Master Lab IoT

Lukas Radovansky
Technische Universität München

20.1.2025

Contents

1 Background for grading (0.5 page)

- **General overview of your schedule including longer leave of absence (e.g., vacations) to explain missing data:**
  - Couple of multiday trips during the semester.
  - Christmas break from 15.12.2024 to 2.1.2025.
  - Unnoticed MQTT broker issue from 13.1.2025 to 18.1.2025.
- For more details, refer to Section 3.10.
- **Hardware problems:**
  - At the beginning of the semester, I encountered a problem with my ESPs that couldn't maintain a stable connection to the FRITZBOX router.
  - I spent a lot of time trying to fix the issue, but I couldn't find a solution.
  - In the end, I asked my landlord to give me the router credentials to debug the issue, which he refused.
  - Instead, he provided me with a small router that I connected to the FRITZBOX. This solved the issue, and I was able to continue with the project.
  - However, the router was not able to maintain a stable connection all the time and sometimes had to be restarted.
- **Credentials changed for the Digital Twin:**
  - There were no credentials changed for the Digital Twin app.

2 Your setup at home (1 page)

- **Scale map and sensor placement:**
  - Present the scale map with the placement of the sensor and the observed area.
- **Challenges from positioning:**
  - The Sensor in the kitchen triggered false positives if the door to the kitchen were opened and someone was in the hallway.
  - The sensor placed in bathroom could be affected by the steam from the shower, however the device seems to survive the humidity well.
  - I have deployed exactly one ESP with a Magnetic Switch and PIR sensor simultaneously in my apartment. It has been placed in the bedroom. The PIR sensor was pointing towards the bed area, and the magnetic switch was placed on the entrance door. The idea was to track the sleep time of the patient. If the doors remained open, the sensor was still active, and no bed area events were recorded. This was done on purpose because I know that every time the patient goes to sleep, he will close the door first.
- **Setup information:**
  - Sharing the flat with another person and a cat presented additional challenges in completing some tasks for this seminar.
  - Occasionally, a third person (the landlord) would be present in the flat once or twice a week to work from the crossed-out room on the apartment map. The landlord also used the kitchen, where one of the PIR sensors was placed.

– Throughout the semester, we hosted guests on several occasions (4-5 times), which impacted the data collection process. The highest number of guests was five, during the weekend of December 12-15, 2024.

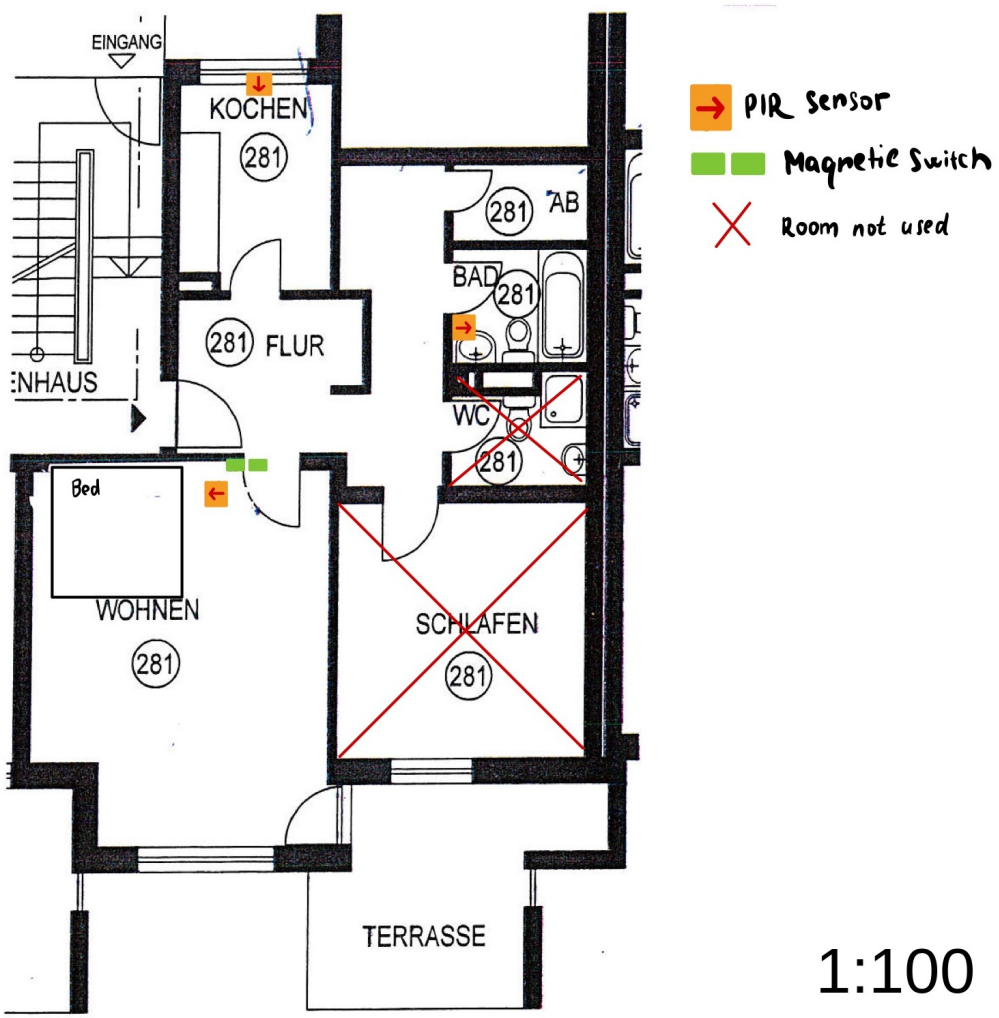

Figure 1: Scale map illustrating the sensor placement and the monitored area. The scale is 1:100. Red arrows indicate the orientation of the PIR sensors. Crossed-out areas are not part of the rented section of the flat.

3 Sensor Development (10 pages)

3.1 Challenges (0.5p)

- **Describe any aspect to consider while grading that impacted the development of your code, such as: programming skills, difficulties in understanding concepts, etc.**

- I had limited experience with embedded systems programming, which required additional time to learn and understand the given tasks.
- I was also new to Docker and Kubernetes.

3.2 Sensor Integration (0.5p)

- **Integration of PIR and Magnetic Sensors**
  The integration of PIR sensors with a magnetic switch sensor was achieved through the following steps:

  1. **Hardware Configuration:**
     - **PIR Sensor** connected to GPIO pin 27.
     - **Magnetic Switch Sensor** connected to GPIO pin 33.
  2. **GPIO Initialization:** Both sensors are configured as RTC GPIOs with pull-down resistors to ensure stable low states when inactive.

```
void configure_rtc_gpio() {
    // Initialize PIR sensor GPIO
    rtc_gpio_init(27);
    rtc_gpio_set_direction(27, RTC_GPIO_MODE_INPUT_ONLY);
    rtc_gpio_pulldown_en(27);

    // Initialize Magnetic Switch GPIO
    rtc_gpio_init(33);
    rtc_gpio_set_direction(33, RTC_GPIO_MODE_INPUT_ONLY);
    rtc_gpio_pulldown_en(33);
}
```

  3. **Wake-Up Configuration:** Both sensors are set as wake-up sources using the EXT1 mechanism, allowing the ESP32 to wake from deep sleep when either sensor is triggered.

```
uint64_t wakeup_pins = (1ULL << 27) | (1ULL << 33);
esp_sleep_enable_ext1_wakeup(wakeup_pins, ESP_EXT1_WAKEUP_ANY_HIGH);
```

  4. **Event Handling:** Upon wake-up, the system identifies which sensor triggered the event and processes it accordingly.

```
void handle_wakeup_reason(){
    if (wakeup_reason == ESP_SLEEP_WAKEUP_EXT1) {
        uint64_t status = esp_sleep_get_ext1_wakeup_status();
        if (status & (1ULL << 27)) {
            // Handle PIR sensor event
        }
        if (status & (1ULL << 33)) {
            // Handle Magnetic switch event
        }
    }
}
```

3.3 Single Code for All Sensors (1p)

- **Generic Code Base for Multiple Sensors**

– **Describe how you achieved a generic code base for all your sensors devices. You can include code snippets.**

  * Generic code for all devices was achieved by identifying the device based on its MAC address and configuring it accordingly.
  * Each device needs to be specied in the ‘main.h‘ file with its device name, MAC address, ID, MQTT topic, security key, battery information availability, and room ID.

```
//Struct definition for device information in main.h
/**
 * @brief Represents the configuration and metadata for a device.
 *
 * This struct is used to define the properties of an ESP device, including its
 * name, MAC address, ID, MQTT topic, security key, and whether battery information
 * is available. The struct can be used to identify devices and manage their specific
 * configurations within the system.
 */
 typedef struct {
    char* device_name;              // < Name of the device (e.g., "Living Room").
    uint8_t mac_address[6];         // < MAC address of the device (6 bytes).
    int device_id;                  // < Unique identifier for the device.
    char* device_topic;             // < MQTT topic for publishing device data.
    char* device_key;               // < Security key for authenticating with the MQTT broker.
    bool battery_info_available;    // < Indicates if the device provides battery information.
    char* room_id;                  // < Id of the room as named in the InFlux database.
} device_info_t;


// Example of definition of the device in main.h
// (name, mac adress, topic, key, battery info available, room_id)
#define ESP_DEVICE_1 {"Living Room", {0xEC, 0x62, 0x60, 0xBC, 0xE8, 0x50}, 4,
"1/4/data", "key", true, "livingroombedarea"}

// Read the MAC address and identify the device in the main function:
uint8_t mac_address[6];
esp_read_mac(mac_address, ESP_MAC_WIFI_STA);
identify_device(mac_address);

// The implementation fo the identify_device function in main.c
identify_device(mac_address);
/**
 * @brief Identifies the current device based on its MAC address.
 *
 * Matches the MAC address of the device with the pre-configured device list in `main.h`.
 * Sets the device's ID, MQTT topic, security key, and battery information availability.
 *
 * @param mac_address Pointer to the MAC address array of the device.
 */
void identify_device(const uint8_t* mac_address) {
```

```
    for (int i = 0; i < sizeof(ESPs) / sizeof(ESPs[0]); ++i) {
        if (memcmp(mac_address, ESPs[i].mac_address, sizeof(ESPs[i].mac_address)) == 0) {
            // Copy the device info into this_device
            memcpy(&this_device, &ESPs[i], sizeof(device_info_t));
            // Logging
            ESP_LOGI("*", "********** Device identified as %s", this_device.device_name);
            return;
        }
    }
    ESP_LOGI("*", "Device not recognized.");
}
```

3.4 Wake-up stub (2p)

3.4.1 Key Points

- **Sensor Trigger Filtering:** It checks if the sensors are newly triggered or still active. If the trigger is repetitive within a short time window, it returns to deep sleep without a full wake-up.

```
if (my_rtc_time_get_us() / 1000000 - last_wakeup_RTC <= SENSOR_INACTIVE_DELAY_IN_WAKE_UP_STUB_SEC) {
        last_wakeup_RTC = my_rtc_time_get_us() / 1000000;
        ets_delay_us(1000000); // Delay in microseconds.
        esp_wake_stub_set_wakeup_time(AUTOMATIC_WAKEUP_INTERVAL_SEC * 1000000);
        // Set stub entry, then go to deep sleep again.
        esp_wake_stub_sleep(&wake_stub);
}
last_wakeup_RTC = my_rtc_time_get_us() / 1000000;
```

- **Conditional Full Wake-Up:** If the PIR event array is full, or if a magnetic switch triggers, the stub decides to fully boot into the main application.
- **Scheduled Timer Wake-Up:** Before returning to deep sleep, it sets a next wake-up timer, thus allowing periodic checks without fully waking the system each time. For this the variable `AUTOMATIC_WAKEUP_INTERVAL_SEC` is used to set the next wake-up time.

3.4.2 Timestamp Handling and Sending PIR Events in Time

- **RTC-Based Timing:** A helper function converts RTC counter ticks into microseconds, which the stub uses to get a "local RTC time" even in deep sleep.

```
/**
 * @brief Retrieves the current RTC time in microseconds.
 *
 * This function reads the RTC time registers to obtain the current time.
 * It operates in the wake-up stub environment and uses low-level register access.
 *
 * @return Current RTC time in microseconds.
 */
RTC_IRAM_ATTR uint64_t my_rtc_time_get_us(void)
{
```

```
    SET_PERI_REG_MASK(RTC_CNTL_TIME_UPDATE_REG, RTC_CNTL_TIME_UPDATE);
    while (GET_PERI_REG_MASK(RTC_CNTL_TIME_UPDATE_REG, RTC_CNTL_TIME_VALID) == 0) {
        ets_delay_us(1); // Wait for RTC time to be valid.
    }
    SET_PERI_REG_MASK(RTC_CNTL_INT_CLR_REG, RTC_CNTL_TIME_VALID_INT_CLR);
    uint64_t t = READ_PERI_REG(RTC_CNTL_TIME0_REG);
    t |= ((uint64_t)READ_PERI_REG(RTC_CNTL_TIME1_REG)) << 32;

    uint32_t period = REG_READ(RTC_SLOW_CLK_CAL_REG);

    // Convert RTC clock cycles to microseconds.
    uint64_t now_us = ((t * period) >> RTC_CLK_CAL_FRACT);

    return now_us;
}
```

- **Unix Timestamp Calculation:** After each successful SNTP sync in the main app, we store:
  - `rtc_time_at_last_sync`: RTC time in milliseconds at sync.
  - `actual_time_at_last_sync`: Real Unix epoch time in milliseconds.

```
// Extern declarations for time synchronization variables during wake up stub in main.h
// These variables will be stored in RTC memory to persist across deep sleep cycles
extern RTC_DATA_ATTR uint64_t rtc_time_at_last_sync;
extern RTC_DATA_ATTR uint64_t actual_time_at_last_sync;

// Storing reference times in the main app (after SNTP sync)
// main.c - after SNTP synchronization
actual_time_at_last_sync = get_current_time_in_ms();
rtc_time_at_last_sync = get_time_since_boot_in_ms();

uint64_t get_current_time_in_ms() {
    struct timeval now;
    gettimeofday(&now, NULL);
    return (uint64_t)(now.tv_sec) * 1000 + (now.tv_usec) / 1000; // Convert to milliseconds
}

uint64_t get_time_since_boot_in_ms() {
    return (my_rtc_time_get_us() / 1000); // Convert microseconds to milliseconds
}
```

The wake stub computes the *actual* timestamp by adding the difference between current RTC time and `rtc_time_at_last_sync` to `actual_time_at_last_sync`.

```
// Computing actual timestamp in the wake stub:
void store_pir_event(void)
{
    // 1. Current RTC time in ms
    uint64_t rtc_time_now = my_rtc_time_get_us() / 1000;
```

```
    // 2. Calculate the time difference since last sync
    uint64_t rtc_time_diff = rtc_time_now - rtc_time_at_last_sync;

    // 3. Compute actual Unix timestamp in ms
    uint64_t actual_timestamp = actual_time_at_last_sync + rtc_time_diff;

    // Store the PIR event with the computed timestamp
    pir_events[pir_event_count].timestamp = actual_timestamp;
    // ...
}
```

- **PIR Event Storage:** The PIR events are stored in an array of structs in RTC memory, until the array is full of a magnetic switch triggers a full wake-up or automatic wake-up interval is reached.

```
/**
 * @brief Represents the struct for a PIR event
 *
 * This struct includes the timestamp of the event and the device information.
 */
typedef struct {
    uint64_t timestamp;       // The actual Unix timestamp in milliseconds
    device_info_t device;     // Device information associated with the event
} PIR_Event_t;

// Define the maximum number of PIR events stored in RTC memory, the array
// to store these events, and the counter for the events in main.h
extern RTC_DATA_ATTR uint32_t MAX_PIR_EVENTS;
extern RTC_DATA_ATTR PIR_Event_t pir_events[CONFIG_MAX_PIR_EVENTS];
extern RTC_DATA_ATTR int pir_event_count;
```

  - If a PIR sensor wakes the device, the stub calls:

```
/**
 * @brief Stores a PIR event with the current timestamp and device information.
 *
 * Calculates the actual timestamp based on RTC time and synchronization data,
 * then stores the event in the PIR events array.
 */
void store_pir_event(void)
{
        //...

    // Store the new event with the calculated actual timestamp.
    pir_events[pir_event_count].timestamp = actual_timestamp;

    // Since we cannot use memcpy in the wake-up stub, we manually copy each field.
    pir_events[pir_event_count].device.device_id = this_device.device_id;
    pir_events[pir_event_count].device.battery_info_available = this_device.battery_info_available;

    // Note: Assigning pointers directly as below is acceptable in the wake-up stub environment.
    pir_events[pir_event_count].device.device_name = this_device.device_name;
```

```
        pir_events[pir_event_count].device.device_topic = this_device.device_topic;
        pir_events[pir_event_count].device.device_key = this_device.device_key;
        pir_events[pir_event_count].device.room_id = this_device.room_id;

        // Copy MAC address manually.
        for (int i = 0; i < 6; i++) {
            pir_events[pir_event_count].device.mac_address[i] = this_device.mac_address[i];
        }

        // Increment the event count.
        pir_event_count++;
    }
```

  - If `pir_event_count` hits its maximum, the stub forces a full wake to upload all stored events via MQTT.

3.5 Sending Battery RSOC (0.3p)

- **RTC Timestamp Check:** The stub keeps a variable `last_battery_info_time_RTC`. On each wake, it checks if enough time has passed:

```
if ((my_rtc_time_get_us() / 1000000) - last_battery_info_time_RTC
        >= BATTERY_INFO_INTERVAL_SEC) {
    last_battery_info_time_RTC = my_rtc_time_get_us() / 1000000;
    esp_default_wake_deep_sleep(); // Force main app to fully boot
}
```

- Once awake, the main application reads the battery gauge and sends the RSOC (Remaining State of Charge) via MQTT. Since `last_battery_info_time_RTC` is in RTC memory, it persists through deep sleeps, ensuring strictly periodic RSOC uploads.

3.6 Monday Problem (0.3p)

- **What is the Monday problem?**
  The "Monday problem" refers to a situation where the Raspberry Pi (running the MQTT broker) is physically taken to class on Mondays, making the broker temporarily unreachable.
- **Approach to solving the Monday problem**
  To handle the broker's unavailability, I introduced a boolean variable, `mqtt_broker_connected`, in `mqtt.c`. Whenever the device cannot connect to the broker, it skips sending data and retains the events in RTC memory. The next time the device fully wakes and finds the broker reachable, it sends all stored data.
- **Alternative attempts?**
  I did not attempt alternate solutions because the above approach—caching unsent events until reconnection—proved both straightforward and reliable.

```
// Wait for connection with a timeout of 10 seconds
EventBits_t bits = xEventGroupWaitBits(
    mqtt_event_group, CONNECTED_BIT, false, true, pdMS_TO_TICKS(10000)
);
```

```
if (bits & CONNECTED_BIT) {
    ESP_LOGI("mqtt", "Connected to MQTT\n");
} else {
    ESP_LOGI("mqtt", "Could not connect to MQTT broker\n");
    mqtt_broker_connected = false;
}

// Example of error handling for PIR events
//(for battery information and magnetic switch data is the approach same)
if (!mqtt_broker_connected) {
    ESP_LOGI("PIR", "Cannot send stored PIR events, MQTT is not connected");
    return;
}
```

3.7 Power Consumption (1p)

- **Phases of Code Execution (160 MHz, No Light Sleep).** Each duty cycle includes five main phases:
  1. **Boot:** $\sim 30\,\mu W\,h$
  2. **Wi-Fi Connection:** $\sim 295\,\mu W\,h$
  3. **Clock Synchronization (NTP):** $\sim 125\,\mu W\,h$
  4. **MQTT Connection & Data Transmission:** $\sim 31\,\mu W\,h$
  5. **Deep Sleep:** $\sim 0\,\mu W\,h$ (negligible)

  The total energy consumption per cycle (Boot + Wi-Fi + NTP + MQTT) is about $481\,\mu W\,h$.
- **Typical Energy Use per Cycle.**
  - **No Light Sleep (160 MHz):** About $481\,\mu W\,h$ per cycle (Boot + Wi-Fi + NTP + MQTT).
  - **Light Sleep (ALS 160 MHz):** Around $435\,\mu W\,h$.
  - **Dynamic Frequency + ALS:** Can drop cycle energy to roughly $443\,\mu W\,h$ if clock sync is done less often.
- **Measurement Variations.** Table 1 illustrates an example of minimum, maximum, mean, and standard deviation of per-cycle consumption ($\mu W\,h$) at different frequencies, with and without Automatic Light Sleep.

Table 1: Power Consumption Summary (Per Cycle) for Various Frequencies

Frequency (MHz)	Min	Max	Mean	Std
80	420	510	465	27
160	460	520	481	20
240	490	585	540	27
160 + ALS	400	460	435	18

- **Battery Life Estimation.** Let:
  - C_{batt} = battery capacity in $\mu W\,h$ (e.g. $1.4\,Ah$ at $3.7\,V \approx 5{,}180{,}000\,\mu W\,h$).

- E_{cycle} = measured energy consumption per cycle in µW h.
- N_{cycles} = number of cycles per day (e.g. 100).

Then the battery can sustain:

$$\text{BatteryLife (cycles)} \;=\; \frac{C_{\text{batt}}}{E_{\text{cycle}}}.$$

Over N_{cycles} per day, total days of operation is:

$$\text{Days} \;=\; \frac{C_{\text{batt}}}{E_{\text{cycle}} \times N_{\text{cycles}}}.$$

For instance, with $C_{\text{batt}} = 5{,}180{,}000\,\mu\text{W h}$ and $E_{\text{cycle}} = 443\,\mu\text{W h}$ at 100 cycles/day, the device can run for roughly 117 days.

3.8 Limitations (0.5p)

The final solution meets the core requirements. However, it can be improved in the following ways:

- Currently, only simple logic is used to filter repeated sensor triggers (If the patient stays active next to the sensor). More sophisticated signal filtering could prevent repeated events.
- The application relies heavily on RTC memory for event storage. If the memory becomes full, older events risk being overwritten. An SD card or external storage could help preserve data when connectivity is lost.
- Adding more sensors might require dynamic sensor configuration and more sophisticated scheduling for wake-ups. Current solution is tailored to a predefined sensors.

3.9 Code Structure

The project uses a standard ESP-IDF layout under the `main` folder, as shown below:

```
|- CMakeLists.txt
|- main
|   |- CMakeLists.txt
|   |- component.mk
|   |- gauge.c / gauge.h        (Battery gauge handling)
|   |- main.c / main.h          (Main application logic)
|   |- mqtt.c / mqtt.h          (MQTT client setup and event publishing)
|   |- rtc_wake_stub.c / .h     (Wake-up stub logic for low-power triggers)
|   |- sntp.c / sntp.h          (SNTP initialization and time synchronization)
|   |- wifi.c / wifi.h          (Wi-Fi provisioning, connection, event handling)
```

Below is a brief overview of the files:

- `main.c, main.h`: Contains application entry-point (`app_main`), handles device identification, event loops, power configuration, and sets up wake-up sources.
- `rtc_wake_stub.c, .h`: Implements the minimal RTC wake-up stub, including time checks, sensor event handling, and re-entry into deep sleep if needed.

- `wifi.c, wifi.h`: Initializes and manages the Wi-Fi connection (WPA2, SSID/password, IP assignment).
- `mqtt.c, mqtt.h`: Sets up MQTT connectivity, handles message publishing (e.g., PIR events, battery status).
- `sntp.c, sntp.h`: Configures SNTP for accurate time synchronization with NTP servers.
- `gauge.c, gauge.h`: Includes battery gauge drivers (`1c709203f`) to measure voltage and state-of-charge (RSOC).

3.10 Data Reporting (2p)

- **Proof of Continuous Deployment:** Figure 2 provides an overarching view of all recorded sensor data (kitchen, living room, bathroom). Despite certain offline periods (data gaps), the devices generally remained deployed and actively reported when connected and powered.

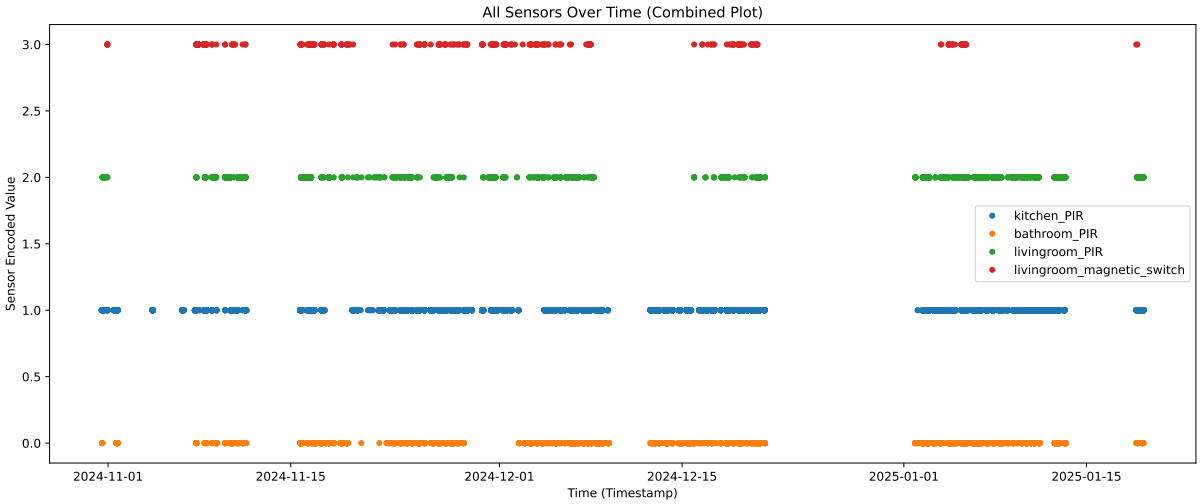

Figure 2: Consolidated view of sensor data (kitchen, living room, bathroom). Each symbol corresponds to a different sensor.

- **Battery Usage Graphs:** Figures 3 and 4 show examples of kitchen and living room battery levels over time. The bathroom ESP32 did not support battery monitoring.

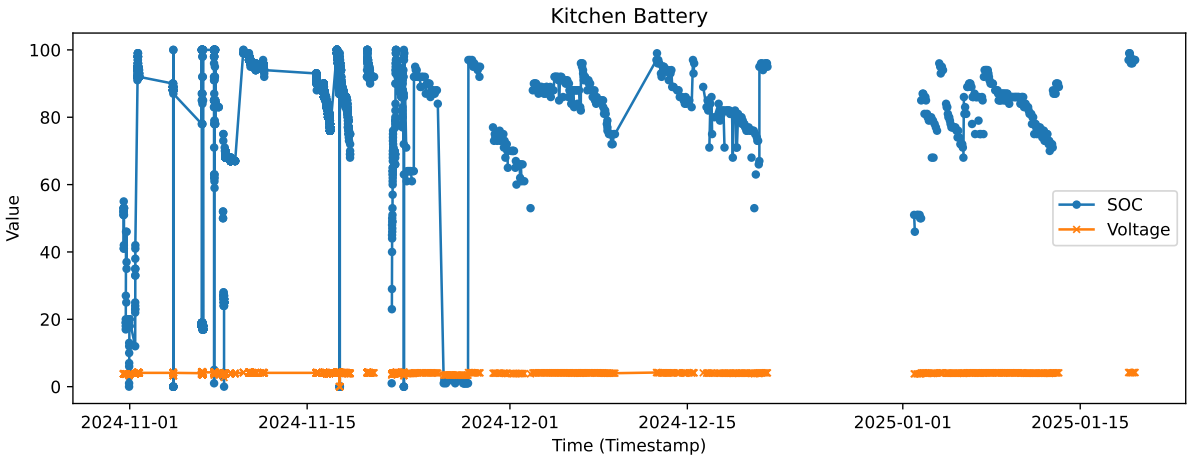

Figure 3: Kitchen device battery usage and PIR event timestamps.

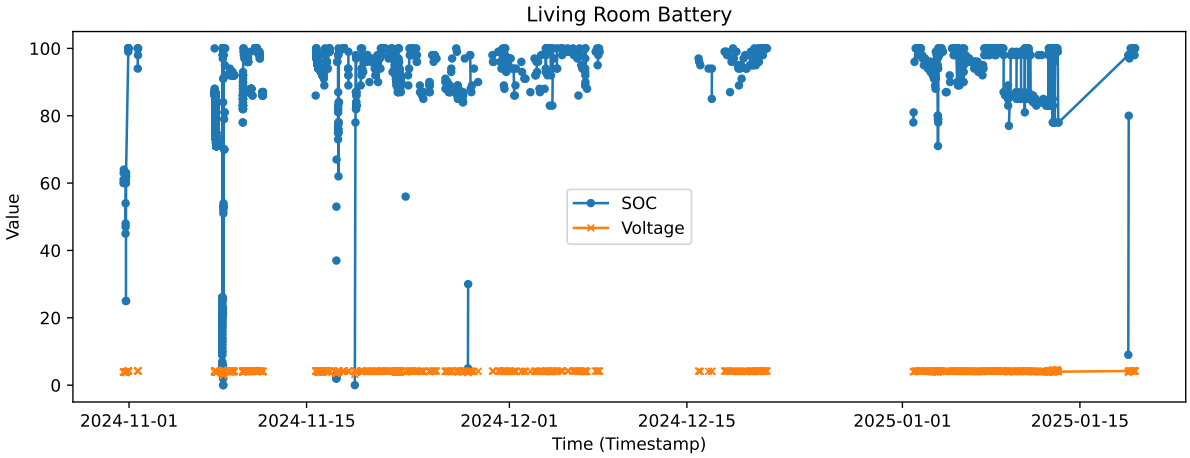

Figure 4: Living Room device battery usage and PIR event timestamps.

3.10.1 Data Gaps Overview

Despite good coverage, there were multi-day outages for each sensor due to initial deployment issues, holiday breaks, or multi-day trips. Tables 2 and 3 summarize these periods, including approximate reasons.

Comment on Gaps. These offline intervals primarily occurred around the initial deployment period (before mid-November), the Christmas holiday (late December), and unnoticed MQTT broker issue in January. Once reconnected, the sensors resumed normal operation, uploading new PIR or magnetic switch events to the MQTT broker.

- These extended offline intervals reflect times when the devices were disconnected or powered down (e.g., holidays). Once reconnected, all events were again captured and sent to the broker.

Table 2: Data Gaps for PIR Sensors

Sensor	Gap Start	Gap End	Duration	Reason
PIR Sensors Gaps				
	2024-11-01 17:53	2024-11-04 09:55	∼ 3 days	Initial deployment
	2024-11-04 10:35	2024-11-06 16:25	∼ 2 days	Initial deployment
	2024-11-11 14:48	2024-11-15 17:14	∼ 4 days	Initial deployment
	2024-11-17 14:50	2024-11-19 17:10	∼ 2 days	Short trip
	2024-12-09 07:30	2024-12-12 13:30	∼ 3 days	Pre-Christmas break
	2024-12-21 08:08	2025-01-02 01:21	∼ 12 days	Christmas break
	2025-01-13 07:54	2025-01-18 19:34	∼ 5 days	Unnoticed MQTT broker issue

Table 3: Data Gaps for `livingroom_magnetic_switch`

Gap Start	Gap End	Duration	Reason
2024-10-31 22:42	2024-11-07 18:10	∼ 7 days	Initial deployment
2024-11-11 12:57	2024-11-15 17:17	∼ 4 days	Initial deployment
2024-11-17 18:48	2024-11-22 19:36	∼ 3 days	Short trip
2024-12-08 00:18	2024-12-15 22:10	∼ 8 days	Unnoticed cable disconnection
2024-12-20 18:45	2025-01-03 19:49	∼ 14 days	Christmas break
2025-01-05 18:48	2025-01-18 19:32	∼ 13 days	Unnoticed MQTT broker issue + cable disconnection

4 Digital Twin (17 pages)

4.1 Event-driven Programming (1p)

- **Challenges with a Distributed, Event-driven System**
  I was already comfortable with Python and machine learning, but I had limited experience with distributed systems, containers (like Docker), and orchestration platforms (like Kubernetes). Working with an event-driven approach also introduced new concepts, such as triggers and message flows between microservices. The limited time to get everything set up and tested made this more challenging, especially when debugging issues that could happen in different services at once.
- **Difficulties and Debugging Approach**
  One of the main hurdles was figuring out how to trace errors across multiple containers or services. I used Jupyter notebooks locally to prototype and test smaller parts of the project, especially the data processing and model training code. For remote debugging, I added detailed logging statements to monitor the flow of events, making it easier to see where a service was failing or how data was being passed around. I relied on logs from each service to understand what happened when a specific event was triggered. This helped me quickly track down configuration problems or code bugs. Over time, I learned how to rebuild and redeploy services faster, so I could iterate on fixes without wasting too much time.

4.2 Our Digital Twin Basics (2p)

- **Events in Each Component:**

- **Modeling (modeling/main.py):**
  - `TrainOccupancyModelEvent`: Triggers the `create_occupancy_model_function`, which fetches sensor data (e.g. room usage) and trains an occupancy model (`train_occupancy_model`).
  - `TrainMotionModelEvent`: Invokes `create_motion_model_function`, training a model that analyzes movement patterns and stores it (via `save_model_to_minio`).
  - `TrainBurglaryModelEvent`: Calls `create_burglary_model_function`, training a burglary detection model for home security.
- **Monitoring (monitoring/main.py):**
  - `CheckEmergencyEvent`: Triggers `check_emergency_detection_function`, which runs `emergency_detection_workflow` and may emit an `EmergencyEvent` if thresholds are exceeded (e.g. no movement detected for a prolonged period).
  - `AnalyzeMotionEvent`: Triggers `motion_analysis_function`, executing `analyse_motion_patterns` to detect unusual motion trends.
  - `CheckBurglaryEvent`: Fires `check_burglary_detection_function`, running `detect_burglary` logic and potentially dispatching a `BurglaryEvent`.
- **Actuation (actuation/main.py):**
  - `EmergencyEvent`: Calls `create_emergency_notification_function`, which sends a high-priority (`send_todo`) message to caregivers or relevant stakeholders.
  - `BurglaryEvent`: Invokes `create_burglary_notification_function`, raising an alert for suspected unauthorized entry.

- **Frequency and Purpose of Events:**
  - *Periodic Training*: The *modeling* events (e.g. `TrainOccupancyModelEvent`) are fired at regular intervals (e.g. every few hours or days). This ensures the ML models remain up-to-date with evolving household patterns and occupant behavior.
  - *Monitoring Checks*: The *monitoring* events (`CheckEmergencyEvent`, `CheckBurglaryEvent`, `AnalyzeMotionEvent`) typically run more frequently (e.g. every 15 minutes or hourly) to catch urgent situations promptly (medical emergencies, suspicious motion, etc.).
  - *Actuation Alerts*: Whenever an emergency or burglary is detected, a one-shot event triggers *actuation* logic, causing immediate notifications (`EmergencyEvent`, `BurglaryEvent`). These events focus on critical, time-sensitive responses, such as sending notifications to a caretaker or homeowner.

- **Data Generated by the Custom Event Fabric:**
  - *Sensor Data Retrieval*: On the modeling side, each training function fetches historical sensor data (e.g. from InfluxDB) via `fetch_all_sensor_data` and prepares it for analytics (`prepare_data_for_occupancy_model`).
  - *Learned Models*: After training, models and statistical summaries are uploaded to MinIO (`save_model_to_minio`) and can be re-used by the monitoring flows to detect unusual patterns or emergencies.

– *Alarms & Notifications*: `EmergencyEvent` and `BurglaryEvent` produce JSON payloads with risk-level and context (e.g. "Patient is in bathroom for some amount of time"). The *actuation* component receives these payloads and relays them to external services (via `send_todo`) for display in dashboards or push notifications.

Reasoning Behind Configurable Intervals

- **Retraining Intervals (`TRAIN_OCCUPANCY_MODEL_INTERVAL`, `TRAIN_MOTION_MODEL_INTERVAL`, `TRAIN_BURGLARY_MODEL_INTERVAL`):**
  By default, these are set to `12h`. Retraining every 12 hours balances the need to adapt to new sensor data with the cost of computational resources. If the environment changes more rapidly, one could reduce the interval to improve model responsiveness. Conversely, if conditions are stable, retraining could be scheduled once a day or once a week to save resources. I chose 12 hours also for faster debugging.
- **Analysis Intervals (`ANALYSE_MOTION_INTERVAL`):**
  Motion data analysis is set to `24h`, reflecting that aggregated motion patterns may be more meaningful over a full day's worth of data. This can be shortened if near real-time detection is necessary (e.g. motion anomalies every few hours).
- **Emergency & Burglary Checks (`CHECK_EMERGENCY_INTERVAL`, `CHECK_BURGLARY_INTERVAL`):**
  These intervals (`30m` for emergencies, `1h` for burglaries) reflect the urgency of the events. Emergency checks run every 30 minutes to promptly detect inactivity or other critical signals. Burglary checks run hourly, as unauthorized-entry patterns may be less frequent but still demand timely detection.
- All these parameters reside in `monitoring/config.py` so that different environments (e.g. a small home vs. a large facility) can easily tune these intervals. Adjusting intervals can reduce system load, optimize battery usage on sensor devices, or improve detection responsiveness, depending on deployment requirements.

4.3 Use Case 1: Emergency Detection (2p)

- **Environment and Modeling Approach**
  Our setup monitors an apartment with two people and a cat, which introduces variability in sensor data. I chose a straightforward approach to detect possible emergencies based on how long someone remains in a single room. Specifically, I compute the *mean* and *standard deviation* (σ) of room-stay durations, then mark an event as an "emergency" if the measured time in that room falls outside a configurable threshold range:

$$(\text{mean} - k \times \sigma,\ \text{mean} + k \times \sigma).$$

  Here, k is a tuning parameter (the number of standard deviations), set in code as `THRESHOLD_FOR_EMERGENCY_DETECTION` (defaulting to 3). The idea is that if a resident stays significantly longer (or shorter) in a room than usual, it might indicate a problem (e.g., a fall, health emergency, or inability to leave).
- **Metrics Used for Comparison**

- **Mean Duration:** Average time spent in each room, as calculated from historical sensor data.
- **Standard Deviation (σ):** Measures the typical variability around that mean.
- **Threshold:** A multiple of σ. If the new data point is outside:

$$[\text{mean} - k \times \sigma,\ \text{mean} + k \times \sigma],$$

it raises an alert that might indicate an *emergency*.

- **Evolution by Re-Training the Model**
  Our "model" is primarily a statistical table (`room_stats`) of `(mean, std)` for each room. When re-trained:
  1. I fetch a fresh window of sensor data (e.g. the last two weeks).
  2. I recalculate the mean and standard deviation for each room based on new data.
  3. Next time an event comes in, I use this updated `room_stats` to decide if the current room duration is out of range.

  This *simple statistical approach* gradually adapts to changes in living patterns. For example, if a resident starts spending more time in the bedroom for legitimate reasons, over subsequent re-training cycles, the mean bedroom-stay duration will increase, reducing false emergency alerts.
- **Handling Two Residents and a Cat**
  The presence of multiple residents and a pet can add noise. However, our focus was on the *primary occupant's time in each room.* Thus, I aggregate all motion events but rely on repeated triggers (like repeated `PIR` detections) to indicate continuous presence. In a real deployment, I could refine the sensor placement or logic to better distinguish individuals. For now, I maintain a single model that treats *any* motion in a room as occupant presence.
- **Code Snippets Illustrating Emergency Detection**

```
# Constant for allowed deviation from mean:
THRESHOLD_FOR_EMERGENCY_DETECTION = 3  # e.g., 3 std devs

def detect_emergency(room: str, duration: float, stats: pd.Series, threshold: int = 3) -> (bool, str):
    mean = stats.get("mean", 0)
    std  = stats.get("std", 0)

    # If standard deviation is zero, any difference from mean is unusual
    if std == 0:
        if duration != mean:
            return True, "Time spent differs from mean; potential emergency."
        else:
            return False, "Time spent matches mean; no emergency."
    else:
        # Calculate lower/upper bounds
        lower_bound = max(mean - threshold * std, 0)  # not below 0
        upper_bound = mean + threshold * std
        if duration < lower_bound or duration > upper_bound:
            return True, "Time spent is outside allowed range; emergency."
```

```
        else:
            return False, "Time is within expected range; all good."
```

Examples of Emergency and Non-Emergency Messages

Below are some sample messages generated by the `detect_emergency` function, illustrating the "Emergency Alert" scenario.

```
Emergency Alert!
Room: kitchen,
Patient has spent 03:45:00 here.
Expected duration (mean): 02:00:00,
Standard deviation (std): 00:30:00,
Threshold used: 3,
Allowed duration range: 00:30:00 - 03:30:00,
Duration is outside the allowed range!
```

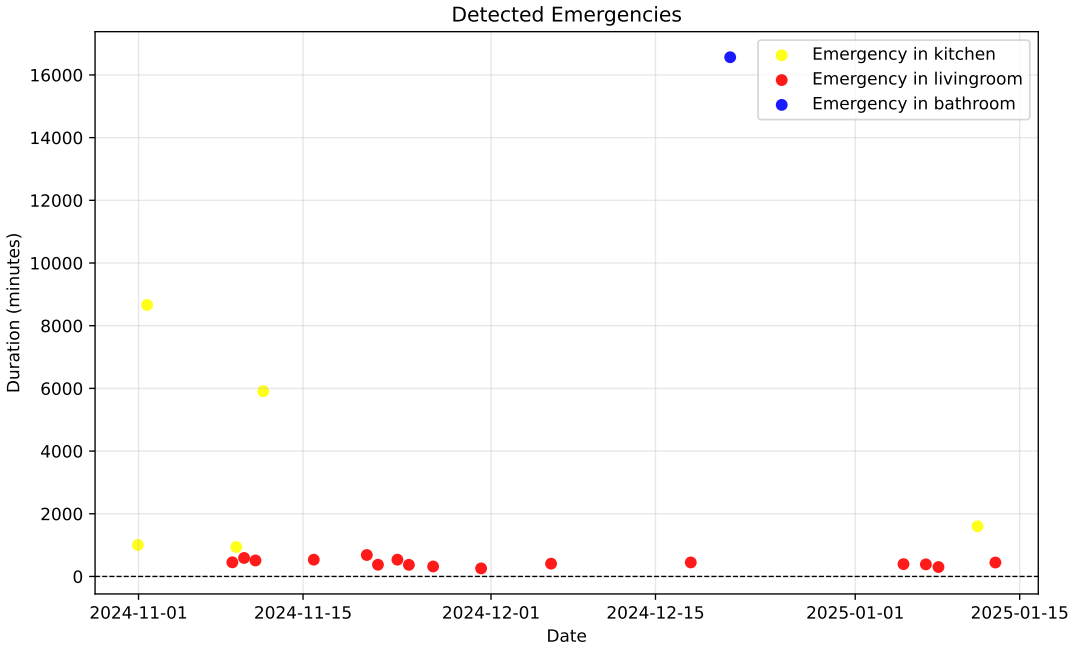

Figure 5: Detected emergencies based on sensor data. Each point represents a detected emergency event.

- Weekly means show slight variations, reflecting natural behavioral changes, such as more time in the kitchen during meals or in the living room during leisure.
- Larger σ for the bathroom indicates irregular usage patterns compared to other rooms.
- Weekly retraining adapts to routine changes, minimizing false alarms.
- Smaller σ in the kitchen room indicates consistent use patterns.

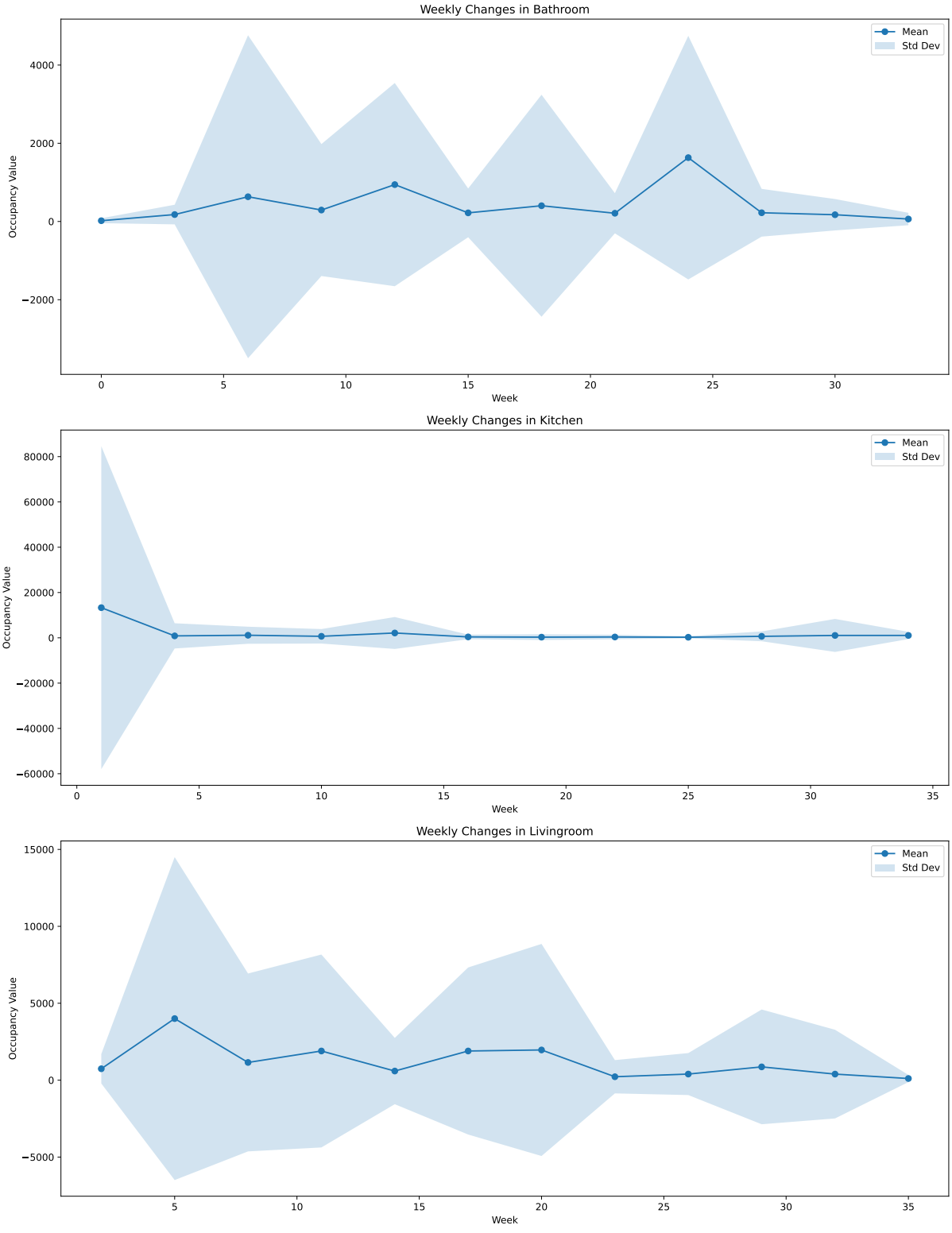

Figure 6: The weekly changes in the mean and std of the model.

4.4 Use case 2: Detection of behavioral changes (3p)

- Describe the modeling technique followed to build an understanding of stay duration in the rooms of your apartment.
- What does your model cannot capture?
- Explain the selection of your modeling approach.
- What is important to be aware of regarding your system and models?
- What metrics and values are considered to differentiate between an "information" and "To-do" message?
- How is the information communicated to the visualization component?
- How is the severity calculated?
- Is your modeling able to determine emergencies?

4.5 Use case 3: Behavioral change based on paths (3p)

Only present this if you did the path analysis.

- Give your path definition, the model structure, and the criteria for behavioral changes.
- Could you use path information also to report emergencies?

4.6 Use case 3: your own use case (4p)

Provide this section only, if you developed an own use case.

- If you implement another use case, explain the model, its use, important metrics, your modeling design decisions, and short-comings of the model given the amount of data, seasonality, style of data.

4.7 Performance Analysis (2p)

- Measure response times (time from trigger calling the event fabric until delivering the event to the scheduler or time from calling event fabric till actuation communicates an emergency) and execution times (time taken from dispatching the invocation until completing the invocation) for each event, break it down in terms of fetching data from storage and compute times.
- How long does it take to train your model?
- How does it perform when your model considers more data? Correlate increasing days of data vs time.
- What is the ratio between preprocessing your data and training?
- What is the size of your models?

4.8 Limitations (1p)

- Describe current limitations of your implementation.

4.9 Code Structure (2p)

This project—named *Digital Twin App*—is split into separate folders for each major component: `actuation`, `modeling`,`monitoring`, `viz-component` and `homecare-hub`. Each folder contains a `main.py` file that implements FastAPI endpoints and logic specific to that component. A common `base/` folder (symlinked or copied into each component) provides shared utilities such as event classes, triggers, and helper functions to interact with databases, object storage, and the HomeCare Hub utils to send the info and todo messages.

For additional information also refer to the README.md file in the code repository. (There is one common file and then also separate README.md files in each component folder.)

- **Individual Folders per Component**
  - `actuation/`
    - `main.py`: Defines and deploys `create_emergency_notification_function` and `create_burglary_notification_function`, each responsible for generating notifications (e.g., via `send_todo`) when emergencies or burglaries are detected.
    - `Dockerfile`: Builds and containers this microservice.
  - `modeling/`
    - `main.py`: Exposes endpoints for training the Occupancy, Motion, and Burglary ML models. Internally uses local modules like `occupancy_model.py`, `motion_model.py`, and `burglary_model.py`.
    - `occupancy_model.py`: Responsible for preparing and training the occupancy model, computing room-stay durations, etc.
    - `motion_model.py`: Contains logic for analyzing transitions and motion events.
    - `burglary_model.py`: Implements an Isolation Forest-based burglary-detection model.
  - `monitoring/`
    - `main.py`: Registers asynchronous functions for checking emergencies (`check_emergency_detection_function`), burglary events (`check_burglary_detection_function`), and general motion analysis (`motion_analysis_function`). Also configures periodic triggers for each event based on intervals specified in `config.py`.
    - `config.py`: Defines environment variables and intervals (e.g., `TRAIN_OCCUPANCY_MODEL_INTERVAL = "12h"`, `CHECK_EMERGENCY_INTERVAL = "30m"`). These parameters are adjustable to match system load and user needs.
- **Common `base/` Folder**
  - `gateway.py`: Offers the `LocalGateway` class, enabling the components to define endpoints in a consistent manner.
  - `event.py`: Contains all event classes (e.g., `TrainOccupancyModelEvent`, `CheckEmergencyEvent`, `BurglaryEvent`), used for the event-driven architecture.
  - `trigger.py`: Implements `PeriodicTrigger` and `OneShotTrigger` for scheduling or one-time event invocation.

- `homecare_hub_utils.py`: Provides communication methods like `send_info` (for general notifications) and `send_todo` (for alerts needing user action).
- `influx_utils.py`: Defines `fetch_all_sensor_data` and other functions to query InfluxDB for sensor, battery, or aggregated data.
- `minio_utils.py`: Manages saving/loading DataFrames (model results, stats) to/from MinIO object storage.

Challenges and Organization Choices. Initially, I had some confusion on how to structure the code for different services (e.g. `actuation` vs. `monitoring`). I also had to learn how to keep track of common logic in the `base/` folder so I would not duplicate code in each microservice.

I found Docker and Kubernetes orchestration somewhat complex at first, but following the microservices approach allowed me to build and deploy each component independently. This structure made debugging easier: if something failed in `monitoring`, it rarely affected `actuation` or `modeling` directly. Additionally, relying on `docker-compose` or `k8s/` manifests simplified local testing.

Important Files and Their Roles.

- `main.py` in each folder: The entry point for that service, registering relevant routes and triggers.
- `config.py` (particularly in `monitoring/`): Specifies intervals/wait times for each scheduled job, plus external IP addresses or credentials for InfluxDB, MinIO, etc.
- `*model.py`: Contains the logic for training each ML model (occupancy, motion, burglary).
- `docker-compose.yml` or `k8s/*.yaml` (not shown in detail above): Describes how to run each microservice or scale them using Docker/Kubernetes.

5 Declaration of data usage

Specify, whether you allow us to keep your data for tests with our own implementation. We make sure, that the data cannot be tracked down to your name.

5.1 Data Donation Agreement

Under this agreement, you understand that we, the Chair CAPS, can use (read, modify, delete) any sensor data generated during the WiSe2024 in future research and educational activities or produce derivative works. Given the current structure of the stored data in the procured Raspberry Pi, we ensure the data cannot reveal any personal information of the donor as it refers to a generic user: `iot-user`, while the fields contain generic names and values.

MTK, Signature

Date, Place