Politecnico di Torino

Master's Degree in Communications and Computer Networks Engineering

Master's Degree Thesis

KRules: Empowering IoT Platforms with Advanced Automation and Rule-Based Intelligence

Supervisors:

Prof. Matekovits Ladislau (DET)
Prof. Allegretti Marco (Docente esterno)

Candidate:

Fattaneh Pasand Hafshejani

June 2023

Acknowledgment

I would like to express my deepest gratitude to my professor, Matekovits Ladislau, for his invaluable guidance and support throughout my thesis journey. His introduction to the CSP company played a pivotal role in shaping the direction of my research and provided me with a remarkable opportunity to explore the industry firsthand.

I am immensely thankful to my colleague, Roberto Politi, whose unwavering support and collaboration made this thesis possible. His expertise and insights have been instrumental in refining my ideas and enhancing the overall quality of my work. I would also like to extend my gratitude to the entire team at Airspot company, whose partnership and assistance proved to be invaluable throughout the project.

I am indebted to Alberto Degli Esposti and Lorenzo Campo for their significant contributions and unwavering support. Their dedication and expertise played a vital role in helping me navigate the complexities of working with their product. Their guidance and insights were truly invaluable and greatly enhanced the outcome of this thesis.

I would like to express my heartfelt thanks to my love, Saman, for his unwavering support and encouragement throughout this challenging endeavor. Your constant belief in my abilities and your love has been a constant source of motivation. Your presence and understanding made the journey all the more rewarding.

Finally, I would like to express my heartfelt gratitude to my beloved family. To my father, mother, two brothers, and my lovely sister, even though we may have been physically apart, your unwavering love and unwavering support have always resided within my heart. It is through your emotional encouragement that I have been able to embark on and successfully complete this incredible journey. Thank you for being there for me every step of the way.

In conclusion, I am deeply grateful to everyone who has contributed to the successful completion of this thesis. Your support, guidance, and encouragement have been instrumental in shaping my academic and personal growth. Thank you all for your invaluable assistance.

Objective

The objective of my thesis is to enhance the existing IoT platform, specifically the ThingsBoard IoT platform, by automating certain actions and offering advanced services. The IoT segment is rapidly expanding due to the widespread use of IoT devices, which have impacted various fields including healthcare, the industrial sector, home automation, environmental monitoring, and retail. The capture of data by IoT devices allows for separate analytics based on the collected data, making life easier across different sectors. However, my focus is on improving the ThingsBoard platform by incorporating a subset of rulesets in the form of Python data structures. This will be achieved by implementing KRule, which is a useful tool for automating actions and improving the functionality of the platform.

Abbreviations

Abbreviation	full name
IOT	Internet Of Things
CRI	Container Runtime Interface
IaC	Infrastructure as Code
GUI	Graphical User Interface
CI/CD	continuous Integration/Continuous Delivery
GCP	Google Cloud Platform
EDA	Event Driven Architecture
SDK	Software Development Kit
CRDs	Kubernetes Custom Resources Definitions
GPIO	General Purpose Input Output

Contents

1 Introduction **3**

2 IOT Platform **6**
2.1 IOT . . . 6
2.1.1 IOT network . . . 6
2.1.2 IOT platform . . . 7
2.2 LoRa . . . 8
2.2.1 LoRa sensors . . . 8
2.2.2 LoRa gateway . . . 8
2.2.3 LoRa server(ChirpStack) . . . 8
2.3 Thingsboard . . . 9

3 Google Cloud Platform (GCP) **11**
3.1 Containerization . . . 11
3.1.1 what is the difference between containers and virtualization? . 12
3.2 Kubernetes . . . 13
3.2.1 Kubernetes Architecture . . . 13
3.2.2 Events in Kubernetes . . . 15
3.3 GCP Services . . . 16
3.3.1 Terraform . . . 16
3.3.2 Pub/Sub . . . 17
3.3.3 Eventarc . . . 17
3.3.4 Kafka broker . . . 17
3.3.5 CI/CD (continuous integration/continuous delivery) . . . 17
3.3.6 Google Artifact Registry . . . 18
3.3.7 Cloud Logging API . . . 19
3.3.8 Firebase Cloud Messaging (FCM) . . . 19
3.3.9 Firestore . . . 20

4 Knative-eventing **22**
4.1 Introduction . . . 22
4.2 Event-Driven Architecture . . . 23
4.3 CloudEvents . . . 25
4.4 Knative . . . 26
4.4.1 Knative Serving . . . 26
4.4.2 Knative Eventing . . . 28
4.5 Event Mesh . . . 29
4.5.1 Knative Event Mesh . . . 30
4.6 Knative Broker for Apache Kafka . . . 31

5 KRules **33**
5.1 Introduction . . . 33
5.2 KRules . . . 34
5.3 Subject . . . 35
5.4 Rules . . . 37
5.5 Filters . . . 38
5.6 Processing . . . 39

6 Project Architecture and results **41**
6.1 LoRa server and Thingsboard implementation . . . 41
6.2 GCP implementation . . . 43

7 Conclusions and future works **49**
7.1 Future Works and Implementations . . . 50

List of Figures

1.1 Whole Project overview. 4

2.1 Thingsboard rule engine. 9

3.1 Difference btw containerization and virtualization [1] 12
3.2 Kubernetes Architecture [2] 15
3.3 GCP Architecture Of Project 16

4.1 "Event-Driven Architecture" 24
4.2 "Knative serving" 27

5.1 35
5.2 36
5.3 CloudEvents format 36

6.1 Thingsboard integration 41
6.2 publish lorasource-dev events to pub/sub 42
6.3 our project directory 43
6.4 Redis configuration 44
6.5 pub/sub Terraform configuration 45
6.6 Eventarc configuration [3] 46
6.7 Rulesets developments 47

Chapter 1

Introduction

In conclusion, the significance of an IoT platform becomes evident in situations where tracking the status of an environment manually is impractical, time-consuming, hazardous, and energy-intensive for humans. The need to rely solely on human intervention for monitoring purposes becomes increasingly challenging when faced with complex and large-scale environments. However, mere tracking of sensor data is often insufficient in addressing critical issues and ensuring human safety. To address this limitation, there is a compelling motivation to automate actions based on real-time situations. Automation enables timely responses to problems, creating a secure environment for individuals. Furthermore, the demand for a generalized solution becomes apparent, as numerous companies utilize different IoT platforms that often lack comprehensive functionality beyond sensor status tracking and emergency notifications. Hence, the aim of this thesis is to introduce a rule-based logic solution that can be implemented across various IoT platforms, adding an automation layer to enhance their capabilities. This approach goes beyond mere information dissemination through messages, recognizing the need for a more robust and efficient solution to optimize the potential of IoT technologies in diverse applications.

During this thesis, as Figure 1.1 shows, we utilized LoRa sensors due to their advantageous capabilities, such as long-range communication and lower power consumption. These sensors effectively transmitted messages containing crucial environmental data, including temperature and humidity. The messages were then received by a LoRa server, which served as a centralized dashboard for device management and data monitoring. To ensure seamless integration and future identification of message sources, we leveraged Thingsboard, a powerful platform that enabled us to define metadata and data components of the messages. This platform facilitated the transmission of messages with specific topics to the Pub/Sub GCP services, ensuring that subscribers would receive only the relevant project-related messages.

As the publisher did not possess direct access to the Kubernetes cluster, we employed the Eventarc broker. This allowed us to utilize Knative eventing, enabling the creation of an event-driven architecture. The utilization of KRules, built on top of Knative eventing, perfectly aligned with our objective of implementing rule-based logics. With a concentrated focus on writing rule-based logic in Python, we achieved success in formulating four rules that continuously monitored the sensor status and effectively responded to temperature changes. These responses were exe-

cuted through HTTP post requests to smart devices, thereby triggering appropriate actions. Simultaneously, users were promptly notified through Firebase notifications, ensuring real-time awareness of critical environmental changes.

This thesis explores the utilization of Kubernetes and containerization to en- able efficient pod autoscaling in an IoT environment. By leveraging Kubernetes, we created a cluster of Worker Nodes responsible for running containerized applications, achieving optimal performance and resource utilization. We wrote each Rule in a separate pod. Additionally, the integration of Knative Eventing that provides a higher-level abstraction for deploying and running serverless applications. In Knative Eventing, the "broker" and "trigger" are key components that enable event-driven architectures and the routing of events between event producers and consumers. The introduction of KRules which is a higher-level application logic on top of the Knative -eventing. KRules implementation added a layer of automation to the IoT platform, allowing for the definition of multiple Rules and enhancing overall system efficiency. This research contributes to the development of a robust and scalable system that leverages cloud-native principles for improved performance and functionality in the IoT domain.

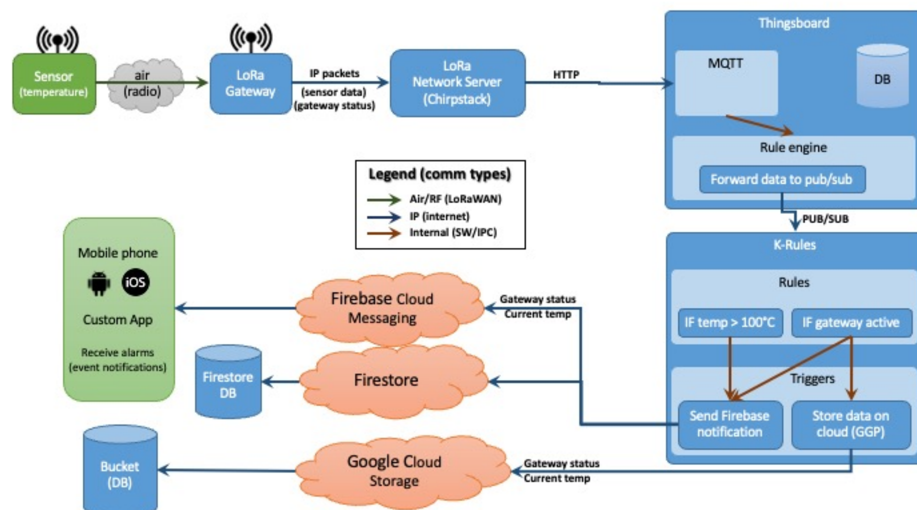

Figure 1.1: Whole Project overview.

Chapter 2

IOT Platform

2.1 IOT

IoT stands for the Internet of Things, a network infrastructure that uses standard communication protocols. It comprises billions of physical devices worldwide, all of which are interconnected through the internet to gather and exchange data. With the advent of affordable computer chips and the widespread accessibility of wireless networks, these devices can be connected and equipped with sensors, providing them with a level of digital intelligence. This allows them to communicate real-time data without requiring human intervention, resulting in a more intelligent and responsive world that merges the digital and physical universes.

2.1.1 IOT network

The IOT network has three layers:

- Sensors - These are devices that gather data and can either send the information to the internet or receive commands to perform an action.
- Gateways - These devices act as a connection between the sensors and the internet. They receive data from the sensors and transform it into a format that can be sent to the IoT platform. There are different types of gateways, depending on the communication technology used by the IoT devices. In this case, we are discussing LoRa sensors that use LoRaWAN protocol, which is a low-power, long-range wireless communication technology, enables IoT devices to establish connections over several kilometers.
- Internet - This is where the IoT platform is located and is the final destination for the data collected by the sensors. It is a software that acts as a hub for devices and allows multiple operations to be performed on them.

2.1.2 IOT platform

An IoT platform is a comprehensive system consisting of interconnected components that empower developers to design, develop, and deploy applications, securely gather and analyze data from connected devices, manage sensors and other devices, and establish reliable connectivity between devices and the platform. There are various types of IoT platforms available, such as TIG, Kibana, and Thingsboard. In this thesis, I will focus on Thingsboard and explore ways to enhance its capabilities.

2.2 LoRa

I think I must start explaining about LoRa with this question, why LoRa?

LoRa technology is particularly suitable for applications that require the transmission of small amounts of data at low bit rates. Compared to other wireless technologies such as WiFi, Bluetooth, or ZigBee, LoRa can transmit data at longer ranges with lower battery consumption. These characteristics make LoRa ideal for sensors and actuators that operate in low-power modes.

2.2.1 LoRa sensors

LoRa sensors are IOT devices because they can transmit data from a large number of sensors distributed over a wide geographical area. This capability makes LoRa well-suited for applications where data needs to be collected from multiple locations and transmitted to a central hub or server for analysis.

2.2.2 LoRa gateway

A LoRa gateway is a physical device that acts as a bridge between LoRa devices and the internet. Similar to a Wi-Fi router, it receives data from LoRa sensors and transmits it to a cloud-based application server through a wired or wireless network. Conversely, it receives commands from the server and sends them to LoRa devices. A LoRa gateway includes a LoRa concentrator, which enables it to receive RF signals sent out by LoRa devices. The signals are converted to a compatible format, such as Wi-Fi, to transmit data to the cloud-based server.

2.2.3 LoRa server(ChirpStack)

LoraServer offers a user-friendly web interface for effectively managing gateways and devices. Additionally, it facilitates the configuration of data integrations with popular cloud providers, databases, and services commonly employed for handling device data. With LoraServer, you can view activities and configure all devices and gateways present in LoRa networks. Additionally, you can create applications for decoding and encoding LoRa messages.

2.3 Thingsboard

Thingsboard is an open-source IoT platform that simplifies the process of building and deploying IoT solutions, while also solving common technical challenges. The platform can be hosted in the cloud or on-premises, giving you full control over your IoT technologies and the freedom to choose any cloud vendor.

Thingsboard supports both compact monolithic and robust fault-tolerant microservices setups, enabling seamless scalability from hundreds to millions of devices without the need for application redesign. The platform enables device connectivity by providing out-of-the-box HTTP, MQTT, and COAP protocols, and also allows you to use IoT gateway APIs to connect non-IP devices.

With Thingsboard, you can model complex physical world objects via assets, devices, and relations. This data model enables you to produce more valuable insights from your data faster and easier. The platform stores common telemetry in either SQL or NoSQL databases, ensuring scalable and fault-tolerant storage of data.

Thingsboard also enables bi-directional commands to be sent to devices, allowing for remote control. It provides the most comprehensive dashboard editor in the market, with dynamic actions that can be configured with zero coding efforts. Dashboards can be assigned to multiple customers, and each customer's users can see and control only their own devices and assets, without having access to other customer data.

The heart of Thingsboard is its rule engine, which allows you to filter, enrich, and transform system events and device telemetry, as well as trigger actions. For example, you can create alarms, execute REST- API calls, or push filtered data to external message queues for advanced analytics.

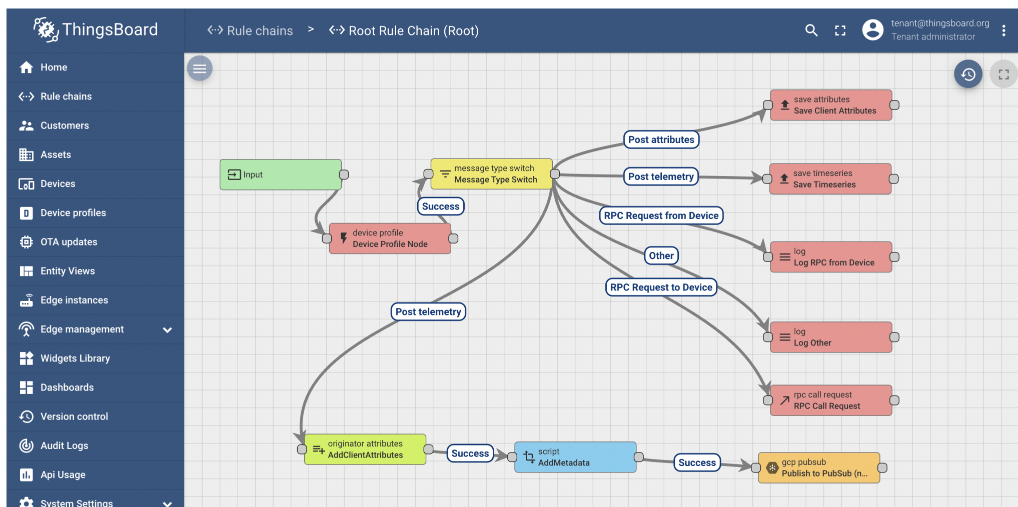

Figure 2.1: Thingsboard rule engine.

Chapter 3

Google Cloud Platform (GCP)

3.1 Containerization

Containers provide a robust solution for ensuring reliable software execution across diverse computing environments. Developers often face challenges when transitioning seamlessly from their laptops to testing environments, resulting in the application not functioning properly elsewhere. This issue can occur due to three primary reasons:

- Missing Files: If one or more files are missing, the application may not be deployed completely, leading to functionality issues.
- Software Version Mismatch: Differences in software versions between the developer's environment and the testing environment can cause compatibility problems and hinder proper execution.
- 3-Different Configuration Settings: Differences in configuration settings, such as network configurations or system dependencies, can impact the application's functionality when transitioning from one environment to another.

Docker is a purpose-built platform, offers streamlined capabilities for application development by facilitating the creation of isolated and virtualized environments. These environments serve as self- contained units for building, deploying, and testing applications. A container, as a fundamental element, encompasses two distinct components:

- Docker image : Composed of a series of immutable, read-only layers, the image operates as a template. Each layer originates from the previous one, forming a hierarchical structure. Notably, an image alone cannot be directly executed; it serves as a blueprint for constructing containers.
- Docker container: Representing a virtualized runtime environment, the container provides a secluded space to isolate applications from the underlying system. The container layer, a writable component, is added upon the instantiation of a container, thus enabling a functional virtual environment.

Overally, the use of containerization technology, such as Docker, has become increasingly popular in software development, enabling developers to work in a reliable, secure, and isolated environment, ensuring seamless application execution across various computing environments.

3.1.1 what is the difference between containers and virtualization?

Virtualization means abstraction of physical hardware, Each virtual machine is resource intensive and it takes a slice of actual physical hardware resources like CPU, memory and disk and we have limit in terms of the number of VM running on host. In a physical server that runs three virtual machines using a hypervisor (software for create and manage virtual machines) each virtual machine running its own separate operating system and they share memory and disk .

On the other hand, when using Docker to run three containerized applications, only a single operating system is utilized. In virtual machines virtualization happens at the hardware level, while containerization occurs at the application layer, allowing multiple containers to share the kernel and virtualize the operating system actually they don't need full operating system. This results in containers being extremely lightweight, often occupying only tens of megabytes, while a virtual machine can be several gigabytes in size. As a result, a single server can accommodate a larger number of containers compared to virtual machines. Furthermore, containerized applications can be initiated nearly instantaneously, whereas virtual machines often require several minutes to boot up their operating systems and launch the hosted applications.

Containerization plays a crucial role in enabling the concept of microservices. Rather than running an entire complex application within a single container, the application can be divided into separate modules, such as the database and the application front end. This modular approach simplifies the management of microservices, as each module is relatively simple and modifications can be made to individual modules without the need to rebuild the entire application.

Containers and virtual machines give the same kind of isolation ,we can run multiple application one host.

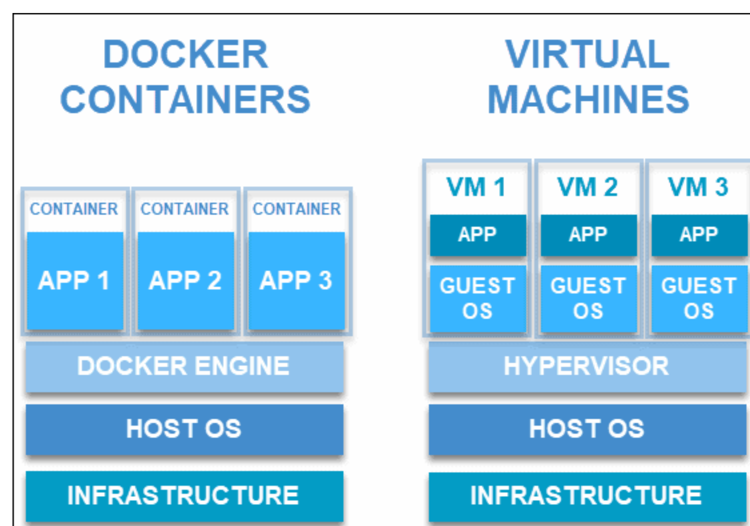

Figure 3.1: Difference btw containerization and virtualization [1]

3.2 Kubernetes

Kubernetes is an open-source container orchestration engine that automates the deployment, scaling, and management of containerized applications. [4].

Kubernetes is product of Google that it provides a solution for managing applications composed of hundreds or even thousands of containers. As I mentioned earlier, containers play a crucial role in the emergence of microservices. However, manually managing a large number of containers across multiple environments using scripts and self-made tools can be extremely challenging. This is where container orchestration comes into play.

Kubernetes offers several key features that make it an invaluable tool for container management. Firstly, it provides high availability, ensuring that the platform remains accessible to users at all times. Additionally, Kubernetes is highly scalable, allowing applications to scale up or down in response to changes in workload. This flexibility ensures optimal performance and resource utilization.

Another important aspect of Kubernetes is its disaster recovery capabilities. In the event of infrastructure issues, such as data loss or server failures, having reliable backups and the ability to restore the system to its most recent state is crucial. Kubernetes provides mechanisms for disaster recovery, enabling the restoration of operations and minimizing downtime.

The primary objective of using Kubernetes in this thesis is to automatically scale up the infrastructure whenever multiple sensors send data to the cloud. Subsequently, after performing the necessary actions, as I will explain in the following sections, the infrastructure should be scaled down again. Kubernetes handles this scaling process automatically. Furthermore, if a container crashes, Kubernetes automatically redeploys the affected container to its desired state, ensuring the continuity of operations.

3.2.1 Kubernetes Architecture

By deploying Kubernetes, you can create a cluster consisting of multiple machines known as nodes. These nodes are responsible for running containers managed by Kubernetes. A typical Kubernetes cluster comprises at least one master node, which connects to several worker nodes. Each worker node runs a Kubelet process that enables communication among the nodes and facilitates the execution of tasks, such as running application processes.

The Worker Nodes host pods that execute user workloads, serving as the primary location for actual work to take place. The number of Docker containers running on the worker nodes can vary depending on how the workload is distributed.Worker Nodes are where that actual work is happening.

On the other hand, the Master Node runs several essential Kubernetes processes responsible for proper cluster management. These processes oversee the Worker

Nodes and all activities occurring within the cluster. To ensure high availability and failover capability, multiple master nodes are utilized.

The Virtual Network is a crucial component of Kubernetes as it facilitates communication between all the Master Nodes and Worker Nodes. It turns all the nodes inside of the cluster into one powerful machine.

Worker Nodes bear a heavier workload compared to the Master Node as they are responsible for running applications and hosting hundreds of containers. In contrast, the Master Node operates only a few master processes and requires fewer resources. However, despite their workload disparity, the Master Node holds greater significance within the cluster. Losing the Master Node would result in the loss of cluster access entirely. Therefore, it is essential to maintain regular backups of the Master Node to ensure the cluster's continuous availability and functionality.

Kubernetes worker Node Components:

- kubelet: it serves as an agent that operates on each node and is responsible for managing the deployment of pods to Kubernetes nodes. It continuously receives new or modified pod specifications from the API server and ensures that the pods and their containers maintain a healthy state and run according to the desired configuration. Additionally, it instructs the container runtime to initiate or terminate containers as necessary.
- kube-proxy: is a network proxy that operates on every Kubernetes node. It is responsible for managing network rules on each node, facilitating network communication between nodes and pods. kube-proxy can either directly forward traffic or leverage the packet filter layer of the operating system for efficient routing.[5].
- Container runtime :is the software layer that manages the execution of containers. Kubernetes supports multiple container runtimes, such as Containerd, CRI-O, Docker, and various other implementations of the Kubernetes Container Runtime Interface (CRI) [5]

Master Node Components:

- API Server: which is also a container. It is the entry point to the Kubernetes cluster, this the process which different Kubernetes clients will talk to like UI if using Kubernetes dashboard, an API if using some scripts or command line tool, all of them will talk to API server.
- Controller Manager: keeps an overview of what is happening in the cluster, whether something need to be repaired or maybe if a container died it needs to be restarted.
- Schedular: is responsible for efficiently assigning containers to different worker nodes based on workload and available server resources. It employs intelligent decision-making algorithms to determine the optimal worker node for scheduling each container, considering factors such as the available resources on the worker nodes and the resource requirements of the containers.

- etcd:is a key-value storage that houses all the configuration and status data of each container within a Worker Node. As mentioned earlier, Kubernetes offers a backup and restore feature, enabling the recovery of the entire cluster through snapshots taken from etcd.

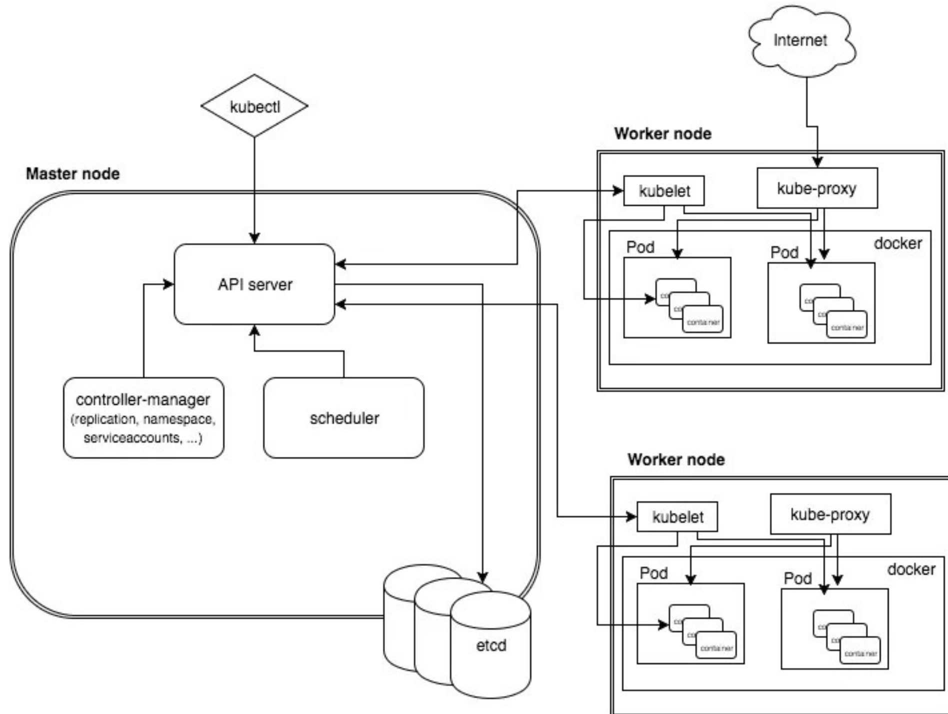

Figure 3.2: Kubernetes Architecture [2]

3.2.2 Events in Kubernetes

Events give you valuable insight into how your infrastructure is running while providing context for any troubling behaviors. This is why events are often useful for debugging [5]. Kubernetes events are generated automatically in response to various events such as changes in the state of resources, errors, or other important messages that need to be communicated to the system. While primarily serving as service messages for introspection and debugging purposes, these events carry a wealth of valuable and insightful information.

3.3 GCP Services

GCP is Google's cloud computing platform that offers a wide range of scalable and reliable services for deploying and managing applications and data in the cloud. I explain in brief about some important services of GCP that will be used in this thesis.

3.3.1 Terraform

The Google provider is utilized for configuring your Google Cloud Platform infrastructure. Terraform is an infrastructure-as-code (IaC) tool developed by HashiCorp. IaC tools enable you to manage infrastructure using configuration files instead of relying on a graphical user interface (GUI). With IaC, you can build, modify, and manage your infrastructure in a secure, consistent, and reproducible manner by defining resource configurations that can be versioned, reused, and shared.

Terraform enables us to define resources and infrastructure using easily readable, declarative configuration files. It not only defines the desired state of your infrastructure but also manages the entire lifecycle of your infrastructure, ensuring consistent provisioning and management. The Terraform state keeps track of changes made to resources. Remote backends, such as Terraform Cloud, enable teams to collaborate on infrastructure by version-controlling configurations, ensuring safe collaboration.

Terraform leverages specialized plugins known as "providers" to interact with various cloud platforms and services via application programming interfaces (APIs). The Terraform configuration language follows a declarative approach, allowing you to define the desired final state of your infrastructure. This differs from procedural programming languages that rely on step-by-step instructions for task execution.

Terraform providers automatically determine dependencies between resources, allowing them to be created or destroyed in the correct order. This ensures the integrity and consistency of your infrastructure deployment.

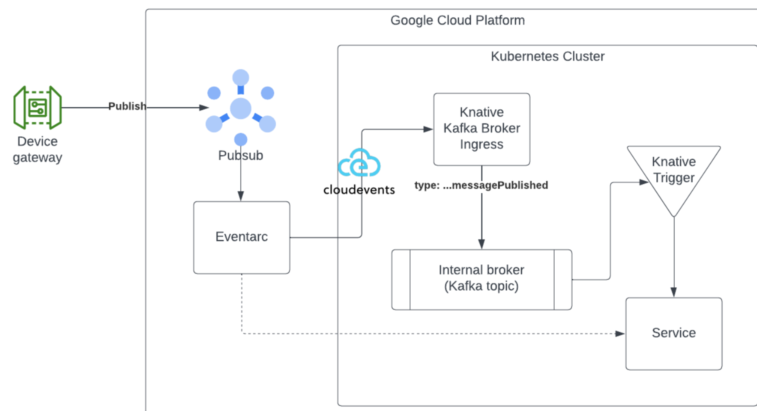

Figure 3.3: GCP Architecture Of Project

In Figure 3.3, the essential infrastructure of GCP services required for the goal

of this thesis is depicted. Instead of manually creating these services through a command line or GUI, they were conveniently provisioned using Terraform's declarative language within the Terraform directory.

3.3.2 Pub/Sub

Pub/sub helps in building robust and scalable systems by facilitating asynchronous integration of applications. Cloud Pub/Sub is a fully managed real-time messaging service that enables multiple applications to send and receive messages. The underlying concept of asynchronous integration is to respond to events represented as messages. Pub/sub simplifies the setup between services or applications. This is achieved by defining topics and creating subscriptions, which allow services to receive messages published on those topics. Consequently, one-to-many communication becomes much simpler.

3.3.3 Eventarc

Eventarc empowers you to deliver events from Google services, SaaS platforms, and your custom applications in an asynchronous manner. It facilitates the utilization of loosely coupled services that respond to state changes. With Eventarc, you can focus on building a modern, event-driven solution without the need for infrastructure management. This allows for enhanced productivity and cost optimization.[6].

3.3.4 Kafka broker

Apache Kafka is a widely adopted event streaming platform utilized for the collection, processing, and storage of streaming event data. It excels in handling data that lacks a distinct start or end. By leveraging Kafka, developers can build a new breed of distributed applications that effortlessly scale to handle billions of streaming events every minute.[7].

3.3.5 CI/CD (continuous integration/continuous delivery)

Google Cloud Deploy simplifies and enhances continuous delivery to GKE, Cloud Run, and Anthos. With Google Cloud Deploy, you can easily define releases and seamlessly progress them across various environments like test, stage, and production. The configuration file or files associated with Google Cloud Deploy define the delivery pipeline, specify the deployment targets, and outline the progression of those targets.

Delivery Pipeline and targets

The delivery pipeline describes a progression of deployment targets. In this Project, there are two targets dev and stable.

Skaffold

Google Cloud Deploy uses Skaffold to render your Kubernetes manifests. The service supports rendering of raw manifests and more advanced manifest-management

tools [8].

Skaffold is a powerful command-line tool designed to streamline continuous development for container-based and Kubernetes applications. With Skaffold, you can effortlessly manage the entire workflow of building, pushing, and deploying your application. It also provides essential building blocks for creating robust CI/CD pipelines. This allows you to concentrate on iterating and refining your application locally, while Skaffold automatically handles continuous deployment to your preferred environment, be it a local or remote Kubernetes cluster, local Docker environment, or Cloud Run project.[9]. Skaffold's file watcher is one of its most powerful features. It actively monitors one or more local directories on file systems, detecting any changes in the code. Whenever Skaffold detects a code change, it automatically initiates a new build process, creating a fresh container image from the updated source code. Once the image is built, Skaffold seamlessly deploys it to your Kubernetes cluster, ensuring that your application stays up to date with the latest changes in real time. This automated process greatly enhances development efficiency and enables rapid iteration and testing within the Kubernetes environment. In this thesis, Skaffold's YAML configuration file " skaffold.yaml" is utilized for each rule within its respective directory. This approach significantly simplifies our development workflow by eliminating the need to manually handle the lengthy process of building and pushing a new container image to the Kubernetes cluster every time code changes occur. Skaffold's automatic detection of code changes and its ability to initiate the image build and deployment process greatly reduces the time and effort involved in this process, enhancing overall development efficiency. As a result, Skaffold proves to be a valuable tool in streamlining the continuous integration and deployment workflow, allowing for more efficient and rapid iteration of our application within the Kubernetes environment.

3.3.6 Google Artifact Registry

Artifact Registry represents the evolution of Google Container Registry, serving as a comprehensive solution for managing container images and language packages. It seamlessly integrates with the cloud's build, test, and deployment processes, while being fully compatible with Google Cloud's tooling and runtimes. Additionally, Artifact Registry provides support for native artifact protocols, enabling effortless integration with your CI/CD tooling for establishing automated pipelines.[10].

The Container registry provides a centralized platform for managing Docker images, conducting vulnerability analyses, and implementing precise access control. By integrating with existing CI/CD systems, you can establish fully automated Docker pipelines, enabling rapid feedback on deployments.

The Artifact registry offers even more granular permissions, granting control over access at the project or registry level. Within a single Google project, you have the option to create multiple regional repositories or multiple repositories with independent registry permissions. This capability allows you to dictate where artifacts are stored and determine who has access to them.

To ensure security, Artifact registry roles can be utilized to apply the principle of least privilege. By assigning appropriate permissions to service accounts and users, they only gain access to the specific permissions required for their tasks.

3.3.7 Cloud Logging API

Cloud Logging is seamlessly integrated with Cloud Monitoring, Error Reporting, and Cloud Trace, empowering you to effectively troubleshoot issues across your services. By leveraging these integrated tools, you can easily identify and resolve any problems that arise. Additionally, you have the flexibility to configure alerts for specific logs, ensuring that you stay updated on critical events and can promptly respond to them [11]. Cloud logging is a comprehensive solution that efficiently stores logs from various Google Cloud products, while offering advanced search, monitoring, and alerting capabilities. With the ability to ingest custom log data from any source through its API, it provides a flexible and scalable log management solution.

Being a fully managed service, cloud logging eliminates the need for manual provisioning of hard drives or resizing partitions. This hassle-free approach allows you to focus on analyzing log data in real-time, without the need to synchronize server pods or manage time zones.

Logs in cloud logging consist of entries generated by Google Cloud services, third-party applications, or application code. These entries contain valuable information known as the payload, which can range from a simple string to structured data. By effectively organizing and storing log entries, cloud logging enables efficient log analysis and troubleshooting.

Log Explore

Logs Explorer provides a powerful platform to search, sort, and analyze logs using flexible query statements. It also offers rich histogram visualizations, a user-friendly field explorer, and the capability to save queries for future reference. You can set up alerts to receive notifications whenever specific messages appear in your logs or utilize Cloud Monitoring to create alerts based on logs-based metrics that you define. This ensures proactive monitoring and timely response to important log events.[11].

3.3.8 Firebase Cloud Messaging (FCM)

Firebase Cloud Messaging (FCM) is a versatile cross-platform messaging solution that enables you to send messages reliably, with no additional cost involved. With FCM, you have the ability to notify a client app about new emails or other data ready for synchronization. It also allows us to send notification messages to enhance user re-engagement and retention. For instant messaging scenarios, FCM supports message payloads of up to 4000 bytes, allowing for efficient data transfer to client apps.[12].

3.3.9 Firestore

Firestore is a serverless NoSQL document database designed specifically to simplify the storage, synchronization, and querying of data for IoT applications on a global scale. Its robust features make it an excellent choice for storing data collected from IoT devices.

- Scalability: Firestore is designed to handle large amounts of data and scale automatically. It can save high amount of incoming data from IoT devices without compromising performance. Firestore scales horizontaly so it ensures that as your IoT deployment grows, the database can handle the increased workload seamlessly

- Real-time updates: Firestore provides real-time synchronization, allowing you to receive instant updates whenever data changes. This feature is crucial for IoT applications where real-time data analysis, monitoring, and alerting are essential. In this thesis, we subscribe to changing data and react rapidly.

- Low latency: Firestore offers low-latency reads and writes, ensuring that data retrieval and storage operations are fast. This is important for IoT applications that require quick response times, especially in scenarios where immediate actions or notifications need to be done based on sensor data.

- NoSQL document model: Firestore is NoSQL document model, allowing you to store IoT data as structured documents.

- Offline support: Firestore provides offline support, enabling IoT devices to continue storing data even when there is no internet connectivity. Once the connection is restored, the data is automatically synchronized with the cloud.

- Firestore seamlessly integrates with other Google Cloud services, enabling the creation of comprehensive end-to-end IoT pipelines. By leveraging this integration, we can efficiently process, analyze, and gain insights from IoT data in real-time. This facilitates the construction of powerful IoT solutions that leverage the full capabilities of the Google Cloud ecosystem.

Chapter 4

Knative-eventing

4.1 Introduction

The structure of a monolith refers to a traditional software architecture that a single application that contains everything. In a monolithic architecture, all the components of the application, such as the user interface, business logic, and data access layer are integrated and deployed together in a single unit. This means that any changes or updates to one part of the application require redeploying the entire monolith.

Monolithic architectures are simple to develop and deploy, but they have challenges as the application grows larger and more complex like Scalability which means scaling the applications horizontally is difficult and they can be scaled just vertically. Another challenge with Monolithic architecture is when applications need changes to a specific component or introducing new features can be time-consuming and risky, as any modifications require retesting and redeploying the entire monolith. Using different technologies and frameworks within a Monolith is challenging and team collaboration is really difficult and can lead to coordination and collaboration challenges.

To solve these challenges, microservices are good solution. Microservices architecture dismantles the monolithic structure into smaller, autonomous services that can be developed, deployed, and scaled independently. Each microservice is dedicated to a specific business capability and communicates with other microservices through APIs. This modular approach allows for greater flexibility, agility, and scalability in building and managing complex systems. Events play a crucial role in a microservices architecture. Events represent occurrences or changes within the system and are used to communicate and coordinate actions between microservices. Event-driven architecture (EDA) relies on events as a means of communication and loose coupling between services.

4.2 Event-Driven Architecture

Event-Driven Architecture (EDA) is a software architectural model or paradigm that facilitates the production, detection, consumption, and response to events or significant changes in the system's state. It emphasizes the importance of events as the primary means of communication and coordination between different components of the system. By leveraging event-driven principles, applications can be designed to be more reactive, flexible, and loosely coupled, enabling efficient handling of complex and dynamic business scenarios.[13]. For years harnessing the power of data has been key to the success of organizations. Today you need to react to data in time, users expect actions right away using up-to-date real-time data and insights. As applications expand horizontally, connections become more complex and difficult to understand. While the microservices pattern provides a great deal of advantages, they also come with a great deal of complexity, Event-Driven Architecture produces a solution for this problem. It provides the ability to reduce dependencies and complexity in your application while still allowing the creation of more microservices.

In Event-Driven pattern there are three key concepts

- producers
- Intermediary
- consumers

Services don't communicate directly to each other, instead they communicate to an event intermediary. Producer does not need to be a service within application, they could be a source that you input to react to the occurrence and deliver the corresponding event to the intermediary. Services in Kubernetes can be either be a producer or consumer or both.

In Event-Driven Architecture, a producer does not need to know how an event will be consumed or take on any dependency from downstream service, similarly consumer only need to know that the event will be raised and understand how to utilize it actually none of detail of upstream service.

An event-driven architecture is a software design pattern where microservices respond to changes in state through events. These events can either contain the state itself or act as identifiers. When an event occurs, it triggers the relevant microservices, which collaborate to achieve a shared objective without needing knowledge about each other beyond the event format. While working together, each microservice can employ distinct business logic and generate its own output events, allowing for modular and decoupled processing within the architecture[14].

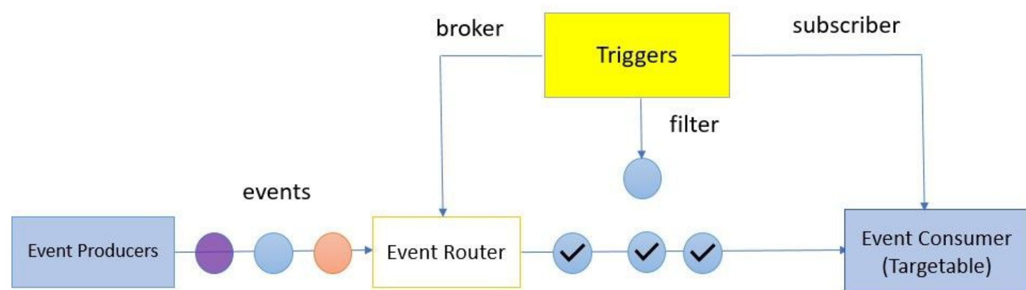

Figure 4.1: "Event-Driven Architecture"

4.3 CloudEvents

The absence of a standardized event description format necessitates developers to write custom event-handling logic for each event source. Without a common event format, the availability of shared libraries, tools, and infrastructure for seamless event data delivery across different environments is limited. To address this challenge, CloudEvents offers software development kits (SDKs) for multiple programming languages, including Go, JavaScript, Java, C#, Ruby, PHP, PowerShell, Rust, and Python. These SDKs empower developers to build event routers, tracing systems, and other related tools.

CloudEvents represents a specification aimed at establishing a common and simplified approach to describing event data. Its objective is to streamline event declaration and delivery across diverse services, platforms, and beyond. Although CloudEvents is still undergoing active development, it has garnered considerable interest from various industry players, ranging from major cloud providers to popular SaaS companies[15].

4.4 Knative

Knative is an open-source framework developed by Google that it simplifes the deployment and management of serverless workloads on Kubernetes. It provides a set of middleware components that enables developers to build, deploy, and scale containerized applications without the need for manual infrastructure management. Knative shines in enabling functions as a service and enabling higher productivity in the fast environment.

Knative plays a crucial role in the Kubernetes ecosystem due to the increasing adoption of cloud-native services and microservices architecture among organizations. Developing a microservices architecture involves creating multiple services, each requiring deployment YAML files, service YAML files, load balancers, and other related components. However, consolidating all these elements to deploy a single service becomes a time-consuming process, hindering the path to production. Additionally, scaling and configuring production settings can be challenging in Kubernetes.

Knative simplifies working with Kubernetes by offering higher-level abstractions and automating scaling and event processing. It seamlessly integrates with popular build tools and provides a serverless-like experience. With Knative, developers and organizations can utilize the strength and adaptability of Kubernetes without dealing with the complexity and manual work of handling and deploying applications.

Knative is composed of two components: Serving and Eventing. In this thesis, only the Eventing part is used.

4.4.1 Knative Serving

Knative Serving introduces Kubernetes Custom Resource Definitions (CRDs) to define and manage a specific set of objects. These resources play a crucial role in determining and governing the behavior of your serverless workloads within the cluster. By leveraging Knative Serving, you gain fine-grained control over how your serverless applications operate, allowing for enhanced scalability, auto-scaling, and other runtime behaviors tailored to your specific requirements[16].

Knative Serving facilitates fast deployment and automatic scaling of containers by utilizing a demand-driven approach, delivering workloads as needed.

Knative Serving simplifies the management of routes and configurations. It allows for multiple active routes, each pointing to a specific revision of a service. A revision represents a snapshot of the service's configuration at a given time. It's possible to have multiple snapshots of services running concurrently, with multiple routes pointing to each of these revisions. The Knative Serving component facilitates scaling, routing, and the management of these snapshots.

- Services: takes care of the complete lifecycle management of our workloads by automatically handling various aspects. It orchestrates the creation of necessary objects, such as routes, configurations, and new revisions, to ensure

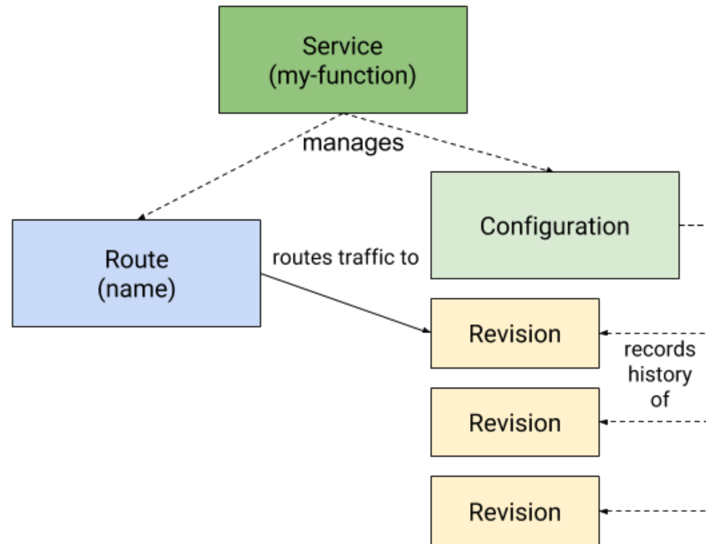

Figure 4.2: "Knative serving"

seamless updates to your application. With Knative Serving, we have the flexibility to define services that either direct traffic to the latest revision or pin traffic to a specific revision, enabling you to control how traffic is routed within our application. This simplified management process ensures smooth updates and efficient deployment of our services.

- Routes: it facilitate the mapping of a network endpoint to one or multiple revisions of our application. They offer various methods to handle incoming traffic, providing flexibility in traffic management. With routes, we can define rules and strategies for routing traffic, such as splitting traffic between multiple revisions, applying canary deployments, or using weighted routing to distribute traffic based on specific criteria. This versatile approach enables us to implement sophisticated traffic handling mechanisms tailored to our application's requirements.

- Configurations: play a vital role in maintaining the desired state of our deployments. They serve as a clear separation between code and configuration, aligning with the principles of the Twelve-Factor App methodology. By utilizing configurations, we can manage and update our application's settings independently from the codebase. Whenever a configuration change occurs, Knative Serving creates a new revision, ensuring that each deployment iteration is distinct and traceable. This approach promotes a modular and scalable deployment process while maintaining a consistent and reliable application state.

- Revisions: represent snapshots of the code and configuration for each modification of your workload. These revisions are immutable objects, preserving their state as they are created. Revisions provide a reliable and consistent foundation for our application's deployment history. Moreover, revisions offer the advantage of automatic scaling, allowing them to dynamically adjust their resource allocation based on incoming traffic. This scalability feature ensures

optimal performance and efficient resource utilization for our application, enabling it to seamlessly handle varying workloads without manual intervention.

4.4.2 Knative Eventing

Knative Eventing is a set of composable APIs that facilitate the implementation of event-driven architectures within applications. These APIs enable the creation of components responsible for routing events from event producers to event consumers, also known as "Sinks," which receive the events. Sinks can also generate a response event to fulfill HTTP requests.

To ensure seamless communication between producers and consumers, Knative Eventing leverages standard HTTP POST requests for event transmission. It adopts the CloudEvents specification as the message envelope to establish a shared understanding and eliminate the need for custom code-based agreements. Events are formatted according to the CloudEvents specification, enabling the creation, parsing, sending, and receiving of events in any programming language.

One of the key advantages of Knative Eventing is the loose coupling of its components, which allows for independent development and deployment. Producers can generate events without requiring active event consumers to be present and waiting for those events. Similarly, event consumers can show interest in specific classes of events before the producers create them.

In fact Knative eventing is our intermediary of our choice as I explained in the Event-Driven Architecture part (4.2).

4.5 Event Mesh

Event Mesh is an architectural layer designed to facilitate the dynamic routing and reception of events between different applications, regardless of their deployment locations (such as private cloud, public cloud, or on-premises). This layer is comprised of a network of event brokers, forming a configurable and dynamic infrastructure for distributing events among decoupled applications, cloud services, and devices. Its primary purpose is to enable flexible, reliable, and fast event communications, which are crucial for the continuous agility of digital businesses operating in an event-driven computing paradigm.

The concept of Event Mesh holds significant importance because it provides optimization and governance for distributed event interactions. In the context of transitioning to an event-driven model, legacy on-premises applications often act as static data silos, impeding the seamless sharing of information among different line of business, applications, and individuals. The objective of becoming event-driven is to set this data in motion and enable its sharing across the entire enterprise.

However, the challenge lies in efficiently moving events between on-premises applications, cloud environments, and IoT devices. Creating individual connections between each application and service across various locations would be a complex and cumbersome undertaking in terms of building, managing, securing, and scaling. To address this challenge, the solution is to introduce event brokers into each of these environments. Each application connects to the relevant broker and utilizes it to publish and subscribe to events. An essential aspect of establishing a fast and reliable event distribution system is the ability to connect these event brokers together, forming an Event Mesh. Event Mesh provides asynchronous (store-and-forward) delivery of messages.

The characteristics of an Event Mesh can be summarized as follows: Interconnected Event Brokers:

- An Event Mesh consists of a network of interconnected event brokers that facilitate the seamless flow of events between applications.

- Environment Agnostic: The Event Mesh is designed to be deployed in any environment, including public clouds, private clouds, Platform-as-a-Service (PaaS) environments, or non-cloud environments. It operates consistently across all these environments.

- Dynamic Event Routing: The Event Mesh dynamically learns which events should be routed to specific consumers and facilitates real-time event routing. It accomplishes this without being dependent on the location of producers and consumers within the Mesh or requiring manual configuration for event routing. It decouples the producer and recipient from the underlying event transport infrastructure. In an Event Mesh, both producing and consuming applications do not need to implement event routing or subscription management.

Event producers can publish all events to the Mesh, which can route events to interested subscribers without needing the application to subdivide events to channels. Event consumers can use Mesh to receive events of interest using filter expressions rather than needing to implement multiple subscriptions and application-level event filtering to select the events of interest.

4.5.1 Knative Event Mesh

Broker API offers a discoverable endpoint for event ingress and the Trigger API completes the offering with its event filtering and delivery capabilities. Brokers provide a discoverable endpoint for event ingress while Triggers facilitate the delivery of events. Knative eventing offers an Event Mesh using these APIs. Event producer and sinks are supporting components of the eventing ecosystem but are not directly part of the Event Mesh.

Using Brokers and Triggers abstracts the details of event routing from the event producer and event consumer.

In this thesis is used an Event Mesh of three brokers:

Pub/sub-broker that gets the events that come from the outside cluster and Eventarc broker that publish events on knative broker inside the cluster for connecting pub/sub to the internal Kafka broker because the publisher does not have access to the Kubernetes cluster, where the Kafka broker resides.

4.6 Knative Broker for Apache Kafka

The Knative Broker for Apache Kafka is a dedicated implementation of the Knative Broker API designed specifically for Apache Kafka. It aims to minimize network hops and provide seamless integration with Apache Kafka, ensuring efficient communication between the Broker and Trigger API model.

In the Knative Kafka Broker, incoming CloudEvents are stored as Kafka records, utilizing the binary content mode for enhanced efficiency in transport and routing. By employing this mode, the Kafka Broker optimizes its performance and eliminates the need for JSON parsing. In the binary content mode, all attributes and extensions of the CloudEvent are mapped as headers on the Kafka record, while the data of the CloudEvent corresponds to the actual value of the Kafka record. This approach offers the advantage of being less obstructive and ensures compatibility with systems that may not have a complete understanding of CloudEvents. [17].

Chapter 5

KRules

5.1 Introduction

In the previous chapter, I discussed Knative and its ability to define producers and consumers of events. However, Knative alone is insufficient to fulfill the constraints set by our goal. The objective of this thesis is to define a logic for handling incoming events and generating corresponding outputs. This functionality is not provided by Knative, as it primarily focuses on event management infrastructure. On the contrary, our goal requires the system to be capable of adapting its behavior based on the specific use case in a simple and declarative manner.

To illustrate this, consider the "riscaldamento centrale comunale" project, where an increase in the level of Radon radioactive gas in the reservoir tanks or canals poses a risk to workers who may need to enter those areas. By installing sensors, we can monitor the radon level in those areas and create rulesets that dictate actions such as opening the chamber covers or activating filtering fans to reduce the radon level.

In this chapter, we will delve into the detailed analysis of the KRules framework, which perfectly aligns with the requirements of our objective. KRules allows us to define rulesets using Python, enabling us to achieve the desired functionality in a seamless manner.

5.2 KRules

KRules is an open-source framework, developed by Airspot Cloud Native Development, that provides to Python developers, a flexible and fast way to build cloud native applications, creating event driven, context aware, and reactive microservices in a Kubernetes cluster. KRules adopts a rules-based approach based on paradigm events-condtions-actions. KRules is inspired by reactive manifesto taking full advantages of the Kubernetes cluster and Knative eventing[18]:

- Responsive: The system responds in real time
- Resilient: The system stays responsive in the face of failure
- Elastic: The system is responsive with varying workload
- Event Driven: the system relies on sending asynchronous events

These characteristics render KRules as an excellent solution for cloud-native and event-driven applications.

KRules seamlessly integrates with Knative eventing and Kubernetes, making it a perfect fit for our environment. Kubernetes allows for the automatic scaling of our system in response to an increase in the number of messages (events) generated by sensors.

While Knative provides a serverless infrastructure for event management, KRules operates at a higher level, enabling the creation of application logic through a set of rules defined as Python data structures. This empowers us to define and manage our application behavior in a flexible and intuitive manner.

5.3 Subject

One of the fundamental concepts in the KRules programming paradigm is subject. Whenever an event is generated, it can always be attributed to a specific entity that either produced it or is somehow associated with it.

KRules offers the capability to track the state related to subjects and react to any changes that occur. The state of a subject is defined through its reactive properties. Once a value is assigned to these properties, the subject comes into existence. Working behind the scenes is a component called the Subject Property Store, which not only ensures responsiveness but also provides efficient functionality in a highly competitive system. When a new value is assigned to a reactive property of the subject, an event is generated, carrying both the new value and the previous one called subject-property-changed. This enables the system to react not only to the new state but also to every state transition, facilitating more comprehensive event-based reactions.

In essence, any element within the application domain that can produce events and possess state has the potential to be treated as a subject.

Apart from reactive properties, KRules also introduces extended properties, which serve as metadata for subjects. These extended properties are recognized by the Knative eventing infrastructure and are primarily used to define the logic of the transport layer. For example, they can be employed to route events related to a specific subset of subjects within the same class to a designated broker, thereby triggering a distinct group of microservices within the same cluster or even in a separate one.

Each Rule resides on a Pod where is deployed the KRules base environment. With this command:

```
>>kubectl exec pod-name -ti --ipython
```

We open a python interactive shell on a Pod ruleset container to understand better some concepts with an example.

First, we create a new subject named “foo”.

```
>>foo =subject_factory(\foo")
```

As I explained before the subject start to exist when we give it a value

```
>>foo.moo
```

```
>> foo.moo
...
AttributeError: moo
```

Figure 5.1

Because the property moo does not exist (Figure 5.1), so we assign a value to it and try again (Figure 5.2)

```
>> foo.moo = 1
>> foo.moo
1
```

Figure 5.2

As before I explained, the property assignment generates an event which then can be processed by one or more Rules. The Figure 5.3 displays cloudEvent format of event

```
☁️  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: subject-property-changed
  source: my-ruleset
  subject: foo
  id: bd198e83-9d7e-4e93-a9ae-aa21a40383c6
  time: 2020-06-16T08:16:57.340692Z
  datacontenttype: application/json
Extensions,
  knativearrivaltime: 2020-06-16T08:16:57.346535873Z
  knativehistory: default-kne-trigger-kn-channel.iot-demo-gcp-01.svc.cluster.local
  originid: bd198e83-9d7e-4e93-a9ae-aa21a40383c6
  propertyname: moo
  traceparent: 00-d571530282362927f824bae826e1fa36-a52fceb915060653-00
Data,
  {
    "property_name": "moo",
    "old_value": null,
    "value": 1
  }
```

Figure 5.3: CloudEvents format

The event comprises a collection of attributes that are defined as an integral part of the standard, along with any extended attributes tailored to meet specific application requirements. Together, these attributes play a crucial role in defining filters for Knative triggers. Please note that the modified property is duplicated in both the payload and as an extended property. This redundancy exists because, at the transport and addressing level of the event, the message content is not considered. Instead, it is utilized by a ruleset. By looking at the payload content event contains, the name of the property and its new value, also the old property value. This is very useful for implementing a logic established not just on the new value of the property but also its old value.

The subject requires a storage solution that provides persistence and, in certain cases, handles concurrency effectively to manage its state. This solution will primarily focus on storing properties and managing access, and it will be utilized by the subject's high-level interface responsible for handling caching and responsiveness. For this purpose, Redis, an in-memory key-value database with optional durability, is employed in this thesis. Redis is well-suited for managing many concurrent accesses, ensuring efficient storage and retrieval of data while maintaining consistency.

5.4 Rules

Rules are grouped into rulesets, which function as microservices deployed on a cluster and operate independently, responding to specific event types and attributes through Knative's triggers. The establishment of these triggers and the selection of events to be received by a ruleset are not constrained in any particular way. The resilience of the resulting system increases as the triggers and corresponding rulesets are defined with greater granularity, allowing each ruleset to scale independently. Within a ruleset, multiple rules can be defined, each subscribing to different event types (if captured by the triggers) and having distinct activation criteria based on the received payload but in this thesis each Rule is deployed in a different pod with a separate trigger that subscribes to a specific event. It is important to note that each rule is always contextualized to a subject. Events can originate from either outside or inside the cluster, the events with the aim of this thesis come from outside sensors.

A Rules is a logic written with python data structure and has different attributes:

- name: is of type string indicating the name of the rule
- subscribe_to: is a string type that indicates the event to which the rule reacts. It corresponds to the attribute type of the cloudEvent, it allows the rule to react to multiple events with different types
- filters: refers to a list of functions designed to filter events based on specific criteria.
- processing: refers to a list of functions that are executed as a result of the filters being applied

5.5 Filters

When a rule receives an event and its type matches any of the types specified in the "subscribe_to" section, the functions listed in the "filters" section are sequentially executed first. These functions serve the purpose of further filtering the events, allowing for highly specific rules and promoting code reusability. Each function is expected to return a boolean value. If a function returns true, the rule proceeds to the next function or moves to the "processing" section. If a function returns false, the rule execution stops.

KRules offers several built-in filtering functions that enable working with events. However, if you require a specific function, you have the flexibility to create a custom one and call it within this section.

- Filter: Evaluate the Boolean expression passed returns its value
- SubjectNameMatch: returns true if the subject name matches the regular expression passed as an argument to the function
- SubjectNameDoesNotMatch: returns True if the subject's name matches the given regular expression
- CheckSubjectProperty: Returns True if the given subject property exists and, if provided, match the given value.
- PayloadMatch: Processes the payload with a given jsonpath expression to check its content.
- OnSubjectPropertyChanged: specific function to filters events of type subject-property- changed. This event is produced whenever a subject property changes and its data contains the property name, the new property value and the old one.

5.6 Processing

Once all the functions in the "filters" section have been successfully passed, the functions in the "processing" section are executed sequentially. These functions form the processing core of the Rule and define the desired actions to be taken.

Similar to the "filters" section, KRules offers a set of basic functions in the "processing" section that perform generic operations. However, since KRules is a framework, it cannot cover all possible processing functions. Nonetheless, it provides a solid infrastructure for managing these functions and includes several versatile built-in functions. Consequently, you have the flexibility to extend KRules by creating custom functions tailored to your specific use case.

- Route: Produce an event inside and/or outside the ruleset. In the rulesets are defined for the aim of this thesis as I explained pub/sub retrieve events from outside then eventarc broker for receiving events that come from pub/sub then we use the Route function in the first Rule to publish an event on the internal kafka broker.
- SetSubjectProperty: Set a single property of the subject, supporting atomic operation.By default, the property is reactive unless is muted (muted=True) or extended (extended=True).
- SetSubjectProperties: Set multiple properties in subject from dictionary. This is allowed only by using cache and not for extended properties.
- StoreSubject: Store alla subject properties on the subject storage and then flush the cache, which Usually happens at the end of the ruleset execution.
- FlushSubject: Remove all subject's properties. It is important tho recall that a subject exists while it has at least one property.
- SetPayloadProperties: Set the given properties in the payload, if some of that already exist will be overridden.

Chapter 6

Project Architecture and results

6.1 LoRa server and Thingsboard implementation

For the purpose of this thesis, we utilized a sensor called "Dragino_Temp_Hum" to gather temperature and humidity data from the environment where it was deployed. As mentioned earlier, we opted for LoRa sensors due to their extended range and lower battery consumption compared to other sensor types.

The sensor transmits its data to a LoRa gateway, which converts the radio frequency signals into IP packets for transmission to the LoRa server. The LoRa server serves as a dashboard for managing various devices and monitoring their data and status.

While our objective was to send messages to GCP Pub/Sub directly, we encountered limitations with the Pub/Sub integration. As a result, we decided to leverage an alternative integration. We directed the messages to the InfluxDB integration solely for the purpose of creating a historical record on an older but still used company IoT Platform. At the same time, the messages were forwarded to Thingsboard through the Thingsboard integration.

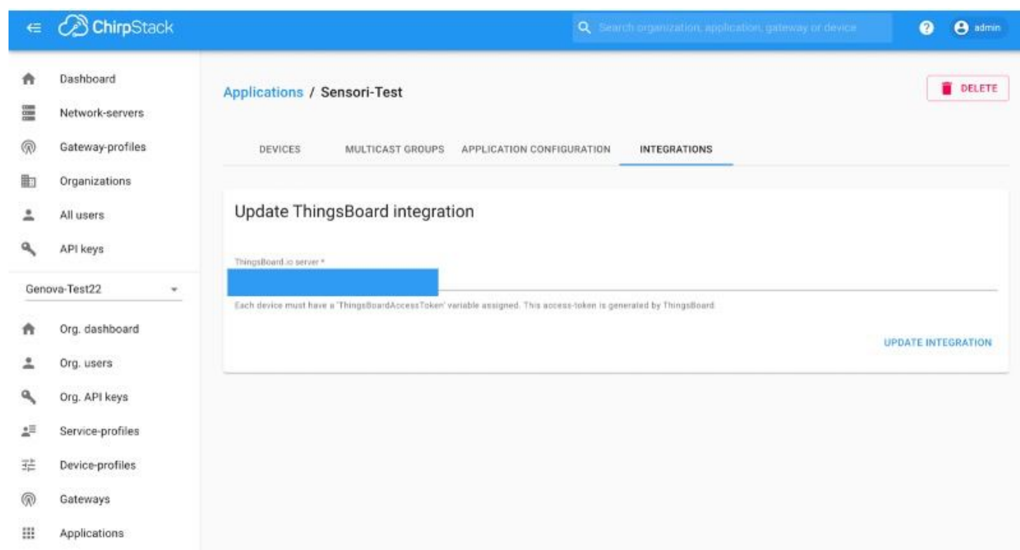

Figure 6.1: Thingsboard integration

Figure 2.1 showcases the rule engine architecture of Thingsboard. As messages enter Thingsboard, we assign metadata to each one. This metadata serves the purpose of future identification of the message source, and it also specifies the payload of the message received from the LoRa server as data. Finally, the data and metadata are published on a Pub/Sub broker within our GCP project on a topic that we named named “lorasource-dev”. The use of a topic is useful to organize messages passing through the brokers, and to allow subscribers to only receive messages related to our project/sensors.

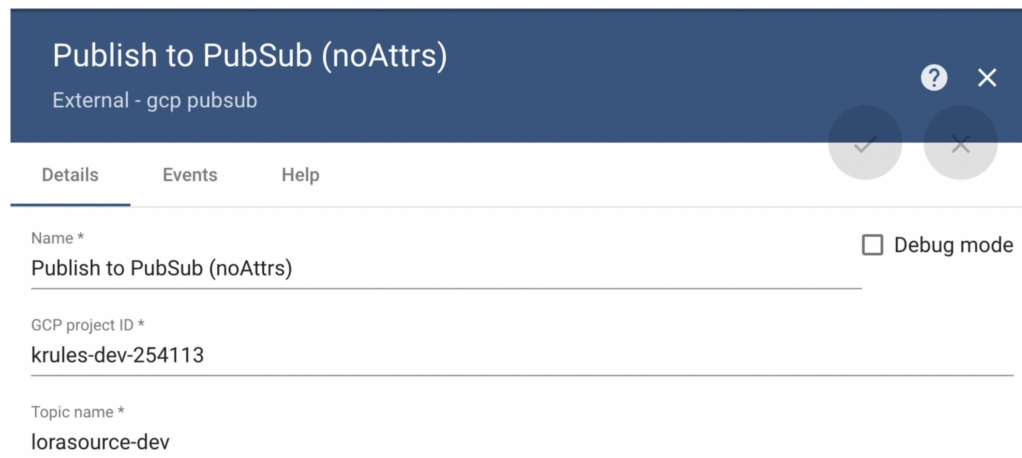

Figure 6.2: publish lorasource-dev events to pub/sub

6.2 GCP implementation

KRules is built on top of Knative and in particular it wants to make the most of its eventing part.

A KRules project resides in his own namespace. Knative eventing provides brokers to deliver events to subscribes. By default, KRules requires a default broker for all general events and procevent for special kinds of events. These events are produced when rules are processed carrying useful information about their execution.

The folder that contains the project structure as Figure 6.3 demonstrates:

```
my-project/
├── base/
│   ├── Dockerfile.j2
│   ├── .env
│   ├── ipython_config.py
│   ├── k8s/
│   │   └── config-krules-subjects-redis.yaml.j2.example
│   └── make.py
├── env.local
├── env.project
└── rulesets/
```

Figure 6.3: our project directory

Env.project consists of variables globally related to the project rather than the local development environment. This file must be included in the git repository. For all developers working on the project and for different deployment environments should be the same.

Env.local: it defines our local environment, and it is not included in the git repository. It takes priority over the project environment allowing you to override variables if neede.

Base directory is where the common resources of the project are located.

Rulesets directory is the suggested location in which rulesets will be created.

KRules extensively utilizes Jinja2 templates to offer flexibility for defining Kubernetes resource files. When a ruleset is deployed, all files with the *.j2 extension in the k8s folder of the base (as well as the ruleset's k8s folder) are rendered and applied to the cluster.

KRules extensively utilizes Jinja2 templates to offer flexibility for defining Kubernetes resource files. When a ruleset is deployed, all files with the *.j2 extension in the k8s folder of the base (as well as the ruleset's k8s folder) are rendered and applied to the cluster. We utilize a script called "make.py" to render and apply resources for any required tasks. This script is consistently present in our workflow. Behind the

scenes, we leverage the powerful ”sane-build” library to handle task dependencies and ensure maximum customizability throughout the process. For getting a list of the available tasks to render in each directory we run

```
>>./make.py --list
```

As I explained in the chapter 5, subject requires a storage for preventing concurrency and enable persistency, we installed Redis that it will be available for all rulesets. ConfigurationProviders serve as the standard method for injecting configurations

```yaml
apiVersion: krules.dev/v1alpha1
kind: ConfigurationProvider
metadata:
  name: config-krules-subjects-redis
  namespace: {{ namespace }}
spec:
  key: subjects-backends.redis
  appliesTo:
    krules.dev/type:
      - ruleset
      - generic
  data:
    url: redis://$REDIS_USER:$REDIS_AUTH@$REDIS_HOST:6379/4
    key_prefix: {{ project_name }}
  container:
    envFrom:
    - secretRef:
        name: redis-auth
```

Figure 6.4: Redis configuration

into Rules. They utilize labels to associate configurations with pods where Rules are deployed. In this scenario, all Rules will receive this configuration since they share the ”krules.dev/type: ruleset” label, which is specified in the appliesTo property of ConfigurationProviders.

KRules is built upon Knative eventing, which relies on CloudEvents being delivered through brokers. In our namespace, we have identified two brokers. Pub/Sub broker, receives messages of topic ”lorasource-dev” from Thingsboard. To establish a functioning pub/sub system, Terraform is utilized in this project to create and configure the necessary resources for the pub/sub broker.

Then we need to Enable Eventarc to manage GKE clusters so the messages send to Eventarc because the publisher does not have access to Kafka broker.

Eventarc offers a standardized solution to manage the flow of state changes, called events, between decoupled microservices. For sending messages from pub/sub to Eventarc, a trigger is defined to subscribe to the messages that come from pub/sub.

```
resource "google_pubsub_topic" "lorasource-dev" {
  name = "lorasource-dev"
  project = "krules-dev-254113"

  labels = {
    app = "krules-lab"
  }


  message_retention_duration = "86600s"
}
```

Figure 6.5: pub/sub Terraform configuration

In this thesis we could put all the rulesets in the same container inside pod but for reasons like more flexibility and easier creation new Rules within new pod, easier debugging and faster processing of the Rules we decided to separate Rules in different pods with specified trigger to subscribe to just the specific events that that Rule need to does some action or actions.

In this thesis, we developed four Rules that are logics that they react to changing sensor temperature and humidity.

After receiving a message from Pub/Sub, Eventarc generates an event of type "google.cloud.pubsub.topic.v1.messagePublished". We have two rules in place to handle these events.

The first rule, named "lorasource-dispatcher", sets up a trigger that subscribes to the Eventarc broker for events of type "google.cloud.pubsub.topic.v1.messagePublished". In the processing step, it invokes the Route function, which publishes an event to the internal Kafka broker, creating a new event called "lorasource-event".

The second rule, named "lorasource-warning-detector", subscribes to events of type "lorasource-event" in the default Kafka broker. In the processing step, the payload of the message is saved in a new variable called "lastReceivedPayload". The temperature sent by the sensor is then checked. A reactive property named "sensorStatus" is defined, and its value is set to "high" if the temperature is above 30°C, otherwise, it is set to "normal". We utilize Redis as an in-memory storage to store the value of "sensorStatus". Whenever the sensor status changes (e.g., from high to low or vice versa), another event is triggered called "subject-property-changed". This event updates the payload, including the property name, the new value, and the old value. This is the reason why we had to store the original payload (containing the sensor data) it in the "lastReceivedPayload" variable.

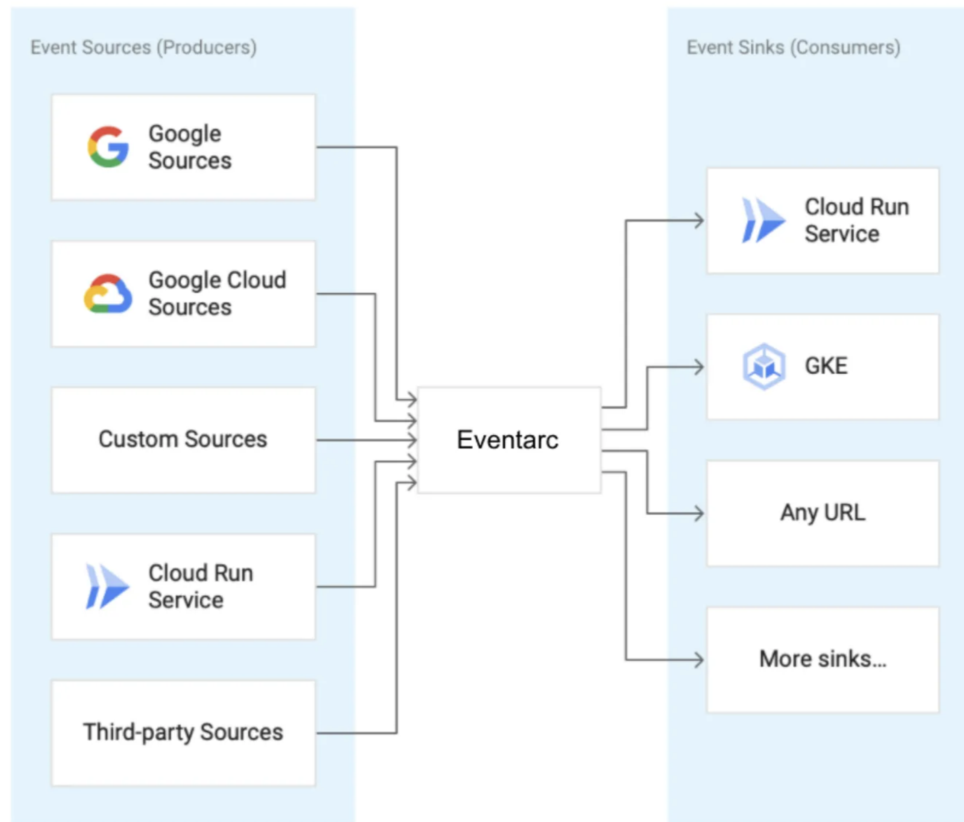

Figure 6.6: Eventarc configuration [3]

The third Rule subscribes to events of type "subject-property-changed". It also includes a filter that checks if the changed reactive property is "sensorStatus". If the filter is satisfied, a notification is sent to a mobile app or web application using the Firebase FCM service, to inform the users. The third and fourth rules work in parallel.

To achieve automation, we employ a Raspberry PI, a small single-board computer capable of running Linux, equipped with 40 General Purpose Input Output (GPIO) pins that can be programmed in read/write mode using code. In the fourth rule, we send an HTTP POST request to the IP address of the smart device to display the sensor status. The status is shown in red when the temperature is too high, and green when it is normal.

This test demonstrates how to connect and automate actions with a smart device. The same process, with minor modifications, can be applied to control any IP-connected smart device that supports HTTP API. For instance, we can make API calls to a smart air conditioning system to turn it on when the temperature is too high and turn it off when the temperature returns to normal.

During this thesis and the deployment of Rules, our objective went beyond simply turning on a red or green light. Our focus was on exploring the capabilities of KRules and implementing automated tasks, as well as understanding how to connect and interact with smart devices. The process we developed can be adapted,

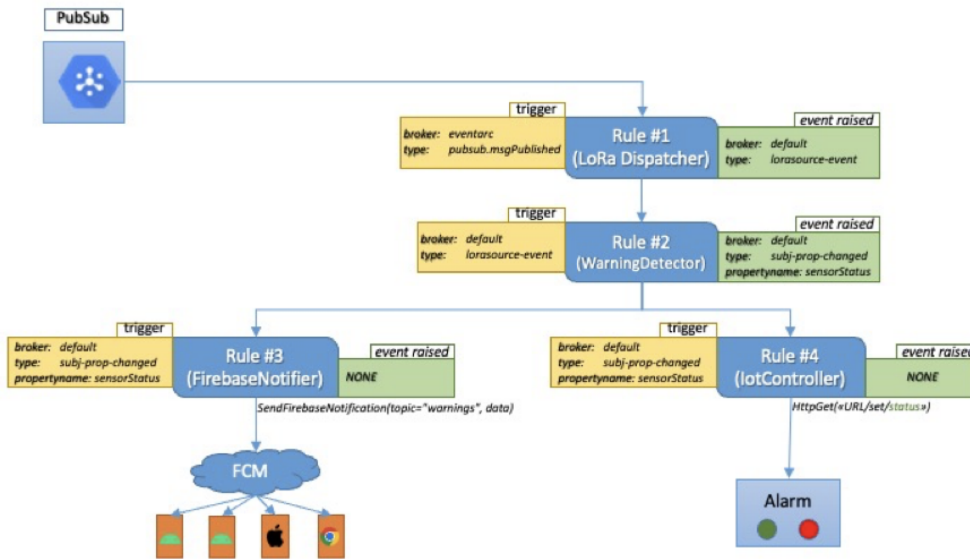

Figure 6.7: Rulesets developments

with minor modifications, to control any IP-connected smart device that supports HTTP API (or, using some of gateways, also with smart devices based on other communication protocols such as Modbus, Canbus, ODBC, Bluetooth, etc..).

For instance, we demonstrated how an API call can be made to a smart air conditioning system to activate it when the temperature is too high and deactivate it when the temperature returns to normal. This showcases the potential for automating actions based on real-time data from various sensors and devices.

KRules, developed by Airspot company [4], is a revolutionary product that provides a rapid and flexible solution for developing cloud-native applications. Integrating Krules with into Thingsboard, users can easily configure and automate actions based on specific conditions in a much flexible way than just using Thingsboard's internal rule engine. Unlike other complex and challenging configuration options available in some dashboarding systems, KRules offers a user-friendly interface and seamless integration with different services.

In conclusion, our project demonstrates the power of KRules in enabling efficient automation and control of smart devices, paving the way for innovative applications in various domains.

Chapter 7

Conclusions and future works

The potential of IoT devices, coupled with technologies like LoRa and platforms such as Thingsboard, has revolutionized industries across various domains. Our aim is to present a new practical solution that caters to the needs of all users and seamlessly integrates with any preferred IoT platform and/or dashboarding system.

During the time I am writing my thesis, the region of Emilia Romagna faced a critical situation due to heavy rainfall, posing a threat to the lives of its residents as the regional water levels began to rise. Although some people received messages via WhatsApp or SMS, many were asleep and unaware of the danger. Instead of relying solely on messages, an automated alarming system could have been implemented to activate sirens in high-risk areas and effectively warn the population.

Another significant concern related to worker safety arises from the "teleriscaldamento" project in Turin. The project recycle the hot water used to activate the turbines in power plants and distribute it to the citizen using a network of pipes running underground and connected to all the buildings of the city. This hot warer is used for both heating and for domestic use. The underground chambers and reservoir tanks involved in the project sometimes experience an increase in the level of the Radon radioactive gas, that can be dangerous for workers who may need to access these areas. By installing sensors to monitor the radon levels, automated actions such as opening chamber covers or activating filtering fans could be triggered to reduce the radon concentration.

These examples highlight the potential of the Iren's "Smart City" project, which aims to actively monitor and address various hazards through automated measures such as activating security sirens, opening vents, or operating filtering fans.

The objective of this thesis is to simulate the IoT platform within the project, allowing sensors to effectively communicate their data to the data centers, and to automate some critical aspects of specific projects, obtaining what could be called a I2I (IoT-2-IoT) solution.

7.1 Future Works and Implementations

Enhancing Future Thesis Work: Creating a Universal Event-Driven Processing Engine for IoT Platforms.

In the scope of this project, the future objective is to leverage KRules to develop a versatile event-driven processing engine that can be applied to any IoT platform. This engine will offer seamless integration and efficient processing of events across diverse IoT ecosystems.

To address scalability challenges, the plan is to implement autoscaling capabilities for the entire Thingsboard infrastructure, which currently operates as a monolithic instance. By incorporating KRules and employing a load balancer, the system will be able to dynamically handle increasing message loads with optimal resource allocation.

Furthermore, utilizing KRules will enable the creation of multiple instances of specific modules or sub-modules within the Thingsboard framework. This modular approach allows for better resource distribution and alleviates overloaded components, ensuring smooth and efficient operation.

Looking ahead, an alternative option under consideration is the adoption of Google Cloud Run as a replacement for GKE. This potential shift aims to simplify the configuration and development of rules, streamlining the deployment and management process. Also, the use Clud Run could also result in lower service costs if the number of messages to be managed is low (Cloud Run service is free for low usages, whereas Kubernetes clusters have to be active 24/7 anyways, so the service cost might be higher)

By pursuing these future directions, the thesis work endeavors to revolutionize event processing in IoT platforms, enabling enhanced scalability, modular flexibility, and simplified configuration for rule-based systems.

Bibliography

[1] Knowledge zone. https://knowledgezone.co.in/posts/62694b8075c1691ba69b9cc5.

[2] Cloud Native Wiki. https://www.aquasec.com/cloud-native-academy/kubernetes- 101/kubernetes-architecture/.

[3] Medium Journal. Available: https://medium.com/google-cloud/event-driven-architecture- eventarc-on-gcp-2c051c27e0d9.

[4] Kubernates Documentation. https://kubernetes.io/docs/home/.

[5] Cloud Native Wiki. https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-nodes.

[6] Google Cloud Documentation. https://cloud.google.com/eventarc/docs.

[7] Google Cloud Documentation. https://cloud.google.com/learn/what-is-apache-kafka.

[8] Google Cloud Documentation. https://cloud.google.com/deploy.

[9] Skaffold Documentation. https://skaffold.dev/docs/.

[10] Google Cloud Documentation. https://cloud.google.com/artifact-registry.

[11] Google Cloud Documentation. https://cloud.google.com/logging.

[12] Fire Base Documentation. https://firebase.google.com/docs/cloud-messaging.

[13] Xenon Stack. https://firebase.google.com/docs/cloud-messaging.

[14] Google Cloud Documentation. https://cloud.google.com/eventarc/docs/event-driven-architectures.

[15] Cloud Events. https://cloudevents.io/.

[16] Knative documentation. https://knative.dev/docs/concepts/#what-is-knative.

[17] Knative documentation. https://knative.dev/docs/eventing/brokers/broker-types/kafka-broker/.

[18] Krules documentation. https://intro.krules.io/en/0.8.3/overview.html.

Bibliography Description: All the websites have been accessed on April/May 2023