Public Transportation Efficiency with Smart Stop Density Monitoring System

Institution: Sixfab Company

Internship Period: 08.07.2025-19.10.2025

Submission Date:

Student Full Name	Student Surname	Student ID	Student University	Student Department	Student Unit
Ömer Ercan	DAYAN	150122050	Marmara University	Engineering Faculty	Computer Engineering

Content

Abstract

This report presents the design and implementation of the PTE_SSDMS (Public Transportation Efficiency with Smart Stop Density Monitoring System), an intelligent bus stop density monitoring system. The system, developed using Raspberry Pi 4 and the Sixfab modem kit, provides basic image processing capabilities and real-time data transmission. This enables real-time analysis of passenger density at bus stops, aiming to inform both passengers and management units.

The application was developed using Python-based software and integrated with ThingsBoard, allowing data to be processed and visualized on a cloud-based platform. Multiple versions of the project were tested, and system performance was continuously improved through version control managed on GitHub. The results demonstrate that the system functions successfully and indicate that future versions can be enhanced by integrating more advanced image processing techniques and AI-based analytics, thus expanding the overall scope of the project.
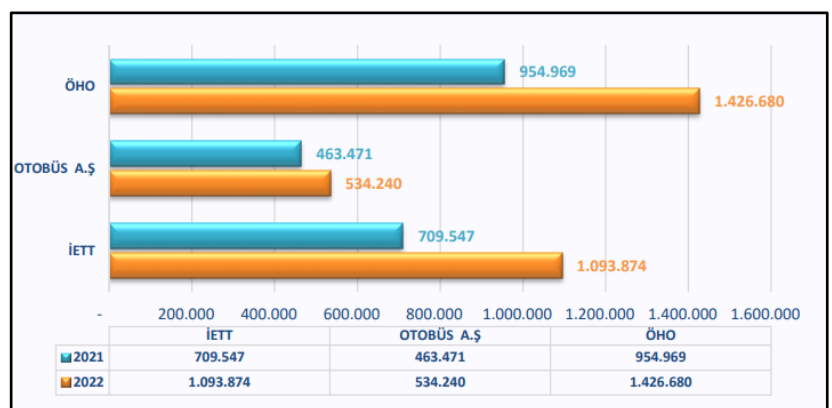
Problem Description

Imagine working at a company where you're expected to be in the office every weekday at 9:00 AM. A citizen wakes up early one morning but doesn't even have time to make a cup of coffee—let alone enjoy a proper breakfast. "I'll grab a coffee on the way," the person thinks, rushing out the door. Still half-asleep from a poor night's rest, the person aims to catch the 08:00 bus, leaving home at 07:45.

But at the stop, the person finds at least 25 other people waiting to board the same bus.

The person thinks, "Maybe I should've taken the other line from the next stop—would I have made it in time?" As the person hesitates, the bus that arrives is already nearly full. Squeezing in is the only option. And worse still, this is just the beginning: after the bus, the person needs to transfer to the metro—which will be even more crowded.

This isn't just one person's bad morning; it's a daily struggle shared by thousands of city residents. And this struggle is reflected in actual data: in Istanbul alone, between January and June 2022, the number of bus passengers reached millions each month—ranging from 2.5 million in January to over 3.1 million in May. These figures clearly show that in a megacity like Istanbul, managing such

Graph 1: Number of bus trips by month in 2022

a massive volume of passengers is not optional; it requires serious planning and constant adjustments to prevent inefficiencies and frustration.

To make things worse, the same overcrowded bus stop becomes almost completely deserted later in the day. The buses pass by empty. While the line is overloaded in the morning, it runs nearly idle at other times, leading to fuel waste and inefficiency.

But what if we could see the crowd levels at bus stops in advance?

What if just one person could avoid being late by choosing a different route?

This system is designed to minimize exactly that problem.
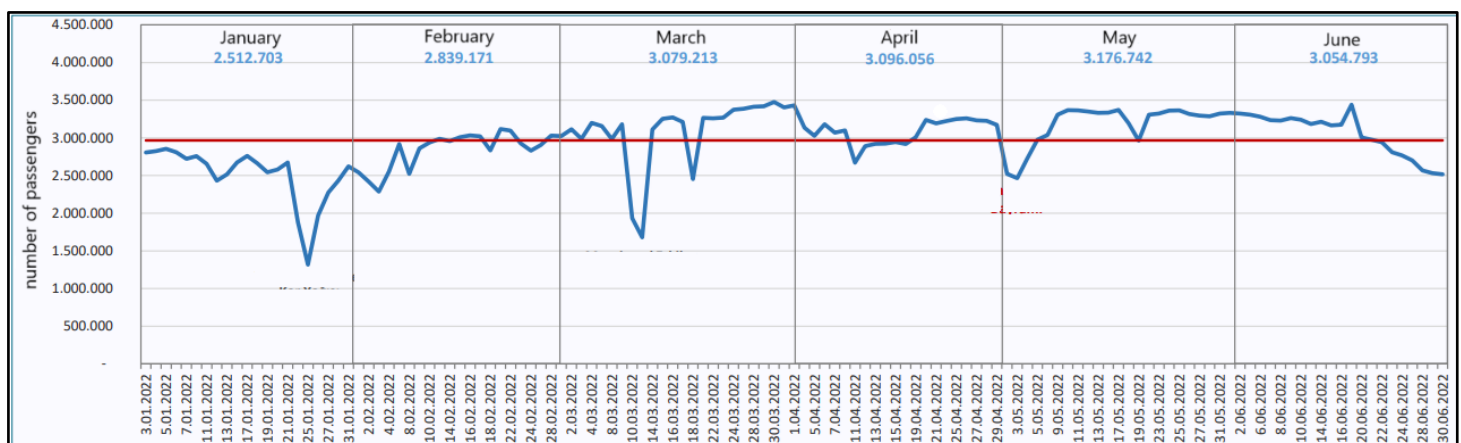
Graph 2: Number of bus passengers by month in 2022

Solution Approaches

1. Description of General Solution Idea

To address the challenges faced in urban public transportation, this project explores an automated approach for monitoring passenger density at bus stops. After evaluating various solution ideas and technical approaches, I developed a system that combines image processing and IoT integration to offer a practical and scalable monitoring solution. This resulted in the creation of the **Public Transportation Efficiency with Smart Stop Density Monitoring System (PTE_SSDMS)**.

The core objective of the project is to enable smarter and more efficient management of bus-based transportation systems, especially during peak hours like morning and evening rush periods. By providing real-time information on bus stop occupancy, the system can help both passengers and transit authorities make more informed decisions, reducing overcrowding and improving overall service quality.

The system is built using a Raspberry Pi 4 and a camera module, which captures periodic images of bus stops. These images are analyzed using lightweight image processing techniques to detect changes in the scene. When a significant change is detected, the system determines whether it involves human activity, and if confirmed, it sends the event data to a cloud-based IoT platform like ThingsBoard. Captured images are also archived to Google Drive for later review.

Although currently focused on detecting the presence of people, the system establishes a foundation for more advanced capabilities such as passenger counting, density estimation, and prediction-based route planning. With further improvements, it can be transformed into a fully autonomous tool for managing and optimizing public transportation flow in real-time

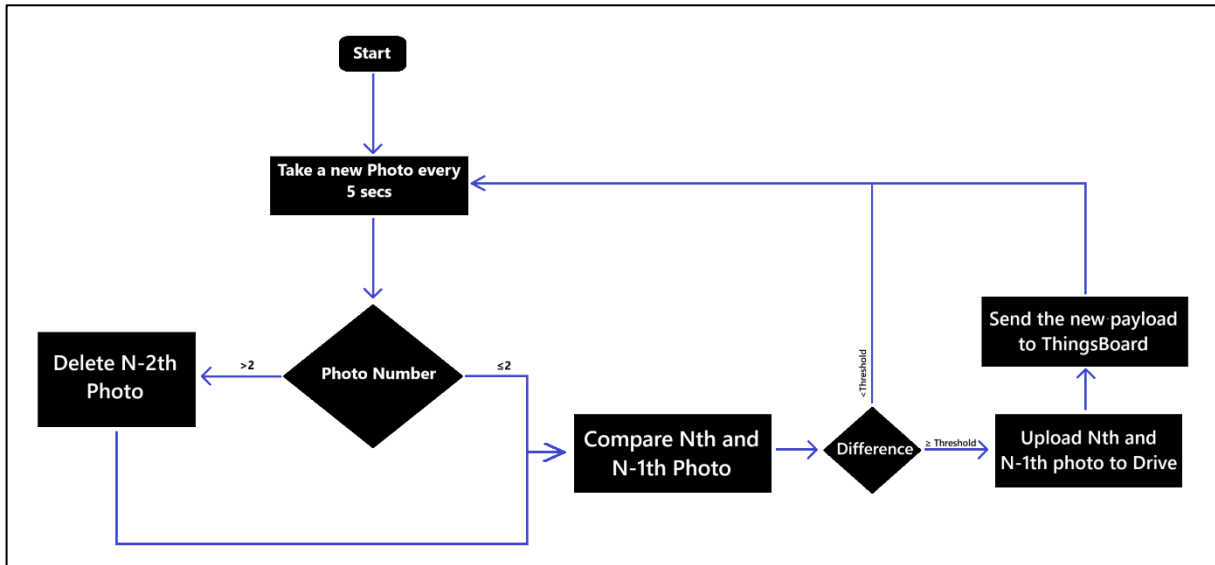## 2. Improved Solutions

2.1.   Basic Change Detection System

Image 1: FlowChart

- ## The Idea Description

The first solution, Basic Change Detection System, was designed as a fundamental prototype to validate the overall workflow of an automated monitoring system using a Raspberry Pi and camera module.

The primary objective was to determine whether a visible change occurred in the environment by comparing consecutive images, without focusing on identifying the exact nature or source of the change. This approach aimed to create a quick, low-cost starting point for more complex solutions in the future.

It integrated three core steps: capturing images at fixed intervals, performing a pixel-level comparison between current and previous images, and sending change notifications to ThingsBoard along with uploading photos to Google Drive for record-keeping.

- ## Solution Steps

1. **System Initialization:** The Raspberry Pi powers up, initializes the camera module, and checks for internet connectivity to ensure ThingsBoard and Google Drive integration.
2. **Image Capture Loop:** Photos are taken every 5 seconds and stored temporarily in local memory for quick access.
3. **Image Comparison:** The most recent photo is compared with the previously saved one using a pixel-based difference algorithm.
4. **Threshold Evaluation:** If the computed difference exceeds a defined threshold (default 30%), the system concludes that a significant environmental change has occurred.
5. **Cloud Actions:** The triggered images (current and previous) are uploaded to Google Drive, then a change event notification is sent to ThingsBoard, updating its dashboard.
6. **Cleanup and Repeat:** The system keeps only the last two images for memory efficiency, deletes older files, and restarts the loop from step 2.

- ## Detailed Explanation, CodeSnapshot & GitHub Link

This version operated on a simple yet functional pipeline:

1. **Image Capture:** Photos were taken every 5 seconds using the Raspberry Pi camera module.
2. **Comparison Algorithm:** The algorithm compared the most recent image with the previously saved one, using pixel-level difference to detect significant environmental changes. A configurable threshold value determined whether a change was relevant or a false trigger caused by lighting variations or sensor noise.
3. **Cloud Integration:** Upon detecting a change, both images were uploaded to Google Drive for archival purposes, while ThingsBoard received a notification, updating its dashboard to reflect the detected event.,

The approach deliberately avoided heavy processing techniques such as object recognition or video analysis to keep system resource usage minimal. By focusing on building a working baseline, it allowed testing of hardware integration, workflow timing, and cloud connectivity before moving on to more complex solutions.

GitHub Link:

The relevant code for image capture, pixel comparison, and Google Drive upload routines is available here:

https://github.com/ErcanPasha/inter_project

Key scripts used:

- *camera_loop.py* – Handles continuous photo capture.
- *compare_image.py* – Implements pixel-difference comparison.
- *upload_drive.py* – Uploads flagged photos to Google Drive.

• Pros/Cons and Issues Encountered

Pros

- ➢ **Simple and Fast Deployment:** Quick to implement and validate the core monitoring workflow without heavy computational requirements.
- ➢ **Low Hardware Requirements:** Uses only a Raspberry Pi 4, a camera module, and internet connectivity.
- ➢ **Cloud Connectivity Demonstration:** Successfully integrated with ThingsBoard for dashboard visualization and Google Drive for image archiving.
- ➢ **Configurable Threshold:** Allowed fine-tuning of change sensitivity without modifying the algorithm logic.

Cons

- ➢ **False Positives:** Pixel-level comparison frequently flagged lighting changes or minor sensor noise as real events.

- ➢ **No Change Localization:** The algorithm detected "a change" but not *where* or *what* changed.
- ➢ **Cloud Dependency:** Upload performance heavily depended on internet stability, sometimes causing delays.
- ➢ **Timing Issues:** Occasional mismatches between camera capture timing and comparison logic caused workflow delays.

Issues Encountered

During implementation, several challenges emerged, primarily involving threshold adjustment and cloud integration. Fine-tuning the sensitivity required multiple test iterations to reduce false positives, while network instability occasionally disrupted ThingsBoard and Google Drive uploads. Additionally, some timing mismatches between image capture and comparison caused minor workflow delays, highlighting the limitations of this basic prototype for larger-scale real-time applications.
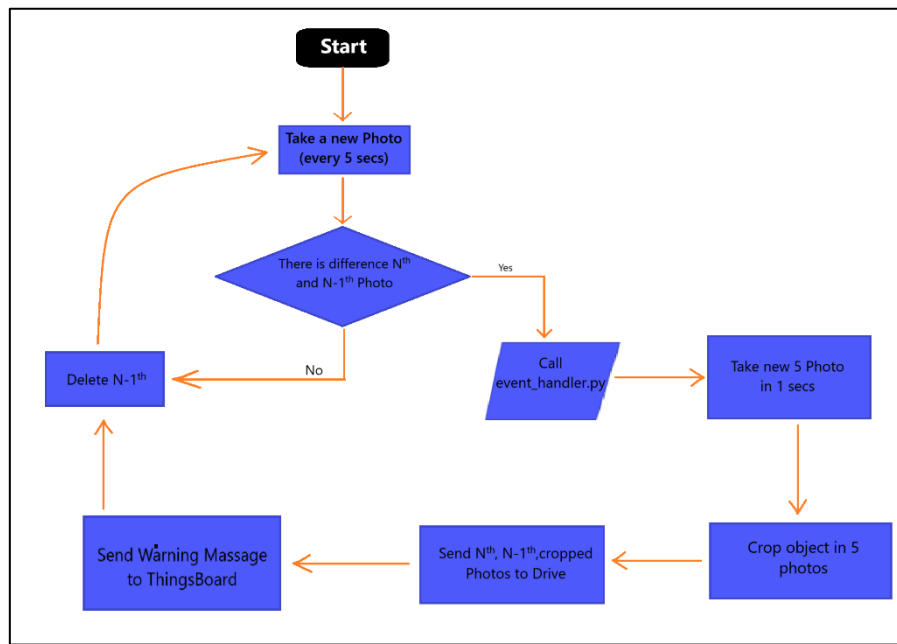
2.2.   Cloud-Integrated Monitoring System

Image 2: FlowChart

- ## The Idea Description

    The second solution, **Cloud-Integrated Monitoring System**, was designed to extend the initial prototype by integrating cloud storage and improving data accessibility.

    While the first version focused primarily on detecting changes and issuing basic alerts, this iteration aimed to create a more robust workflow that could automatically archive captured data to cloud storage, ensuring accessibility for remote monitoring and later analysis. The approach was motivated by the need for a scalable, persistent data storage solution that does not rely solely on the local memory of the Raspberry Pi.

- ## Solution Steps

1. **Image Capture Loop:** The Raspberry Pi camera captured images at fixed intervals (5 seconds) similar to the previous version.

2. **Change Detection Algorithm:** The pixel-based comparison method from Version 1 was reused to determine environmental changes.

3. **Cloud Upload Integration:** Instead of storing images solely on the device, captured images (both "before" and "after" change detection) were automatically uploaded to **Google Drive** using an automated upload pipeline.

4. **ThingsBoard Notification:** Change detection results were sent to ThingsBoard, updating the cloud dashboard in near real time.

5. **File Management & Cleanup:** Local storage was managed by retaining only the most recent two images, ensuring the device remained optimized and functional during continuous operation.

- ## Detailed Explanation, CodeSnapshot & GitHub Link

    This version maintained the lightweight design philosophy of Version 1 but introduced a cloud integration pipeline using **rclone**. The image capture and difference detection scripts

remained largely unchanged, focusing on quick pixel-based comparison for speed and simplicity. The major enhancement was the seamless transfer of detected event images to Google Drive, ensuring long-term storage without overloading the Raspberry Pi's local disk.

Additionally, data notifications continued through ThingsBoard, allowing the visualization of detection events in a user-friendly dashboard. With cloud integration, the system improved accessibility, enabling off-site inspection of images and event logs.

GitHub Link:

The full implementation can be accessed here:

https://github.com/ErcanPasha/inter_project

Key scripts include:

- *camera_loop.py* – Continuous image capturing.
- *compare_image.py* – Image difference algorithm.
- *upload_drive.py* – Automated upload script via rclone to Google Drive.

• Pros/Cons and Issues Encountered

.Pros

➢ **Remote Accessibility:** Detected change events and associated images were stored on Google Drive, allowing easy remote access and long-term data retention.
➢ **Improved Workflow:** The new main cycle improved process handling by managing capture, comparison, and upload sequences more reliably.
➢ **Integrated Event Handling:** The change detection and event handling scripts (*change_detector.py*, *event_handler.py*, and *main_cycle.py*) worked together for a smoother end-to-end operation**..**
➢ **Dashboard Notification:** ThingsBoard integration remained intact, providing real-time event visualization.

Cons

➢ **Cloud Dependency:** The system heavily relied on an active internet connection; unstable networks caused delays or failed uploads.
➢ **Token Management:** rclone-based Google Drive integration occasionally required manual token refreshing, reducing full automation potential.
➢ **Limited Processing Depth:** Like Version 1, the solution still only detected changes, not their specific context or type.

Issues Encountered

During development, the integration of cloud upload functionality introduced several challenges. Network instability sometimes disrupted the automated pipeline, requiring retry logic. Token refresh requirements occasionally halted the upload process until manual intervention was performed. Additionally, balancing timing between image capture, change detection, and event handling required fine-tuning, but once stabilized, the system performed consistently and reliably for prototype-level deployment..
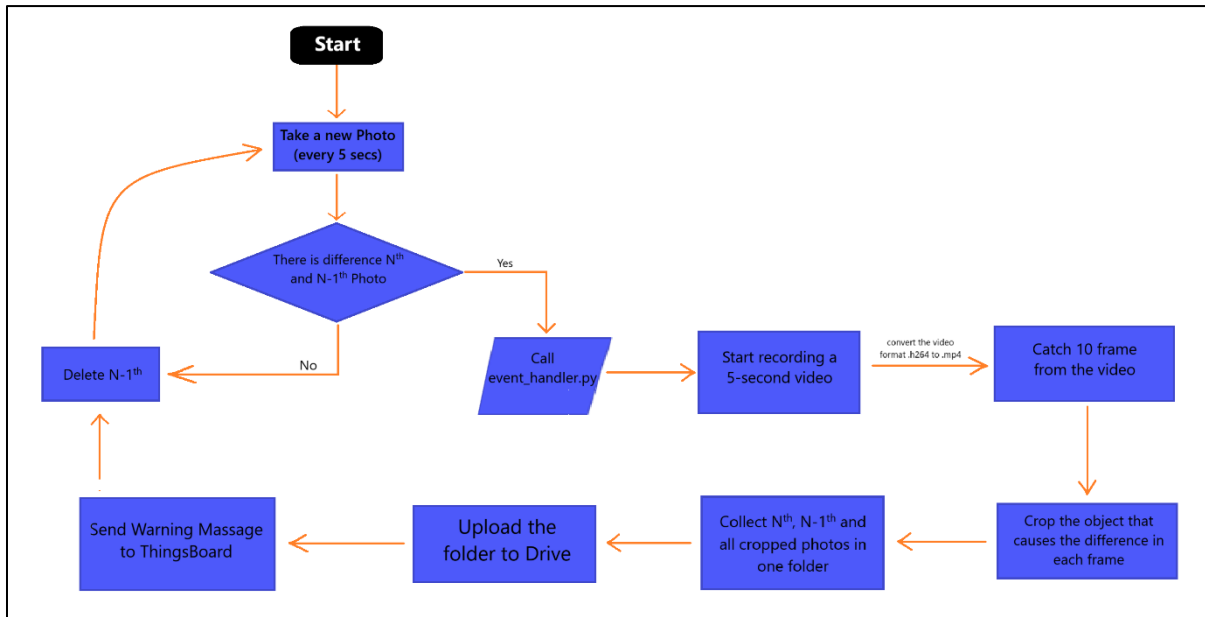
2.3.   Optimized IoT Observation Framework

Image 3: FlowChart

• The Idea Description

The third and final solution, **Optimized IoT Observation Framework**, represents the production-ready version of the project. Building upon the core change detection and event handling logic of previous versions, this iteration focused on improving system performance, reducing redundant file storage, and streamlining cloud integration.

The final version implemented automated cleanup mechanisms to minimize disk usage, improved timing within the main cycle, and optimized event handling logic for faster decision-making. With these enhancements, the system became more stable and efficient, making it suitable as a deliverable prototype.

• Solution Steps

1. **System Initialization:** The Raspberry Pi initializes required modules, including camera setup, network checks, and script dependencies.
2. **Image Capture Loop:** *main_cycle.py* captures images periodically and triggers processing workflows.
3. **Change Detection:** *change_detector.py* compares current and previous images to detect significant differences.
4. **Event Handling:** When a change is detected, *event_handler.py* determines the appropriate response, including preparing flagged images and sending change notifications.
5. **Cloud Actions:** Images and event data are automatically uploaded to Google Drive, while ThingsBoard receives change notifications for dashboard visualization.
6. **File Cleanup:** Automated deletion of old images ensures the system retains only the most recent two, maintaining minimal local storage usage.

- ## Detailed Explanation, CodeSnapshot & GitHub Link

This optimized version refined the original workflow to create a stable and resource-efficient solution. A dedicated image cleanup routine reduced storage overhead by systematically removing old files after processing. Synchronization between *change_detector.py*, *event_handler.py*, and *main_cycle.py* was improved to prevent timing conflicts and skipped comparisons.

Cloud integration remained intact but was optimized for stability: uploads to Google Drive were queued to prevent failures on slower connections, and ThingsBoard notifications were confirmed before removing temporary files. The final system delivered reliable event detection, near real-time cloud notifications, and robust memory management.

GitHub Link:

The final solution implementation is available here:

https://github.com/ErcanPasha/inter_project

Key scripts:

➢ *change_detector.py* – Core change detection algorithm.

➢ *event_handler.py* – Handles response actions after detection.

➢ *main_cycle.py* – Orchestrates capture, detection, and cloud integration workflows.

- ## Pros/Cons and Issues Encountered

Pros

➢ **Optimized Performance:** Improved timing and automatic file cleanup reduced processing delays and storage overhead.
➢ **Stable Workflow:** Enhanced synchronization between modules ensured reliable event handling and fewer missed detections.
➢ **Cloud Integration:** Google Drive and ThingsBoard connections were stabilized, providing consistent uploads and dashboard updates.
➢ **Deliverable Ready:** Designed as the final, production-oriented version, making it suitable for real deployment.

Cons

➢ **Limited Detection Capability:** Like previous versions, it still focused only on detecting changes, not identifying specific objects or causes.
➢ **Cloud Dependency:** Continuous internet access remained critical; offline operation was limited.
➢ **Initial Setup Complexity:** Required proper configuration of cloud credentials and rclone tokens before deployment.

Issues Encountered

During development, synchronization issues between detection and upload tasks occasionally caused image duplication or upload delays. Adjusting queue timing and adding confirmation checks resolved these problems. Cloud upload failures in unstable networks were mitigated by implementing retries, but full offline resilience was outside the scope of this version. Despite these challenges, the final solution achieved consistent, optimized performance suitable for long-term use.
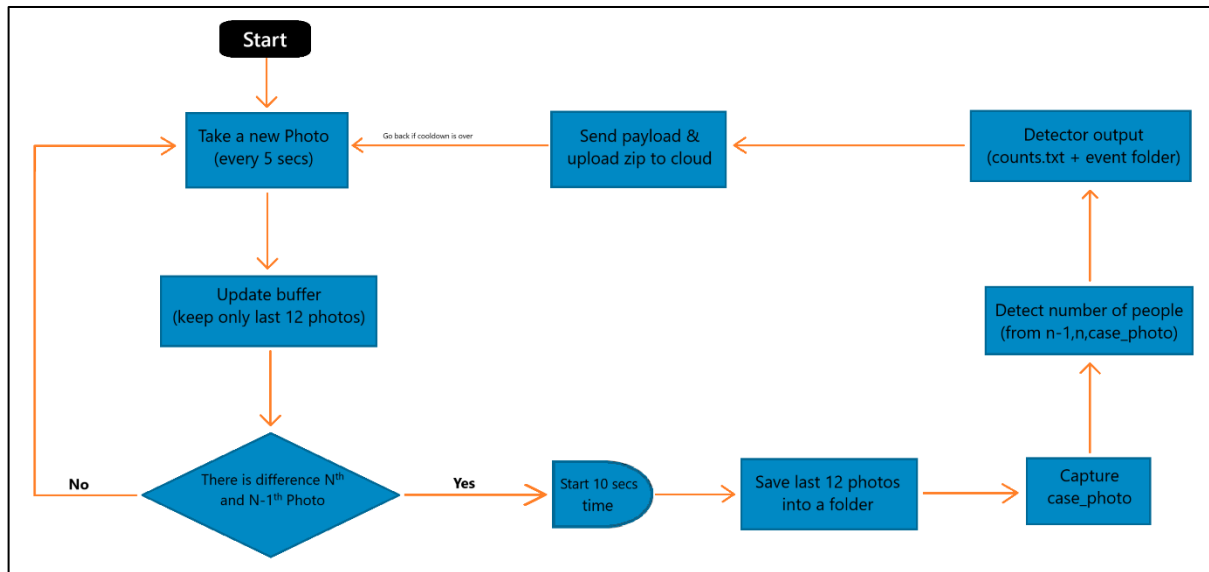
2.4.   Human Detection

Image 4: FlowChart

- ## The Idea Description

  The Human Detection module represents an enhancement over the previous Optimized IoT Observation Framework by introducing context-aware filtering for detected changes. While the earlier versions of the system focused on identifying environmental changes through pixel-based difference algorithms, they lacked the ability to understand *what* caused the change. This often led to false positives due to lighting variations, camera shake, or moving objects unrelated to the system's purpose.

  This improved version addresses that limitation by introducing a lightweight human presence detection algorithm. After a change is detected, the system performs an additional verification step to assess whether the observed change involves the presence of a person. This makes the event data significantly more meaningful and aligns better with the project's goal: monitoring human density at public transport stops.

  By implementing this enhancement, the system reduces noise in the collected data, improves event accuracy, and lays the foundation for future integration of object detection or AI-based counting models.

- ## Solution Steps
1. **System Initialization:** The Raspberry Pi powers up and initializes the camera module, internet connection, and all core Python scripts including *main_cycle.py, change_detector.py, event_handler.py*, and *detector.py*.
2. **Image Capture Loop:** The system captures images at regular intervals (every 5 seconds) using *camera_manager.py* and temporarily stores them locally for processing.
3. **Change Detection:** The *change_detector.py* script compares the most recent image with the previous one using a pixel-level difference algorithm. If the computed difference exceeds a configurable threshold (changeable value), a change is detected.

4. **Human Verification:** When a change is detected, control is passed to *detector.py*, which applies a basic image processing technique (to assess whether the change involves a human figure. Only if a person is likely present does the pipeline proceed.

5. **Event Handling and Logging:** If human presence is confirmed, *event_handler.py* prepares relevant image data and formats a payload for cloud services. It also logs the event locally, and prints feedback for debugging**.**

6. **Cloud Upload and Notification:** When a person is detected, the system packages the last 12 captured images (stored in the Images/ directory) into a compressed .zip file. This archive provides a richer context for post-event review and is uploaded to Google Drive using the *uploader.py* module via rclone. Simultaneously, a simple telemetry payload such as { "Folder Name": "event_20250807_105300", "Payload Sequence Number": 4, "Person ": 1 } is sent to the ThingsBoard cloud dashboard using MQTT, allowing real-time visualization of the event.

7. **Cleanup and Loop Reset:** To maintain storage efficiency, the system continuously manages the Images/ folder as a rolling buffer. After each new image capture, it deletes the image that is 13 steps old (e.g., img_1.jpg when img_13.jpg is created), ensuring that only the most recent 12 images are retained at all times. After this cleanup, the loop restarts following a short cooldown period to avoid redundant detections.

- ## **Detailed Explanation, CodeSnapshot & GitHub Link**

This version adds a simple **person-aware check** on top of the existing change detection. Previously, any large visual difference could trigger an event. Now, we verify **whether the change is caused by a human** before we upload anything. This cuts false alarms from lighting shifts, moving shadows, or passing objects.

How the pipeline runs (production):

1. camera_manager.py saves a photo every **5 seconds** into Images/ and auto-cleans older files (keeps the most recent 12).
2. change_detector.py compares the latest two images. If the difference score exceeds the **threshold**, we proceed.
3. detector.py loads a lightweight **YOLOv4-tiny** model and **counts people** in the most recent image (case_photo).
4. If ≥1 person is found, uploader.py zips the last 12 images and uploads to Google Drive (via **rclone**). In parallel, an MQTT payload is sent to ThingsBoard for logging and dashboard display.

The newly added *detector.py* module loads a lightweight **YOLOv4-tiny** model and performs a basic inference to count how many people are visible in the most recent image (case_photo). This approach allows the system to avoid false triggers caused by irrelevant movements, such as lighting shifts, passing objects, or shadows. Although basic false trigger issues have been solved, the simple-level human detection used may still not always give accurate results for complex or unclear images.

The core of the project's idea is based on *human detection*, handled by the detector.py script, which uses the YOLOv4-tiny model. To run, this model requires the **yolov4-tiny.cfg**, **yolov4-tiny.weights**, and **coco.names** files to be present locally. On devices like the

Raspberry Pi, this concept is still open to İMPROVEMENT, but in its current form, it depends on having these files available.

To allow a simple, hands-on experience with detector.py, an additional script called **manual_detector.py** was developed. This script provides a way to test the *detector.py* functionality using image inputs without requiring the other main files.

For a manual quick test (local or on Pi):

➤ manual_detector.py lets you bypass the full pipeline and test only the human detection step.

```bash
python manual_detector.py path/to/image.jpg
```

➤ While running, manual_detector.py uses the original detector.py file as its source. For detector.py to work, the previously mentioned files (yolov4-tiny.cfg, yolov4-tiny.weights, and coco.names) must be in the same directory.

➤ Below are the CMD commands for installing these files. Make sure to navigate to the relevant directory before running the installation commands.

```
curl -L -o coco.names https://raw.githubusercontent.com/pjreddie/darknet/master/data/coco.names

curl -L -o yolov4-tiny.cfg https://raw.githubusercontent.com/AlexeyAB/darknet/master/cfg/yolov4-tiny.cfg

curl -L -o yolov4-tiny.weights https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v4_pre/yolov4-tiny.weights
```

Note: Large model files are committed to Git to keep the repo light.

GitHub Link:

The final solution implementation is available here:

https://github.com/ErcanPasha/inter_project

Key scripts:

➤ *camera_manager.py* – Orchestrates the entire capture/detect/upload

➤ *change_detector.py* – Detects visual difference between two consecutive images

➤ detector.py – Uses YOLOv4-tiny to count people in the image.

➤ *uploader.py* – Compresses last 12 images and uploads archive to Google Drive.

- **Pros/Cons and Issues Encountered**

.Pros

➢ **More Meaningful Detection:** This version not only checks if there is a change in the image but also verifies if the change is caused by a person. This improves the quality of event detection.

➢ **Clear Task Separation:** Each script in the system (*camera_manager.py*, *change_detector.py*, *detector.py*, *uploader.py*) has a specific job. This makes the code easier to understand, test, and update in the future.

➢ **Efficient Storage Management:** The system keeps only the latest 12 images and deletes older ones automatically. This prevents storage problems.

➢ **Improved Zip Handling:** The image zipping process was moved from *camera_manager.py* to *uploader.py*. This made the upload process more stable and reduced errors.

➢ **Threshold Optimization:** The change detection threshold was improved from around 50,000 to 120,000. This reduced false triggers caused by small lighting or background changes.

Cons

➢ **Simplified Detection Model:** Uses a lightweight YOLOv4-tiny model (not heavier models like YOLOv5/YOLOv8), so it may miss people in crowded or low-light scenes.

➢ **Cold Camera Start:** Sometimes, the first few images are too dark because the camera needs time to adjust after starting.

➢ **Sequential Detection:** The system first checks for change and then checks for people. In fast-moving scenes, this may not react quickly enough.

Issues Encountered

During development, a few technical challenges had to be addressed. The change detection threshold required several adjustments to find the right balance between too many false alarms and missing real activity. Early on, the image zipping logic was handled inside *camera_manager.py*, which caused problems when preparing the correct files for upload. This issue was solved by moving the zip process into *uploader.py*, making the upload more reliable. In addition, the human detection module sometimes gave inaccurate or unstable results. These were improved by tweaking detection settings and ensuring the image quality was suitable for analysis. Overall, these updates made the system more stable and suitable for real-world use.

General Hardware

The hardware setup used in all solution iterations was minimal yet sufficient for prototype-level development. The core platform was a **Raspberry Pi 4** single-board computer, chosen for its compact size, integrated GPIO support, and compatibility with various IoT applications. A **Raspberry Pi Camera Module** was connected for image capture, serving as the primary sensor for change detection. For network connectivity, a **Sixfab Cellular Modem Kit** was integrated, ensuring reliable internet access even in remote deployment scenarios.

Additional components included a standard **microSD card** (32 GB) for the operating system and local storage, a power provider (5V power adapter i.e.) for stable operation, and basic peripheral devices (keyboard, mouse, HDMI cable) used only during the initial setup phase. However if you do not have basic peripheral devices, you can setup with Raspberry Pi Imager.

This hardware configuration provided a balanced combination of processing power and connectivity, enabling the project to run image capture, basic image comparison, and cloud integration tasks effectively without requiring external high-performance systems.
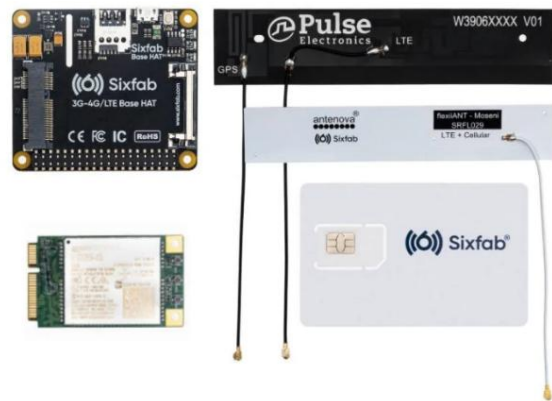

Image 5: Raspberry Pi 4

Image 6: Sixfab Cellular Modem Kit

Image 7: Raspberry Pi Camera Module

Image 8: microSD card

Future Work

To further improve the system and expand its real-world applicability, several enhancements can be considered in future iterations. First, the current human detection module can be replaced or upgraded with a more robust, AI-based model, such as YOLOv5 or MediaPipe Pose, to increase accuracy and reduce false positives. Additionally, a live video stream feature could be added to monitor real-time activity at bus stops, alongside still image analysis.

Another potential improvement is the implementation of automatic crowd density estimation, which would allow the system to provide not just presence detection but also population metrics. This could be combined with predictive analytics, enabling route optimization and better scheduling for public transportation services .

Moreover, implementing offline buffer storage and retry mechanisms for unstable network conditions would make the system more reliable in remote or high-traffic locations. Lastly, a user-facing mobile or web dashboard could be developed to allow citizens to access live stop occupancy data and make informed travel decisions in real time.

Terminology and References

Istanbul Metropolitan Municipality. *Istanbul Ulaşım ve Hareketlilik Raporu – Haziran 2022*. İstanbul Ulaşım Yönetim Merkezi (UYM), 2022. Accessed August 2025. Available at: https://cdn-uym.ibb.gov.tr/cdn/uym/NewReports/Istanbul_Ulasim_ve_Hareketlilik_Raporu_Haziran_2022.pdf

→ **Raspberry Pi 4:** A compact single-board computer used as the main processing unit of the system, providing camera interface, GPIO access, and internet connectivity capabilities.

→ **Sixfab Cellular Modem Kit:** A modular hardware kit enabling LTE/4G cellular connectivity, ensuring internet access in remote deployment scenarios where Wi-Fi is unavailable.

→ **ThingsBoard:** An open-source IoT platform used to collect, visualize, and manage data from IoT devices. In this project, it was utilized for real-time monitoring and event visualization.

→ **Google Drive Integration (rclone):** Rclone is an open-source command-line tool used to synchronize and upload files to cloud storage services like Google Drive, enabling remote storage of captured images.

→ **IoT (Internet of Things):** A technology concept referring to interconnected devices communicating and exchanging data, forming the basis of this project's design philosophy.

→ **Threshold:** A configurable numeric value used in change detection to determine whether detected differences between images should trigger event actions.

→ **Human Detection:** A computer vision process used to identify the presence of human figures in images. In this project, it is used after detecting visual changes to verify if the change involves a person.

→ **Payload:** The data package sent to the cloud (ThingsBoard) after a detection event. It includes information such as folder name, sequence number, and number of people detected.

→ **Rolling Buffer:** A memory management technique where only the most recent set of files (in this case, 12 images) are kept, and older ones are deleted as new ones are added.

→ **Zip Compression:** The process of packaging multiple images into a single compressed file format (.zip) before uploading them to cloud storage.