

Towards Smart Home Automation Based on Containerization

Mounira TARHOUNI^{1,2*} and Ibrahim ALOUI²

^{1*}University of Gabes, Faculty of Sciences of Gabes, Hatem Bettahar
IRESCoMath Research Lab, LR20ES11, 6072, Gabes, TUNISIA.

²Higher Institute of Computer Science and Multimedia Gabes (ISIMG),
University of Gabes, TUNISIA.

*Corresponding author(s). E-mail(s): mounira.tarhouni@isimg.tn;
Contributing authors: ibrahim.aloui@isimg.tn;

Abstract

This paper presents an advanced smart home automation system optimized for managing IoT devices and sensors through container-based architecture. The system is deployed on a Raspberry Pi using Docker and Docker Compose, which enhance operational efficiency, scalability and interoperability with other smart home systems. A Dockerized Django backend is employed for robust data processing and user authentication. Furthermore, AI-powered programs integrated within the system enable advanced functionalities. A Flutter-based mobile application provides a user interface that includes features like facial recognition for login, comprehensive weather control, voice command integration, and enhanced smart surveillance. The containerized approach ensures improved system performance and scalability, making this solution highly suitable for modern home automation applications.

Keywords: Containerization, Docker, IoT, AI, face recognition, Voice recognition

1 Introduction

The advancement of smart home technology has greatly increased the comfort, safety, and energy efficiency of modern life. Compared to other settings like industries and cities, homes have been identified as one of the major contexts for the widespread adoption of IoT devices [1]. Smart homes combine many Internet of Things (IoT) gadgets,

like security cameras, connected appliances, and smart thermostats, to offer homeowners automation and remote control. Nevertheless, there are performance, scalability, and deployment issues with controlling these devices.

Containerization provides a practical solution to these issues. Docker is a platform that ensures consistent performance across environments by packaging apps and their dependencies into lightweight, independent containers. An orchestration tool called Docker compose increases system scalability and stability by automating container deployment, scaling, and operation. This approach tackles the problem of controlling IoT devices and paves the way for a more efficient and seamless smart home ecosystem. Artificial intelligence (AI) solutions like integrated regression models can further improve functionality by providing predicted insights and optimizing workflows. The distinctive contributions are as follows:

- The implementation of a deep learning algorithm-based face recognition system
- The application of AI to the development of a voice command system
- Regression model-based weather prediction systems or those that incorporate the Weather API.
- The development of a Flutter app that support face recognition login and allow user to interact with Smart Home.
- Dockerization of Django server
- Dockerization of the IoT system (micro-services):it is an effective way to manage and deploy IoT systems that involve microservices. It consists on Break down our IoT system into smaller, self-contained microservices. Each microservice should handle a specific part of our system's functionality.
- Orchestration of micro-services:This involves optimally managing the deployment, scaling, and operation of containerized applications, thereby ensuring their performance, reliability, and security.

Our approach's strength lies in its combination of advanced security via face recognition for both the system and the Flutter app login, easy-to-use voice commands, proactive weather prediction, control over portable apps, scalable and resilient Dockerized server and micro-services, and adaptable, cross-platform Docker containers. All of these features come together to create a comprehensive, safe, and effective smart home system. The paper is organized as follows. In Section 2, we will review the challenges faced in IoT systems and underscore the necessity of adopting a microservice-based architecture. In Section 3, the architecture of our Securing Smart Home is explained by presenting the components and the interaction between them. Implementation details of some of the services are given in Section 4. The implemented testbed and the results of functional experiments are addressed in Section 5. The overview of our proposed platform and possible improvements are reported in Section 6.

2 Related Work

Nowadays, automation plays a crucial role in all work places and living homes. The primary goal of smart home automation is to simplify and enhance the convenience of daily life for homeowners and users. It plays a crucial role in supporting healthcare,

as well as contributing to the social and economic well-being of users, by creating a comfortable and efficient living environment. Authors in [2] provide a comprehensive taxonomy of IoT-based smart home automation systems, offering in-depth discussions on the technologies, trends, and challenges in their design. It presents a thorough review of existing literature, organized by various application areas of smart home automation. Finally, the paper highlights the key approaches, technologies, and the strengths and weaknesses of different smart home automation systems. Vamsikrishna et al. (2017) developed a system that allows users to control all home appliances via smartphone and PC, given that an Internet connection is available. The system employs a Raspberry Pi due to its capability to interface with multiple sensors and establish an Internet connection. Additionally, computer vision methods are utilized in this system [3]. In [4] authors present a cost-efficient integrated smart home system leveraging IoT and the Edge Computing paradigm. This system enables remote and automated control of household appliances while ensuring both security and safety. To safeguard customer privacy, sensitive data is stored locally using the edge-computing model. Furthermore, visual and scalar data generated by sensors are processed and managed on an edge device (Raspberry Pi) to minimize bandwidth usage, computational load, and storage costs. This paper [5] presents a smart home framework built on an Internet of Things (IoT) platform. The system enables control of devices such as lamps, fans, and irrigation pumps, along with remote monitoring of temperature and humidity levels. An IoT platform (ThingSpeak), integrated with the Blynk application, has been utilized for remote monitoring and control of home appliances. This system enables interaction with the appliances via a mobile application when specific conditions are met. Home security systems have evolved from basic setups to highly sophisticated solutions, incorporating advanced technologies such as artificial intelligence to enhance their effectiveness and capabilities. In [6], authors develop a prototype of an IoT-based smart home security system with a Flutter mobile app. The system designed is an IoT-based smart home security system, equipped with a series of sensors including temperature, light, flood, and motion sensors. These sensors are controlled by an Arduino UNO, which acts as a device manager and is connected to a Raspberry Pi microcomputer.

The aforementioned works rely on hard-coded traditional software architectures, which face challenges in terms of interoperability with other existing smart home systems. Microservices represent one of the latest software architecture paradigms, designed to address numerous IOT challenges [7].

Microservice architecture has gained significant traction in cloud computing and enterprise applications, as it addresses the limitations of traditional monolithic software in terms of scalability and maintainability. Unlike monolithic systems, which have reached the boundaries of their manageability, microservices break down applications into a set of distributed, loosely coupled services. By leveraging service-oriented architecture (SOA) principles and advancements in software virtualization, microservices allow for more flexible and scalable solutions.

The increasing number of IoT devices and applications across diverse domains has introduced new demands on the IoT ecosystem. This paper [8] explores the patterns

and best practices commonly applied in the microservices architecture and examines how these can be leveraged in the context of the Internet of Things (IoT). In [9], authors present a general microservice framework for IoT applications, offering a more scalable, extensible, and maintainable architecture. they detail the system’s design and the associated microservices, with a focus on core services and the communication between the service layer and the physical layer of IoT devices.

Microservices represent an architectural style that breaks down an application into a collection of independent services, each serving a specific business function. This independence allows for more agile development, testing, and deployment. Containers, on the other hand, are lightweight, standalone packages that include everything needed to run software, ensuring portability across different environments. By combining these two approaches, organizations can benefit from simplified management, efficient scaling, and service isolation, leading to more dynamic and resilient systems. Tools like Docker and Kubernetes further facilitate this integration, enabling automated orchestration and deployment that are essential for modern microservices architectures.

There is considerable research dedicated to performing performance tests that aim to analyze how edge devices behave when implementing containerization. In this article [10], authors present a modular and scalable architecture that leverages lightweight virtualization. This modularity, along with Docker’s orchestration capabilities, facilitates easier management and supports distributed deployments, resulting in a highly dynamic system. The author of [11] conducts benchmark tests on Raspberry Pi B+ and Raspberry Pi model B, focusing on Disk and Network I/O for both native and containerized approaches. While the overhead is significantly high for the Raspberry Pi B+, notable performance improvements are observed with the Raspberry Pi 2. These benchmarks demonstrate that containerization is feasible on System on Chip (SoC) devices like the Raspberry Pi 2, proving to be more efficient than VM-based virtualization. Chesov et al. [12] present a multi-tier approach to containerize various functionalities in the context of smart cities, such as data aggregation, business analytics, and user interaction with data, and deploy these containers on the cloud. Kovatsch et al. [13] simplify the programmability of IoT applications by exposing scripts and configurations through a RESTful CoAP interface from a deployed container. In [14] authors explore the architectural aspects of containerization and examines the suitability of available containerization tools for multi-container deployment in the context of IoT gateways. By evaluating the performance, flexibility, and efficiency of these tools, it provides valuable insights for optimizing the infrastructure of IoT gateways. The paper [15] introduces an intelligent system for smart home environments that controls the network of nodes and services, activating or deactivating them based on predictions of the user’s service consumption. Recent research has investigated the use of containerization technology in smart home systems to improve the deployment, management, and scalability of IoT devices and applications. FOGHA [16] is a Fog-based Home Automation Platform that utilizes software services as Docker containers. It employs real-time containerized service deployment and task offloading across its computing nodes. Zhong et al. [17] created a SmartHome system with containerization technology to handle apps and devices, highlighting its benefits in terms of deployment and scalability. In [18], authors utilized Docker Swarm and Kubernetes

containers to manage smart home gateways, highlighting the advantages of modern containerization technologies for better scalability and deployment. In [19], authors developed a smartphone application for evaluating household appliances, highlighting the significance of user-friendly interfaces in managing smart home systems.

The paper [20] presents a container-based approach to facilitate the integration of smart home systems that are compatible with multiple vendors. The goal is to create a flexible and scalable solution to address interoperability issues between different devices and communication protocols used in smart home ecosystems. Paper [21] propose a collaborative processing algorithm strategy that integrates cloud and edge computing, utilizing Kubernetes for unified system deployment. Experimental results demonstrate that this approach offers improved operational efficiency compared to systems centered solely on cloud or single-edge computing, thereby better meeting the real-time requirements of smart homes.

In this context, our system is designed to integrate microservice concepts to implement a connected and intelligent smart home using Docker Compose. This approach allows us to manage different services independently, ensuring scalability, flexibility, and ease of deployment. Each component of the smart home is implemented as a distinct microservice, which can communicate with others through defined APIs. Docker Compose facilitates the orchestration of these services, enabling seamless integration and management across various devices in the smart home environment.

3 Proposed Approach

We propose a container-based architecture for our smart home systems, utilizing a modular approach that enables the independent development, testing, and deployment of services. This design enhances both agility and scalability. The implemented services include facial recognition, weather monitoring, and voice command, all orchestrated using Docker Compose. The adoption of containers provides a lightweight and isolated environment for running these services, encapsulating all necessary components to ensure consistency across different environments and facilitating seamless deployment. By containerizing microservices with container technology, we significantly improve the overall resilience and flexibility of IoT applications.

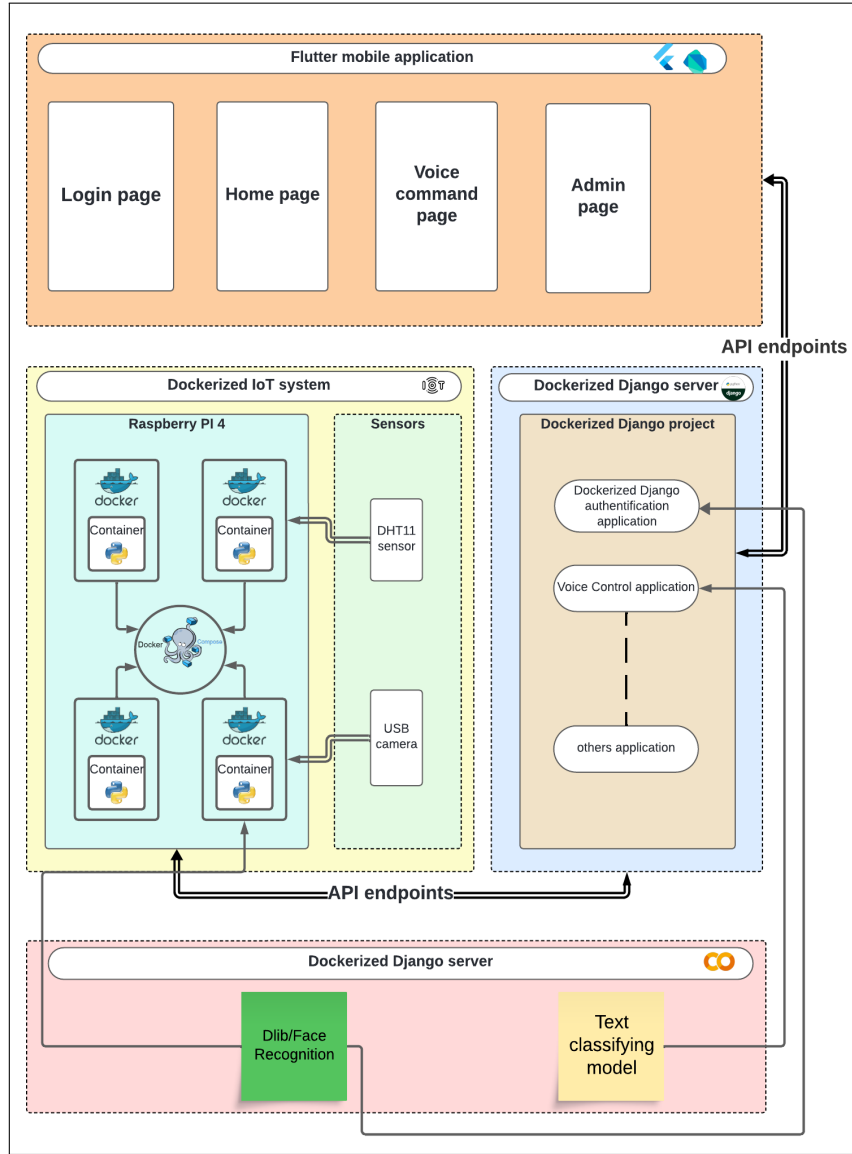


Fig. 1: Diagram hierarchical of SmartHome system

Our proposed approach includes the improvement of a coherent and comprehensive framework that brings together a Django backend server, an IoT framework, AI relapse for climate expectation, a Shudder versatile application, and coordination utilizing Docker Compose. We start by building a vigorous Django backend server, followed by its containerization to guarantee adaptability and ease of sending. Parallel to

this, we plan an IoT framework with point by point engineering that coordinates consistently with the backend, leveraging containerization to preserve consistency across various environments. Within the domain of AI, we center on building and assessing a relapse show pointed at giving exact climate forecasts, typifying the demonstrate in a holder to upgrade movability and unwavering quality. Nearby these specialized endeavors, we create a Shudder versatile application to offer clients an instinctive and lock in interface, covering all basic client cases and interfacing. To guarantee smooth and productive organization of the complete framework, we utilize Docker Compose. This permits us to oversee and facilitate the arrangement and operation of numerous holders, guaranteeing that all framework components work concordantly together. This coordination approach guarantees a cohesive, adaptable, and viable framework that leverages the qualities of each component. The chart underneath outlines the high-level engineering of our proposed framework. In outline, our approach combines backend improvement, IoT integration, AI modeling, mobile application advancement, and coordination to form a consistent and vigorous framework custom-made to supply dependable climate forecasts and user-friendly involvement. This comprehensive technique not only addresses the specialized prerequisites but also guarantees that the framework is adaptable, viable, and prepared for future improvements. All those specifications and functionalities are organized in the diagram that was illustrated in Figure 1.

4 Django backend server based on container

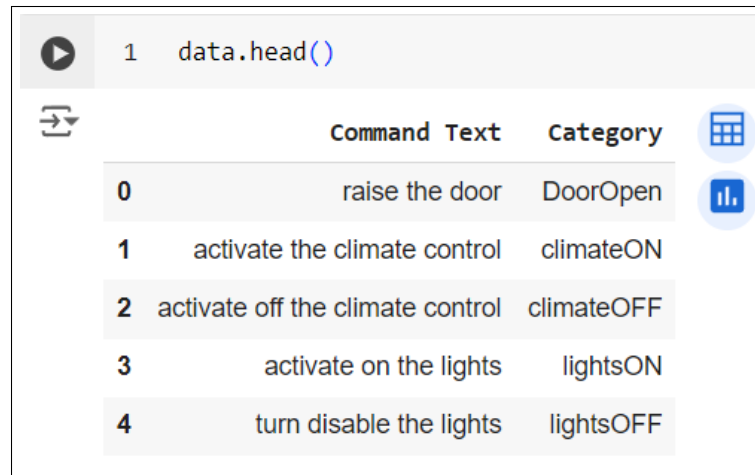
4.1 Building of Django backend server

In our method, we'll build a backend server using Django. Actually, Django is a high-level Python web framework that makes it possible to create safe, enduring websites quickly. In keeping with the "batteries-included" concept, it offers an extensive feature set and an ORM (Object-Relational Mapping) to facilitate the efficient development of reliable web applications by developers. At the beginning, we will create a project organized under the structure bellow : The structure of this Django project was developed with the aim of optimally managing data and user interactions. Here is an overview of the structure:

- **frame receiver:** contains code related to receiving frames from a real time face recognition frames.
- **media:** Typically used to store media files such as images, videos, or other static assets.
- **myapp:** includes code for fetching and displaying weather-related data.
- **myproject:** This is the main project directory containing core logic and configurations.
- **user authentication:** Contains code related to user authentication processes.
- **voice control:** includes code for handling voice control features.
- **weather:** includes code for fetching and displaying weather prediction related data.
- **db.sqlite3:** The SQLite database file used to store data for the project.

- **manage.py**: A command-line utility that lets you interact with this Django project in various ways.

Django applications are divided into multiple files, each with a distinct purpose, making code administration and maintenance easier. **admin.py** sets up the Django admin interface, and **apps.py** provides the application's fundamental setup. **models.py** creates models for database tables and optional **serializers.py** supports data communication in forms such as JSON with the Django REST Framework. **tests.py** guarantees that the application works properly, whereas **urls.py** defines URL paths for navigation. **Views.py** handles HTTP requests and responses, whereas database migration files are located in the **migrations/** directory. The **init.py** file identifies the directory as a Python package. This ordered structure improves the application's scalability and maintainability by isolating concerns and assigning explicit roles to each component.



The screenshot shows a Jupyter Notebook interface. At the top, a code cell contains the command `data.head()`. Below it, a table displays the first five rows of a dataset. The table has two columns: 'Command Text' and 'Category'. The rows are indexed from 0 to 4. To the right of the table, there are two icons: a calendar icon and a bar chart icon.

	Command Text	Category
0	raise the door	DoorOpen
1	activate the climate control	climateON
2	activate off the climate control	climateOFF
3	activate on the lights	lightsON
4	turn disable the lights	lightsOFF

Fig. 2: Text classifying dataset structure

Integrating speech recognition and text classification into a Django application involves setting SpeechRecognition library and using a text classification model saved as a pickle file within the voice control module. This integration enables users to interact with the application using voice commands, converting speech to text, and performing classification based on the recognized content.

In Django, a dedicated module handles speech input, where views manage the conversion process and utilize the pickle file to classify the recognized text accurately, ensuring a seamless user experience. Our text classification model was constructed using a text classification dataset. The structure of our dataset is shown in figure 2. After utilizing this dataset to train our pickle model based on logistic regression and classifying the SpeechRecognition result, the voice command is now ready for use and can be integrated into the Django voice control application.

4.2 Containerization of Django server

Although there are other solutions available for containerization, we choose to use Docker for our method. Docker is an open-source framework for containerization that facilitates the development, deployment, and management of lightweight, portable programs. Applications and all of their dependencies are packaged into these containers by Docker. The Dockerfile contains instructions on how to create Docker images that are used as container templates. Programs can operate in isolated settings thanks to these images, which also optimize deployment and scalability and guarantee consistency across many environments. In addition to increasing the software development and deployment process's effectiveness, security, and portability, this strategy works well for controlling a range of IoT devices and sensors in smart home automation systems. Algorithm 1 describes how to create and initiate

Algorithm 1 Creation of a Docker Container

```
1: procedure CREATECONTAINER
2:   Container Creation:
3:   Write a Dockerfile specifying OS, dependencies, and environment.
4:   Include necessary libraries and dependencies in the Dockerfile.
5:   Build a Docker image from the Dockerfile.
6:
7:   Application:
8:   Write an application that uses the included libraries and dependencies.
9:   Add the application to the Docker image.
10:
11:  Testing and Deployment:
12:  Test the image to ensure the application is working as expected.
13:  Run the image as a container on the chosen environment.
14:  return container                                ▷ Output: Container of the application
15: end procedure
```

After going over the entire procedure, let's get started with Dockerizing our Django project. Making a Dockerfile is the first step towards Dockerizing the Django server. The Django application build and run process in a Docker container is outlined in this file. It includes instructions on how to copy the application code, install dependencies, set up the environment, configure settings, and launch the Django server. For our Django application to be consistently deployed and portable across many environments, this Dockerfile is essential.

Let's get started on building a Docker image now. For consistent and portable deployments across various environments, we use a Dockerfile to specify the build parameters for our Django server.

Now, we build and execute a container that encapsulates the Django application's environment using a Docker image that complies with the Dockerfile specification.

The program is guaranteed to run reliably across many platforms and to retain all required dependencies and configuration thanks to the container.

5 IoT System based on container

Our IoT system's architecture is built on a Raspberry Pi 4 serving as the main controller. It incorporates a number of sensors, including a DHT11 sensor for temperature and humidity monitoring and a camera module for facial recognition, along with output devices, which are relays or LEDs that represent things like doors, windows, lights, and air conditioners. Through API endpoints, the Raspberry Pi and Django server can interact remotely and for monitoring purposes. The hardware configuration and its link to the Django server are shown in Figure 3:

5.1 Architecture of IoT system

Installing the required libraries and frameworks to allow communication and control between the hardware components and the central controller is part of the software setup for our Internet of Things system. To handle the camera, read data from the DHT11 sensor, and control the GPIO pins for the LEDs/relays, we install libraries using a package manager (apt-get for system packages, pip for Python packages). The Raspberry Pi 4 is used to implement the processing and control logic after the software has been configured. The camera module and the DHT11 sensor provide input data that the system processes and uses to carry out specific activities. The camera takes pictures, which are subsequently processed to identify and detect faces in order to perform facial recognition. Through the implementation of actions based on recognized faces, this can enhance security. Temperature and humidity data are periodically read by the DHT11 sensor, analyzed, and displayed to keep an eye on the surrounding circumstances.

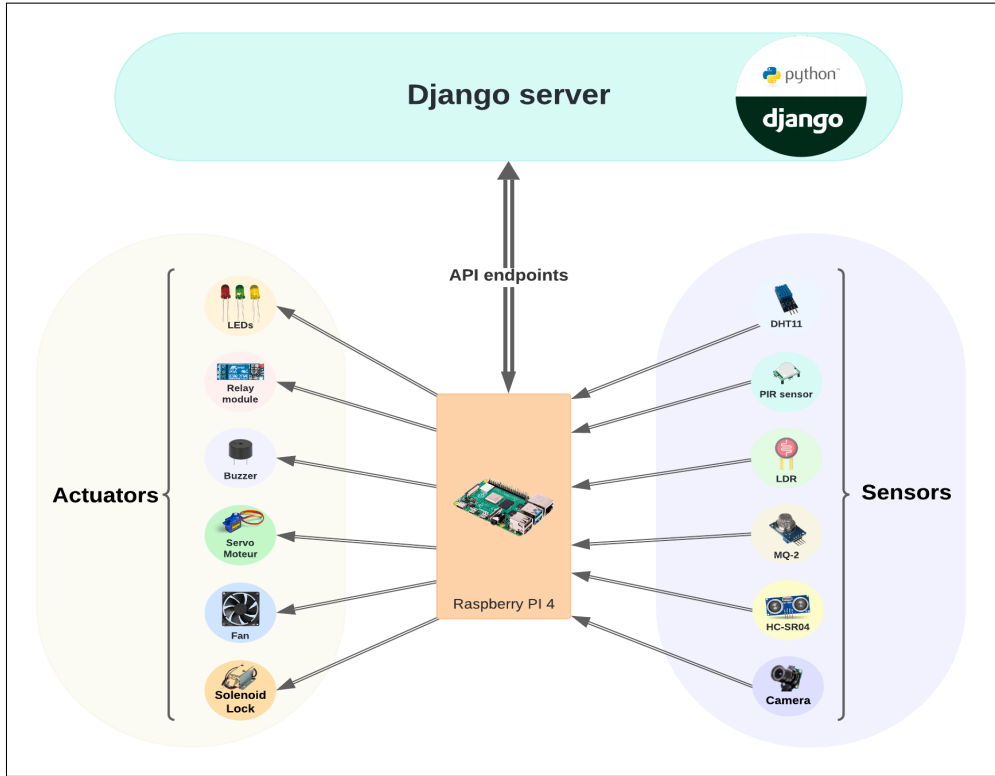


Fig. 3: IoT System Architecture

Additionally, the Raspberry Pi regulates the status of linked LEDs or relays, which stand in for appliances like windows, doors, lights, and air conditioning units. Receiving values from the Django server—which serves as the main hub for remote monitoring and control—determines these statuses. The Raspberry Pi modifies each device’s state by sending a query to the Django server’s API endpoints. The use of LEDs, temperature and humidity sensor, and a camera connected to a Raspberry Pi, serves merely as a prototype to validate our concept. However, this innovative idea has the potential to be scaled by deploying multiple sensors and actuators, allowing it to cover every aspect of a fully integrated smart home.

We plan to containerize the complete IoT setup in order to improve the deployment and scalability of our system. This entails putting the device state management, facial recognition, control logic, and sensor data processing of the Raspberry Pi into Docker containers. Containerization facilitates updates and scalability across several platforms, simplifies deployment, and guarantees consistent environments.

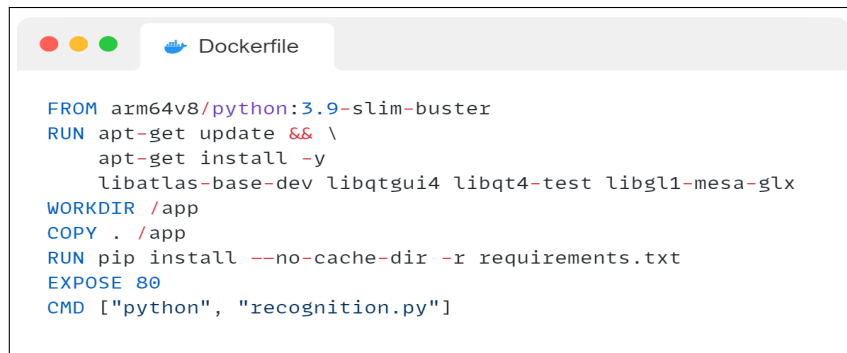
5.2 Containerization of IoT system

To containerize programs on our Raspberry Pi, we need to start by installing Docker with these three commands. This allows us to develop, deploy, and manage containers directly on the device, maximizing the efficiency and flexibility offered by containerization technology.

- `curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -`
- `echo "deb [arch=arm64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list`
- `sudo apt-get install docker-ce`

After installing Docker with the commands provided earlier, we'll be able to bundle our programs into lightweight, portable containers that run flawlessly on our Raspberry Pi.

In our IoT system, we have three different applications: a facial recognition application, a weather monitoring application, and a voice command processing application. We encapsulate each application in its own Docker container to ensure its isolation and independent management. This approach improves the modularity, scalability, and maintainability of the system. Figure 4 below shows the structure of the Dockerfile for the facial recognition application.

A screenshot of a code editor window titled 'Dockerfile'. The code is as follows:

```
FROM arm64v8/python:3.9-slim-buster
RUN apt-get update && \
    apt-get install -y \
        libatlas-base-dev libqtgui4 libqt4-test libgl1-mesa-glx
WORKDIR /app
COPY . /app
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 80
CMD ["python", "recognition.py"]
```

Fig. 4: Face Recognition Dockerfile Structure

The other Dockerfiles for the weather monitoring app and the voice command processing app follow a similar structure, customized for each app's specific dependencies and execution commands. We initiate the build process to create Docker images for each application and ensure that they contain all required dependencies and configurations. Figure 5 illustrates the build process for the facial recognition application Docker image and shows the terminal commands used to build the image.

```

ibrahim@raspberrypi:~/face-recognition $ docker build -t face_recognition .
Sending build context to Docker daemon  6.656kB
Step 1/7 : FROM arm64v8/python:3.9-slim-buster
--> bb1bad240fdf
Step 2/7 : RUN apt-get update && apt-get install -y libatlas-base-dev libqtgui4 libqt4-te
--> Using cache
--> 541ffc32300e
Step 3/7 : WORKDIR /app
--> Using cache
--> 0305c10e2247
Step 4/7 : COPY . /app

```

Fig. 5: Building Docker Image for Face Recognition Application

Docker performs the instructions supplied in the Dockerfile while building the Docker image for the facial recognition application. This comprises installing the necessary libraries, configuring the runtime environment, and launching the application. These stages ensure that the resultant Docker image is thorough and self-contained, allowing it to be deployed reliably across multiple settings. After building the Docker images for each application using custom Dockerfiles, we move on to creating and launching Docker containers. Docker executes commands to initialize the container based on the previously prepared image while it creates and launches a Docker container for the facial recognition software. This entails launching the application in the container environment, allocating resources, and creating network connections. By using the encapsulation that Docker containers offer, these procedures make sure that the facial recognition application operates reliably and safely. The process of creating and starting Docker containers for the weather monitoring application and the voice command processing application follows a similar structure. Each Docker container is configured to encapsulate its respective application, ensuring that they work independently while interacting seamlessly in the IoT system.

6 Flutter Mobile Application Development

Our strategy entails creating a Flutter mobile application that allows customers to communicate smoothly with their protected Smart Home system via containerization. This software offers a simple interface for monitoring and controlling home automation functions, including lighting, temperature, and security. By utilizing containerized apps, we ensure that each component runs independently and securely. The Flutter app provides real-time updates, customizable notifications, and insights, allowing customers to swiftly and conveniently control their Smart Home on both Android and iOS smartphones.

6.1 User cases diagram

To better understand the different interactions and activities in our Flutter mobile application, let's have a look at the use case diagram. The User and the Super User are the two types of actors that the program supports. Normal users may visualize both indoor and outdoor weather conditions, view weather forecasts, operate security cameras, and control home devices with voice and manual commands thanks to this software. However, Super Users have greater powers, including the ability to manage

user accounts and register new users. This broad range of features guarantees efficient interaction and management of the Smart Home system by administrators and end users alike. The interactions and workflows of the program are clearly illustrated in the use case diagram shown in Figure 6.

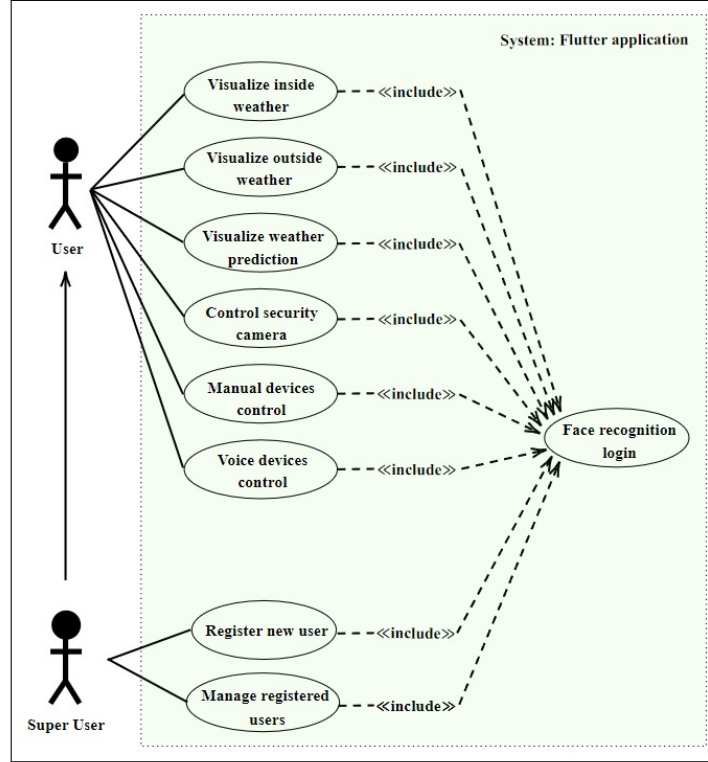


Fig. 6: User Cases Diagram

6.2 Implementation

Google launched Flutter, an open-source UI software development kit that enables developers to create natively built mobile, web, and desktop applications from a single codebase. Dart is a programming language developed by Google that is geared for creating quick and responsive applications on any platform. These two powerful technologies will allow us to deliver robust and feature-rich user experiences. To begin, we need to download and install the Flutter SDK on our PC. The Flutter SDK provides us with the necessary tools and frameworks for developing Flutter applications. Next, we use the following command line to start a new Flutter project.

```
flutter create sh
```

This command initiates the construction of the "sh" default Flutter application structure. It contains all of the necessary files and directories to provide us with a strong

framework on which to build our Flutter application. The structure of our Flutter project consists of various Dart files placed in the "lib" directory. These Dart files are critical for defining various components of our program, such as screens, models, services, and reusable widgets, which contribute to its functionality and user experience. In addition, a "pubspec.yaml" file exists in our project's root directory. This file serves as a configuration hub for our Flutter application, allowing us to handle dependencies and assets.

The final step in our development process is to run our software on a real device and evaluate its performance. Running the application on actual hardware helps us to see how it operates in a real-world setting, ensuring that all features function properly and the user experience is smooth. To deploy and launch our Flutter application on a connected device. The **flutter run** command compiles the application, installs it on the device, and launches it so that we may interact with it in real time. Testing the application on a physical device allows us to discover and address any potential performance, compatibility, or user interface issues that may not be apparent in the emulator. This stage is critical for delivering a powerful and dependable application to our users. Furthermore, a Dockerized Django server will be able to connect with our Flutter mobile application via API endpoints, enabling smooth data transfer and interaction between the mobile app and the backend server. We need to set up the Django server to handle Cross-Origin Resource Sharing (CORS) headers in order to facilitate this connection. This guarantees that the server may receive safe and permitted API queries from our mobile application, preserving the integrity.

6.3 Interfaces of Flutter application

This section describes the design of a Flutter mobile application, with an emphasis on four main pages: the login page, the home page, the voice command page, and the admin page. Each page has a defined purpose and is loaded with components that improve user experience and functionality. The Login page serves as a secure entry point for users, allowing them to authenticate and access the application. The Home page provides an overview of major features and information and serves as the primary center for user interaction. The Voice Command page allows users to operate numerous smart home devices with voice commands, increasing ease and accessibility. Lastly, super users are meant to use the Admin page, which gives them the ability to manage user registrations and keep an eye on system activity. As seen in Figure 7.

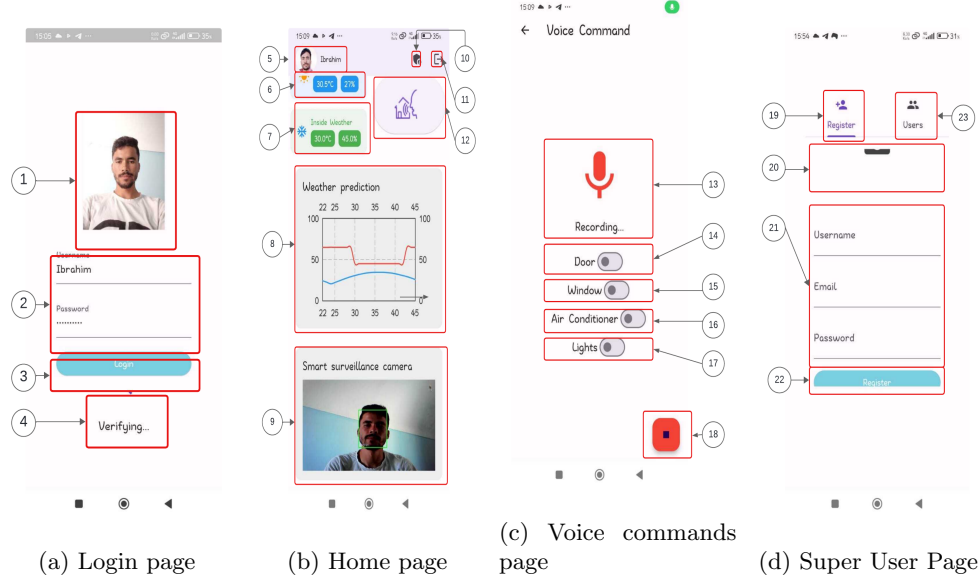


Fig. 7: Flutter mobile application interfaces

(a) Login page :

The Login page is the initial interface where users authenticate their identity. The components on this page include:

1. User Temporary Picture: Allows login using face recognition with a temporary picture.
2. Username and Password Fields: Input fields where users can enter their username and password.
3. Login Button: A button for submitting the login credentials.
4. Verifying Indicator: A status indicator that appears when the login process is in progress.

(b) Home page :

Once authenticated, users are directed to the Home page, which serves as the main dashboard. The components of this page include

1. User Greeting and Profile: Welcomes users with their name and profile picture, fostering a personalized environment.
2. Outside Weather Values: Offers real-time weather conditions outside, keeping users informed.
3. Inside Weather Values: Displays current indoor weather conditions, providing immediate environmental insights.
4. Weather Prediction Graph: Illustrates weather forecasts graphically, aiding users in planning ahead.

5. Smart Surveillance Camera: Enhances security monitoring by showcasing the status or live feed from a smart surveillance camera.
6. Superuser Button: Provides access for superusers to administrative functions, linking to the Admin Page.
7. Logout Button: Allows users to securely log out of their account.
8. Voice Command button: Enables users to control smart home devices by moving the voice command page .

(c) Voice commands page :

the Voice command is designed to allow users command and control several devices in our smart home using their voice or manually. The components of this page include:

1. Voice Command Indicator: Indicates that the voice command feature is active.
2. window switcher : Indicates the state of door and allow manual control of it.
3. Door switcher : Indicates the state of window and allow manual control of it.
4. Air Conditioner switcher : Indicates the state of Air Conditioner and allow manual control of it.
5. Lights switcher : Indicates the state of Lights and allow manual control of it.
6. Voice Command Recording Button: A button to start recording voice commands.

(c) Super User page :

The Super User page is designed for administrative users to register and manage users of the application. The components on this page include:

1. Register Tab: Navigates to the user registration section.
2. Choose Profile Picture: Allows users to select a profile picture from their phone.
3. Username, Email, and Password Input Fields: Input fields for entering user registration details, including username, email address, and password.
4. Register Submit Button: A button to submit the registration form and create a new user.
5. Users Tab: Allows viewing and management of existing users.

7 Orchestration using Docker Compose

7.1 Concept of orchestration

Docker Compose is a tool for orchestrating multi-container applications that uses YAML files to define services, networks, and volumes. Docker Compose is incredibly useful on the Raspberry Pi for controlling the startup, shutdown, and operation

of numerous containers as a single entity. This technique simplifies container lifecycle management, enables scaling, assures high availability, manages networking, and aids in service discovery. Docker Compose is ideal for our lightweight containerized applications due to its ease of use and efficacy in managing smaller deployments.

Figure 8 illustrates how Docker Compose orchestrates containers on the Raspberry Pi. The YAML configuration specifies services, networks, and volumes, allowing Docker Compose to manage containers seamlessly.

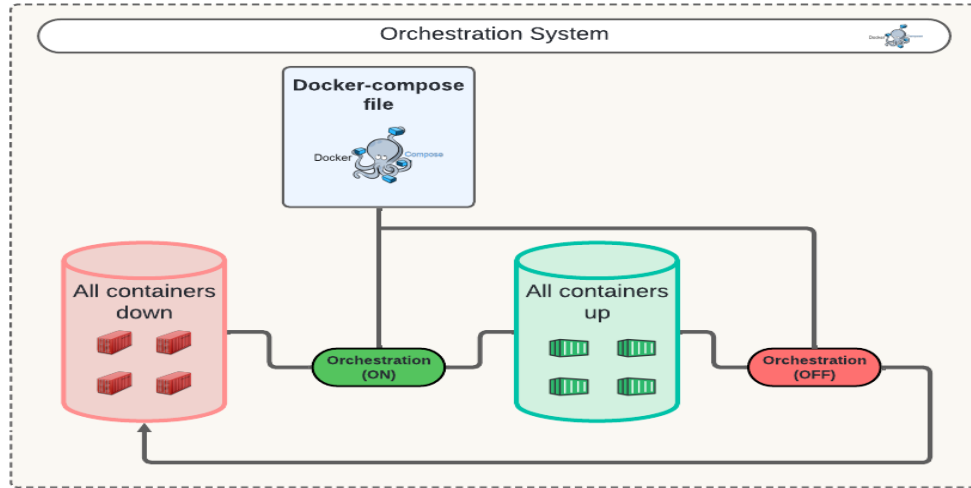


Fig. 8: Diagram of orchestration cycle

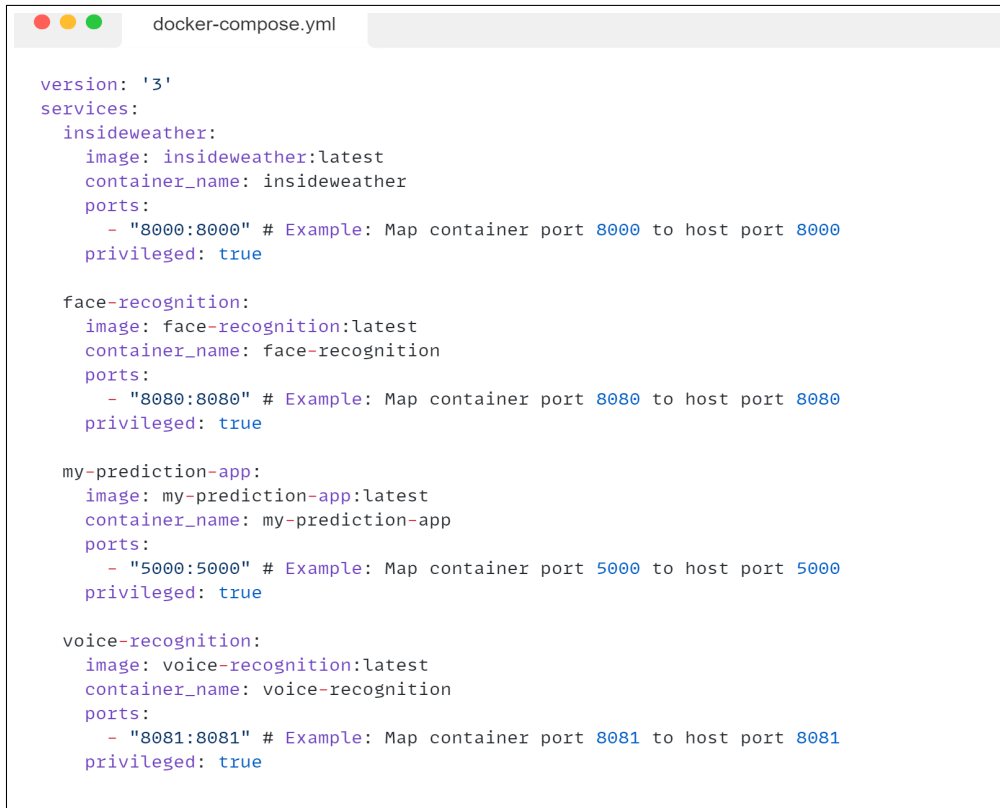
7.2 Implementation of orchestration system

In our approach, we use Docker Compose to orchestrate various services. The provided docker-compose.yml file outlines the configuration for four services: insideweather, face-recognition, my-prediction-app, and voice-recognition. Each service is defined with a specific Docker image, container name, port mappings, and is set to run in privileged mode. This setup enables seamless integration and communication between the services, ensuring efficient deployment and management, as illustrated in Figure 9.

7.3 Results of orchestration

After developing our Docker Compose file, deploying the orchestration process is simple. Docker Compose has two main modes: initiating and coordinating all containers defined in the YAML configuration, and gracefully ending them. This method makes it easier to manage multi-container applications on the Raspberry Pi by ensuring that services start in the correct order and that their dependencies are appropriately specified. Once deployed, Docker Compose monitors the containers' health and interactions, giving a unified interface for managing their entire lifecycle. This method improves

efficiency and reliability, making it suitable for managing our lightweight containerized applications.



```
version: '3'
services:
  insideweather:
    image: insideweather:latest
    container_name: insideweather
    ports:
      - "8000:8000" # Example: Map container port 8000 to host port 8000
    privileged: true

  face-recognition:
    image: face-recognition:latest
    container_name: face-recognition
    ports:
      - "8080:8080" # Example: Map container port 8080 to host port 8080
    privileged: true

  my-prediction-app:
    image: my-prediction-app:latest
    container_name: my-prediction-app
    ports:
      - "5000:5000" # Example: Map container port 5000 to host port 5000
    privileged: true

  voice-recognition:
    image: voice-recognition:latest
    container_name: voice-recognition
    ports:
      - "8081:8081" # Example: Map container port 8081 to host port 8081
    privileged: true
```

Fig. 9: Structure of docker compose orchestration file

7.3.1 Deploying and Verifying Docker Compose Orchestration

```
ibrahim@raspberrypi:~/Orechestration $ sudo docker-compose up -d
Creating network "orechestration_default" with the default driver
Creating my-prediction-app ... done
Creating voice-recognition ... done
Creating face-recognition ... done
Creating insideweather ... done
ibrahim@raspberrypi:~/Orechestration $ docker ps
```

CONTAINER ID	IMAGE	COMMAND	NAMES	CREATED
9d7da7db77e1	insideweather:latest	"python weather.py"	insideweather	18 seconds ago
b0ceb013ee79	my-prediction-app:latest	"python prediction.py"	my-prediction-app	18 seconds ago
05358469223b	face-recognition:latest	"python recognition..."	face-recognition	18 seconds ago
fa4d02549772	voice-recognition:latest	"python voice.py"	voice-recognition	18 seconds ago

```
ibrahim@raspberrypi:~/Orechestration $
```

Fig. 10: Deploying and Verifying Docker Compose Orchestration

Once we've produced our Docker Compose file, we can deploy the orchestration process by issuing a single command that starts the complete configuration. This command starts the creation and execution of all defined containers, ensuring that they are properly coordinated and begun in the correct order. After running the command, we can see the containers being created, and Docker Compose will send status updates as each service starts. To ensure that the procedure was successful, we can use Docker commands to list all of the containers that are operating on our system. This helps us to validate that all containers are operational as planned. The Figure 10 depicts the process of launching the Docker Compose configuration and verifying the status of running containers, giving a clear visual representation of the deployment process and its results.

7.3.2 Stopping the System Using Docker Compose Orchestration

After successfully installing and running our containers, there may be a need to shutdown the system. Docker Compose simplifies this operation. By using a simple command, we may gracefully terminate all operating containers, assuring a controlled shutdown. The Figure 11 depicts the command and process for halting the Docker Compose orchestration.

```
ibrahim@raspberrypi:~/Orechestration $ sudo docker-compose down
Stopping insideweather ... done
Stopping my-prediction-app ... done
Stopping voice-recognition ... done
Removing insideweather ... done
Removing face-recognition ... done
Removing my-prediction-app ... done
Removing voice-recognition ... done
Removing network orechestration_default
ibrahim@raspberrypi:~/Orechestration $ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS
```

Fig. 11: Stopping the System Using Docker Compose Orchestration

8 Conclusion

The primary goal of this paper was to showcase an integrated approach that combines innovation and versatility through strategic containerization of the Django backend, a robust IoT ecosystem, AI-driven weather forecasting, and facial recognition and voice command capabilities. The Flutter mobile app enhances user interaction, while Docker Compose orchestrates the seamless deployment of container components. This comprehensive approach boosts system reliability, user engagement, and future-proofing by continuously adapting to evolving technology trends and user needs. Our commitment is to deliver a consistent, optimized experience that meets the dynamic needs of our stakeholders.

As a forward-looking perspective, I propose to enhance our system by developing a cloud-native application through a CI/CD pipeline. This strategy will optimize the development process by facilitating continuous integration and delivery, thereby improving collaboration among teams and expediting the deployment of updates and new features. By adopting cloud-native principles, we can achieve greater scalability, resilience, and efficient resource utilization, ultimately elevating the performance and reliability of our smart home IoT solutions. This approach not only streamlines operations but also positions us to swiftly adapt to evolving user needs and technological advancements.

9 Data Availability Statement

The data that support the findings of this study are openly available:

- OpenWeatherMap API: Used for weather prediction systems. <https://openweathermap.org/api>
- Docker: Used for containerization to ensure consistent performance across environments by packaging applications and their dependencies into lightweight, independent containers. <https://www.docker.com/>
- Docker Compose: An orchestration tool to increase system scalability and stability by automating container deployment, scaling, and operation. <https://docs.docker.com/compose/>

- Django: A high-level Python web framework used for building the backend server. <https://www.djangoproject.com/>
- Flutter: A UI toolkit from Google used to build the mobile application with features such as facial recognition login, complete weather management, and voice commands. <https://flutter.dev/>
- SpeechRecognition library: Integrated into the Django application for handling speech input. <https://pypi.org/project/SpeechRecognition/>
- SQLite: Used as the database to store data for the project. <https://www.sqlite.org/>
- Logistic Regression Model: Used for text classification in the voice command system. https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
- dlib: Used for facial recognition and image processing in the mobile application. <https://pypi.org/project/dlib/>
- face_recognition: A library built on top of dlib for easier implementation of facial recognition functionalities in the mobile application. <https://pypi.org/project/face-recognition/>

10 Conflict of Interest

The authors declare that they have no conflict of interest.

References

- [1] Gomez C, Chessa S, Fleury A, Roussos G, Preuveneers D (2019) Internet of things for enabling smart environments: a technology-centric perspective. *J Ambient Intell Smart Environ* 11(1):23–43
- [2] Taiwo, O., Gabralla, L.A., Ezugwu, A.E. (2020). Smart Home Automation: Taxonomy, Composition, Challenges and Future Direction. In: Gervasi, O., et al. *Computational Science and Its Applications – ICCSA 2020*. ICCSA 2020. *Lecture Notes in Computer Science()*, vol 12254. Springer, Cham. https://doi.org/10.1007/978-3-030-58817-5_62
- [3] V. Patchava, H. B. Kandala and P. R. Babu, "A Smart Home Automation technique with Raspberry Pi using IoT," 2015 International Conference on Smart Sensors and Systems (IC-SSS), Bangalore, India, 2015, pp. 1-4, doi: 10.1109/SMARTSENS.2015.7873584.
- [4] Yar, Hikmat and Imran, Ali and Khan, Zulfiqar and Sajjad, Muhammad and Kastrati, Zenun. (2021). Towards Smart Home Automation Using IoT-Enabled Edge-Computing Paradigm. *Sensors*. 21. 10.3390/s21144932.
- [5] F. Alsuhaym, T. Al-Hadhrami, F. Saeed and K. Awuson-David, "Toward Home Automation: An IoT Based Home Automation System Control and Security," 2021 International Congress of Advanced Technology and Engineering (ICOTEN), Taiz, Yemen, 2021, pp. 1-11, doi: 10.1109/ICOTEN52080.2021.9493464.

- [6] Hamzah NA, Saad MRB, Ismail WZBW, Bhunaeswari T, Abd Rahman NZ (2019) Development of a prototype of an IoT based smart home with security system. *Journal of Engineering Technology and Applied Physics* 1(2):34–41
- [7] Hassaan Siddiqui and Ferhat Khendek and Maria Toeroe (2023), Microservices based architectures for IoT systems - State-of-the-art review, *Internet of Things*, 23,100854, <https://doi.org/10.1016/j.iot.2023.100854>.
- [8] B. Butzin, F. Golasowski and D. Timmermann, "Microservices approach for the internet of things," 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), Berlin, Germany, 2016, pp. 1-6, doi: 10.1109/ETFA.2016.7733707.
- [9] L. Sun, Y. Li and R. A. Memon, "An open IoT framework based on microservices architecture," in *China Communications*, vol. 14, no. 2, pp. 154-162, February 2017, doi: 10.1109/CC.2017.7868163.
- [10] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah and Y. Chen, "Orchestration of Microservices for IoT Using Docker and Edge Computing," in *IEEE Communications Magazine*, vol. 56, no. 9, pp. 118-123, Sept. 2018, doi: 10.1109/MCOM.2018.1701233.
- [11] A. Krylovskiy, "Internet of Things gateways meet linux containers: Performance evaluation and discussion," 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), Milan, Italy, 2015, pp. 222-227, doi: 10.1109/WF-IoT.2015.7389056.
- [12] R. Chesov, V. Solovyev, . Khlamov, and . Prokofyev, "Containerizedcloud based technology for smart cities applications," *Journal ofFundamental and Applied Sciences*, vol. 8, no. 3S, pp. 2638–2646,2016.
- [13] M. Kovatsch, M. Lanter, and S. Duquennoy, "Actinium: A RESTfulruntime container for scriptable Internet of Things applications," in*Internet of Things (IOT)*, 2012 3rd International Conference on the.IEEE, 2012, pp. 135–142
- [14] K. Dolui and C. Kiraly, "Towards Multi-Container Deployment on IoT Gateways," 2018 IEEE Global Communications Conference (GLOBECOM), Abu Dhabi, United Arab Emirates, 2018, pp. 1-7, doi: 10.1109/GLOCOM.2018.8647688.
- [15] Rego, A., Ramírez, P.L.G., Jimenez, J.M. et al. Artificial intelligent system for multimedia services in smart home environments. *Cluster Comput* 25, 2085–2105 (2022). <https://doi.org/10.1007/s10586-021-03350-z>
- [16] Ozmen HA, Isik S, Ersoy C (2021) A hardware and environment-agnostic smart home architecture with containerized on-the-fly service offloading. *Computers & Electrical Engineering* 92

- [17] Zhong Y, Yang H, Fan Y, Zhang Y (2017) SmartHome system based on containerization technology. In: 2017 5th International Conference on Mechatronics, Materials, Chemistry and Computer Engineering (ICMMCCE 2017), Atlantis Press, pp 724-730
- [18] Kang B, Jeong J, Choo H (2021) Docker swarm and kubernetes containers for smart home gateway. *IT Professional* 23(4):75-80
- [19] Moita CJ (2023) A consumer-oriented mobile application for assessment of household appliances. PhD thesis
- [20] J. Fleck, J. Sorgalla, F. Katzenberg and S. Sachweh, "A Containerized Template Approach for Vendor-Friendly Smart Home Integration," 2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Dortmund, Germany, 2023, pp. 352-355, doi: 10.1109/IDAACS58523.2023.10348797.
- [21] Ma, Q., Huang, H., Zhang, W., Qiu, M. (2020). Design of Smart Home System Based on Collaborative Edge Computing and Cloud Computing. In: Qiu, M. (eds) Algorithms and Architectures for Parallel Processing. ICA3PP 2020. Lecture Notes in Computer Science(), vol 12454. Springer, Cham. https://doi.org/10.1007/978-3-030-60248-2_24