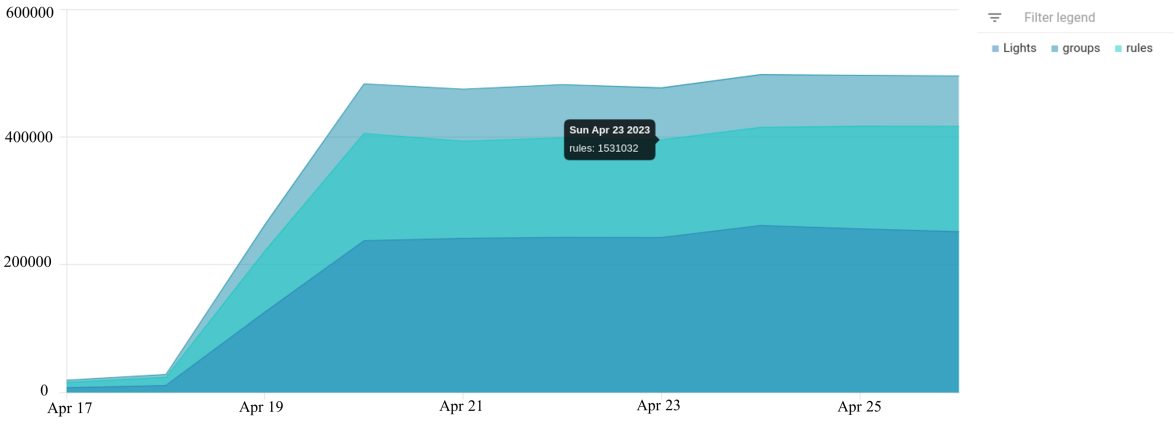


Figure 6.1: **Number of Delta messages per sub-table(Virtual Bridges)**

Due to the large volume of data, it is not feasible to fully demonstrate the data in this thesis. Therefore, we conducted statistics on the data from various

sub-tables, and the statistical results are shown in Figure 6.1. During the experimental period, the Hue cloud database collected delta messages generated by virtual bridges. The figure demonstrates the data totals for three sub-tables (lights, rules, groups), which correspond to the three threads of the Random Commands Generator. It can be observed that the delta messages for the three sub-tables showed a significant increase on the day of deployment, followed by a uniform distribution over time. Our research findings indicate that virtual bridges, like Beta Testers' Bridges, are capable of generating analytical data (delta messages). This result supports our hypothesis.

Validation Experiment 2: *Will applying the Frequent Pattern Mining algorithm to Delta Messages generated by Virtual bridges result in the same Use Case?*

In Chapter 4, the Use Case Model is generated based on the Delta_Messages dataset, combined with the Full_Config dataset to enhance the completeness and objectives of the Use Case. In Experiment 1, we have already verified that Virtual Bridges, when applied with the Use Case Model, can generate user data similar to that of Beta Testers (same sub-tables, similar data distribution). Based on the validation results from Experiment 1, we couldn't help but have a further idea: How does the similarity and coverage of the Use Cases generated by the model compare to the Beta Testers' Cases if we follow the Use Case Modeling Process (Data preprocessing, Pattern Mining, Pattern analysis, and Validation) described in Chapter 4?

Table 6.3: **Number of Use Cases (Frequent itemsets) derived by Beta Testers and Virtual Bridges**

	Beta Testers	**Virtual Bridges**	**Coverage %**
Lights Control	76	51	67.1%
Groups Control	159	140	88.1%
Rules Triggered	280	185	66.1%
Other Case	15	5	33.3%
Total Case	530	381	71.9%
Filtered Total Case	461	381	82.6%

To validate our hypothesis, we selected the Delta_Messages data (generated by Virtual Bridges) with a duration of two weeks for comparative experiments. Taking the Rules Triggered Use Case as an example, the comparison results are shown in Table 6.4. The Table lists the results of the Rules Triggered Use Case from the chapter (part of the Use Cases is extracted due to paper length constraints) and the identical Use Cases discovered by Virtual Bridges in this part. The identical portions are highlighted in orange. Based on the results, we found a high degree of similarity between the two. The statistical results of each sub-table are shown in Table 6.3, and the overall Use Case coverage reached 71.9% (381/530), with the results from Virtual Bridges being a subset of the Beta Testers' results. Through our analysis of Delta Messages, we discovered that the reason why some Use Cases could not be generated in Virtual Bridges is mainly due to certain sensor types that cannot be simulated through the HSE (Hue System Emulator), such as daylight sensors and ClipGeneric Sensors. The Random Command Generator attempts to send corresponding Hue commands,

but the Sever cannot find the object and return error Response. If we filtered these Use Cases, the overall Use Case coverage can reach 82.6% (381/461).

Based on the experiments in this section, we have verified two hypothetical questions regarding the Use Case Model. It has been confirmed that after applying the Use Case Model in the form of a Random Command Generator to the system deployment of Kubernetes, it can generate analysis data and corresponding Use Cases. Although the coverage is not 100% (82.6%), given the current available resources, it successfully simulates the real behavior of Beta users to the maximum extent possible.

6.2 Automated Feedback Validation

In Section 6.1, we validated the effectiveness of the Use Case Model in improving firmware update efficiency through focused analysis and diagnostic data. In this section, we further validate the system. For validation engineers, discovering feedback from diagnostic data is an important validation step. Automated feedback is a generic term for issues and bugs discovered through diagnostic data in software development (as opposed to user feedback actively sent to developers through feedback portals). Developers analyze the validation data and compare it horizontally with previously released firmware data to identify anomalies in the current firmware. In this section, we will follow the validation engineer's process of analyzing firmware to validate if virtual bridges perform similarly to beta testers in discovering feedback. Compared with the experiment in Section 6.1, the most essential difference between the verification experiment in this Section is the difference in the target dataset. The data set in Section 6.1 is the Analysis dataset (Delta Message) of Hue Bridge, and the corresponding dataset in this Section is the Diagnostic dataset (Diagnostic Messages) of Hue Bridge, the diagnostic dataset is the final object of the entire Beta Test stage.

To validate our hypothesis, we designed the following experiment setup (Applies to all experiments in this section):

- **Experimental Objects:** 30 virtual bridges (run 24 hours per day, managed by Kubernetes)
- **Experimental Purpose:** To verify the consistency or similarity between the visualization results of diagnostic data during the operation of virtual bridges and the historical results of beta testers.
- **Experimental Duration:** 1 weeks to 4 weeks (Beta Testers use historical Diagnostic data, and the Virtual Bridges' date corresponds to the experimental date, so the Duration is the same but the date is different.)
- **Experimental Tools**: Minikube, Docker, Kubectl, Hue SQL Tool
- **Use Case Model:** The Use Case Model was applied in this experiment.

Validation Experiment1: *Does the user diagnostic result obtained after deploying a specific firmware version to the virtual bridge resemble that of the Beta Testers?*

This experiment aims to compare the analysis results of diagnostic data (Automated Feedback) between virtual bridges and beta testers under the same

Table 6.4: **Comparison Use Case Results between Beta Testers and Virtual Bridges (Identical use cases is displayed in orange)**

Rules Triggered Use Case (via Beta Testers)	**Validation Use Case (via Virtual bridges)**
When the user press the Hue dimmer switch, modify the state of sensor#id	When the user press the Hue dimmer switch, modify the state of sensor#id
When the user press the Hue wall switch module, alter the scene of groups#id	When the Hue motion sensor detect presence, store the state value of light#id
When the Hue motion sensor detects presence, store the state value of light#id	When the Hue motion sensor detects presence, turn on the light of#id
When the user press the Hue Smart button, modify the state of sensor#id	When the user press the Hue Smart button, modify the state of sensor#id
When ClipGneric Sensor been triggered, alter the scene of groups#id	When the user press the Hue dimmer switch and ambient light sensor achieve specific brightness level, modify the state of sensor#id
When the user press the Hue dimmer switch and Lutron_Aurora Sensor rotate, modify the state of sensor#id	When the user press the Hue dimmer switch and ambient light sensor achieve specific brightness level, modify the state of sensor#id
When the user press the ZLL switch, modify the state of sensor#id	When the Hue outdoor motion sensor detects presence, stores the state value of light#id
When the Hue outdoor motion sensor detects presence, stores the state value of light#id	When the user press the Hue tap switch, turn on the lights of group#id
When the Hue motion sensor detect presence, turn on the lights of group#id	When the Hue outdoor motion sensor detects presence, alter the scene of groups#id
When the Hue motion sensor detect presence, store the state value of light#id	When the user press the Hue tap switch, turn on the lights of group#id
When ClipGneric Sensor been triggered, modify the state of sensor#id	When ClipGneric Sensor been triggered, modify the state of sensor#id
When daylight sensor be triggered, modify the state of sensor#id	When the Hue motion sensor detect presence, turn on the lights of group#id
When the Hue motion sensor detect presence, alter the scene of groups#id	When the Hue motion sensor detect presence, alter the scene of groups#id
When user press the Friends of Hue switch, alter the scene of groups#id	When user press the Friends of Hue switch, turn on the lights of group#id
When the user press the button of Hue Wall switch module , turn on the lights of group#id	When the user press the Hue Tap switch, turn on the lights of group#id
When user press the Friends of Hue switch, turn on the lights of group#id	When the user press the Hue wall switch module, alter the scene of groups#id
When the user press the Hue wall switch and Lutron_Aurora Sensor rotate, alter the scene of groups#id	
When the Hue outdoor motion sensor detects presence, turn on the lights of group#id	
When the user press the Hue wall switch and Lutron_Aurora Sensor rotate, alter the scene of groups#id	
When the Hue motion sensor detect presence, turn on the config of sensor#id	
When the user press the Hue tap switch, turn on the lights of group#id	

firmware. Prior to the experiment, we conducted research on the process of validation engineers obtaining automated feedback. After collecting diagnostic data from beta testers, validation engineers visualize the data using the Dashboard feature of the Hue Data Management Platform. The diagnostic data is grouped and demonstrated based on various dimensions such as time, firmware version, error type, reset_reason, and reconnection, according to validation requirements. By visualizing the diagnostic data and comparing it with past diagnostic data, anomalies in the current firmware can be more intuitively identified, which is the main source of automated feedback. According to our hypothesis, after deploying the same firmware, virtual bridges and beta testers should have similar or identical results in the visualization of diagnostic data.

In this experiment, we will select representative feedback types for comparative analysis. The control group consists of the validation results of Beta users in the specified firmware version phase. Firstly, we conducted experiments on the Automated Feedback corresponding to the "Daemon Reset Reason" sub-table. Typically, validation engineers would gather and compare the occurrences of frequent daemon resets among the current version's Beta Testers. Each diagnostic data contains a "ResetCode" for presenting the specific reasons for the daemon reset, such as user restart or memory errors. By observing the change curve/bar of the total number of Daemon Resets (Resets per day of all Beta Testers), verification engineers can intuitively judge the stability of the Daemon in the currently deployed firmware version. If there is a noticeable increase in the number of resets during the test, it indicates an anomaly. Engineers will investigate the specific cause of the reset and release an update in subsequent firmware versions to solve the problem.

Following the diagnostic data analysis process, we created a "Reset Reason Dashboard" for Virtual Bridges. The dashboard uses the same SQL logic to visually present the reset reasons for each daemon. As a comparative experiment, we also queried the historical dashboard of Beta Tests within a specified time period for comparison. Taking the Daemon A as an example, Figure 6.2 shows the reset situation of the daemonA during a certain firmware version's Beta Test period. The image displays the data grouped by Reset Reason/Code and stacked mode.

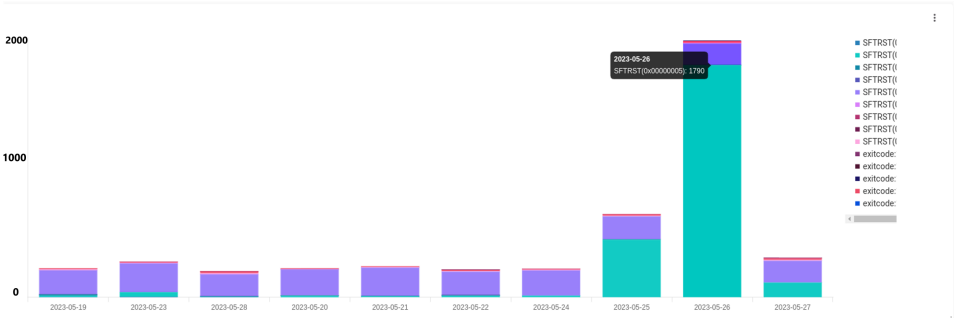


Figure 6.2: **Beta Testers: All daemonA reset reasons (Grouped by ResetCode)**

Figure 6.3 shows the results obtained by querying the Virtual Bridges data using the same method. In May 25th and May 26th (shown in Figure 6.2), which

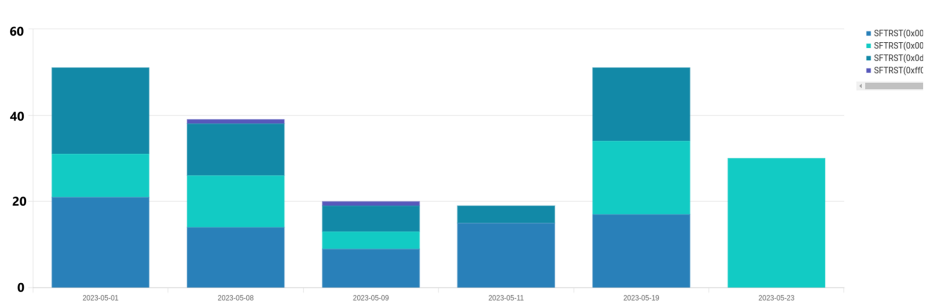


Figure 6.3: **Virtual Bridges: All daemonA reset reasons (Grouped by ResetCode)**

are the Beta Deploy days in the firmware cycle, there were a large number of daemonA resets caused by firmware updates (green corresponding to Code) and bridge restarts (purple corresponding to Code). However, during the rest of the time, excluding the Deploy days, the daily average number of daemonA resets for Beta Testers (about 6000) is 210, of which more than 90% are caused by restarts. As for virtual bridges, there were no resets or interruptions due to container restarts or other reasons during the experimental period. Therefore, we did not collect any daemonA reset messages on non-Deploy days. Figure 6.3 records all instances of daemonA occurrences throughout May during our experimental period. After verification, we found that all these dates correspond to the dates when we switched firmware versions, which are the deploy days in the Beta Test. Furthermore, there were no instances of any Daemon Resets during the remaining time, indicating that daemonA operated stably and normally. This is consistent with the analysis results of this version regarding daemonA.

Next, we selected MQTT and WebSocket protocols as the second set of control experiment objects.The MQTT and Websocket protocols are the core protocols for the communication between Hue Bridge and Hue Cloud/Hue App. The concetivity of communication directly affects the user experience and the stability of cloud data forwarding. Therefore, we choose to demonstrate the comparative experiment of these two protocols.We conducted a comparative analysis of the MQTT and WebSocket communication conditions between virtual bridges and beta testers under the same firmware conditions. The visualization of this data was used to analyze whether firmware updates would result in frequent disconnections in the Hue Bridge's MQTT/WebSocket communication. In the commercial communication field, the communication connection rate (statistical average of large-scale IoT devices) for these protocols typically focuses on the range of 95% to 100%, with particular emphasis on achieving incremental improvements in the 99% to 100% range every 0.1%. Figure 6.4 shows the WebSocket percentage connected during the experimental period, with different colors representing the WebSocket communication performance of beta testers using different firmware versions. It can be observed that different versions have some impact on communication stability within a range of 1%.

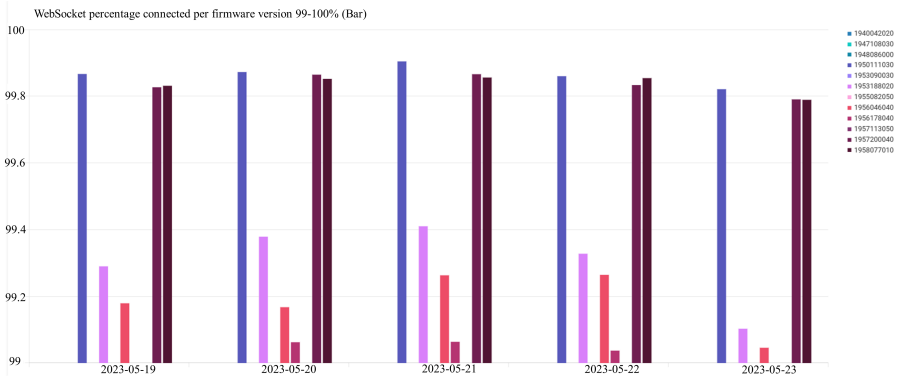


Figure 6.4: **Beta Testers: WebSocket Percentage connected per software version**

We selected two firmware versions, as shown in Figure 6.5, and deployed these firmware versions on virtual bridges. Firmware 1 (1956046020/1956046040) was deployed on virtual bridges from 2023-05-20 to 2023-05-24, and firmware 2 (1957200040) was deployed from 2023-05-24 to 2023-05-28. The experimental results are shown in Figure 6.6. The green bars correspond to Firmware Version 1, and the percentage is noticeably lower by approximately 0.3-0.4% compared to the blue bars, which is consistent with the difference observed during beta testing. From a statistical perspective, the mean percentage of WebSocket connectivity in beta testing was 99.1995% (Version 1)/99.8025% (Version 2), while the mean percentage for virtual bridges was 99.1918% (Version 1)/99.7674% (Version 2). Version 1 had a difference of 0.0077%, while Version 2 had a difference of 0.0351%. After verifying and cross-checking the data with validation engineers, we can conclude that this experimental result provides verification for WebSocket connectivity as a single factor.

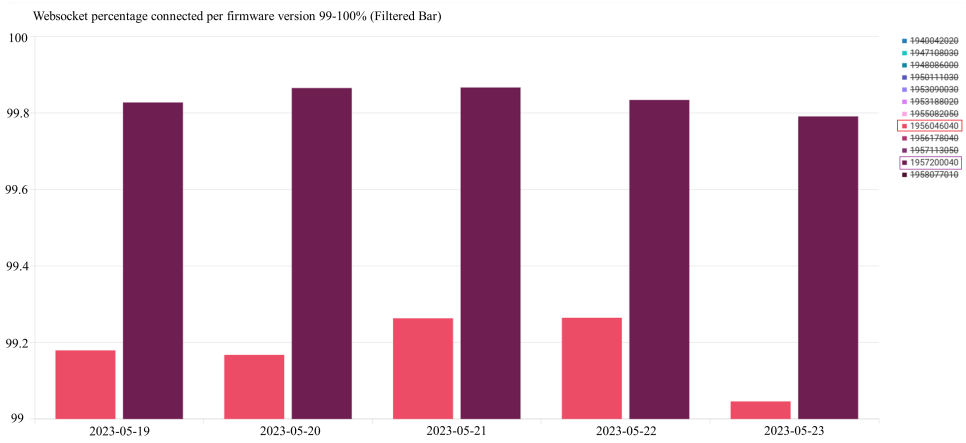


Figure 6.5: **Beta Testers: WebSocket Percentage connected per software version (filtered)**

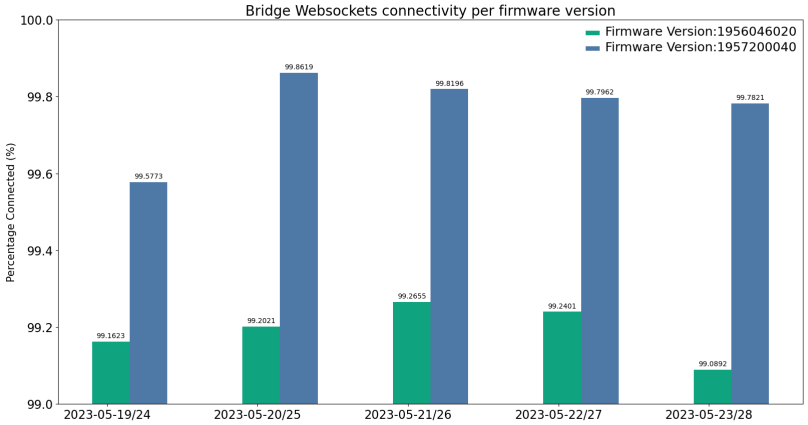


Figure 6.6: **Virtual Bridges: WebSocket Percentage connected per software version**

For MQTT Connectivity, the diagnostic data includes the Hue Bridge MQTT disconnected time and connected time. Therefore, when verifying MQTT Connectivity, the MQTT average connection time per day percentage visualization graph is obtained based on these two times. Figure 6.7 demonstrates the MQTT status during the Beta Test phase, grouped according to firmware. It can be observed that the percentage is distributed in the range of 98% to 100%. With the exception of a few firmware versions, the majority are within 99.5%.

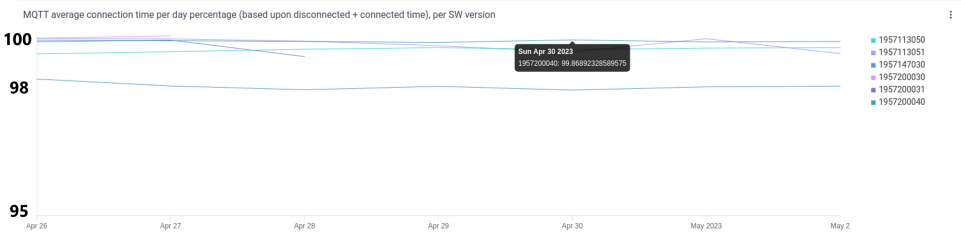


Figure 6.7: **Beta Testers: MQTT average connection time per day percentage(per SW version**

We deployed two firmware versions to the Virtual Bridges to validate WebSocket Connectivity, and we deloyed these two version for MQTT Connectivity validation as well. The MQTT average connection time percentage for these two versions is shown in Figure 6.8. The average percentage for Version 1 is 99.9124%, and for Version 2 it is 99.9243%. Both percentages are within 99.9% and consistent with the data from the Beta Testers. After verification and cross-checking of the data with the validation engineer, we can conclude that the experimental results validate the MQTT Connectivity in this aspect.

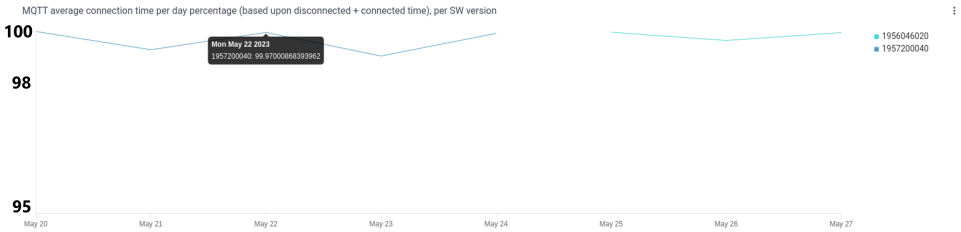


Figure 6.8: **Virtual Bridges: MQTT average connection time per day percentage**

In this experiment, we conducted a comparative validation of the Reset_reasons and Mqtt/Websocket Connectivity in the diagnostic data of the Hue Bridge. Our experimental results indicate that when we deploy firmware to the virtual bridge, the user diagnostics obtained have the same results as the Beta Testers.

Validation Experiment2: *Can we identify the existing bugs and some abnormal situations in the current firmware after deploying a specific version of the firmware to the virtual bridge?*

In Experiment 1 of this section, we conducted a comparative analysis of the visualization results of partial diagnostic data between Virtual bridges and Beta Testers. These results provide evidence of the reference value and significance of the diagnostic data from Virtual bridges. However, most of the visualizations compared in Experiment 1 were in a stable state, and no issues were discovered from them (as the selected firmware versions were relatively new and bug-free). Now, if we deploy a firmware version that has already been found to have bugs, can Virtual bridges also detect this bug? To answer this question, we carefully screened the DevOps Issues repository and ultimately selected the bug discovered in the open-source Daemon FluentBit in Firmware Version (1949107040).

- **Bug Name:** A big increase in FluentBit reset Reasons (Exit Code 139, which means segmentation fault)
- **Caused Reason:** FluentBit software update
- **Firmware Version:** 1949107040
- **Fixed Version:** 1949203000
- **Priority:** Major
- **Related Daemon:** Fluent-Bit Daemon
- **Description:** In beta there is a clear big increase in fluentbit reset reasons. The exitcode 139 was linked to the software update which causes a restart of the bridge. On average there are around 4 exitcodes 139 per bridges.

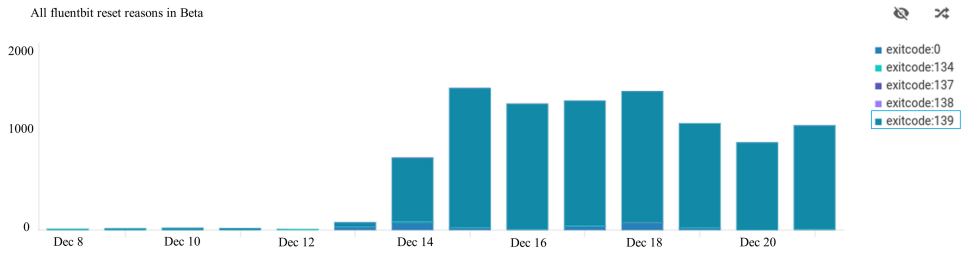


Figure 6.9: **Beta Testers: Number of total Fluent-Bit Reset Resaons per day (group by reset reasons)**

Figure 6.10 presents the visualization of FluentBit reset reasons during the Beta testing period for Firmware Version (1949107040). On Dec-14th, the Beta Tester Bridges deployed this version of the firmware, and it can be observed that there was a sudden increase in the number of exit code: 139 (represented by the blue bars in the graph) in what was originally a stable Reset Reason. Normally, during the Beta, FluentBit resets occur in the tens, so encountering more than 1000 resets during Beta testing is considered highly abnormal.

Following the deployment process, we proceeded with the virtual bridge testing. We deployed this bugged Firmware Version to the Virtual Bridges and monitored the reset status of the FluentBit Daemon through a cloud-based database. We discovered that after deploying this firmware to the Virtual Bridges, upon reading the real-time log inside the Virtual Bridge Container, we observed that FluentBit encountered an abnormal exit with an exit reason of Code 139. This exit reason aligns with the bug identified during the Beta testing phase. Similarly, to verify if the remaining Virtual Bridges would also encounter such situations, we visualized the Reset Reason subtable from the cloud-based database, specifically filtering for reset messages from the FluentBit daemon. The result is shown in Figure 6.10:

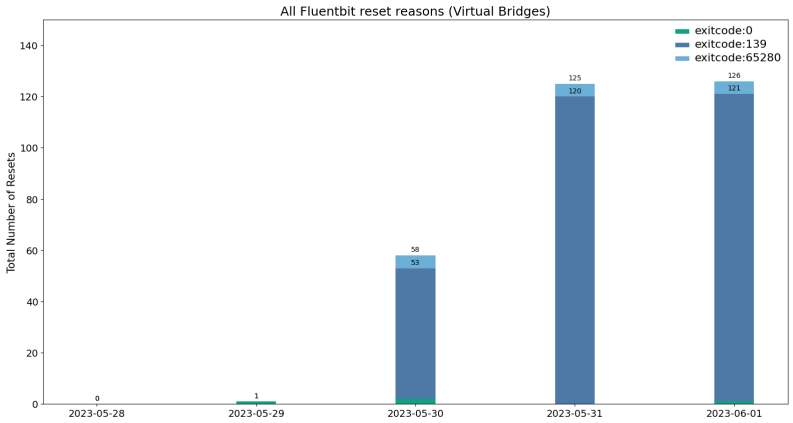


Figure 6.10: **Virtual Bridges: Number of total Fluent-Bit Reset Reasons per day (group by reset reasons)**

Our experimental results demonstrate that when using Virtual Bridges to perform the tasks conducted by Beta Testers during the Beta Test, we consistently discovered the same Automated Feedback (Bugs/Issues). This outcome strongly confirms the value of Virtual Bridges in obtaining Automated Feedback.

In addition to the previous FluentBit bug, we also conducted tests on other types of bugs. However, due to privacy concerns, the detailed presentation of these tests is not included in the paper. To further support our conclusions, we made new attempts in this experiment. While the previous bug was based on known issues in historical firmware versions, we explored the possibility of detecting a manually injected bug in the diagnostic data visualization charts by deploying a bug-free firmware version to the Virtual Bridges.

To validate our hypothesis, we designed the following bug:

- **Bug Name:** Increase in WebSocket nr of Disconnect in %
- **Caused Reason:** Manually Cut the WebSocket Connection
- **Firmware Version:** 1957200040
- **Related Daemon:** WebSocketd Daemon
- **Description:** We send Kubectl Commands to all the virtual bridges to manually cut the WebSocket Connection

The Validation Engineer keeps track of the daily WebSocket reconnection count of Bridge devices, classified into four categories based on the count. Under normal circumstances, during Beta testing, over 95% of Bridges reconnect only once per day. If there is a widespread occurrence of two or more reconnections, it indicates frequent disconnections of WebSocket.

Based on this pattern, we introduced a bug on May 22nd and May 23rd. On May 22nd, we selected five out of thirty Virtual Bridges to inject this bug, while on May 23rd, we forcefully restarted the WebSocket daemon for all thirty Virtual Bridges to inject this bug. The results are shown in Figure 6.11, where we can observe the following: prior to injecting the bug, the WebSocket disconnection count remained consistently at one (indicated by the green color), whereas after injecting the bug, the reconnection percentage on May 22nd reached nearly 50%, and on May 23rd, it exceeded 100% with reconnections greater than two times. After verifying and cross-checking the data with the Validation Engineer, we can conclude that this experimental result provides validation specifically on the WebSocket number of disconnects.

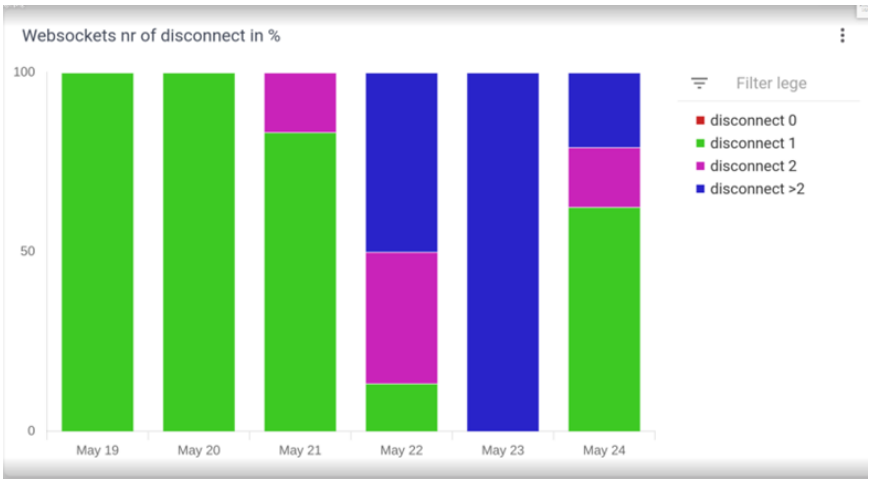


Figure 6.11: **Virtual Bridges: Number of WebSockets disconnection in % per day (group by disconnect times)**

Validation Experiment 3: *After applying the Use Case Model to Our Kubernetes Deployment, can the virtual bridges obtain diagnostic data more quickly?*

We followed the experimental design (time and method) of Experiment 1 and narrowed down the experimental dataset to the "raw" sub-table of the Diagnostic Dataset. The "raw" sub-table records the raw diagnostic data sent to the cloud (the cloud further splits the "raw" sub-table into other sub-tables based on the data type). The purpose of our experiment was to observe and compare the total amount of diagnostic data. Therefore, we first needed to preprocess the data for the experimental subjects. The "raw" sub-table mainly consists of fields such as message_type, sw_version, bridge_id, message, and date. The message_type corresponds to different sub-table names, and the total count of messages represents the total amount of diagnostic data for the current bridge after deploying a specific version of the firmware (counted by date). Therefore, our query logic for data retrieval is to limit the start and end times and the firmware version, and then use the COUNT keyword to calculate the total count of messages. We separately counted the total number of messages collected from the Virtual Bridges (without the Use Case Model), the Virtual Bridges (with the Use Case Model applied), and the Beta Testers' Bridges within the specified time period. The comparative experimental results are shown in Table 6.5.

Table 6.5: **Table: Average number of Diagnostic Messages per bridge**

Experiment Object	**Beta Testers' Bridges**	**Virtual Bridges with model**
Avg number of Messages per bridge	170 messages per day	305 messages per day

In Table 6.5, the total number of diagnostic data messages per bridge collected during the experimental period is presented. With the application of a use case model, there is a significant increase in diagnostic messages. Compared to beta

testers, there is a 79.4% increase. Based on the experimental results, we find that the application of a use case model leads to a significant improvement in diagnostic data. It also shows an improvement in mean values compared to the data generated by beta testers.

Therefore, we approached the engineer responsible for diagnostic data and consulted with them regarding the generation and forwarding mechanism of diagnostic data. They informed us that diagnostic data mainly consists of event-driven and periodic diagnostic types, with a ratio of 4:6. Event-driven diagnostic data is generated immediately upon the occurrence of predefined events (e.g., daemon error/exit/resource create). On the other hand, periodic diagnostic data involves the Hue Bridge sending analysis reports at regular intervals, typically once every n hours. For example, the MQTT connection stability report is sent every eight hours. Even if we apply a use case model, the amount of periodic diagnostic data will not increase. This explains why the previous experimental results did not show a significant increase in the total count.

To facilitate a clearer comparison, we categorized and reanalyzed the experimental data. In this analysis, we focused only on event-driven diagnostic data and selected five sub-tables for horizontal comparison.

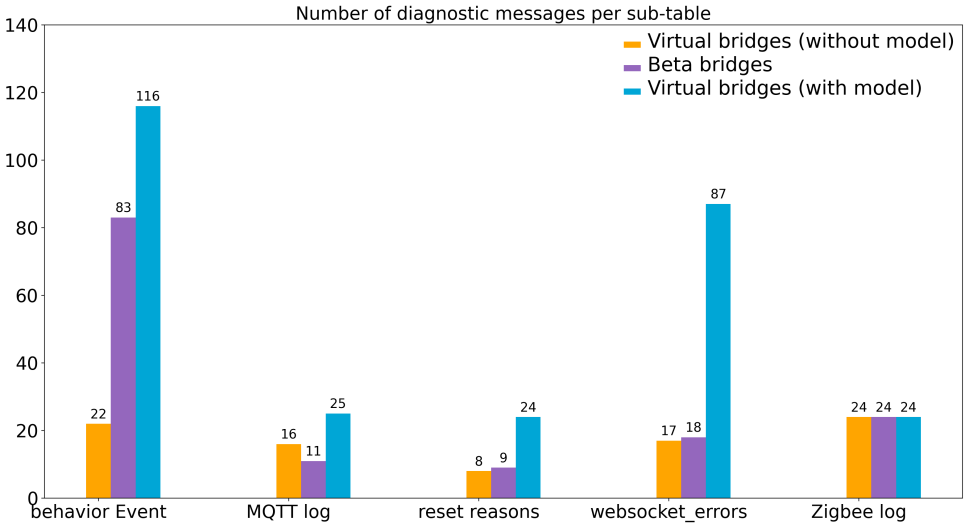


Figure 6.12: **Number of diagnostic messages per sub-table per bridge**

Figure 6.12 displays the daily number of diagnostic messages for five sub-tables. It is evident that, except for the fifth sub-table, all other tables exhibit significant improvements after applying the Use Case Model. The Bridgezigbeelog sub-table represents periodic diagnostic data, where the Hue bridge records 24 messages per day at hourly intervals, regardless of the application of the Use Case Model. This indirectly validates our previous hypothesis that the diagnostic type may not yield noticeable improvements.

According to data analysis, we found significant differences in the performance of Diagnostic messages across different sub-tables. Some sub-tables are strongly correlated with user behavior (Use Case Model), such as behaviordaemon and wserrors shown in Figure 6.12. On the other hand, some sub-tables show only

slight changes after applying the Use Case Model, and these sub-tables account for a small proportion of the total. Experimental results indicate that there are significant differences in the improvement magnitude among different sub-tables. Therefore, we do not present the improvement magnitude table for all sub-tables. However, we can observe that Virtual Bridges can be used to enhance event-driven diagnostic data, and the periodicity of regular diagnostic data can be shortened based on testing requirements to expedite the generation of diagnostic data. Compared to Beta Bridges, Virtual Bridges provide higher fault tolerance.

Based on the results of this experiment, we can draw the following conclusions: Virtual Bridges can obtain diagnostic data more quickly compared to Beta Tester, and the improvement magnitude varies significantly depending on the sub-table category, with a total improvement around 79.4%. The differences among different firmware versions are within 5%.

Validation Experiment 4: *What is the coverage of KPIs and user behavior when deploying Bridge firmware to a virtual bridge?*

In our previous experiments, we compared the visual diagnostic data following the process of validation engineers and verified the feasibility of Virtual Bridges in automated feedback discovery. In this experiment, we will complete the final step of the validation process: KPI analysis. KPI stands for Key Performance Indicator and serves as the validation standard for the Hue Bridge firmware established by various research and development teams. These KPIs are divided based on different teams responsible for specific daemon development. During the final stage of the Beta Test, validation engineers create a Beta Report based on the diagnostic data. In the Release Meeting, the engineers determine whether to release the firmware based on the performance of the KPIs mentioned in the report. This experiment aims to explore the coverage of Virtual Bridges in KPIs and discover key indicators for comparison.

We cloned all the Dashboards from the Hue SQL tool used by validation engineers to analyze diagnostic data. These Dashboards belong to different development teams and are used to analyze whether different firmware versions meet the KPIs. By transforming the cloned Dashboards to include only Virtual Bridges (with limited Bridge IDs), we obtained the Beta Test Dashboards for Virtual Bridges. We converted and organized these Dashboards one by one and collected statistics on the coverage of Virtual Bridges in different types of KPIs, as shown in Table 6.6.

Table 6.6: **Virtual Bridges: KPIs Analysis Coverage**

	Beta Tester Bridges	**Virtual Bridges**
Number of Hue Bridge	∼6k	30
Reset Reason KPIs	17	15
Connectivity KPIs	9	7
Diagnostic Messages KPIs	10	8
Automation KPIs	11	6
Other KPIs	6	1
Total KPIs	53	37
Total KPIs (Except Hardware related)	46	37

In Table 6.6, we can see that Virtual Bridges have a certain coverage in various types of KPI analysis. Most KPIs related to hardware analysis of the Hue Bridge, such as memory usage and electronic component temperature, cannot be covered by Virtual Bridges. These are beyond the scope that a virtual Bridge can cover. The limitation of virtualized devices lies in their inability to perform diagnostic analysis on these aspects of real IoT devices. Excluding hardware factors, the overall coverage of Virtual Bridges in KPIs can reach 80.43% (37/46). This coverage meets our expectations for the project results and has been affirmed by validation engineers.

Chapter 7

Conclusions

In this paper, we propose a virtualization solution based on containerization and Kubernetes to address the firmware update cycle efficiency issues of Hue Bridge devices. The slow firmware updates hinder Signify's ability to iterate firmware versions quickly and timely discover and validate automated feedback during the Alpha and Beta testing phases. Our goal is to use the virtualized Hue System to perform tasks assigned to beta testers and thereby reduce the time required. The existing virtualization solutions in Signify have significant deficiencies in terms of system functionality and scalability. Moreover, the inability to collect user diagnostic data poses a challenge for validating and analyzing test results in the current approach.

To address the issues in existing projects, we present the following research questions for this study:

- **Question I:** Based on the existing projects, how can we improve the virtualization of Hue Bridge devices?
- **Question II:** Can the virtual bridge improve firmware update efficiency and provide valuable automated feedback?

Regarding Question I, we first establish the transmission of analysis/diagnostic data between the virtual bridge and Hue Cloud. This diagnostic data is crucial for bridge engineers to verify and test their work, making the collection of analysis/diagnostic data from the virtual bridge in the cloud a foundation for project validation. Additionally, to enhance the scalability and convenience of the existing virtualization projects, we design a Kubernetes deployment solution for the virtual Hue system based on Kubernetes container management technology. With our deployment solution, developers can quickly deploy thousands of virtual Hue systems according to their needs, with the entire deployment process taking only 5 minutes.

Furthermore, we propose a Use Case Model to analyze the everyday behavior of real Signify users when using the Hue System, such as light control and automation scenes. Based on the analysis and modeling of the Delta_message database, we not only analyze the daily interaction patterns between users and the Hue System but also understand user preferences for advanced light control. We convert the generated Use Case list from the Use Case Model into machine commands (Hue Commands) recognizable by the Bridge Daemon. We design a

random command generator based on the Use Case Model, which can send Hue Commands to all deployed virtual bridges at a specified frequency. According to the verification by bridge experts, the improvements made to the virtual Hue System and Dockerized Bridge significantly enhance the efficiency of automated testing and partially simulate real user behavior.

For Question II, regarding the feasibility of virtual bridges in improving firmware update efficiency and providing valuable automated feedback, we gradually validate the following verification questions:

- Will virtual bridges generate similar analysis data (Delta Messages) when the Use Case Model is applied to Kubernetes Deployment?
- Will applying the Frequent Pattern Mining algorithm to Delta Messages generated by virtual bridges result in the same Use case?
- Does the user diagnostic result obtained after deploying a specific firmware version to the virtual bridge resemble that of the Beta Testers?
- Can we identify the existing bugs and abnormal situations in the current firmware after deploying a specific version of the firmware to the virtual bridge?
- After applying the Use Case Model to our Kubernetes Deployment, can the virtual bridges obtain diagnostic data more quickly?
- What is the coverage of Key Performance Indicators (KPIs) and user behavior when deploying Bridge firmware to a virtual bridge?

Based on the experimental results in Section 6, we can conclude that the analysis data collected from virtual bridges aligns with our assumptions about the Use Case Model. Through visualizing the diagnostic data generated by the virtual bridges, we obtained diagnostic data analysis results specific to the virtual bridges. By comparing the results, we found consistent patterns and outcomes between the virtual bridges and beta testers.

Additionally, according to our results, using our virtual Hue system allows us to collect automated feedback present in buggy firmware versions. This finding suggests that engineers can use our virtual Hue system to identify or reproduce issues in the firmware earlier. The earlier issues are identified and addressed, the faster

Chapter 8

Future Work

In this thesis, the implemented system design demonstrates feasibility in the Firmware Update Cycle. However, there are aspects where the system could be further improved and validated. We outline several remaining works that need to be addressed:

1. **Resource limitations of virtual bridges:** During the validation phase, due to resource constraints in the cloud, we were only able to deploy 30 virtual bridges for validation experiments. However, for Beta Testing, the automated feedback generated by thousands of test bridges is essential for comprehensive testing. Therefore, we propose resolving the resource limitations by collecting diagnostic data from thousands of virtual bridges in the cloud.

2. **Enhancement of the Use Case Model:** We employed Frequent Pattern Mining to analyze and extract use cases from user data. However, our model is based on a short period of user data. To obtain a more comprehensive user behavior model, it is necessary to have a more complete dataset covering a longer duration, ideally spanning a year.

3. **Limitations of the random command generator:** During the validation of the Use Case Model, we found that the use case coverage generated by virtual bridges was around 80%. This is mainly due to the inability of the HSE to simulate certain sensors. Therefore, improving the simulation of sensors (such as third-party sensors or virtual clip sensors) in the HSE can enhance the coverage of virtual bridges in terms of user behavior. This would allow us to cover more scenarios of users using the Hue System and maximize the likelihood of detecting firmware issues.

4. **Evaluating resource utilization:** Our project did not analyze and quantify the software and hardware resources consumed by the large-scale deployment of virtual Hue Systems. Additionally, we propose conducting a comparative analysis and cost evaluation between Beta Testers and Virtual Bridges in terms of time and monetary costs. This analysis will provide insights into the cost savings for Signify after adopting our virtual system.

Bibliography

[1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.

[2] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499. Santiago, Chile, 1994.

[3] Sunha Ahn and Sharad Malik. Automated firmware testing using firmware-hardware interaction patterns. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–10, 2014.

[4] Sergey Brin, Rajeev Motwani, Jeffrey D Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 255–264, 1997.

[5] Antonio Celesti, Davide Mulfari, Maria Fazio, Massimo Villari, and Antonio Puliafito. Exploring container virtualization in iot clouds. In *2016 IEEE international conference on Smart Computing (SMARTCOMP)*, pages 1–6. IEEE, 2016.

[6] Hong-Yi Chang, Jia-Chi Lin, Mei-Li Cheng, and Shih-Chang Huang. A novel incremental data mining algorithm based on fp-growth for big data. In *2016 International Conference on Networking and Network Applications (NaNA)*, pages 375–378. IEEE, 2016.

[7] Docker. Use containers to build, share and run your applications. `https://www.docker.com/resources/what-container/`, 2023. Last accessed: May. 15, 2023.

[8] Docker docs. About storage drivers. `https://docs.docker.com/storage/storagedriver/`, 2023. Last accessed: May. 14, 2023.

[9] Fluentbit. An end to end observability pipeline. `https://fluentbit.io/`, 2023. Last accessed: May. 15, 2023.

[10] Kanwal Garg and Deepak Kumar. Comparing the performance of frequent pattern mining algorithms. *International Journal of Computer Applications*, 69(25), 2013.

[11] Jim Grundy. Firmware validation: challenges and opportunities. In *FM-CAD*, page 11, 2013.

[12] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2):1–12, 2000.

[13] Jeff Heaton. Comparing dataset characteristics that favor the apriori, eclat or fp-growth frequent itemset mining algorithms. In *SoutheastCon 2016*, pages 1–7. IEEE, 2016.

[14] José L Hernández-Ramos, Gianmarco Baldini, Sara N Matheu, and Antonio Skarmeta. Updating iot devices: challenges and potential approaches. In *2020 Global Internet of Things Summit (GIoTS)*, pages 1–5. IEEE, 2020.

[15] Eric Hitimana, Gaurav Bajpai, Richard Musabe, Louis Sibomana, and Kayavizhi Jayavel. Containerized architecture performance analysis for iot framework based on enhanced fire prevention case study: Rwanda. *Sensors*, 22(17):6462, 2022.

[16] Kang Huaishuai. Docker practice: kubernetes. `https://yeasy.gitbook.io/docker_practice/kubernetes`, 2020. Last accessed: May. 10, 2023.

[17] Kostas Kolomvatsos. An intelligent, uncertainty driven management scheme for software updates in pervasive iot applications. *Future generation computer systems*, 83:116–131, 2018.

[18] Miguel A. López-Peña, Jessica Díaz, Jorge E. Pérez, and Héctor Humanes. Devops for iot systems: Fast and continuous monitoring feedback of system availability. *IEEE Internet of Things Journa*, 7:10695–10707, 2020.

[19] Lucille McMinn and Jonathan Butts. A firmware verification tool for programmable logic controllers. In *Critical Infrastructure Protection VI: 6th IFIP WG 11.10 International Conference, ICCIP 2012, Washington, DC, USA, March 19-21, 2012, Revised Selected Papers 6*, pages 59–69. Springer, 2012.

[20] Igor Muzetti Pereira, Tiago Carneiro, and Eduardo Figueiredo. A systematic review on the use of devops in internet of things software systems. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1569–1571, 2021.

[21] Igor Muzetti Pereira, Tiago Garcia de Senna Carneiro, and Eduardo Figueiredo. Understanding the context of iot software systems in devops. In *2021 IEEE/ACM 3rd International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*, pages 13–20. IEEE, 2021.

[22] Kai Petersen, Claes Wohlin, and Dejan Baca. The waterfall model in large-scale development. In *Product-Focused Software Process Improvement: 10th International Conference, PROFES 2009, Oulu, Finland, June 15-17, 2009. Proceedings 10*, pages 386–400. Springer, 2009.

[23] Pranshu Gupta. On virtualization and containers. `https://medium.com/@pranshugupta_61104/on-virtualization-and-containers-6f9683fea1c1`, 2023. Last accessed: Mar. 18, 2023.

[24] Sebastian Raschka. Mlxtend: Providing machine learning and data science utilities and extensions to python's scientific computing stack. *The Journal of Open Source Software*, 3(24), 2018.

[25] Ruth Hadari. How is pi planning different from sprint planning? `https://www.goretro.ai/post/pi-planning/`, 2023. Last accessed: May. 15, 2023.

[26] Collins Sey, Hang Lei, Weizhong Qian, Xiaoyu Li, Linda Delali Fiasam, Ruchao Sha, and Zirui He. Firmblock: A scalable blockchain-based malware-proof firmware update architecture with revocation for iot devices. In *2021 18th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, pages 134–140. IEEE, 2021.

[27] Signify. Become a philips hue developer to light up your ideas. `https://developers.meethue.com`, 2023. Last accessed: May. 14, 2023.

[28] Signify NV. Meet the hue. `https://www.philips-hue.com/nl-nl`, 2023. Last accessed: Mar. 1, 2023.

[29] Alanoud Subahi and George Theodorakopoulos. Detecting iot user behavior and sensitive information in encrypted iot-app traffic. *Sensors*, 19(21), 2019.

[30] Synopsys. Why is ci/cd important? `https://www.synopsys.com/glossary/what-is-cicd.html#:~:text=Definition,and%20continuous%20delivery%2Fcontinuous%20deployment.`, 2023. Last accessed: May. 18, 2023.

[31] Maryam Tavakkoli. Analyzing the applicability of kubernetes for the deployment of an iot publish/subscribe system. Master thesis, Aalto University, 2019.

[32] TechTarget. Comparing waterfall vs. agile vs. devops methodologies. `https://www.techtarget.com/searchsoftwarequality/opinion/DevOps-vs-waterfall-Can-they-coexist`, 2023. Last accessed: Mar. 18, 2023.

[33] Chun-Wei Tsai, Chin-Feng Lai, Ming-Chao Chiang, and Laurence T Yang. Data mining for internet of things: A survey. *IEEE Communications Surveys & Tutorials*, 16(1):77–97, 2013.

[34] Woei-Jiunn Tsaur, Jen-Chun Chang, and Chin-Ling Chen. A highly secure iot firmware update mechanism using blockchain. *Sensors*, 22(2):530, 2022.

[35] University of Regina. Support measure. `http://www2.cs.uregina.ca/~dbd/cs831/notes/itemsets/support.html`, 2012. Last accessed: May. 10, 2023.

[36] Wikipedia. Kubernetes. `https://en.wikipedia.org/wiki/Kubernetes`, 2014. Last accessed: May. 14, 2023.