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.

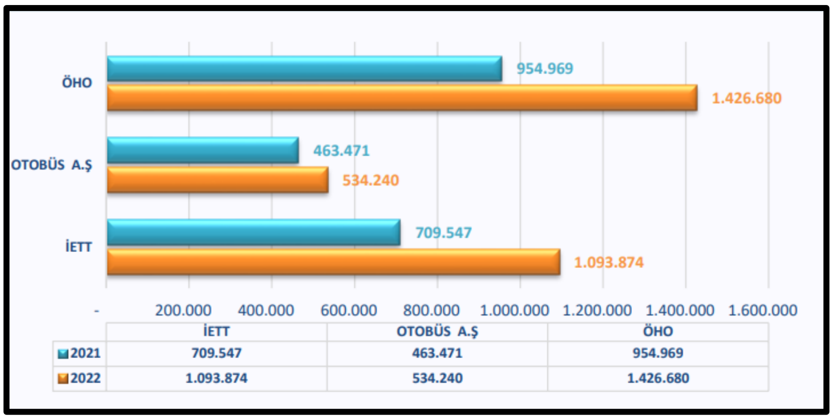
Problem Description

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



	İETT	OTOBÜS A.Ş	ÖHO
🟦 2021	709.547	463.471	954.969
🟧 2022	1.093.874	534.240	1.426.680

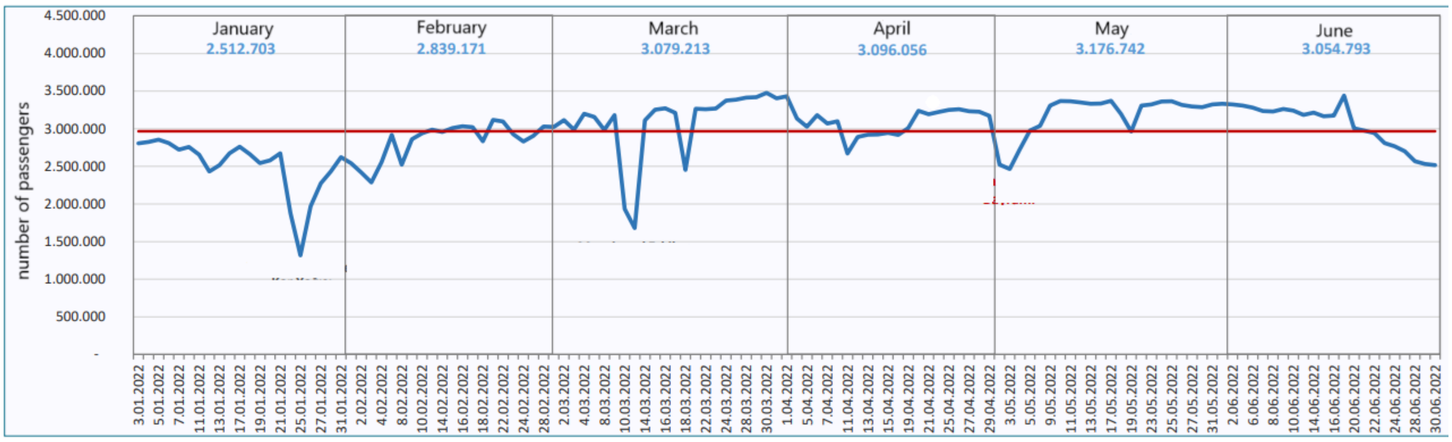
Grahp 1: Number of bus trips by month in 2022

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.



Grahp 2: Number of bus passengers by month in 2022

Solution Approaches

1. Description of General Solution Idea

To explore potential solutions for the main problem, I reviewed various ideas aimed at addressing the issue. Each approach proposed a different method and perspective, so I analyzed their common points to determine whether a more efficient and sustainable solution could be developed. As a result, I designed and implemented the **Public Transportation Efficiency with Smart Stop Density Monitoring System** project.

As the name suggests, the main goal of this project is to enable more effective and smarter management of public transportation. Its primary application area is bus-based public transport systems, which are widely used around the world. The key aim is to reduce congestion and crowding, particularly during peak hours such as the morning and evening rush periods.

Our system is built on a Raspberry Pi with a camera module that captures real-time images of bus stops and applies simple image processing techniques to detect changes. The Python-based software connects to an IoT platform, such as ThingsBoard, to send change notifications and upload captured data to the cloud. In its current form, the system can detect changes in bus stop activity and lays the groundwork for future enhancements like density analysis, prediction, and advanced optimization studies.

2. Improved Solutions

2.1. Basic Change Detection System

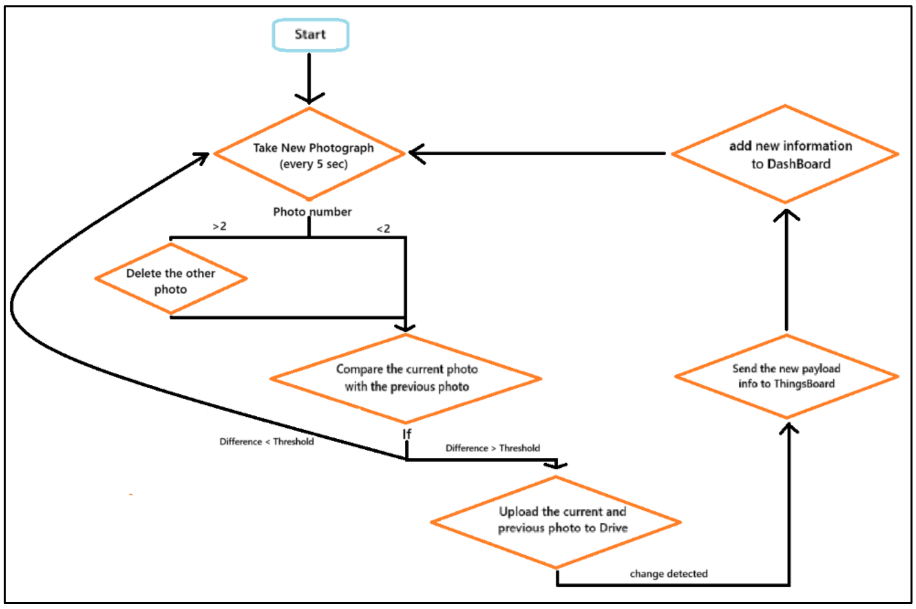

Start
Take New Photograph (every 5 sec)
add new information to DashBoard
Photo number
>2
<2
Delete the other photo
Compare the current photo with the previous photo
Send the new payload info to ThingsBoard
If
Difference < Threshold
Difference > Threshold
Upload the current and previous photo to Drive
change detected

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.

Code Snapshot & 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/Cros 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

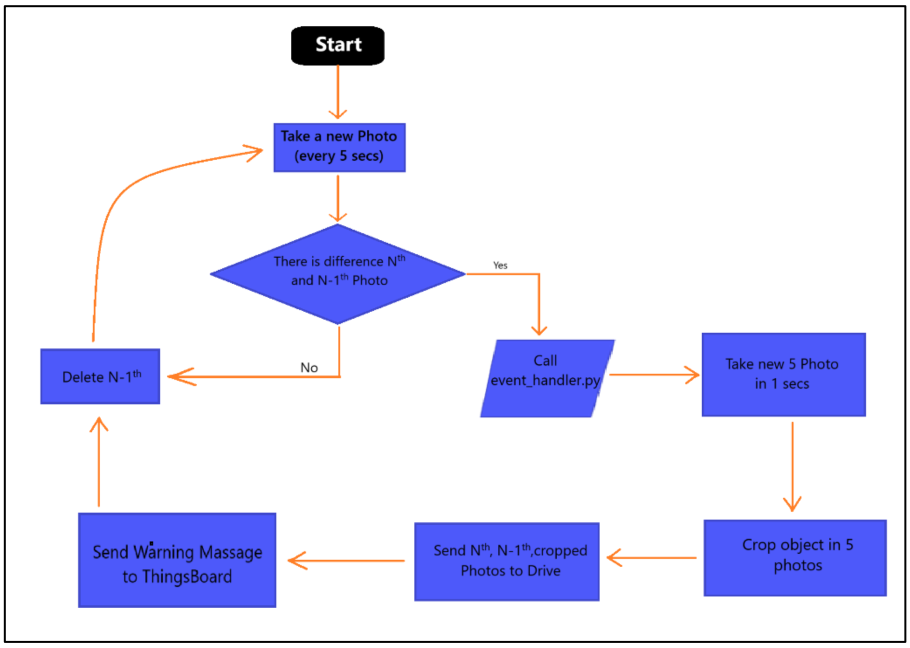

Start
Take a new Photo (every 5 secs)
There is difference Nth and N-1th Photo
Yes
No
Delete N-1th
Call event_handler.py
Take new 5 Photo in 1 secs
Crop object in 5 photos
Send Nth, N-1th,cropped Photos to Drive
Send Warning Massage to ThingsBoard

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.

Code Snapshot & 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/Cros 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.

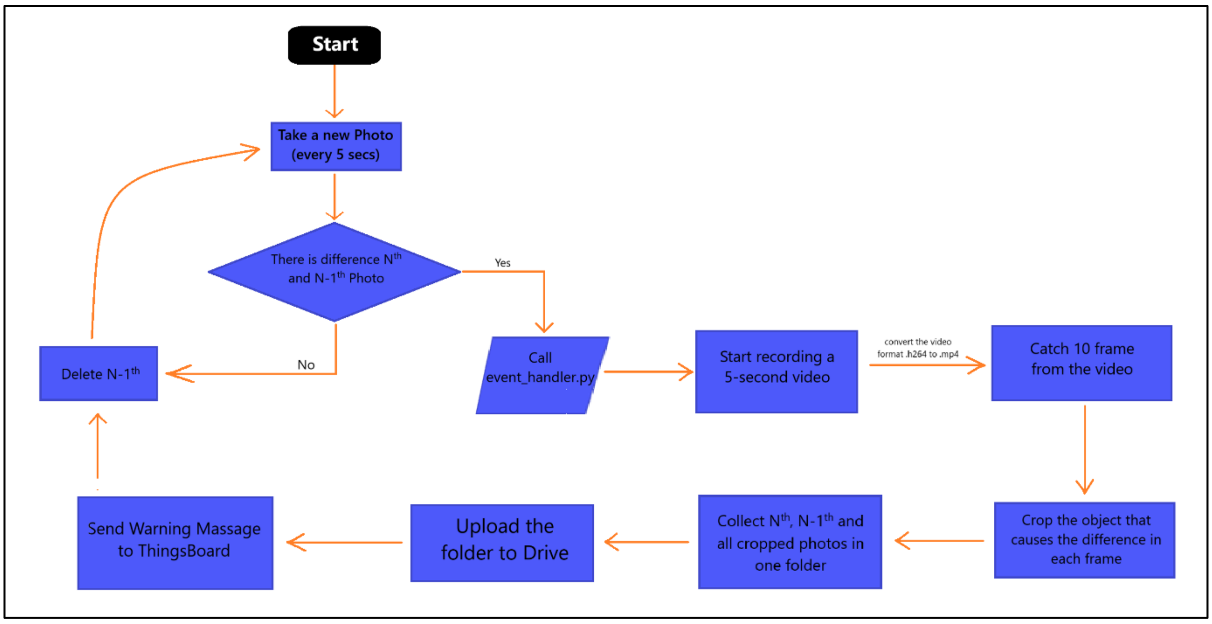
Cons

- **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



Start
Take a new Photo (every 5 secs)
There is difference Nth and N-1th Photo
Yes
No
Delete N-1th
Call event_handler.py
Start recording a 5-second video
convert the video format .h264 to .mp4
Catch 10 frame from the video
Crop the object that causes the difference in each frame
Collect Nth, N-1th and all cropped photos in one folder
Upload the folder to Drive
Send Warning Massage to ThingsBoard

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.

Code Snapshot & 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/Cros 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.

General Hardware and Cots

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 4: Raspberry Pi 4

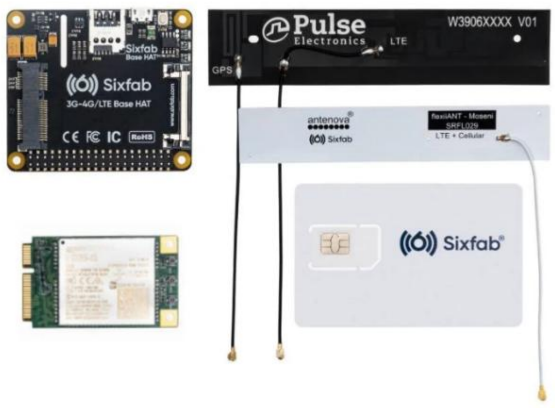


Image 5: Sixfab Cellular Modem Kit

Image 6: Raspberry Pi Camera Module

Image 7: microSD card

Future Work

Terminology and References