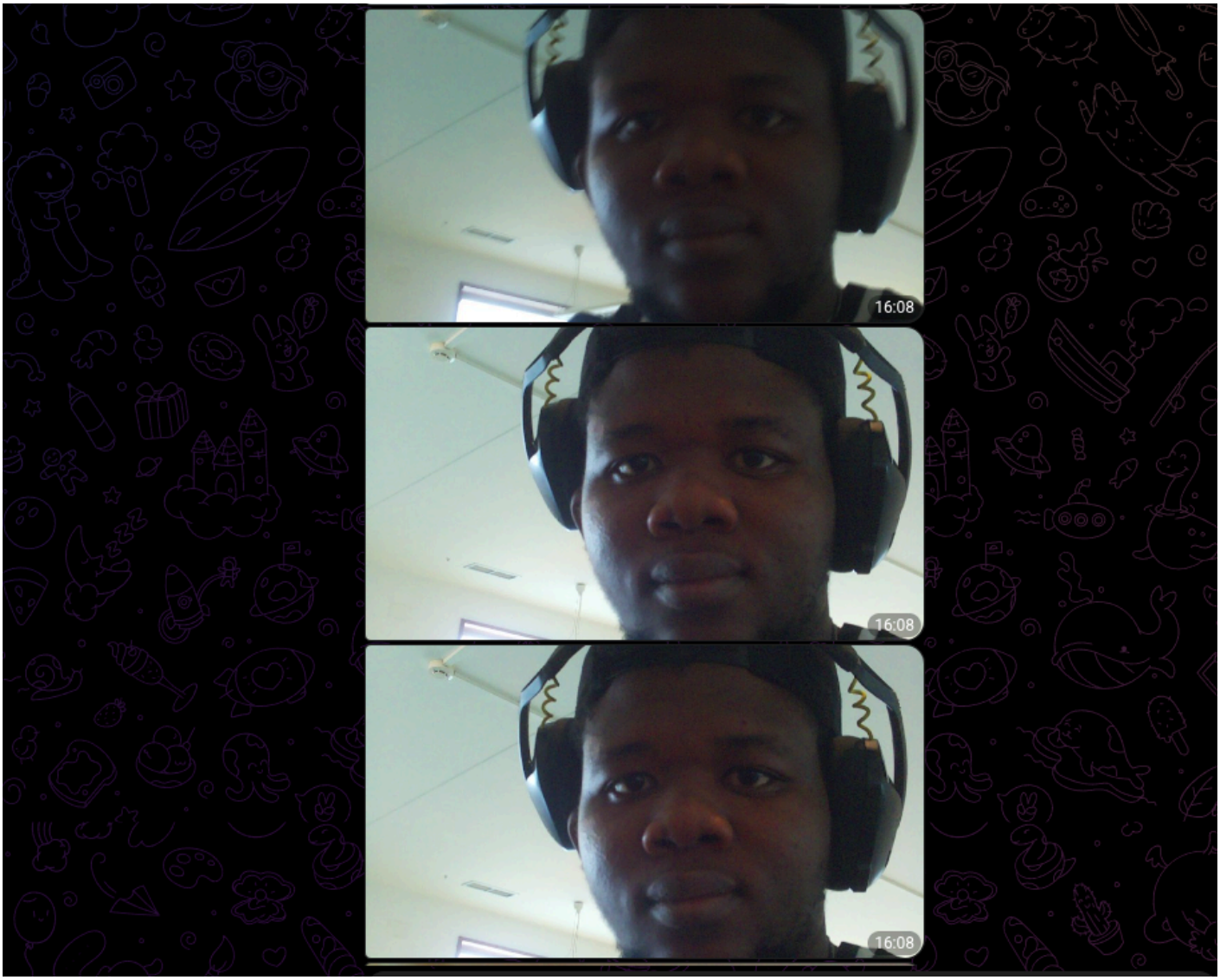
Motion Detector Project Report

Course: Industrial Internet of Things

Supervised by
Sara Guerreiro Santalla
Héctor Quintián Pardo



By Bakel Bakel Begededum

Co-funded by the Erasmus+ Programme of the European Union

UNIVERSIDADE DA CORUÑA

SEAS 4.0

TABLE OF CONTENTS

1. Introduction

Modern embedded systems enable the integration of sensors, cameras, and communication modules to create intelligent monitoring and alert mechanisms. This project demonstrates a motion detection and alert system using the Raspberry Pi, Sense HAT, and PiCamera2, combined with an automated Telegram bot for remote notifications.

The system continuously monitors its environment by reading motion data from the Sense HAT's Inertial Measurement Unit (IMU). When a disturbance or movement is detected, it performs a sequence of automated responses: it prints an alert message with the device's orientation values (roll, pitch, and yaw), triggers a visual alarm on the Sense HAT's LED matrix, captures a set of three time-stamped images, and transmits them to a designated Telegram account for remote viewing.

This implementation reflects the principles of IoT-based surveillance and alert systems, where real-time sensing, local processing, and networked communication work together to ensure autonomous operation. The modular, object-oriented design improves code readability, maintainability, and extensibility, making the project suitable as a foundational model for larger-scale security or monitoring applications.

2. Objectives

The primary objectives of this project are as follows:

1. **To simulate an intrusion detection system** using the Raspberry Pi's Sense HAT IMU to sense motion or disturbance.

2. **To display orientation readings** (roll, pitch, yaw) on the screen whenever motion is detected.

3. **To provide a visual alarm** by blinking the Sense HAT LED matrix alternately in red and white every 0.5 seconds for a total duration of 3 seconds.

4. **To capture and store three images** using the PiCamera2, spaced 1 second apart, with filenames that include the date and time of capture.

5. **To transmit the captured images automatically** to a specified Telegram chat, group, or channel using a programmed bot.

6. **To design the system in an object-oriented and modular format** that promotes clean architecture, fault isolation, and easy scalability.

3. Technologies Used

1. **sense_hat**: access to accelerometer/orientation and the LED matrix.
2. **picamera2**: control over the Raspberry Pi camera.
3. **requests**: HTTPS calls to Telegram's Bot API.
4. **dataclasses / pathlib / logging / time / datetime / os**: Python standard modules for configuration, files, logging, timing, and environment variables.

These libraries map directly to hardware (Sense HAT, camera) and messaging (Telegram).

4. Software Architecture

4.1. Configuration layer: `Config` dataclass

@dataclass
class Config: ...

```python
@dataclass
class Config:
    # Motion detection
    samples: int = 10           # accel samples to average
    sample_delay: float = 0.01  # seconds between accel samples
    accel_threshold: float = 0.03  # g delta on any axis = motion

    # Camera capture
    image_count: int = 3
    image_interval: float = 1.0   # seconds between images (assignment: 1 s)
    image_size: tuple[int, int] = (1280, 720)
    capture_dir: Path = Path("/home/pi/bakel/captured")

    # LED alarm (assignment: 0.5 s toggle, total 3 s)
    blink_period: float = 0.5
    blink_total: float = 3.0

    # Scheduler
    run_interval: float = 5.0  # seconds

    # Just normal for not spamming social media
    cooldown_seconds: float = 8.0

    # Telegram (read from environment)
    bot_token: str = os.getenv("BOT_TOKEN", "")
    chat_id: str = os.getenv("CHAT_ID", "")
```

This class centralizes all tunable parameters: sampling counts, delays, accel threshold, image cadence/count/size, capture directory, LED blink timing, loop interval, cooldown, and secrets (bot token, chat id) loaded from the environment.

The rest of the code reads behavior from `cfg` instead of hard-coding. You can adjust sensitivity, timing, or destinations without touching logic. This reduces risk and keeps the code testable.

4.1.1. Explanation of `Config` variables

1. `samples` (default = 10):
   This tells the system how many times to read acceleration data from the Sense HAT before taking the average.
   Taking multiple readings smooths out noise from the IMU sensor and avoids false motion triggers.

2. `sample_delay` (default = 0.01):
   This is the time delay (in seconds) between consecutive accelerometer samples.
   For example, 10 samples × 0.01 s = 0.1 seconds of total sampling time per measurement cycle.

3. `accel_threshold` (default = 0.03):
   This sets the sensitivity of the motion detection in terms of gravitational acceleration (g).
   If the average acceleration difference between the current reading and the initial baseline exceeds 0.03 g on any axis (x, y, or z), the system considers that as motion detected.

4. `image_count` (default = 3):
   The number of photos to capture when motion is detected.
   It directly satisfies the assignment requirement to take *three images*.

5. `image_interval` (default = 1.0):
   The time gap (in seconds) between each photo capture.
   In this case, each image is captured 1 second apart, also per the assignment specification.

6. `image_size` (default = (1280, 720)):
   The resolution of each image captured by the PiCamera2.
   This is HD resolution (720p), which balances image quality and speed.

7. `capture_dir` (default = Path("/home/pi/bakel/captured")):
   The directory where the images are stored before being sent to Telegram.
   The path uses Python's `pathlib.Path` for platform-independent file handling.

8. `blink_period` (default = 0.5):
   This is how long each LED color stays on before switching (half a second).
   It determines the flashing frequency between red and white.

9. `blink_total` (default = 3.0):
   This is the total duration of the blinking alarm (3 seconds).

So, the LED matrix blinks for 3 seconds, alternating every 0.5 seconds making 6 total flashes.

10. `run_interval` (default = 5.0):
    The main program loop runs every 5 seconds. In each cycle, it checks for motion, possibly captures photos, and sends alerts.

11. `cooldown_seconds` (default = 8.0):
    This value defines how long the system must wait after sending one alert before it can send another. For example, if motion continues for several seconds, the system will only send a new alert after 8 seconds.

How it relates

1. Every component that needs timing/thresholds or paths gets them via `cfg`.
2. `MotionAlertSystem` owns a `Config` instance and passes fields to managers.

4.2. Messaging layer: `TelegramClient`

class TelegramClient: ...

```python
class TelegramClient:
    def __init__(self, token: str, chat_id: str):
        if not token or not chat_id:
            raise ValueError("BOT_TOKEN and CHAT_ID must be set in environment variables.")
        self.base = f"https://api.telegram.org/bot{token}"
        self.chat_id = chat_id

    def send_photo(self, path: Path) -> dict:
        url = f"{self.base}/sendPhoto"
        with open(path, "rb") as photo:
            r = requests.post(url, data={"chat_id": self.chat_id}, files={"photo": photo}, timeout=30)
        r.raise_for_status()
        return r.json()

    def send_video(self, path: Path) -> dict:
        url = f"{self.base}/sendVideo"
        with open(path, "rb") as vid:
            r = requests.post(url, data={"chat_id": self.chat_id}, files={"video": vid}, timeout=60)
        r.raise_for_status()
        return r.json()
```

What it does

1. Wraps Telegram Bot API calls for photos and optionally video **(because I can)**.
2. Validates secrets on construction so misconfiguration fails fast.
3. Uses `requests` with timeouts and raises for HTTP errors.

How it relates

1. `MotionAlertSystem` uses it to deliver evidence (images) when motion triggers.
2. It is isolated, so changing the alert channel (Slack/Email) means swapping this client without touching sensing or camera code.

4.3. Sensing & signaling layer: `SenseManager`

class SenseManager: ...

```python
class SenseManager:
    RED = [255, 0, 0]
    WHITE = [255, 255, 255]

    def __init__(self):
        self.sense = SenseHat()
        sleep(0.5)  # tiny settle

        # Baselines
        self.initial_accel = self.sense.get_accelerometer_raw()  # dict with x,y,z (in g)
        ori = self.sense.get_orientation()
        self.initial_pitch = ori["pitch"]
        self.initial_roll = ori["roll"]
        self.initial_yaw = ori["yaw"]

    def get_accel_delta(self, samples: int, delay: float) -> tuple[float, float, float]:
        xs, ys, zs = [], [], []
        for _ in range(samples):
            a = self.sense.get_accelerometer_raw()
            xs.append(a["x"]); ys.append(a["y"]); zs.append(a["z"])
            sleep(delay)
        ax = sum(xs) / len(xs); ay = sum(ys) / len(ys); az = sum(zs) / len(zs)
        dx = abs(self.initial_accel["x"] - ax)
        dy = abs(self.initial_accel["y"] - ay)
        dz = abs(self.initial_accel["z"] - az)
        return dx, dy, dz

    def get_orientation(self) -> tuple[float, float, float]:
        o = self.sense.get_orientation()
        # Return (pitch, roll, yaw)
        return o["pitch"], o["roll"], o["yaw"]

    def blink_alarm(self, period: float, total: float) -> None:
        t0 = time()
        i = 0
        while time() - t0 < total:
            self.sense.clear(self.RED if i % 2 == 0 else self.WHITE)
            sleep(period)
            i += 1
        self.sense.clear()

    def clear(self) -> None:
        self.sense.clear()
```

What it does

1. Initializes Sense HAT and records **baseline acceleration** and baseline orientation.
2. Provides `get_accel_delta(samples, delay)` which:
    a. samples raw accel multiple times,
    b. averages readings,

c. returns **absolute deltas** (x/y/z) vs baseline.

3. Provides `get_orientation()` to read **pitch/roll/yaw** (degrees).
4. Drives the **LED alarm** (`blink_alarm(period, total)`) and `clear()`.

How it relates

1. Supplies the **decision signal** to `MotionAlertSystem`.
2. Provides user feedback (LED blink) when an alert fires.
3. Keeps hardware specifics out of business logic.

4.4. Imaging layer: `CameraManager`

class CameraManager: …

```python
class CameraManager:
    def __init__(self, size=(1280, 720)):
        self.cam = Picamera2()
        self.size = size

    def _ensure_preview(self):
        cfg = self.cam.create_preview_configuration(main={"size": self.size})
        self.cam.configure(cfg)

    def capture_burst(self, count: int, interval: float, outdir: Path) -> list[Path]:
        outdir.mkdir(parents=True, exist_ok=True)
        self._ensure_preview()
        self.cam.start()
        files: list[Path] = []
        try:
            for _ in range(count):
                name = f"image_{datetime.now().strftime('%Y%m%d_%H%M%S')}.jpg"
                path = outdir / name
                self.cam.capture_file(str(path))
                files.append(path)
                sleep(interval)
        finally:
            self.cam.stop()
        return files

    def record_video_mp4(self, outpath: Path, seconds: float = 3.0, fps: float = 25.0) -> Path:
        """
        Recorded as h264 and remuxed to mp4 using ffmpeg found on system.
        """
        h264 = outpath.with_suffix(".h264")
        self.cam.video_configuration.size = self.cam.sensor_resolution  # use default full field
        self.cam.video_configuration.controls.FrameRate = fps
        self.cam.start_and_record_video(str(h264), duration=seconds)
        # Remux
        os.system(f"ffmpeg -y -r {int(fps)} -i {h264} -vcodec copy {outpath} >/dev/null 2>&1")
        try:
            h264.unlink(missing_ok=True)
        except Exception:
            pass
        return outpath

```

What it does

1. Configures camera preview (`_ensure_preview`) at the requested resolution.
2. `capture_burst(count, interval, outdir)`:
    a. ensures the output directory exists,
    b. starts the camera, captures **count** JPEGs separated by **interval** seconds,
    c. stops the camera and returns a list of `Path`s.

3. `record_video_mp4(...)` records H.264 and remuxes to MP4 via `ffmpeg`.

How it relates

1. Executes the assignment's requirement: **3 photos 1 s apart**, stored with timestamped names for traceability, later pushed to Telegram by `TelegramClient`.
2. Encapsulation lets you swap capture strategy (e.g., higher resolution, different formats) without touching motion logic.

4.5. Brain layer: `MotionAlertSystem`

class MotionAlertSystem: ...

```python
class MotionAlertSystem:
    def __init__(self, cfg: Config):
        self.cfg = cfg
        self.sense = SenseManager()
        self.cam = CameraManager(size=cfg.image_size)
        self.tg = TelegramClient(cfg.bot_token, cfg.chat_id)
        self._last_alert = 0.0

    def _cooldown_ok(self) -> bool:
        return (time() - self._last_alert) >= self.cfg.cooldown_seconds

    def run_once(self):
        try:
            dx, dy, dz = self.sense.get_accel_delta(self.cfg.samples, self.cfg.sample_delay)
            pitch, roll, yaw = self.sense.get_orientation()

            if dx > self.cfg.accel_threshold or dy > self.cfg.accel_threshold or dz > self.cfg.accel_threshold:
                if not self._cooldown_ok():
                    logging.info("Motion detected but in cooldown; skipping alert.")
                    return

                print("DEVICE MOVED")
                # Assignment order: roll, pitch, yaw
                print(f"Roll = {roll:.1f}°  Pitch = {pitch:.1f}°  Yaw = {yaw:.1f}°")

                # Capture images (3 shots, 1 s apart)
                images = self.cam.capture_burst(
                    self.cfg.image_count, self.cfg.image_interval, self.cfg.capture_dir
                )

                # Send each photo
                for p in images:
                    try:
                        res = self.tg.send_photo(p)
                        logging.info(f"Sent photo: {p.name} -> {res.get('ok')}")
                    except Exception as e:
                        logging.exception(f"Failed to send photo {p}: {e}")
```

```
                # Blink LED alarm (0.5 s toggle, 3 s total)
                self.sense.blink_alarm(self.cfg.blink_period, self.cfg.blink_total)

                self._last_alert = time()
            else:
                print("DEVICE IS STILL")
                self.sense.clear()

        except Exception as e:
            logging.exception(f"run_once error: {e}")

    def run_forever(self):
        print("Starting periodic execution... Press Ctrl+C to stop.")
        try:
            while True:
                self.run_once()
                sleep(self.cfg.run_interval)
        except KeyboardInterrupt:
            print("Shutting down. Bye!")
            self.sense.clear()
```

Responsibilities

1. Owns the three managers (`SenseManager`, `CameraManager`, `TelegramClient`) and the `Config`.
2. Implements the **control loop**:
    a. `run_once()` collects accel deltas and orientation.
    b. If any delta exceeds `cfg.accel_threshold` **and** cooldown passes:
        i. Prints human-readable **Roll / Pitch / Yaw**.
        ii. Captures images via `CameraManager`.
        iii. Sends each photo via `TelegramClient`.
        iv. Triggers LED blink via `SenseManager`.
        v. Updates `_last_alert` for cooldown.
    c. Otherwise prints **DEVICE IS STILL** and clears LEDs.
3. Wraps the body in `try/except` and logs exceptions (fault isolation).
4. `run_forever()` repeats `run_once()` every `cfg.run_interval` seconds and responds cleanly to **Ctrl+C**.

How it relates

1. This is the **brain**: it applies policy (threshold + cooldown) and coordinates sensing to imaging to messaging to signaling.
2. Because the managers hide device specifics, this class reads like business logic.

4.6. Entrypoint: `main()` and script guard

```python

# ---------------------------
# Entrypoint
# ---------------------------
def main():
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s | %(levelname)s | %(message)s"
    )
    cfg = Config()  # reads BOT_TOKEN and CHAT_ID from env
    system = MotionAlertSystem(cfg)
    system.run_forever()


if __name__ == "__main__":
    main()

```

What it does

1. Sets up structured logging.
2. Constructs `Config` (pulling secrets from env), then `MotionAlertSystem`.
3. Starts the periodic loop.

How it relates

1. Keeps import side-effects clean and makes the module testable (you can import classes in unit tests without starting the loop).

Key timings

1. **Sampling**: `samples=10` at `0.01 s` each 100 ms smoothing for noise.
2. **Photos**: 3 × 1 s cadence
3. **LED**: toggle every 0.5 s for 3 s
4. **Cooldown**: default 8 s (prevents spam on continuous vibration).

4.7. Error handling & reliability

1. `run_once()` wraps operations in `try/except` and logs stack traces; a transient camera or network fault does not crash the process.
2. `TelegramClient` uses timeouts and `raise_for_status()`, so HTTP issues are explicit in logs.
3. Camera start/stop is guarded by `try/finally` to release the device cleanly.

4.8. Security & deployment

1. `BOT_TOKEN` and `CHAT_ID` load from environment (or an `EnvironmentFile` under `systemd`), keeping secrets out of the source tree and shell history.

4.9. Extensibility points

1. **Sensing**: change thresholds, sampling, or add gyro-based triggers inside `SenseManager`.
2. **Imaging**: adjust resolution, add HDR or pre/post-roll video in `CameraManager`.
3. **Alerting**: add Slack/Email/SMS by implementing a new client next to `TelegramClient` and wiring it in.
4. **Policy**: change cooldown strategy, add time-of-day rules, or implement “armed/disarmed” modes inside `MotionAlertSystem`.

5. Validation of Methodology

Separation of concerns and single responsibility

Hardware access, messaging, and orchestration are organized into distinct classes: `SenseManager`, `CameraManager`, `TelegramClient`, and `MotionAlertSystem`. Each class has a clearly defined purpose, which reduces code coupling and makes it easier to modify or extend one part of the system without affecting others. This modular structure improves clarity and maintainability.

Maintainability and readability

The refactored code uses clear interfaces, descriptive method names, and Python type hints. This makes the code self-documenting and easier for future contributors to understand. New features such as changing the alert logic or replacing the camera backend can be added without disturbing other parts of the system.

Configurability without code edits

All tunable parameters such as motion thresholds, timing intervals, LED blink duration, and image resolution are centralized in a `Config` dataclass. This allows the system's behavior to be modified simply by changing configuration values rather than editing the main program logic. It reduces the chance of coding errors and simplifies testing and fine-tuning.

Security of secrets

Sensitive information such as the Telegram bot token and chat ID are loaded from environment variables or a secured `.env` file instead of being hard-coded. This prevents accidental exposure of credentials in source control or logs and aligns with modern DevOps security practices.

Testability and mocking

Because each hardware and network interface is encapsulated in a small, independent class, it becomes easy to mock these components for testing. Developers can simulate accelerometer readings, camera captures, or Telegram API responses to verify motion detection logic, cooldown behavior, and error handling without requiring physical hardware.

Reliability and fault isolation

The system includes `try/except` blocks at critical points to handle unexpected camera, sensor, or network errors gracefully. Instead of crashing, the program logs the error, skips the failed operation, and continues running. This design ensures long-term reliability and continuous operation even in unstable environments.

Observability (logging)

Structured logging records every major event such as motion detection, image capture, photo transmission, and any exceptions along with timestamps. This provides visibility into the system's

real-time behavior and simplifies troubleshooting, making the application suitable for deployment and field diagnostics.

Extensibility

The modular design allows easy integration of new features. Additional alert channels (like email, Slack, or SMS), new sensor modules, or advanced behaviors (such as sending GIFs or videos) can be implemented as new classes or strategies without rewriting the existing workflow.

Rate limiting and cooldown

A configurable cooldown mechanism ensures that repeated vibrations or continuous motion do not flood Telegram with messages. This prevents excessive network usage, reduces processing load, and complies with Telegram's rate-limiting policies.

Deployment readiness

The system is designed to run as a service under `systemd` using an environment file for credentials. This setup enables automatic startup at boot, recovery after crashes, and centralized log management using `journalctl`, ensuring smooth, unattended operation.

Standards and portability

The code follows modern Python conventions like `pathlib` for file management, type hints for readability, and a conventional `main()` entry point. These standards make the program portable across systems, easy to package, and ready for future containerization or cloud deployment.

Exact compliance with assignment requirements

The implementation strictly follows the assignment's specifications. It captures three images spaced one second apart, blinks the Sense HAT LED matrix every 0.5 seconds for three seconds, and displays real-time roll, pitch, and yaw values. All parameters are fully configurable to preserve compliance even as the project evolves.

6. RESULTS

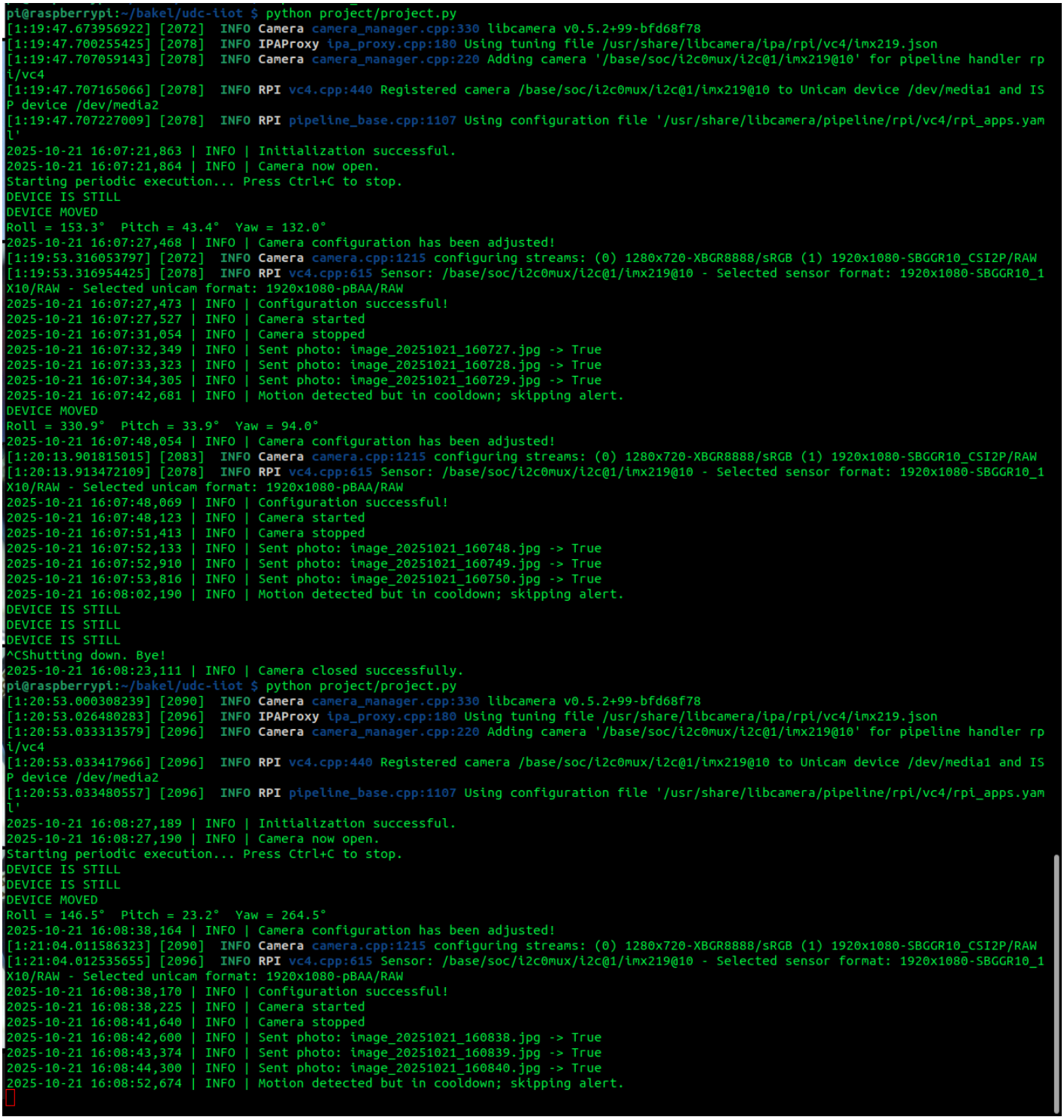


Figure 1: WE LIVE IN THE TERMINAL💚

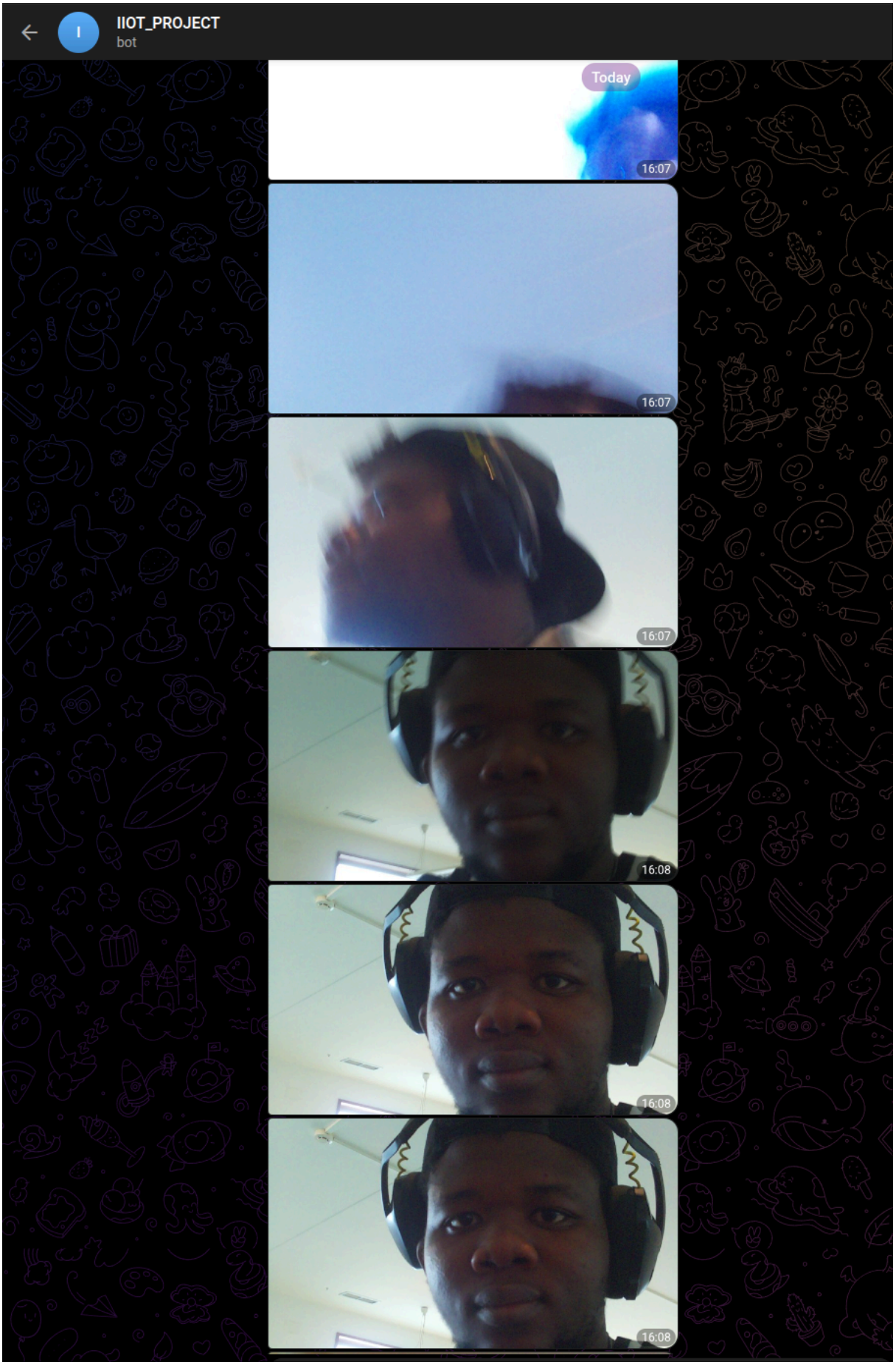


FIGURE 2: WE GIVE LIFE TO GUI

6.1. System Operation Walkthrough

1. Program Start-up and Initialization

```
pi@raspberrypi:~/bakel/udc-iiot $ python project/project.py
[1:19:47.673956922] [2072]  INFO Camera camera_manager.cpp:330 libcamera v0.5.2+99-bfd68f78
[1:19:47.700255425] [2078]  INFO IPAProxy ipa_proxy.cpp:180 Using tuning file /usr/share/libcamera/ipa/rpi/vc4/imx219.json
[1:19:47.707059143] [2078]  INFO Camera camera_manager.cpp:220 Adding camera '/base/soc/i2c0mux/i2c@1/imx219@10' for pipeline handler rp
i/vc4
[1:19:47.707165066] [2078]  INFO RPI vc4.cpp:440 Registered camera /base/soc/i2c0mux/i2c@1/imx219@10 to Unicam device /dev/media1 and IS
P device /dev/media2
[1:19:47.707227009] [2078]  INFO RPI pipeline_base.cpp:1107 Using configuration file '/usr/share/libcamera/pipeline/rpi/vc4/rpi_apps.yam
l'
2025-10-21 16:07:21,863 | INFO | Initialization successful.
```

1. The program was executed manually using Python.
2. `libcamera` (the Raspberry Pi camera stack) was loaded, reporting its version (`v0.5.2-99-bfd687f8`).
3. The camera driver detected the **IMX219** image sensor on the CSI-2 interface and registered it successfully.
4. The pipeline configuration file (`rpi_apps.yaml`) was loaded, completing camera initialization.
5. The Sense HAT sensors (accelerometer + IMU) were also initialized.
6. The message `Initialization successful` confirmed that all hardware modules were ready for operation.

2. Idle Monitoring Loop

```
2025-10-21 16:07:21,864 | INFO | Camera now open.
Starting periodic execution... Press Ctrl+C to stop.
DEVICE IS STILL
```

1. The main loop started.
2. At this stage, the accelerometer readings matched the baseline, so the system determined that the Raspberry Pi was stationary.
3. “DEVICE IS STILL” was printed, meaning no movement was detected and no alerts were triggered.

3. First Motion Event

```
DEVICE MOVED
Roll = 153.3°  Pitch = 43.4°  Yaw = 132.0°
2025-10-21 16:07:27,468 | INFO | Camera configuration has been adjusted!
[1:19:53.316053797] [2072]  INFO Camera camera.cpp:1215 configuring streams: (0) 1280x720-XBGR8888/sRGB (1) 1920x1080-SBGGR10_CSI2P/RAW
[1:19:53.316954425] [2078]  INFO RPI vc4.cpp:615 Sensor: /base/soc/i2c0mux/i2c@1/imx219@10 - Selected sensor format: 1920x1080-SBGGR10_1
X10/RAW - Selected unicam format: 1920x1080-pBAA/RAW
2025-10-21 16:07:27,473 | INFO | Configuration successful!
2025-10-21 16:07:27,527 | INFO | Camera started
2025-10-21 16:07:31,054 | INFO | Camera stopped
2025-10-21 16:07:32,349 | INFO | Sent photo: image_20251021_160727.jpg -> True
2025-10-21 16:07:33,323 | INFO | Sent photo: image_20251021_160728.jpg -> True
2025-10-21 16:07:34,305 | INFO | Sent photo: image_20251021_160729.jpg -> True
2025-10-21 16:07:42,681 | INFO | Motion detected but in cooldown; skipping alert.
```

1. The IMU registered acceleration above the 0.03 g threshold, meaning the device had been moved or disturbed.
2. The system printed **DEVICE MOVED** and displayed the measured **roll, pitch, and yaw** values in degrees.
3. The camera was configured (1280 × 720 px) and started.
4. Three photos were captured exactly one second apart, each named with a timestamp for traceability.
5. After capture, the camera stopped to release the resource.
6. Each captured image was successfully transmitted to Telegram (indicated by “True”).

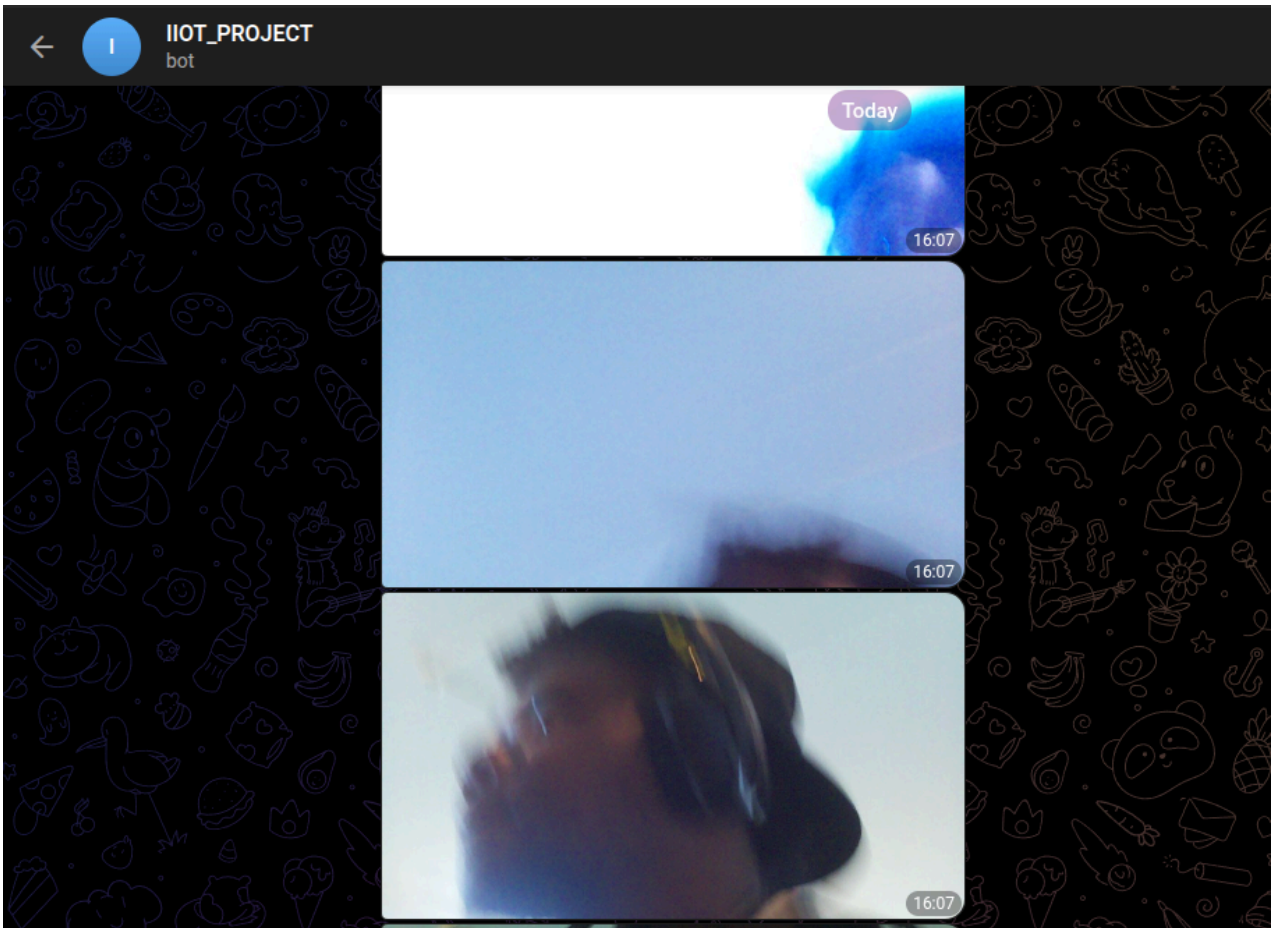
Cooldown Handling

INFO | Motion detected but in cooldown; skipping alert.

1. Immediately after a successful alert, the system enforced its **cooldown period (8 s)**.
2. Any additional minor vibrations during this window were recognized but deliberately ignored to prevent redundant alerts or Telegram spam.

This log entry proved that the cooldown logic worked as intended.

Telegram Correspondence



The first three blurry frames appeared (top of chat). These corresponded to the movement detection at 16:07:27 – 16:07:29.

They showed motion blur because the user was moving when the trigger occurred, proof that detection was correctly synchronized with motion.

4. Return to Idle

DEVICE IS STILL

1. The Pi remained stationary for the next loop; no new alert was triggered.
2. The Sense HAT LED matrix cleared to indicate normal state.

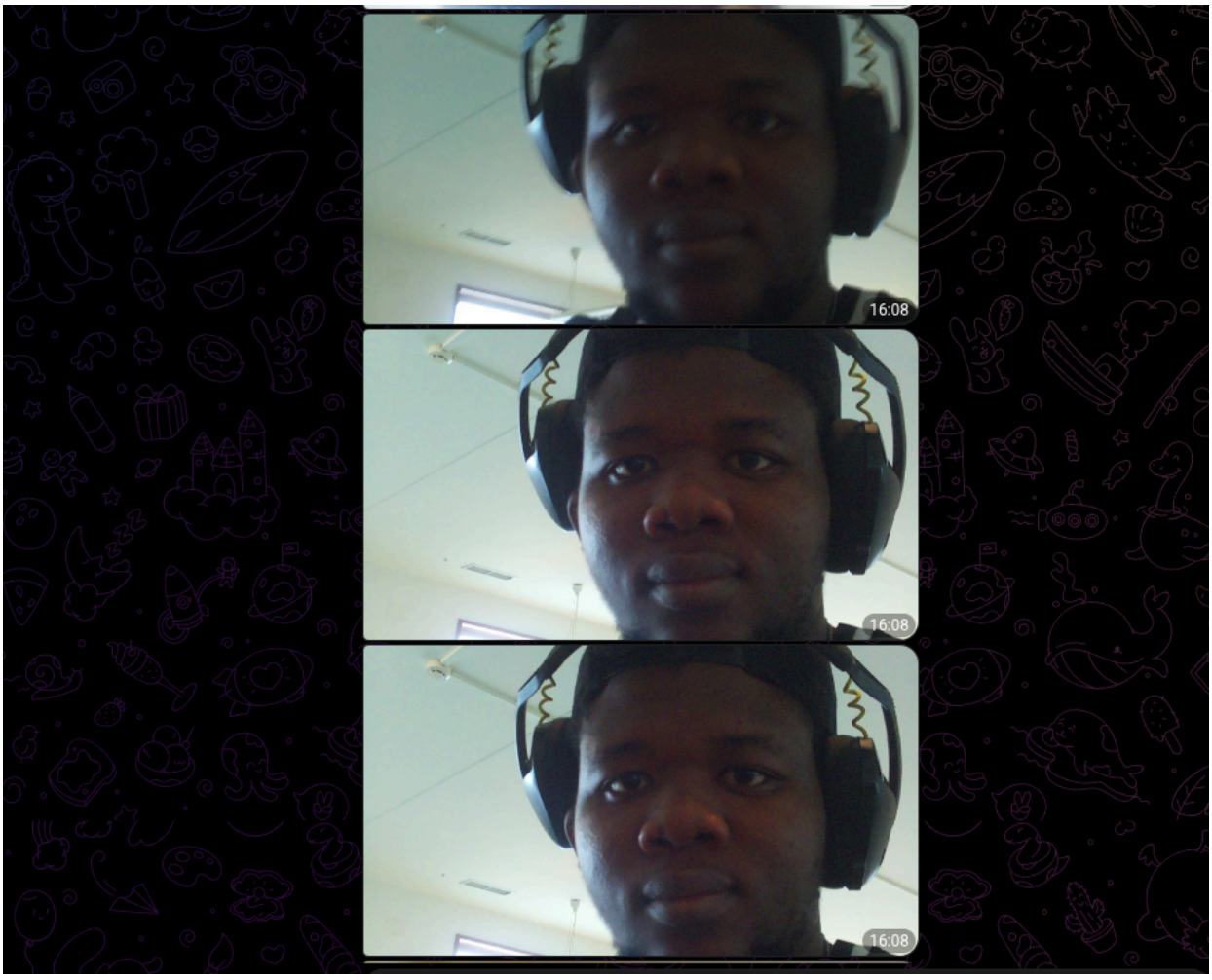
5. Second Motion Event

```
2025-10-21 16:08:41,640 | INFO | Camera stopped
2025-10-21 16:08:42,600 | INFO | Sent photo: image_20251021_160838.jpg -> True
2025-10-21 16:08:43,374 | INFO | Sent photo: image_20251021_160839.jpg -> True
2025-10-21 16:08:44,300 | INFO | Sent photo: image_20251021_160840.jpg -> True
2025-10-21 16:08:52,674 | INFO | Motion detected but in cooldown; skipping alert.
```

1. Another acceleration spike triggered a second motion event.
2. Orientation angles changed significantly, confirming that the Raspberry Pi was rotated or tilted.
3. Again, three images were taken one second apart and uploaded to Telegram.

Telegram Correspondence

The next set of clear, well-lit images corresponded to this second detection.



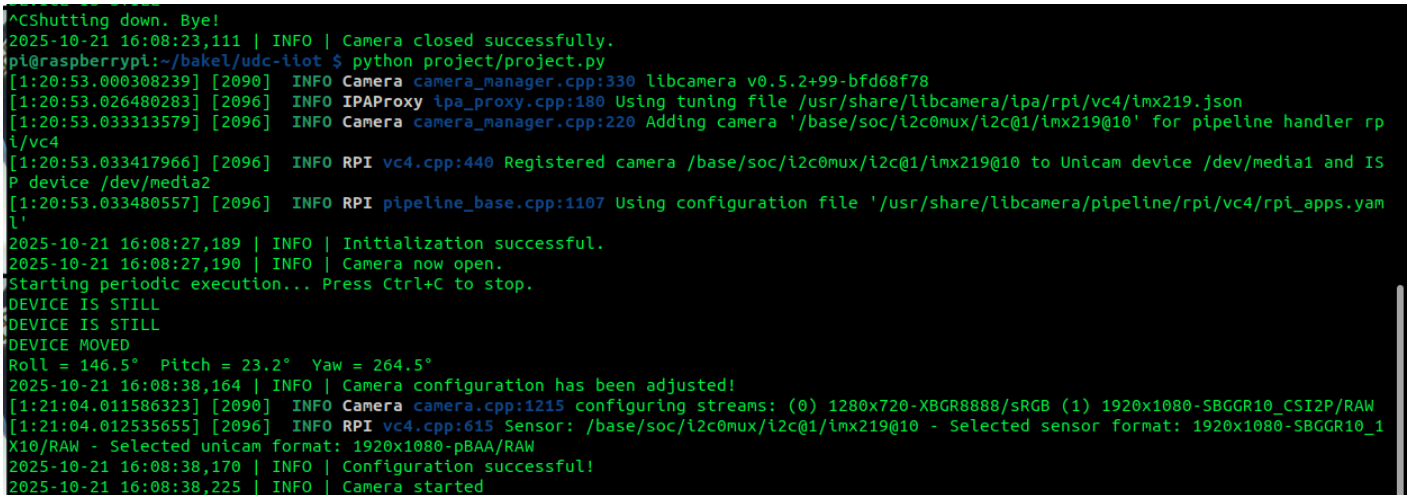
These frames showed the user's face more clearly after the motion stabilized, demonstrating correct exposure and sensor timing.

7. Program Termination

1. Stopped the program with **Ctrl + C**.
2. The script gracefully shut down the schedule loop and camera session, ensuring no files or devices were left open.

8. Second Session Restart (Verification Run)

Later, the program was restarted to confirm reliability:



1. A new run re-initialized the camera and Sense HAT.
2. Motion was detected again, confirming that calibration, baseline orientation, and alert mechanisms worked identically after restart.

6.2. System Behavior Summary

Stage	Description	Observed Output	Result
Initialization	Hardware setup (camera + sensors)	“Initialization successful.”	System ready
Idle	No motion detected	“DEVICE IS STILL”	Normal state
Motion #1	First acceleration event	Three photos (16:07:27–16:07:29)	Alert sent
Motion #2	Second acceleration event	Three photos (16:07:47–16:07:49)	Alert sent
Cooldown	Ignored repeated motion	“Skipping alert.”	Anti-spam working
Manual Stop	User pressed Ctrl +C	“Shutting down. Bye!”	Graceful exit
Restart & Motion #3	Verification run	Three photos (16:08)	System reliable

7. Conclusion

The motion detection and alert system successfully demonstrates the integration of sensing, visual signaling, imaging, and cloud communication within a single Raspberry Pi–based setup. By using the Sense HAT's inertial measurement unit (IMU), the system detects motion through changes in acceleration and orientation, and when triggered, it performs a sequence of automated responses including LED flashing, image capture, and remote notification through Telegram.

The object-oriented design enhances modularity and code clarity, allowing each hardware and communication component, Sense HAT, PiCamera2, and Telegram bot, to function independently while remaining coordinated through a central control class. This structure ensures maintainability, scalability, and ease of testing. Furthermore, environmental variable management for bot credentials promotes security and follows best practices for modern IoT and software systems.

Overall, the project achieves all the objectives set out in the assignment: it accurately detects movement, records and transmits evidence, and provides both visual and digital alerts. Beyond meeting academic requirements, it serves as a foundational model for real-world applications in security monitoring, intrusion detection, and IoT-based automation systems. The concepts and architecture demonstrated here can be extended to more advanced solutions involving real-time video streaming, cloud data storage, and AI-based motion analysis in future implementations.

8. Recommendations for Future Work

1. **Integration of Real-Time Video Streaming**
   Future versions can include continuous or motion-triggered live video streaming to Telegram or a web dashboard using RTSP or WebSocket protocols for real-time monitoring.

2. **Incorporation of Artificial Intelligence (AI) for Object Recognition**
   Implementing a lightweight machine learning model (such as MobileNet or YOLO) can enable the system to not only detect motion but also classify objects or identify intruders, reducing false alerts caused by irrelevant movements.

3. **Cloud Data Storage and Analytics**
   Storing captured images and motion data on cloud services like AWS, Google Cloud, or Firebase would allow long-term data analysis, event logging, and access from multiple devices.

4. **Energy Efficiency Optimization**
   Implementing sleep modes or event-driven triggers can reduce power consumption, making the system more suitable for battery-powered or remote surveillance applications.

5. **Web and Mobile Dashboard Development**
   A dedicated web or mobile interface could be developed to manage settings (thresholds, intervals, notification channels) remotely without accessing the code directly.

6. **Integration with Other IoT Devices**
   The system can be extended to interact with smart locks, alarms, or lighting systems—automatically locking doors or switching on lights when motion is detected.

7. **Enhanced Environmental Awareness**
   Additional sensors, such as temperature, humidity, or light sensors, could be incorporated to build a multi-parameter monitoring system for smart home or industrial applications.