ZilverHeat AI

Denis Timofeev Fornasov (2979756)
Adham Elhabashy (2879751) Hamza Elkady (2946912)
Kristiyan Spirov (2628015) Samer Saleh(2888327)

Supervisor: Alex Chiumento

Contents

1 Introduction

1.1 Problem Definition

Maintaining energy efficiency while ensuring indoor comfort is a key challenge in building management, particularly in office environments. Traditional, manually operated thermostats require continuous user intervention to maintain comfortable indoor temperatures. Occupants who forget to adjust temperatures while away or during non-working hours lead to inefficient heating. As highlighted by **Khalilnejad et al. (2020)**[1], a significant portion of the energy consumption in commercial buildings is dedicated to heating.

Smart thermostats are designed to address this problem. Unlike traditional thermostats, smart thermostats use sensors and presence prediction to automate and optimize temperature control. The Smart thermostat learns the preferences of the occupants over time, responds to environmental conditions and make energy-efficient adjustments automatically.

The report will further explain the research, design, and implementation process of the smart thermostat. We will begin by defining the requirements for a thermostat, followed by the exploration of hardware and software design choices. The subsequent chapters will delve into the specifics of how the system collects data, interprets user needs, learns from historical behavior, and integrates with existing infrastructure to improve efficiency while mainlining comfort.

2 System Design Plan

2.1 Initial Requirements

2.1.1 Necessary Requirements

The system must fulfill the following core requirements to ensure both energy efficiency and user comfort in an office environment:

- **Intelligent Preheating:** The system should be capable of preheating the office space in advance of occupancy. This includes learning and predicting optimal heating start times based on historical data, external temperature and occupancy patterns. The goal is to ensure the room reaches a comfortable temperature precisely when needed, minimizing energy waste from unnecessary early heating.
- **Maintenance of Desired Temperature:** Once the target temperature has been reached, the system must maintain it within a narrow margin of fluctuation. This involves dynamically adjusting heating levels in response to internal and external temperature changes, presence detection, and window/door state if available. Stability and responsiveness are key to maintaining thermal comfort without constant manual intervention.

2.1.2 Additional Requirements

While not strictly necessary for the system's core functionality, the following features can significantly enhance performance, user comfort, and energy efficiency:

- **Detection of Open Windows and Doors:** The system should ideally detect when windows or doors are open, either through dedicated sensors or by inferring from sudden temperature drops. This information can be used to temporarily pause or adjust heating to prevent energy waste. For example, if a window is open for ventilation, the system can avoid heating that area until it is closed, thus optimizing both cost and performance.

2.2 Initial System Design Plan

Based on the requirements, we devised this general system architecture, which incorporates all of the features previously mentioned.

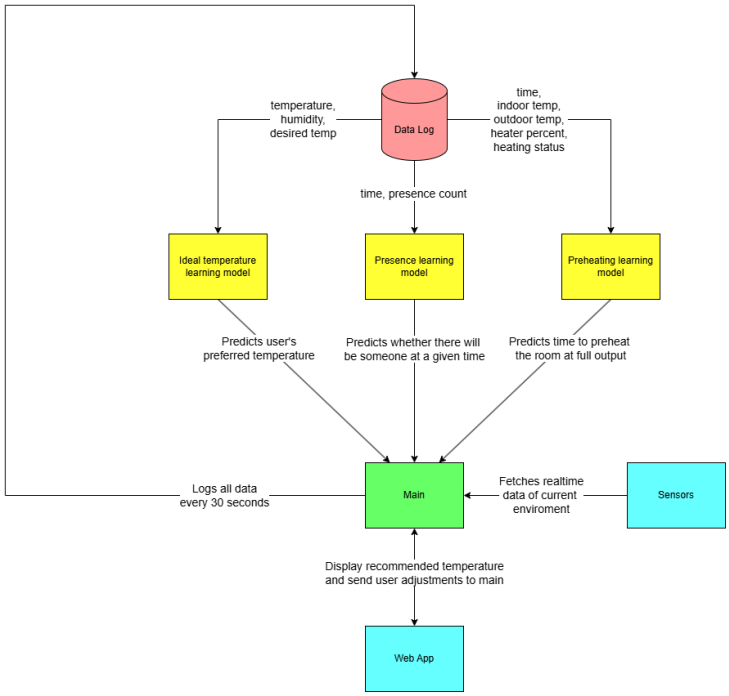

Figure 1: System Architecture using sensors.

The system consists of three machine learning models:

- Ideal temperature learning model: predicts the ideal temperature for the user of the system.

- Presence learning model: predicts the number of people that will be in a room at a given time.
- Preheating learning model: predicts how long it will take to heat the room to the desired temperature.

The main program uses the predicted time that people will arrive, the predicted time to heat the room, and the predicted ideal temperature to balance the heat in the room in a way that reduces unnecessary heating while still maintaining user comfort by making sure the room is adequately preheated. The web app allows the user to manually adjust the temperature and also displays the predicted ideal temperature for the user. The main program logs all of the sensor and actuator data every set period of time; we decided that 30 seconds was a reasonable compromise of plentiful data and scalable log file size. The detailed internals of the main program, web app and all the models will be discussed in the Software Architecture section (3.2).

2.3 Setbacks and limitations

Unfortunately, there were several setbacks, namely:

- **Sensor Malfunction:** The FP2 sensor did not function as expected, it failed to detect presence consistently and could not accurately count the number of individuals in a room. Additionally, we could only interface with it through the Aqara app, which is not suitable for development.
- **Thermostat Replacement and Limitations:** Our original thermostat, the Comet Zigbee model, was found to be defective. It was replaced with a Sonoff thermostat that could be controlled remotely. However, this device does not support direct access to the radiator valve openness level, restricting our ability to flexibly adjust openness levels.
- **Infrastructure Restrictions:** A major external constraint was the university's decision to disable all heaters in our testing environment (due to increasing outdoor temperatures). This prevented us from gathering real-world performance data and validating heating behavior under actual conditions.

2.4 Final requirements

Due to the previously mentioned setbacks, we had to adjust the necessary requirements to fit the new situation.

2.4.1 Necessary Requirements

The system must fulfill the following core requirements to ensure both energy efficiency and user comfort in an office environment:

- **Intelligent Preheating:** The system should be capable of preheating a simulated office space in advance of occupancy. This includes learning and predicting optimal heating start times based on simulated historical data, external temperature and occupancy patterns. The goal is to ensure the room reaches a comfortable temperature precisely when needed, minimizing energy waste from unnecessary early heating.
- **Maintenance of Desired Temperature:** Once the target temperature has been reached, the system must maintain it within a narrow margin of fluctuation. This involves dynamically adjusting heating levels in response to internal and external temperature changes and presence detection. Stability and responsiveness are key to maintaining thermal comfort without constant manual intervention.
- **Modularity for Actual Future Implementation:** The system must use a singular central data log and be modular in a way that adding further features or devices is simple for future developers. The intention here is to ensure a comfortable transition from the new simulation mode to using actual real world data and devices.

The key differences are that the system is no longer expected to run in the real world, but inside of a simulated environment instead. The newly added requirement ensures a smooth future integration to the real world.

2.5 Final System Design Plan

Based on the final requirements, we devised this general system architecture, which accounts for the setbacks (Figure 2).

Instead of fetching data from sensors, the main program now fetches data from the real-time simulation. The subtle difference of the arrow representing this action now facing both ways means that the simulation requires the main function to give it feedback data. The system also now requires supplementary data logs, this is to avoid interfering with the central data log. For example, the presence schedules for both training and testing will be generated ahead of time, and stored in a separate log which the presence learning model can read. In the real-world architecture, the presence learning model would simply read the central log and extract the data it needs, but plugging generated data for only some of the data values ahead of time into the central data log interferes with the secure structure of only allowing all values to be written into the log at once for a timestamp.

2.6 Test Plan

To ensure that the system meets the functional objectives, a testing strategy was defined during the design phase. This test plan outlines the approach taken to verify the correctness of system behavior, logic flow, and predictions.(subsection 4.1 explains the test cases in detail.)

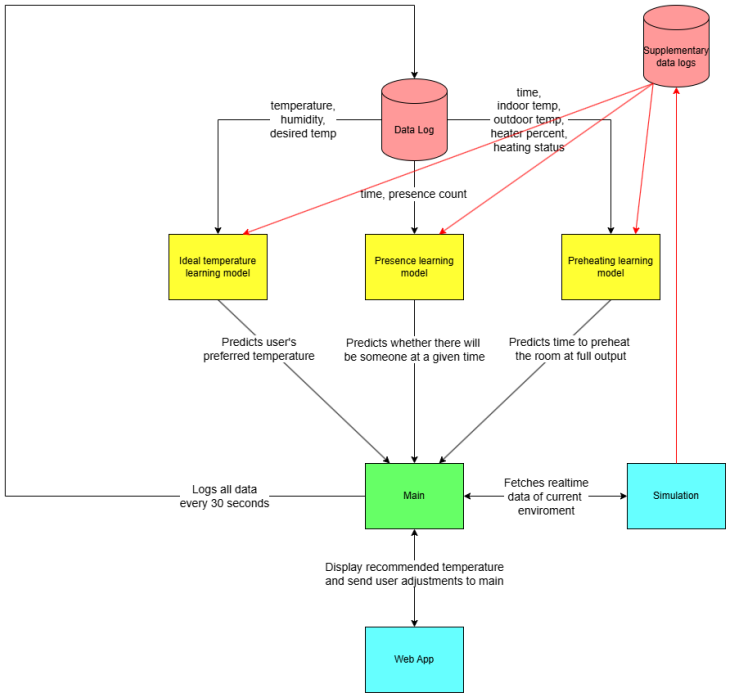

Figure 2: System Architecture using simulation.

Testing Objectives

- Verify the functional correctness of all core control logic components (e.g., PID controller, phase transitions).
- Validate the functioning of machine learning models (ideal temperature, presence prediction, preheating duration).
- Ensure proper system integration through simulation.
- Test endpoints of the web application for correct data exchange and user interaction.

Test Types and Scope

- **Unit Tests:** Individual components are tested in isolation using dedicated test scripts located in `test_scripts/test_logic/`.
- **Model Tests:** The machine learning models are tested to verify prediction output ranges, training consistency, and behavior under edge-case input data. These are also included under `test_scripts/test_model/`.

- **System Tests (Simulation Mode):** Complete end-to-end testing is done using simulated input data. This tests full pipeline behavior — from presence prediction to preheating, PID adjustment, and output logging.
- **Web App / Endpoint Tests:** Manual and automated endpoint testing is performed to ensure that:
  - Setting the temperature in the web app updates the system parameters correctly in the backend.
  - Getting the recommended temperature fetches valid model output.

3 Implementation

This section outlines the realization of the smart heating system. It is divided into hardware and software architecture, explaining the different components of the system, their integration, and the logic that drives the system functionality.

3.1 Hardware Architecture

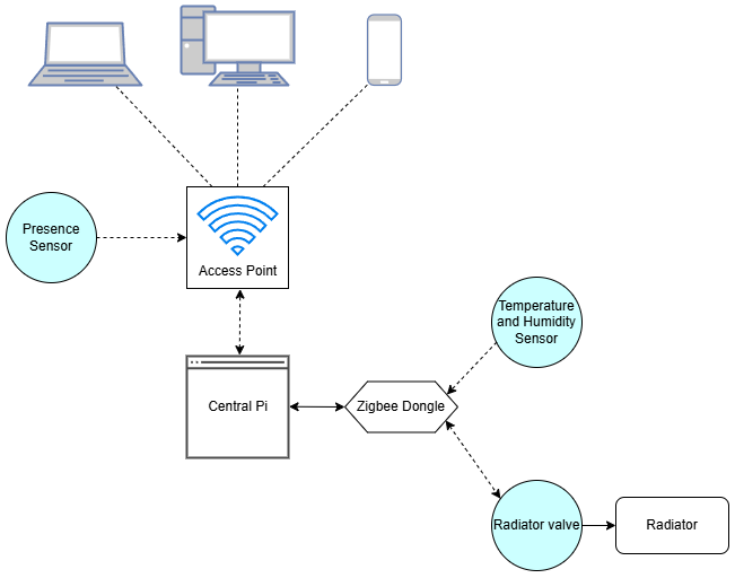

Figure 3: Hardware Architecture of the system. Dashed lines represent wireless connections, solid lines represent wired or otherwise physical connections.

3.1.1 Access Point

Sensors use different protocols, for instance, the temperature and humidity sensor uses the Zigbee protocol, which means it can be connected using a Zigbee

dongle without any issue caused by the university enterprise network. However, as seen in Figure 3, the Aqara FP2 presence sensor uses WiFi. We are using this sensor as an example, even though it was removed due to malfunctioning. Given that the University operates an enterprise network, inter-connectivity over WiFi between devices and sensors is often not possible or limited, despite connection to the university network intended for "Internet of Things" (IoT) devices. To circumvent this problem, the use of an access point is necessary. For the case that other sensors which use WiFi are connected in the future, we deemed it necessary to set up a wireless WiFi access point. We used a Raspberry Pi 4 running Raspberry Pi OS to set up an access point with internet sharing using *hostapd*[4], *dhcpcd*[5] and *iptables*[6].

In addition, the access point allows simultaneous connection between all devices which are connected to it. This means that, as developers, we could all wirelessly access the same Raspberry Pi from our laptops on the enterprise network. Also, for the future, the access point allows for hosting our the web app, which would also allow quick and easy connection through your smartphone. Since internet sharing is set up, being connected to the access point network will not interfere with the user's internet connectivity.

3.1.2 Central Pi

The Central Pi seen in Figure 3 above is where our main program would run in a real world implementation. It is entirely possible to run the main code on any device connected to the access point for the sake of development, but the Central Pi can be left permanently running during real world system deployment. The Central Pi is physically connected via USB to a Zigbee dongle, which allows communication with any devices which use the Zigbee protocol, in our case these are the temperature and humidity sensor, as well as the radiator valve. The Central Pi is connected wirelessly to the access point which allows communication with devices over WiFi.

3.2 Software Architecture

For our software implementation we used the Home Assistant operating system and Python along with the following dependencies:

- scikit-learn: for our machine learning models.
- joblib: for saving and reusing models.
- numpy and pandas: for mathematical operations and data preprocessing.

3.2.1 Structure and Organization

Given that one of the key requirements is for the system to be modular and scalable, we have devised a clean and organized structure for all of the components of the system. The components are organized in the following way:

- **app** directory – this stores boilerplate files for the web app, as well as *routes.py* which manages the API endpoints and background task, both of which will be discussed in detail in further sections.
- **controller** directory - this stores the *phase_controller.py*, *pid_controller.py*, *simulation_controller.py* and *thermostat_controller.py*. These are all modular components which are invoked by the sections of the codebase which need them. They will all be discussed in detail in further sections.
- **preheating, ideal_temp, presence** directories - these store the relevant files for each of the respective prediction models. Each of the directories have a learner file (*preheating_learner.py, ideal_temp_learner.py, presence_learner.py* respectively). Each of those files contain a function for training and another for predicting the respective property. The files can all be run individually if one wishes to force retraining. For example, *python -m presence.presence_learner* for training presence (note that simply running *python presence/presence_learner.py* may not work due to dependency on files in the root directory). The preheating and presence directories also contain the generator files which are used to create simulated data, these are stored as supplementary CSV files within the same directory.
- **models** directory - stores any models or scalars when trained so they can be loaded later on for prediction.
- **test_scripts** directory - contains scripts for testing the functionality of the system. Explained in further detail in the testing section.
- **config.py** - stores all configuration for the system:
  - **HASS_URL**: The URL for Home Assistant consisting of the IP and port number.
  - **TOKEN**: The long-lived access token for Home Assistant (obtained through the Home Assistant security settings).
  - **HEADERS**: The HTTP headers for API requests to Home Assistant.
  - **ENTITY_ID_{device}**: an entity ID for each device that is used (obtained through Home Assistant).
  - **name_to_entity_id**: a map of readable names such as "temperature" to the less readable and long entity ID for the sake of readability of the code.
  - **SYSTEM_TIME_INTERVAL**: interval at which the main system runs in seconds (this is how long the system sleeps after adjusting thermostat, logging data, etc.).
  - **ABSENCE_WINDOW**: window of time (in minutes) for PID to continue working even if there is nobody in the room (for short breaks where heating should stay on).

- **PARAMS_FILE**: file path for storing parameters adjusted by user input (currently just desired temp).
- **LOG_PATH**: file path for storing whole system data log (for training models).
- **PID_DEFAULT_KP, PID_DEFAULT_KI, PID_DEFAULT_KD**: default PID tuning parameters (explained in more detail in Section 3.2.10).
- **PID_INTEGRAL_UPPER_LIMIT, PID_INTEGRAL_UPPER_LIMIT**: PID integral cap values (explained in more detail in Section 3.2.10).
- **SIMULATION_MODE**: when true, the system runs on simulated data using artificial time (turn to false once all working sensors and actuators are in place). Note that some of the behavior with simulation mode off has not been tested due to a lack of functioning components.
- **SIMULATION_SLEEP_INTERVAL**: Sleep interval (seconds) for simulation, simply to slow down demo output. For example with the SIMULATION_TIME_INTERVAL at 30 seconds, and SIMULATION_SLEEP_INTERVAL at 0.1 seconds, the simulation will pass 30 seconds every 0.1 real life seconds (limited by computation time for very short intervals)
- **SIMULATED_PRESENCE_TRAIN_LOG_PATH**: path to simulated training data for presence (what the presence learner training function reads when in simulation mode).
- **SIMULATED_PRESENCE_TEST_LOG_PATH**: path to simulated testing data for presence (what the real-time simulated environment reads to get the current number of people in the room).
- **SIMULATED_PREHEATING_LOG_PATH**: path to simulated preheating cycle data (what the preheating learner training function reads when in simulation mode).
- **SIMULATION_DEMO_TRAIN_START_TIME**: starting datetime object of the simulated presence schedule for training.
- **SIMULATION_DEMO_TEST_START_TIME**: starting datetime object of the simulated presence schedule for testing.
- **LOWEST_OUTDOOR_TEMP, HIGHEST_OUTDOOR_TEMP, OUTDOOR_SIN_AMPLITUDE, OUTDOOR_SIN_VERTICAL_SHIFT, OUTDOOR_SIN_FREQUENCY, OUTDOOR_SIN_PHASE**: parameters for the simulated outdoor temperature sin function.
- **HEATER_COEFFICIENT**: coefficient of heater's contribution to temperature change each simulated temperature step.
- **OUTDOOR_LOSS_COEFFICIENT**: coefficient of outdoor loss contribution to temperature change each simulated temperature step.

  - **PRESENCE_HEATING_COEFFICIENT**: coefficient of a single person's contribution to temperature change each simulated temperature step.

- **data_log.csv** - central data log of the system. Stores the following fields:
  - **timestamp**: The current time when logging stored as yyyy-mm-dd-HH-MM-SS.
  - **indoor_temp**: The indoor temperature (room temperature, also referred to as *current_temp*).
  - **outdoor_temp**: The outdoor temperature (taken from weather API, not sensor).
  - **heater_percent**: The percentage openness of the heater.
  - **presence_count**: The number of people present in the room.
  - **humidity**: The humidity inside the room.
  - **desired_temp**: The desired temperature currently set.
- **data_logger.py** - file containing the function used to write data to the central CSV log file.
- **main.py** - starts the web app (which starts the background task that is the 'actual main' of the system).
- **params.json** - JSON file storing the current desired temperature of the user.
- **test_controller.py** - file for testing basic PID functionality.
- **utils.py** - file containing helper functions for interfacing with Home Assistant as well as fetching the current desired temp parameter from the json file.
- **visualization.py** - script to create a real-time visualization graph of the system.

3.2.2 Home Assistant

For our smart heating system, we used Home Assistant (HA) as the central hub for managing and integrating the peripherals. HA was installed on a Raspberry Pi 4B. The Raspberry Pi was configured to operate over Wi-Fi, connected to the Access Point (AP) to ensure reliable communication with the connected devices.

Once the HA environment was fully operational, we continued to pair all relevant sensors and actuators with the system. The paired devices included:

- **Aqara FP2 presence sensor** – used for detecting occupancy.

- **Sonoff thermostat valve** – responsible for controlling the radiator based on system commands.
- **Aqara temperature sensor** – used to monitor real-time room temperature and humidity data.

After successful pairing, we used HA API endpoints through Python to fetch sensor data and send requests to the thermostat valve. This setup was done before the setbacks previously mentioned. It is not used in any way during simulation, but it is important for future real-world integration.

3.2.3 Main

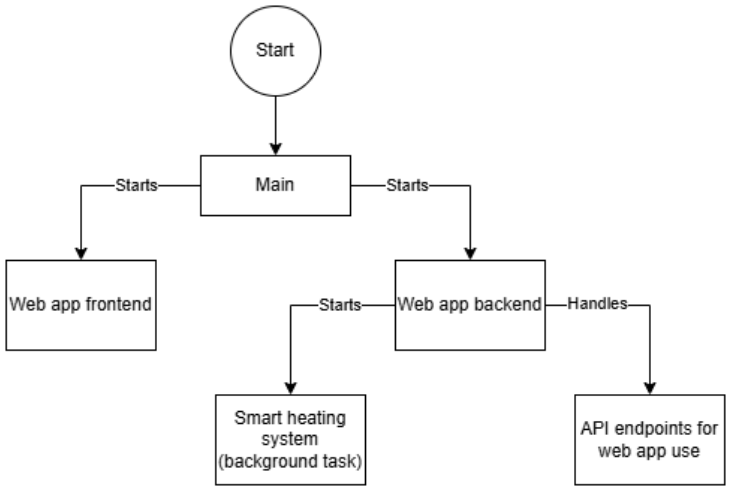

Figure 4: Main execution diagram.

When main is run (*python main.py*) it starts the front-end and back-end of the web app. The front-end is the UI for user feedback which will be discussed further in Section 3.2.4. The backend does two key things: API endpoints for the web app and a background task which runs in parallel to the web app. The background task is the smart heating system itself. This structure was chosen such that only a single program needs to be run, and both separate parts of our system run together.

Figure 5 shows the execution flow path of the background task, which is essentially the main program of the actual smart heating system if we ignore the web app. The program checks whether we are in simulation mode which is an adjustable parameter (in *config.py*). If we are in simulation mode, simulation variables are initialized and the simulated environment is used instead of actual sensors. The program also handles retraining of models which occurs every day, at the first time interval after midnight. The system time interval is an

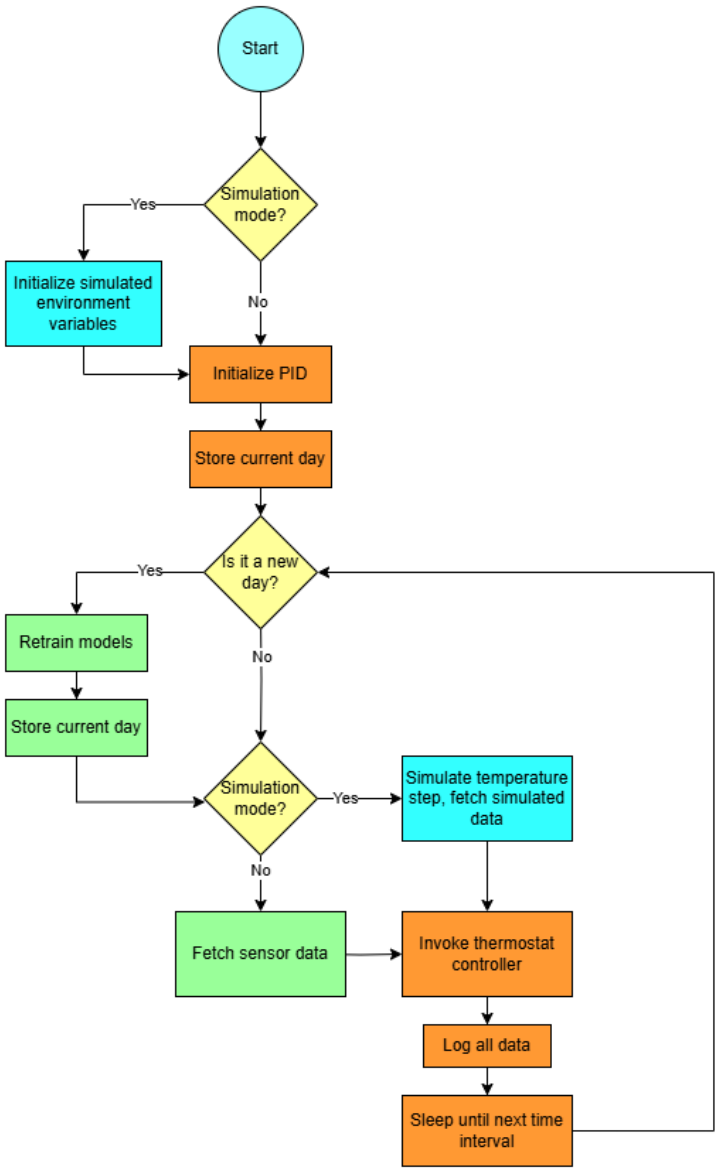

Figure 5: Background task execution path.

adjustable parameter (in *config.py*) which tells the system at which interval to perform its tasks; by default we have set this to 30 seconds which is short enough to capture temperature and presence changes, but long enough to not create immense log file sizes and redundantly call prediction functions. After either simulating environment changes or reading them if we are using real

sensors, the thermostat controller is invoked, which will be discussed in detail in Section 3.2.5. Then the system logs all sensor (or simulated) data and sleeps until the next interval, looping back to check whether it needs to retrain models. In simulation mode, the system logs its simulated environment instead of actual sensor data, but the structure is the same, so all the components (like the logger) do not care if we are in simulation mode or not, they operate the same way for modularity. Also, since in simulation mode we do not sleep and can simulate the next time immediately, there is an additional config parameter (as discussed in Section 3.2): SIMULATION_SLEEP_INTERVAL which gives a time for the system to sleep in simulation mode so that the output is more visible and doesn't happen incredibly fast.

3.2.4 Web App

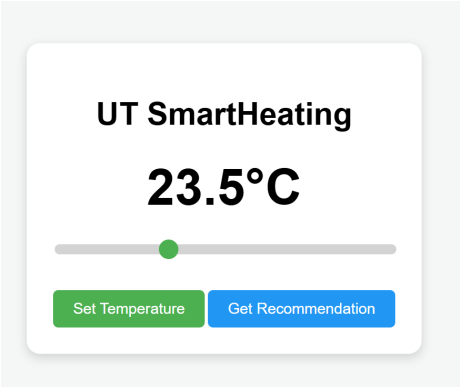

Figure 6: User Interface of the Smart Heating Web App

A simple Flask Web App was created to act as an intermediate layer between the system and users, providing a simple and intuitive interface to enable user feedback in the Smart Heating system. The interface displays the current target temperature "Desired Temperature" and allows users to adjust it using a slider. Two key actions are available to the user: setting a preferred temperature and receiving personalized heating recommendations.

API Endpoints Two Flask routes are used to support front-end interactions:

- `POST /set_temp`
  Triggered by the user pressing on "Set temperature" and receives a JSON object containing the desired temperature selected by the user. This value is rounded and saved via `set_desired_temp_param()`.

- `GET /get_recommendation`
  Triggered by the user pressing on "Get Recommendation" and displays a

prediction of the user's desired temperature temperature using the ideal temperature prediction model *subsubsection 3.2.8*

Behind the scenes, the system records user-selected temperature values, along with the indoor temperature and humidity. This data is continuously fed into a machine learning model designed to learn user preferences over time.

3.2.5 Thermostat Controller

As discussed in Section 3.2.3, the thermostat controller is invoked after fetching the environment data (whether simulated or not). The thermostat controller's *adjust_thermostat()* is called, taking as parameters:

- The current time.
- The phase controller instance (discussed further in Section 3.2.11).
- The desired temperature.
- The current temperature (indoor temperature).
- The outdoor temperature.
- The presence count.
- The PID instance (discussed further in Section 3.2.10).
- The current percentage openness of the heater.

Notice that the thermostat controller does not at all depend on whether the system is in simulation mode. It simply responds to the provided data and is unaffected by whether the system is running in the real world or a simulated environment. The system operates in three phases. The thermostat controller checks which phase is currently in action and responds as follows:

- **Idle Phase:** The system checks for current presence. If presence is detected, the system switches to *PID Phase*. If none is detected, it checks whether the desired temperature is already reached. If desired temperature is reached in *Idle Phase* that means the room is naturally warm enough, so the thermostat is set to 0 (percent openness). If there is no presence and the room is not warm enough, it calculates the heating duration (*time x*) required to reach the desired temperature. If occupancy is predicted in *time x*, the system transitions to the *Preheating Phase*; otherwise, the thermostat remains off (set to 0). Additionally, if the absence window has exceeded (stored in phase controller), the system will also not heat even if presence is predicted. This absence window is reset whenever presence is not predicted because this indicates a future presence block.

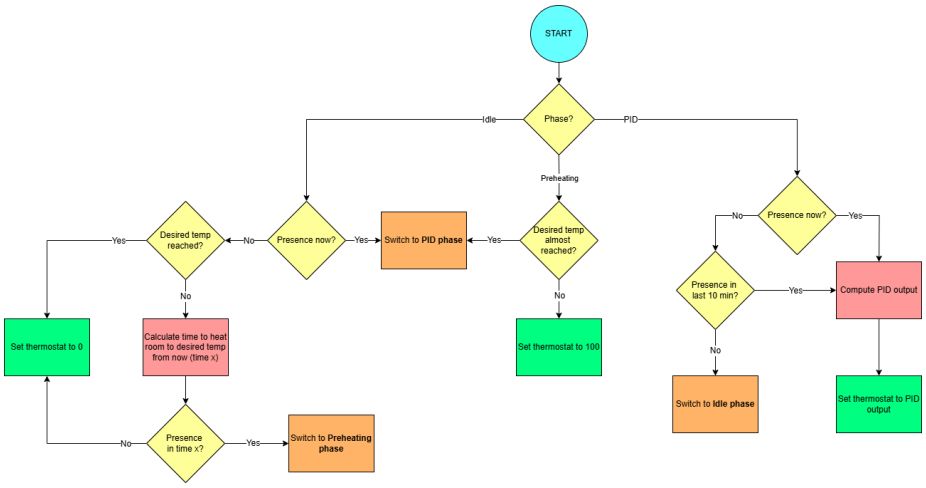

Figure 7: Thermostat Controller execution flow.

- **Preheating Phase:** If the desired temperature is almost reached, the system switches to *PID Phase*, otherwise the thermostat is set to 100 to heat the room as quickly as possible for the user's arrival. The 'almost' reached refers to a preheating margin (by default set to 2°C). This is set to allow a smooth transition between preheating and PID and avoid an overshoot.

- **PID Phase:** If some presence has been detected either right now or in the last 10 minutes (which is an adjustable parameter used to avoid letting the system cool during a short break), then PID is invoked and the thermostat is set to its output, PID will be discussed in further detail in Section 3.2.10. If there is no presence now and hasn't been any for the duration of the break window, the system switches to *Idle Phase*.

Operational Goals:

- *Timeliness:* By predicting occupancy in Idle mode and activating Preheating accordingly, the system ensures that the target temperature is reached right before the expected occupant arrival.

- *Energy Efficiency:* Full heating (100%) is only applied when necessary—during Preheating—and only if the room has not yet approached the target temperature.

- *Comfort:* The smooth transition from Preheating to PID mode guarantees that the room is comfortable at the moment of occupancy with minimal temperature overshoot.

3.2.6 Presence Learner

This model is trained on the entire data log of presence data. The features (X) are the timestamp (reduced to seconds from midnight and normalized using scikit learn StandardScaler to create a mean of 0 and standard deviation of 1) and the day of the week (inferred from timestamp but used as a standalone feature for emphasis on learning patterns based on the specific day of the week). The target variable is the presence count. The model used is the Random Forest Regressor for two key reasons:

- **Captures non-linear relationships**: The relationship between time and presence is not linear due to, for example, certain days having more absence than others. Random Forest Regressor is good at capturing these kinds of relationships when compared to linear models.
- **Robust to outliers**: Random and spontaneous absences should not affect the prediction greatly, because people may be absent for prolonged periods without pattern due to many reasons such as sickness, emergencies, etc. Random Forest Regressor is robust to outliers like these when compared to linear models.

The parameter *n_estimators* (number of decision trees in the forest) is set to 100 because this is a balanced average for both computation speed and accuracy.

3.2.7 Preheating Time Learner

The preheating time learner estimates how long it will take to heat the room to a desired temperature, given current indoor and outdoor conditions and the heater's power setting. This predicted duration is used in Idle mode to determine when to begin preheating so that the desired temperature is reached just in time for occupancy.

Unlike the other models, the preheating learner predicts a heating *rate* (in °C/sec) from past heating cycles and then uses that to compute how long it would take to reach the target temperature. The prediction is based on a simple linear model trained on simulated data.

Training Data and Feature Extraction Training samples are extracted by scanning the central data log for heating cycles—periods of continuous rising indoor temperature above a minimum slope threshold of 0.001°C/sec and a duration of at least 5 minutes. For each valid heating cycle, individual time steps are collected and stored with the following features:

- **indoor_temp**: Indoor temperature at the start of the time step.
- **outdoor_temp**: Outdoor temperature at the same moment.
- **heater_percentage**: Heater valve openness percentage (typically 100 during preheating).

- **delta_to_target**: Difference between the target temperature and the starting indoor temperature.

The target variable is the instantaneous heating rate (°C/sec) during that time step, calculated from the indoor temperature difference over the time delta between readings.

Model Choice and Justification A **Linear Regression** model was chosen for its simplicity and interpretability. Since the features (e.g., heater percent, outdoor temperature) are expected to have a roughly linear influence on heating rate in our simulated environment, this approach offers a good balance between accuracy and generalization.

Prediction Logic Once trained, the model is used to estimate how long it will take to heat the room from the current temperature to the target. The predicted heating rate is plugged into the equation:

$$\text{duration} = \frac{T_{\text{target}} - T_{\text{current}}}{\hat{r}} \cdot 1.2$$

Where $\hat{r}$ is the predicted heating rate and the factor 1.2 is a buffer to account for imperfect modeling and to avoid underestimation. If the predicted rate is negative or zero, the system returns infinity to indicate heating is not possible under those conditions. If the room is already warm enough, the function returns 0.

Limitations and Considerations This model assumes that heating rate is constant throughout the cycle and that all environmental influences are captured by the chosen features. It does not explicitly account for changes in presence, PID transitions, or heating overshoots. However, in simulation, where the heating behavior is relatively stable and consistent, this simplification performs well enough to support timely preheating.

3.2.8 Ideal Temperature Learner

This model is trained on the last two weeks of hourly-averaged historical data, capturing trends in how the user adjusts temperature in response to changes in humidity and indoor temperature.

The model used is Random Forest Regressor. Random Forest Regressor trains multiple decision trees and averages the output. There were 2 main reasons behind this model choice:

- **Training data is simulated**
  - Due to the lack of real world complexity, in simulated data, the chances of overfitting (to ideal conditions) increase. As a result, the model would generalize poorly. Training on multiple decision trees

and averaging them reduces the impact of one overfitting decision tree.

- **Real world systems rarely follow linear relationships**
  - People's preferred temperature does not increase or decrease at a constant rate based on environmental factors. Each decision tree in a random forest splits the data into small chunks using if-else rules, not linear equations. For example, 'If humidity greater than 70% and indoor temp less than 22°C, increase the target temp' or 'If humidity is low and indoor temp greater than 25°C, lower the temp slightly'. There are no assumptions on how features and outputs are related. As a result, more complex patterns can be captured.

3.2.9 Performance metrics after training

- **Mean Absolute Error (MAE):** Indicates the average prediction error in degrees. The lower the MAE, the better.
- **R^2 Score:** Shows how well the model can predict changes (variability). The closer the score is to 1 the better it is. This metric is generally useful to gauge whether a model is predicting what is expected. For example, with presence prediction our scores for different training set sizes are above 0.9, which does verify that weekends are predicted without presence, because if the weekend was predicted wrong that would be a large number of timestamps with wrong prediction. However, this doesn't gauge very well things like small error margins for preheating. A preheating cycle could take only around 15 minutes so, for the sake of this metric score, an error of 15 minutes out of a full day is nothing, but for the sake of a the preheating cycle, those 15 minutes are the entire duration of it, so we could have completely malfunctioning preheating with an R^2 of above 0.9. This is why the metric is useful but we still need to observe the system as a whole, discussed in Section 4.2.

3.2.10 PID controller

Using very simple control systems, too much or too little power is often used which results in over- or undershooting. This not only wastes energy but also creates a potentially uncomfortable environment for the user. PID is used to maintain the temperature steadily and minimize over- and undershooting while the user is present. First we need to discuss exactly how PID works in theory. The theoretical PID formula is the following[2]:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)d\tau + K_d \cdot \frac{de(t)}{dt}$$

Breaking the terms down:

- $u(t)$: **control output** (valve openness calculated by PID).
- $e(t)$: **error** at time t calculated by subtracting the current temperature from the desired temperature (setpoint).
- K_p : **proportional gain** - coefficient to adjust contribution of immediate error (direct response to current error).
- K_i: **integral gain** - coefficient to adjust contribution of accumulated error (reduces long-term drift).
- K_d: **derivative gain** - coefficient to adjust contribution of rate of change of error (dampens oscillations to reduce over- and undershooting).
- $\int_0^t e(\tau)d\tau$: **integral of error** (valve openness calculated by PID).
- $\frac{de(t)}{dt}$: **derivative of error** (valve openness calculated by PID).

For simplicity let's break down the formula into the 3 terms:

- $P(t) = K_p \cdot e(t)$ - this is the Proportional term. It responds to the current error. Meaning the returned heater percent will be greater if we are currently further away from the setpoint (desired temp).
- $I(t) = K_i \cdot \int_0^t e(\tau)d\tau$ - this is the Integral term. It responds to the accumulated error. Meaning the returned heater percent will be greater for a larger amount of accumulated error from previous invocations below the setpoint, and it will be lower for accumulated error above the setpoint.
- $D(t) = K_d \cdot \frac{de(t)}{dt}$ - this is the Derivative term. It responds to the change in error. This means that it will contribute to the adjust the heater percent based on whether the error is growing or shrinking, to counteract over- and undershooting.

Since we are using time steps in our program, we need to discretize each term for implementation in Python instead of using continuous time:

- $P(t) = K_p \cdot e(t) \rightarrow P_n = K_p \cdot e_n$: the time step is denoted by the subscript n, instead of a function of t for continuous time.
- $I(t) = K_i \cdot \int_0^t e(\tau)d\tau \rightarrow I_n = K_i \cdot (I_{n-1} + e_n \cdot \Delta t)$: this follows from the approximation of an integral to a summation. [3]
- $D(t) = K_d \cdot \frac{de(t)}{dt} \rightarrow D_n = K_d \cdot (\frac{e_n - e_{n-1}}{\Delta t})$: this follows from the approximation of a derivative from difference quotient. [3]

Now the PID calculation can be easily implemented and computed in Python without the need for any complex methods. The PID Controller is wrapped in an object-oriented class implementation. The class stores as attributes the following values:

- **setpoint**: The temperature that the PID should balance at. This is passed as the desired temperature at initialization.
- **kp**: The proportional term coefficient passed at initialization.
- **ki**: The integral term coefficient passed at initialization.
- **kd**: The derivative term coefficient passed at initialization.
- **integral**: Since the integral term is a summation of the current and previous term this is stored as an accumulating variable.
- **last_error**: The error of the previous PID invocation.
- **last_time**: The last time that PID was invoked. Used for dt calculation.

As seen in Figure 7, whenever the state "Compute PID output" is reached, the PID controller's *compute()* method is called. This call is given the current temperature (feedback value for PID) and the current time. PID then gives the output as calculated by the previously discussed formula. The integral limits that were previously mentioned in the structure and organization. The limits make sure that the integral doesn't accumulate a very large amount of error skewed in one direction due to, for example, presence appearing before preheating which results in PID accumulating a large amount of error below the setpoint. The PID Controller class also offers the ability to:

- Adjust the setpoint for when the user changes their desired temperature.
- Reset the PID, which clears the accumulated integral term, last error and last time for when the system begins idling to not accumulate a disproportionate dt.
- Forcefully set the integral term, last time and last error for priming the PID. Priming here means giving the PID some inertia by plugging in historical data that it never actually computed. This is used when switching from preheating to PID, giving it the time the preheating took and the temperature change that occurred such that it continues to match that rate instead of letting the temperature fall while accumulating error. Another case where priming is used is when the system is in the idle phase and it goes directly into PID. This means that the user appeared before they were predicted to and before any preheating happened. Here we predict (from preheating learner) the heating duration of the room from the current temperature to the desired temperature and plug that prediction time and temperature difference into the PID historical data.

3.2.11 Phase Controller

The Phase Controller is another wrapper class which has some simple state-control functions:

- Storing the current phase for easier global accessiblility of this value.
- Storing the last time someone was present to keep track of whether the break window has passed.
- Storing the latest preheating start time to prime PID when a transition from preheating to PID occurs.
- Storing whether the absence window has exceeded.

3.2.12 Simulation Controller

The simulation controller handles three key elements:

- **Simulating the room temperature change of a single time step**: the function *simulate_temperature_step()* takes as parameters the current (indoor) temperature, outdoor temperature, presence count and heater percent. It then calculates the heater effect by:

$$heaterEffect = HEATER\_COEFFICIENT \cdot \frac{heaterPercent}{100}$$

The current heater openness percentage (as a decimal) is simply multiplied by the heater coefficient which is an adjustable parameter in *config.py*. By default we selected a value of 0.25, which was selected semi-arbitrarily to give a reasonable heating rate of the room. Then the outdoor loss is calculated by:

$$outdoorLoss = OUTDOOR\_LOSS\_COEFFICIENT \cdot (outdoorTemp - currentTemp + 2)$$

The outdoor cooling loss depends on the current difference between the indoor and outdoor temp such that the room temperature tends to the outdoor temperature when the heating is off and nobody is present. The +2 is used to simulate the fact that the indoor temp never actually drops all the way to the outdoor temp due to insulation and ambient heating, this is a very oversimplified solution to this phenomenon and is only somewhat realistic for colder outdoor temperatures. The outdoor loss coefficient is also an adjustable parameter and is set by default to 0.01. Finally we calculate the contribution of presence to the heating very simply by:

$$presenceEffect = PRESENCE\_HEATING\_COEFFICIENT \cdot presenceCount$$

The presence heating coefficient is also an adjustable parameter set by default to 0.005. Finally the new temperature is calculated by:

$$newTemp = currentTemp + heaterEffect + outdoorLoss + presenceEffect$$

Notice something very important: the coefficients represent a change per time step, not per any actual amount of time. This means that, for example, a heater coefficient of 0.25 and a system time interval of 30 will

increase the temperature of the room by 0.25°C every 30 seconds. So, changing the system time interval requires changing the heater, outdoor loss and presence coefficients to fit that time interval.

- **Simulating the outdoor temperature**: the function *simulate_outdoor_temp_curve()* takes as parameter only the current time. It uses a simple sin curve to give the outdoor temperature based on the time of the day:

  $$outdoorTemp = AMP \cdot \sin\left(FREQ \cdot time\_factor - PHASE) - \frac{\pi}{2}\right) + SHIFT$$

  In this function *AMP*, *FREQ*, *PHASE* and *SHIFT* are all adjustable parameters (with extended names like *OUTDOOR_SIN_AMPLITUDE* for *AMP*) in config. They are currently set to a simple sin curve using

  $$AMP = (HIGHEST\_OUTDOOR\_TEMP - LOWEST\_OUTDOOR\_TEMP)/2$$

  $$SHIFT = (HIGHEST\_OUTDOOR\_TEMP + LOWEST\_OUTDOOR\_TEMP)/2$$

  $$FREQ = 2 * math.pi$$

  $$PHASE = 1/12$$

  Together with default:

  $$LOWEST\_OUTDOOR\_TEMP = 7$$

  $$HIGHEST\_OUTDOOR\_TEMP = 15$$

  This all creates a simple sin curve with a peak at 14:00 and a trough at 02:00 to represent the outdoor temperature throughout the day. The time factor is simply the current time of day normalized as a value between 0 and 1 by converting to seconds from midnight and dividing by the number of seconds in a day.

- **Simulating real-time presence**: The function *simulate_presence_count()* takes as parameter the current time and returns the presence count from the generated presence log (see Section 3.2.13 for presence log generation). Since there may be some time offset between generated presence and the simulated environment (for example, if longer time intervals are used to generate presence data to avoid redundant rows), presence at the exact time is not fetched, but rather presence at the nearest time to the timestamp.

3.2.13 Simulating Presence Data Generation

Presence data is generated using a hard-coded outline with added randomness. During the weekend there is no presence at all, during the weekdays the schedule works as follows:

- The start of the day (1 person present) is at 09:00 with an added range of -5 to +5 minutes.

- There is a lunch break (0 people present) at 12:00 with an added range of -5 to +5 minutes.
- Lunch break ends (1 person present) at 13:00 with an added range of -5 to +5 minutes.
- There is a meeting (3 people present) at 14:00 with an added range of -5 to +5 minutes.
- Meeting ends (1 person present) at 15:00 with an added range of -5 to +5 minutes.
- The day ends (0 people present) at 17:00 with an added range of -5 to +5 minutes.
- A random number of 1 to 3 short breaks, each 2 to 6 minutes long at random times from the start of the work day to the end.

3.2.14 Simulating Preheating Cycle Data Generation

To train the preheating time learner, we required realistic heating cycle data with known conditions and heating trajectories. Since real-world testing was not possible due to disabled radiators, we implemented a data generation script that simulates the thermal behavior of the room under a variety of controlled scenarios.

Each simulated cycle models a realistic heating sequence, consisting of:

- **Warm-up phase:** A short period with no heating effect, representing delay between activation and temperature rise.
- **Heating phase:** The room heats up due to the combined effects of the heater and outdoor temperature differential.
- **Overshoot phase:** The temperature continues rising briefly due to thermal inertia even after active heating ends.
- **Dissipation phase:** The room slowly cools down, modeling natural heat loss.

Each data row logs the `timestamp`, `indoor_temp`, `outdoor_temp`, and `heater_percentage`. The final CSV output is saved to a predefined file path for later use in model training.

Heating Step Model The temperature evolution is driven by a heating step function, similar in structure to the real-time simulation described in Section 3.2. It calculates temperature change per time step (ΔT) as:

$$\Delta T = \underbrace{\text{heater\_coefficient} \cdot \frac{\text{heater\_percent}}{100}}_{\text{active heating}} + \underbrace{\text{outdoor\_loss\_coefficient} \cdot (T_{\text{out}} - T_{\text{in}} + 2)}_{\text{heat loss to environment}}$$

This mirrors the equation used in the full simulation environment, ensuring consistency in thermal behavior between training data generation and system runtime. However, in the generator, the heating speed is optionally perturbed by a small random factor (1.0 to 1.2×) to simulate measurement variance and imperfect control.

Parameter Variation To diversify training data, we simulate cycles across a range of conditions:

- **Outdoor temperatures:** 5°C to 15°C
- **Starting indoor temperatures:** 10°C to 20°C
- **Heater setting:** Always 100% (full power preheating)

Each combination results in one full heating cycle, resulting in dozens of diverse trajectories. This allows the preheating learner to generalize across typical environmental conditions.

Purpose and Integration The generated logs serve as the primary training dataset for the heating rate model (see Section 3.2.7). By ensuring the same simulation logic is used in both training and real-time testing, we reduce domain mismatch between learning and deployment.

3.3 User Manual

To run the system simply first use *python visualization.py* if you wish to visualize the values as they are logged (the graph that was demonstrated in the demo), then run main using *python main.py*. Main can take the 3 PID tuning parameters, running without any parameters will result in default parameters of $K_p = 75, K_i = 0.1, K_d = 50$ which are tuned to the simulated environment, running with some but not all 3 parameters will result in an error since only overriding some parameters could create unexpected behavior (you can also directly change the default parameters in *config.py*). Note that if you use the visualization in simulation mode, it clears the data log when run. This is to avoid multiple graphs layering over each other because in simulation mode, you will likely reuse the same dates for multiple executions. When run with simulation mode set to false, the visualization will continue to show previous data as long as the CSV file is not manually cleared. The visualization shows:

- Time on the x-axis and temperature on the y-axis.
- The setpoint (desired temp) as a dashed orange line.
- The current temperature (indoor temp) as a solid blue line.
- The current outdoor temperature as a solid red line.

- The presence count by deeper shades of red background (white means no presence).

If you wish to generate presence or preheating data, it needs to be run manually.

- **Running Presence Generation**: You can run the file and pass as command line arguments the number of days to generate data for and the mode of either train or test. The initial dates for both train and test generation are, as previously mentioned, in *config.py* because this is cleaner than passing a full date as command line argument. The other values like the start time of the meeting are hard-coded in *presence/generate_presence_data.py*. Let's say we want to generate a week of training data (from the start date in config) and a single day of test data: run *python -m presence.generate_presence_data 7 train*, then run *python -m presence.generate_presence_data 1 test*. Note that it is best to avoid overlapping the train and test dates. There is nothing implemented to warn you or prevent you from doing so.
- **Running Preheating Cycle Generation**: This is run in the same way as presence generation and without any command line arguments using *python -m preheating.generate_heating_data*.

The models are, as discussed, retrained at midnight. But, if you wish to retrain the models manually you can do so.

- **Manually retraining presence model**: *python -m presence.presence_learner*
- **Manually retraining preheating model**: *python -m preheating.preheating_learner*
- **Manually retraining ideal temp model**: *python -m ideal_temp.ideal_temp_learner*

As previously discussed, system parameters are stored in *config.py*. All the parameters have been covered in Section 3.2.1. Changing them in this file adjusts them for all parts of the system that make use of those values. Before using the system with simulation mode off, or if you are changing the system within simulation mode, make sure to look over all of the parameters and adjust them accordingly. For example, if you wish to add a new sensor you must do the following:

- Pair the sensor with Home Assistant.
- Add the sensor's entity ID to *config.py* (both as just the ID and in the mapping).
- Make sure Home Assistant API related information is correct for your environment.
- Adjust *utils.py* however necessary if the data received from the sensor is of a different format than what is currently expected.

- The sensor can now be interfaced using exclusively functions from *utils.py*.

To use the web app, go to *localhost:8080* when main is already running. Beware that, especially if the simulation sleep interval is very low, setting the temperature can cause a concurrency failure when reading the json file. Restart the system if this happens. After adjusting the slider on the web app use the button to actually set the temperature, and use the recommendation button to receive the recommendation from the system which you can choose to accept or ignore.

4 Testing and Evaluation

4.1 Unit Test Cases

The following tests are designed to validate the core functionality of our control components, including the PID controller, thermostat logic, and phase management system.

We have two testing sub-directories, one for testing the ML models (Model testing) and the other for logic testing.

How to Run the Tests To execute the tests locally, follow these steps:

1. Open a terminal in the project's root directory `SmartHeating`.
2. Run the command below, replacing the path with the appropriate test script (e.g., `test_pid_controller.py`):

```
$env:PYTHONPATH="."; pytest .\test_scripts\test_logic\test_pid_controller.py -v
```

4.1.1 Logic Testing

PID controller test This test aims to verify the core functionality and the behavior of the PID controller in the system. This test covers four test cases:

- Supply maximum power when invoked for the first time.
- PID output is a valid floating-point number.
- Updating setpoints for adaptive control.
- Resetting the controller after a cycle has been computed.

This test class verifies the behaviour of the PID controller throughout its lifecycle of initialization, computation, tuning and reset.

Thermostat Controller test This test aims to verify the correctness of the thermostat control system in transitioning modes based on certain conditions:

- Transitioning from IDLE to PID when presence detected.
- Remaining in IDLE if no presence detected and desired temperature is reached.
- Switching from IDLE to PREHEATING when future presence is predicted.
- Transitioning from PREHEATING to PID when the room reaches desired temperature.
- Returning from PID to IDLE when no presence has been detected for a configured duration.

Phase Controller test This test class verifies the functionality of the phase controller. It manages the operational state transitions of a thermostat system, so the following six tests were made to verify its behavior:

- Initializing the controller in the `IDLE` by default.
- Rejecting initialization with an invalid phase.
- Accepting valid phase transitions (`IDLE`, `PREHEATING`, `PID`).
- Raising errors on invalid phase assignments.
- Storing and retrieving the last presence detection timestamp.
- Storing and retrieving the start time of a preheating cycle.

4.1.2 Model Testing

Desired temperature model test The goal of this test is to verify the validity of the output of the desired temperature model. It covers two cases:

- Prediction based on single input set.
- Prediction based on multiple sets of input.

This test ensures that the output is a positive, floating-point number and ensures that the model does not output abnormal results and can handle multiple predictions.

Presence detection model test The goal of this test is to validate the output of the presence detection model. It covers two cases:

- Prediction during the day.
- Prediction during the night.

This test ensures that it produces non-negative, appropriate results across different times of the day. It is expected that it outputs a number that is non-zero during the day, and zero at night, as the building should be closed at this time.

Pre-heating model test This test verifies the behavior of the heating duration prediction model. The scenarios tested are:

- Predicting the duration required to heat from a lower to a higher temperature under realistic outdoor conditions and heater output.
- Handling invalid case of when the target temperature is lower than the current room temperature.

In the first case, the duration is expected to be an integer value within a reasonable range (e.g., under 5 hours). In the second case, the duration should return zero indicating no heating is necessary.

4.2 Results and Observations

This section presents the results of the Smart Heating system under two different scenarios, demonstrating that the system behaves as expected in both cases.

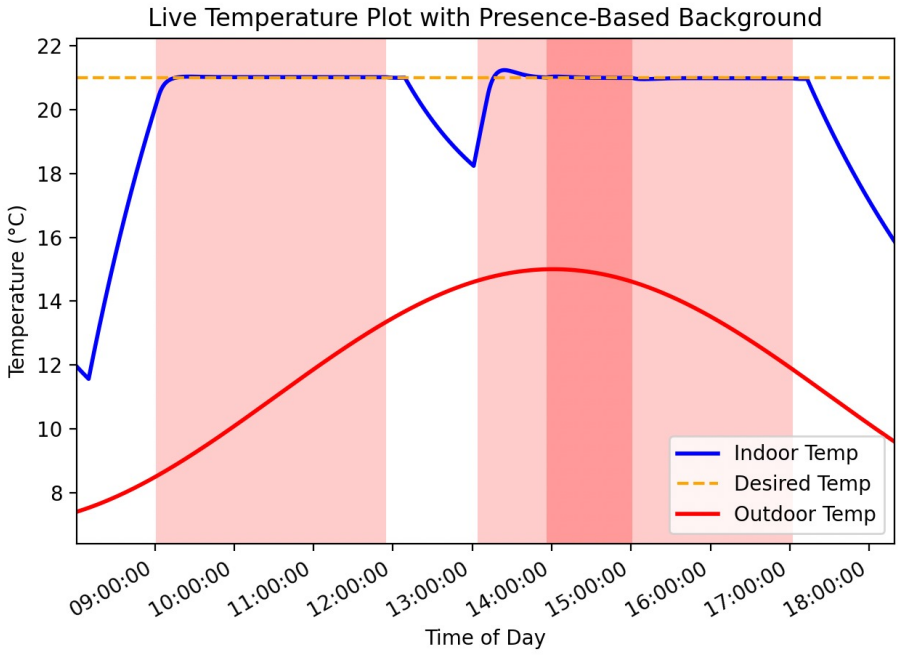

Figure 8: Result simulation of a full day without user interference.

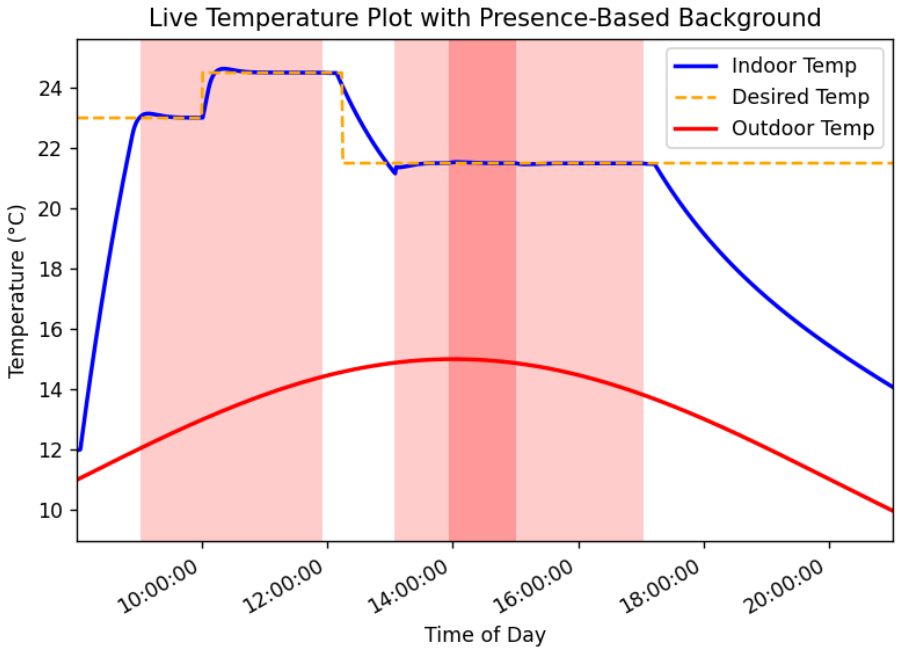

Figure 9: Result simulation of a full day with user interference.

Figures 8 and 9 illustrate the behavior of the smart heating system under two different scenarios: one fully autonomous, and the other with user interference.

Figure 8 shows the system operating without any user input. Anticipating occupancy around 10:00 am, the system initiates preheating and successfully reaches the desired temperature of 21°C before presence is detected. During occupied periods (highlighted in red), the system maintains temperature using PID control. When the user leaves around 12:00 pm, the system waits 10 minutes before heating it turned off due to the detected absence, then heating continues once presence is detected again. The darker red highlight indicates multiple people present, and the system continues to regulate the indoor temperature effectively, showcasing that the PID still operates correctly.

In contrast, Figure 9 demonstrates the system response when the user manually adjusts the desired temperature using the Webapp during the day. After the system preheats to the default setpoint, the user increases the temperature to approximately 24.5°C around 10:30 am. The system immediately adapts and maintains the new setpoint via PID regulation. Later, around 13:00 the user lowers the desired temperature back to 21°C, and the system again updates maintains the temperature successfully. As in the previous scenario, periods of multiple occupancy are handled appropriately.

These results show the system's correctness and that it operates reliably in

both autonomous and user interference modes, maintaining optimal temperature levels while adapting to changes in occupancy and user preference.

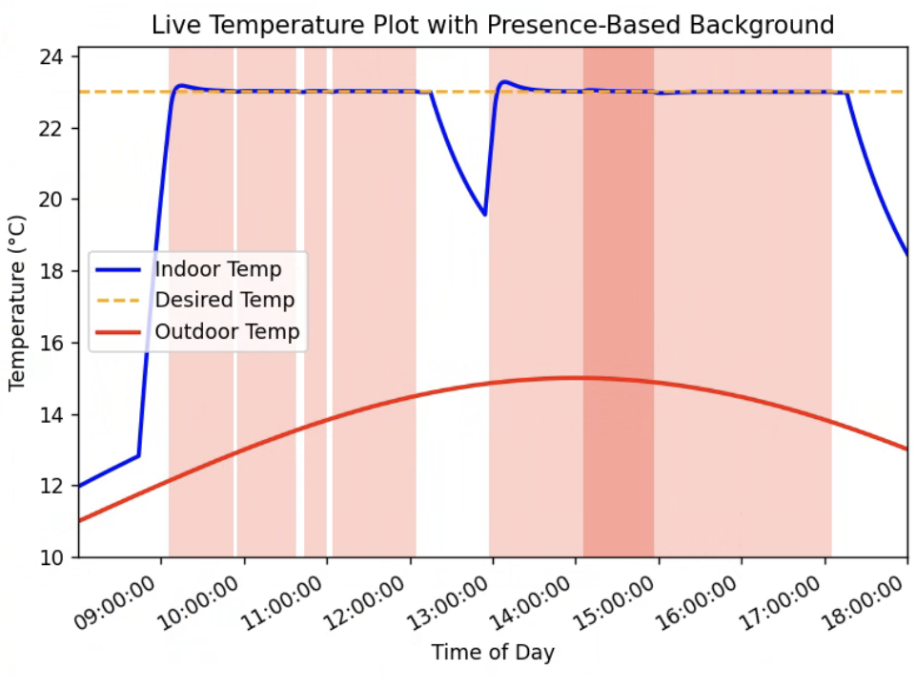

Figure 10: Alternative Schedule.

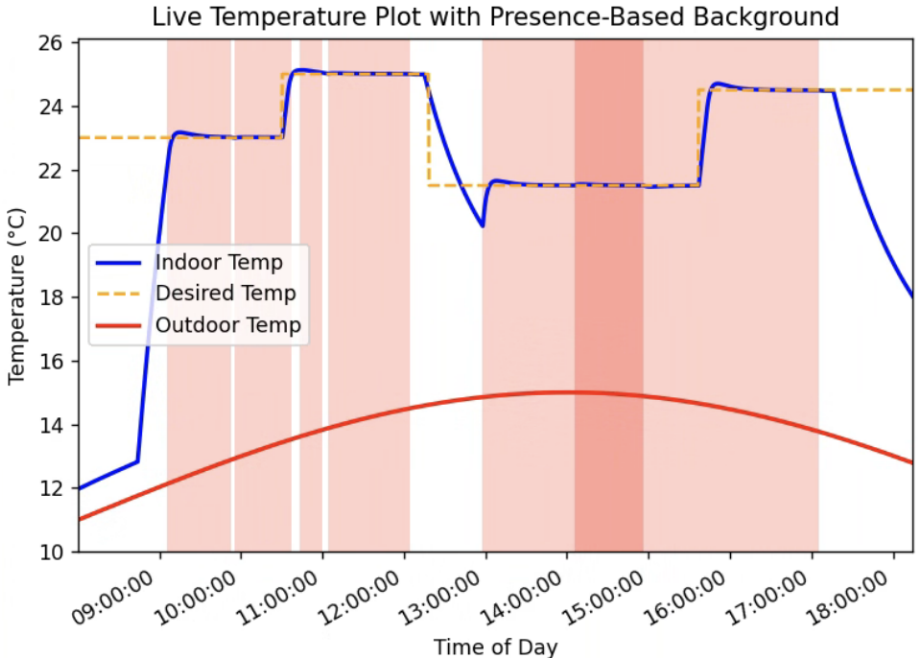

Figure 11: Alternative Schedule With Multiple User Changes.

Figures 10 and 11 shows scenarios of a different presence pattern than Figures 8 and 9. In Figure 10, the white gaps represent short intervals of absence typically under 10 minutes where no presence was detected. During these moments, the PID controller continues to handle temperature regulation without significant disruption. This highlights the system's stability and responsiveness over short absences. On the other hand, Figure 11 demonstrates how the system reacts to frequent manual adjustments. The thermostat responds rapidly handling user interventions effectively handled via the PID.

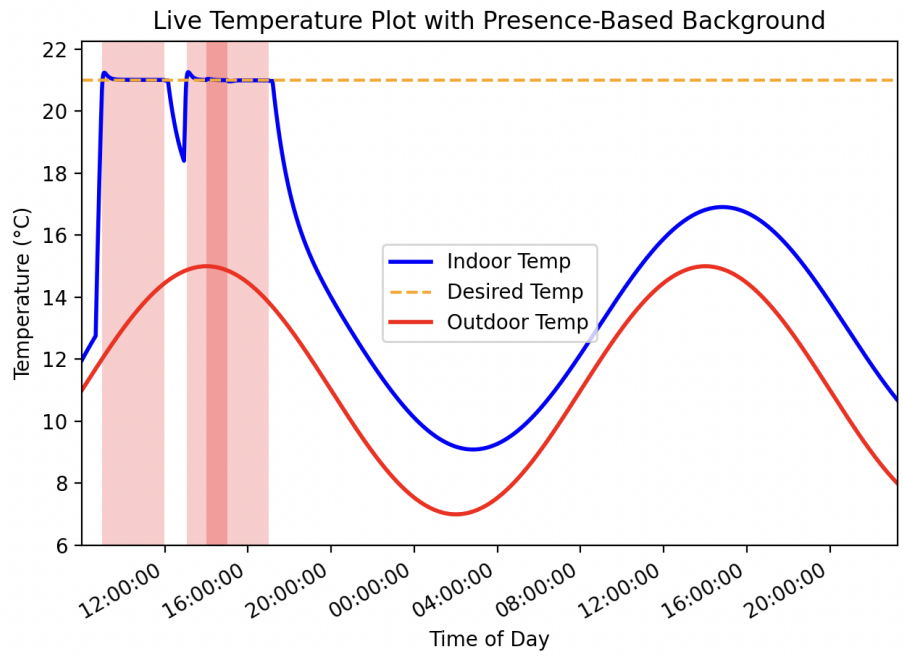

Figure 12: Result simulation of the beginning of a weekend.

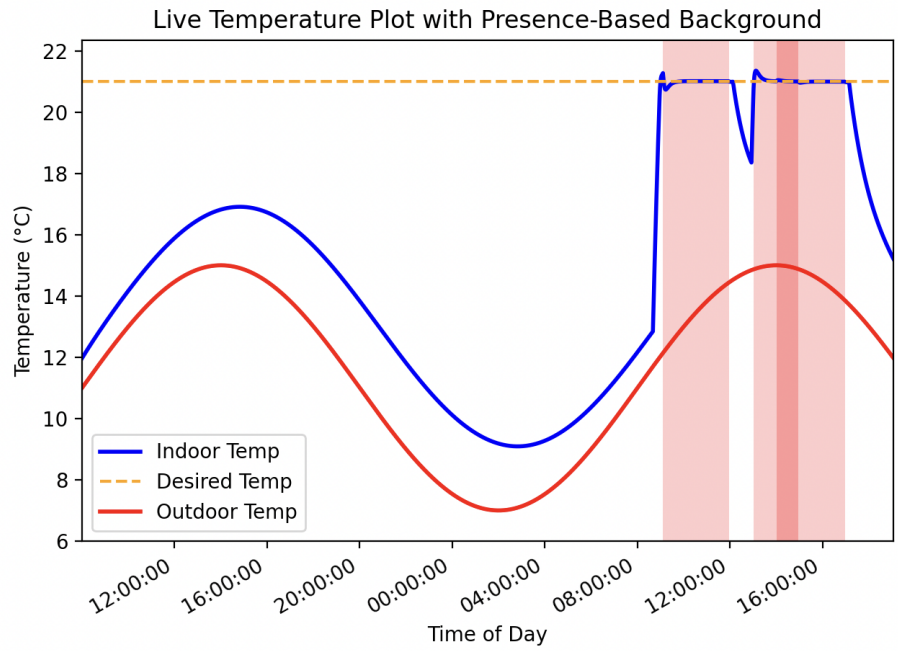

Figure 13: Result simulation of the end of a weekend.

Figure 12 and Figure 13 show the behavior of the system on weekends. Both figures are trained with presence data from 14 days and the exact same preheating data. Figure 12 shows the behavior of the system on a Friday and Saturday, demonstrating how the system predicts presence and preheats on the Friday but stays idle on the Saturday since weekends do not contain presence, as discussed in Section 3.2.13. Figure 13 shows the same behavior but in reverse, transitioning from a Sunday to a Monday.

If you wish to replicate this scenario you can do the following:

- Navigate to *config.py*
- Adjust *SIMULATION_DEMO_TRAIN_START_TIME* to *datetime(year=2025, month=4, day=7, hour=0, minute=0, second=0)*
- Adjust *SIMULATION_DEMO_TEST_START_TIME* to *datetime(year=2025, month=4, day=25, hour=0, minute=0, second=0)*
- Regenerate presence with 14 days train and 2 days test (following instructions in Section 3.3.
- Run the visualization and main program (following instructions in Section 3.3).

- Note that this is only an example scenario for the sake of explanation, the start times and sample sizes can be adjusted to anything.

5 Future Work

To enhance personalization, and real-world effectiveness of our system, the following improvements are planned:

- **Integrating Real Sensors:**
  - Connect a functional presence sensor and switch system to collect real-time environmental and usage data. In addition, deploy the system with a working radiator to log heating behavior.
- **Retrain Models on Live Data:**
  - In order to improve reliability, the models would have to be retrained on live data. This allows the models to adapt to real-world patterns, ensuring better generalization across different environments and user behaviors.
- **Improve Comfort Profiling:**
  - Incorporate real user feedback to fine-tune the prediction of personalized ideal temperatures.
- **Tune PID parameters:**
  - Adjust the PID parameters to fit a real-world environment.
- **Make it user friendly:**
  - Add options to the web app to adjust all parameters found in *config.py* to make the system actually usable for someone without opening the code.
  - Add the visualization graph to the web app such all information is seen in one area.
- **Add detection of open windows and doors:**
  - As we initially planned in our additional requirements, and as suggested by prof.dr. M. Huisman[7] at the poster presentations, add the detection of open windows and doors as this is a problem specifically in the Zilverling building with people leaving windows open.

6 Reflection

6.1 General Reflection

Overall, the project was a really valuable experience. It introduced us to new technical areas such as control systems and smart home automation which all of us had not explored before with our studies as TCS students. Although we faced several challenges, particularly due to hardware limitations and the shift to a simulated environment, it was an engaging learning process. We would have loved the opportunity to fully deploy and test the system in a real-world setting in Zilverling, as it would have provided deeper insight into user interaction and it would have been really nice to see it live in action.

6.2 Supervisor Communication

Throughout the project, we held weekly meetings with our supervisor Alex. Although some meetings had to be canceled or rescheduled due to availability issues, Alex was consistently supportive and flexible with timing. He was always willing to help us with any challenges we encountered and was really flexible throughout the whole thing. Toward the end of the project, we experienced some misunderstandings regarding the simulation environment, however, Alex stepped in and clarified the expectations and guided us on how to adjust our current implementation at the time. His involvement was crucial in helping us get back on track and deliver a final functioning product.

6.3 Team Collaboration

The entire team worked on sensor integration so that the system was properly connected and operational. Upon completion of this fundamental phase, we distributed the rest of the work according to the individual aptitude and preferences of team members, particularly for the software development phase.Overall, the team had good communication throughout the project without any major conflicts, also we held frequent meetings to ensure we are all on the same page alongside whatsapp communication.

- Denis worked on the controllers of the system and the presence learning model.
- Kristyan focused on building the machine learning model responsible for pre-heating prediction.
- Hamza and Adham worked together to create the ideal temperature machine learning model along with the supporting web application. Adham also worked on the unit tests of the logic and models used in the system.
- Samer worked mostly on the initial setup of the system and served as a backup, providing assistance to any team member who required it with their tasks.

The sharing of duties allowed us to function well and guaranteed that each area of the project received due attention and professional experience.

7 Reference list

References

[1] Khalilnejad, A., French, R. H., & Abramson, A. R. *Data-driven evaluation of HVAC operation and savings in commercial buildings*. Applied Energy, 278, 115505, 2020. https://doi.org/10.1016/j.apenergy.2020.115505

[2] Katsuhiko Ogata, *Modern Control Engineering*, 5th ed., Prentice Hall, Upper Saddle River, NJ, 2010.

[3] Steven C. Chapra and Raymond P. Canale, *Numerical Methods for Engineers*, 7th ed., McGraw-Hill Education, New York, 2015.

[4] Jouni Malinen, *hostapd: Wireless Access Point Daemon*, 2020. Available online: https://w1.fi/hostapd/

[5] Roy Marples, *dhcpcd: Dynamic Host Configuration Protocol Client*, 2020. Available online: https://github.com/NetworkConfiguration/dhcpcd

[6] Netfilter Core Team, *iptables: Administration tool for packet filtering and NAT*, 2020. Available online: https://www.netfilter.org/projects/iptables/

[7] prof.dr. M. Huisman, University of Twente (EEMCS). Available online: https://personen.utwente.nl/m.huisman