ViGi

camera surveillance system

Final Project Report

Table of content

Introduction

This report introduces a new software aimed to provide a self-hosted camera surveillance solution for home users. The idea of this project is based on a provided *"Camera Surveillance System"* template. The motivation behind this project is to devise a surveillance system that will be as efficient and convenient as the commercial cloud services but provide the security and privacy of the local self-hosted solutions.

A number of commercial cloud services, such as Nest, Arlo, and others, offer convenient solutions for home surveillance systems. They provide a rich set of features such as motion detection to detect particular events of interest and distinguish them from the background noise or changes in lighting conditions. They can recognize the nature of the moving object whether it is a person or a pet. They provide push notifications to alert users about these events when they are away from home. These features are implemented using computer vision algorithms that are able to detect and recognize objects in the video stream. A user can configure the system to have different sensitivity levels, or to detect only particular objects, such as people, cars, or animals.

Despite their utility, these commercial surveillance solutions exhibit significant drawbacks. Firstly, the pricing model is usually based on the number of cameras and the number of days of video storage. The cost of these services can escalate quickly, especially with an increased number of cameras and extended video storage requirements. Secondly, and more crucially, the processing of a live stream and storage of recordings on private clouds, often in unknown jurisdiction, raises serious privacy concerns. Several leaks and hacks of these cloud services have been reported in the past and some of the vendors have been even accused of spying on their customers. Moreover, IoT devices are known to be vulnerable to cyber attacks due to the short support life cycle and the lack of security updates.

In light of these challenges, an alternative solution that can be installed on a local server with connected cameras and storage is proposed. There is an abundance of free open-source software that can be used to build camera surveillance systems using a local storage, such as Kerberos, Motion, Shinobi, ZoneMinder, and many others. Even entire operating systems MotionEyeOS is dedicated to this single purpose.

However, most of these solutions do not provide the same set of features as the commercial counterparts, such as motion detection, object recognition, and event-based notifications. Some of them require fine-tuning of the parameters, such as the sensitivity of the motion detection algorithm, which can be challenging for non-technical users and not very reliable. Most of them do not provide state-of-the-art AI-based object detection and recognition, or require the additional installation of third-party AI models. It often results in a suboptimal user experience, with a lot of false positives or missed events and continuous tweaking of the parameters.

Furthermore, the installation and configuration of these systems are often complex and require technical expertise. For instance, in order to install the above-mentioned ZoneMinder application, a user is required to install dependencies, which include an Apache web server, a MySQL database, and PHP. The same applies to other solutions, such as Shinobi, which requires the installation of FFmpeg, Node.js, and MongoDB. The installation of these dependencies can be challenging for non-technical users, especially on the Windows platform.

This project aims to bridge the gap between the user-friendly nature of commercial cloud services and the privacy and cost-effectiveness of open source self-hosted solutions. This project aims to have the following characteristics:

- The system should be able to run on resource-constrained single-board ARM computers, such as Raspberry Pi, or on a laptop or a desktop with reasonable hardware requirements. The system should support a wide range of webcams available on the market.
- The system should support a multi-camera setup with two or more cameras connected to the system simultaneously. The system should be able to monitor the video stream from each camera.
- It should be easy to install and use by a user with basic technical skills. The installation process should be as simple as possible, with minimal dependencies and configuration required. Ideally, it should require only software packages available in the standard OS installation, like POSIX-compliant systems.

- It should be able to operate on a local network without the need for an internet connection, and without the need for any cloud services or subscription fees. A connected HDD or SD card should be used to store the video recordings.
- The system should provide a rich set of features, such as motion detection, object recognition, and notifications. The system should be able to distinguish between different types of moving objects, such as people, cars, and pets, and provide the ability to configure the sensitivity of the motion detection algorithm.
- Finally, the system should be able to provide a secure user-friendly web interface to view the live stream and the recorded videos. The web interface should be accessible from most modern web browsers, be responsive to different screen sizes (smartphones, tablets), and be secured with a password to prevent unauthorized access to the system

Such a system will provide a convenient and cost-effective solution for home users who are concerned about their privacy and security, and who are looking for an alternative to commercial cloud services. The system will also be useful for small businesses and organizations that require a surveillance system with similar characteristics.

Literatures Review

During the early stages of research, I identified four areas of this project that need a detailed exploration of existing academic works and software solutions. The review of the literature will help to identify the most suitable approaches and technologies, and to understand the limitations of the existing solutions that can be addressed in this project.

Firstly, the most challenging part of camera surveillance systems is the motion detection algorithm. Motion detection is a challenging problem in computer vision, because it requires the system to distinguish between the events of interest and the background noise, changing lighting conditions, and other factors. There are a number of approaches that can be used to solve this problem. All these algorithms have their pros and cons, and the selection of the most suitable one requires a review of the literature on the topic.

Secondly, this project requires an object recognition module to distinguish between different types of moving objects, such as people, cars, and animals. Object recognition is another challenging problem in computer vision. In recent years, deep learning algorithms have been shown to be very effective in solving this problem. There are a number of open-source deep learning solutions that can be used to tackle this problem. The review of the literature will help to identify the current state-of-the-art approaches and technologies in this area and to select the most suitable one.

Thirdly, the project will require the development of a streaming server to provide live video feeds and recordings to the user. As for motion detection, many approaches have been proposed to solve this problem. There is a standard protocol for streaming video over the network, called RTSP (Real-Time Streaming Protocol). This protocol is supported by most of the modern IP cameras and is widely used in the industry. Another widely adopted protocol is ONVIF (Open Network Video Interface Forum) designed specifically for surveillance cameras. Besides them, there are a number of other protocols, such as RTMP (Real-Time Messaging Protocol), HLS (HTTP Live Streaming), and others. All these approaches have their pros and cons, and the choice of the most suitable one requires a review of the literature.

Finally, the review of the existing open-source and commercial surveillance systems will be conducted. This will help to identify the most demanded features and characteristics of the surveillance systems, and to understand the limitations of the existing solutions that can be addressed in this project. As there are open-source projects available, these source codes will be analyzed to identify used technologies and approaches to the above-mentioned problems. The existing solutions will be evaluated and some of them may be used as an inspiration for this project.

Comparative Analysis of Moving Object Detection Algorithms [1]

This article presents a comprehensive study on motion detection algorithms most commonly used in video surveillance systems with stationary cameras. The primary goal of these algorithms is to distinguish moving pixels (foreground) from the environment (background), a crucial step for detecting events of interest. The challenge is to make these algorithms sensitive enough to notice moving objects while also being

strong against background noise like changes in light, shadows, camera vibrations, and sensor noise.

The paper discusses traditional computer vision algorithms that rely on pixel-wise comparison and do not use deep learning: frame/temporal differencing (FD), simple adaptive background subtraction (BS), Mixture of Gaussian Model (MoG), and approximate median filter. These methods have been around for decades but are still widely used in video surveillance for being simple, efficient, and reliable. Some of the post-processing techniques are also discussed, such as thresholding and morphological operations. These techniques are used to improve the quality of the output and to reduce the number of false positives.

The authors provide a detailed description of each algorithm, including the mathematical foundations and block diagrams. They also created a basic GUI application to visually compare the output of these algorithms on various video clips. This comparison helps to assess how well each algorithm distinguishes moving objects and handles noise. The results of this comparison are presented in Figure 1.

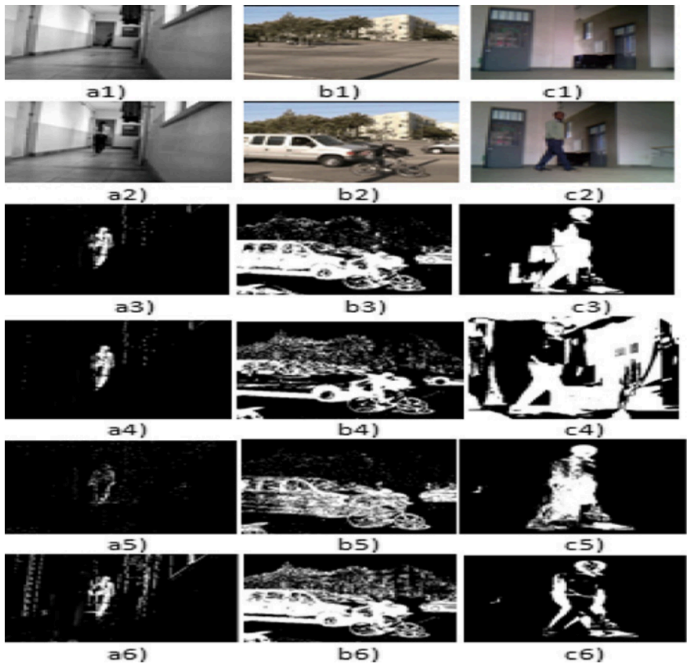

Figure 1: Comparison of different motion detection algorithms output [1]. a3, b3, c3 is a approximate median filter; a4, b4, c4 is a background subtraction; a5, b5, c5 is a mixture of Gaussian (MoG); a6, b6, c6 is a frame differencing.

Unfortunately, the authors do not provide any quantitative evaluation of the algorithms, which would be useful to objectively compare their performance. Also, the code for the GUI application nor for the algorithms themselves is not provided, which makes it difficult to reproduce the results. Some popular background subtraction algorithms, such as KNN (K-nearest neighbors), CouNT, and ViBe, are not discussed in this paper, as well as optical flow-based approaches. Anyways, this paper provides a good overview of the motion detection algorithm landscape, their mathematical foundations, and can be used as a starting point for further research.

BSUV-Net 2.0: Spatio-Temporal Data Augmentations for Video-Agnostic Supervised Background Subtraction [2]

In recent years, deep learning-based methods have emerged as alternatives to traditional background subtraction methods in computer vision, particularly for motion detection. A prominent approach is Supervised Background Subtraction, where a deep learning model learns to separate background from foreground in video clips. The paper proposed BSUV-Net 2.0, a recent advancement in this field, as for 2023 considered a state-of-the-art model in this field.

BSUV-Net 2.0 builds on the original BSUV-Net architecture from 2019, which was based on the well-known U-Net architecture widely used in semantic segmentation. The new version uses spatio-temporal data augmentations to enhance its performance. The spatial augmentations help the model adapt to different camera angles and lighting conditions, while the temporal augmentations improve its handling of varying motion patterns, noise, and vibrations. These augmentations aim to improve the model's generalization capabilities and reduce overfitting.

The authors report that BSUV-Net 2.0 surpasses both its predecessor and other state-of-the-art models in performance on the CDNet 2014 dataset. The paper provides a thorough explanation of the model's design, training process, and source codes, alongside detailed evaluation results.

While the deep learning approach to motion detection is promising and worth considering, it may not be suitable for this project for a number of reasons. Firstly, it is

more computationally intensive, which may not be suitable for inexpensive hardware like Raspberry Pi devices. Secondly, the pre-trained models are not freely available, so it would require training from scratch on a large dataset, which may not be feasible for this project. Thirdly, the model's performance is highly dependent on the training dataset, which may not be representative of the real-world scenarios. Finally, the model's performance is difficult to evaluate, as it requires a large dataset of video clips with and without motion, which is not publicly available.

MotionPlus, a software motion detector [3]

MotionPlus is a new version of Motion, a GNU-licensed software package that for a long time was considered a de-facto standard to build camera surveillance systems on Linux. It was included in most Linux distributions and was widely used in the industry. MotionPlus is a fork of the original Motion project, intended to modernize it, add new features, and improve its performance.

It is a command-line Unix daemon that can be configured via a configuration file and provides a web interface to view the live video stream and access the recordings. The original Motion project uses a simple frame differencing approach to detect motion in the video stream. MotionPlus provides a secondary method motion detection method, in addition to the original frame differencing method. The secondary methods are Haar cascades, HOG (Histogram of Oriented Gradients), and Deep Neural Networks that are used for object recognition. This shift to modern computer vision algorithms makes MotionPlus more accurate and reliable in detecting motion than the original Motion.

The MotionPlus project provides an important insight into the features that are expected from a modern camera surveillance system. It also provides a time-proved implementation of motion detection algorithms that can be used as a reference for this project. However, the original Motion approach to motion detection is quite simplistic and relies on the user to fine-tune the parameters to achieve the desired results. For example, the configuration file has a parameter called "threshold" that is a minimum number of changed pixels to consider as motion. This parameter is difficult to guess and it is quite camera and scene dependent.

As a conclusion of a review of the literature on motion detection approaches, it can be said that while a number of approaches have existed for decades, there is no one approach that is considered to be the perfect solution. Each approach has its significant limitations and requires fine-tuning of the parameters to achieve the desired results in a particular scenario. This is an ongoing research problem in computer vision and is another manifestation of Moravec's paradox, which states that a seemingly trivial task for humans, such as recognizing a moving object, is a challenging problem for computers.

DETRs Beat YOLOs on Real-time Object Detection [4]

In surveillance systems, recognizing objects that cause motion is crucial. In indoor settings, false alarms often occur due to pets, insects, or other moving objects, but the key interest is in identifying people as potential intruders. Real-time object detection models can help in accurately labeling moving objects.

Deep learning models are currently leading in object recognition, surpassing traditional computer vision approaches in accuracy and even inference speed. In the recent decade, a lot of deep learning models have been proposed for object detection and recognition. These models compete in accuracy and speed on public image datasets, like COCO and ImageNet. Until recently, YOLO (You only look once) object detection model was regarded as the state-of-the-art model for real-time object detection due to its speed and accuracy.

However, in recent years, new models were proposed. One notable is DETR (DEtection TRansformer) based on the Transformer architecture, initially used in Natural Language Processing (NLP) but gained popularity in computer vision as well. This particular paper proposes an optimized version of DETR, called RT-DETR (Real-Time DETR), which reportedly outperform recent YOLO versions in both speed and accuracy. According to the provided in the paper comparison, RT-DETR-L, the smallest version of RT-DETR with 32 million parameters, performs 1.6 times faster than YOLOv8-L with 43 million parameters, while maintaining similar or slightly greater accuracy on the COCO dataset (Figure 2).

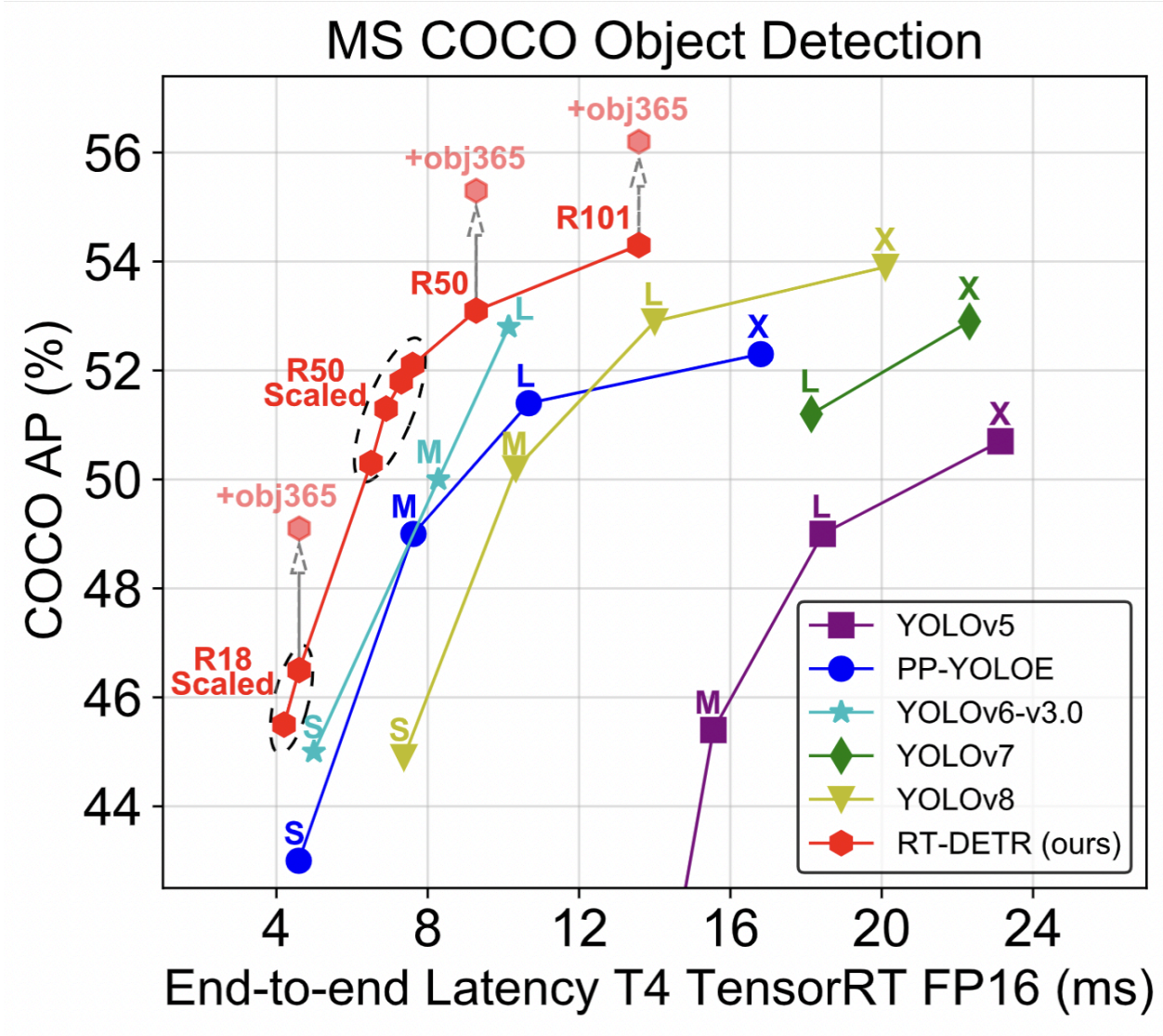

Figure 2: Performance comparison between RT-DETR and YOLO [4]

The paper details the model's architecture, training process, provides source codes, and evaluation results on the COCO dataset. It all suggests that RT-DETR can be used for object recognition in surveillance systems, instead of the popular YOLO model. However, it is noted that this model faces challenges, especially in training and optimizing for small objects. These issues are expected to be resolved in future developments. Given our project's constraints for deployment on local servers or Raspberry Pi devices, a model's computational complexity is crucial. Thus, different models should be evaluated for their performance on small devices and the final choice will be made based on the evaluation results.

Streaming Protocols: 6 Preferred Formats for Professional Broadcasting [5]

The article provides a comprehensive overview of the most common and widely adopted video streaming protocols, including RTSP, RTMP, HLS, WebRTC, and others. It explains the differences and relationships between streaming protocols, video codecs (vodecs), and container formats. It gives a detailed description of each protocol, including the history, the technical details, and the use cases. It also discusses the pros and cons of each protocol and provides recommendations on which one to use in different scenarios.

Based on this overview, WebRTC seems to be the most promising protocol for this project. It is a modern protocol that is supported by most of the modern browsers and mobile devices. It also offers near real-time streaming with low latency, which is crucial for surveillance systems. However, the focus of the article is primarily on professional broadcasting, not surveillance systems. Most arguments are based on broadcasting needs.

Also, the article does not provide information on software or hardware compatibility for each protocol. For instance, it's unclear which protocols are supported by Apple devices. Additionally, the article overlooks the security features of these protocols, a critical factor for surveillance systems. While the article is useful as a high-level overview of the video streaming protocols landscape, it does not offer enough detail for deciding on a protocol for this project. A more thorough study of each protocol's specifications is necessary.

ZoneMinder, an open-source CCTV software [6]

Another open-source surveillance software that is worth mentioning is ZoneMinder. It is one of the most feature-rich solutions available on the market, used primarily in professional settings, but can also run on a home server. It provides a web interface, as well as Android and iOS apps. It can utilize virtually any number of cameras of any type, including infrared cameras. There is an impressive list of features, including

advanced AI-powered detection, cloud and local storage, and many others. It is a mature project with a large community and a number of commercial partners.

As for MotionPlus, ZoneMinder and other open-source solutions have tendencies to accumulate complexity over time. As a general-purpose, fit-for-all solution, it comes with a lot of features that may not be relevant to the home users, targeting primarily enterprise customers. Also, some of the features are available only in the paid version. The installation and configuration of ZoneMinder is not a trivial task and requires advanced technical skills.

Design

Domain and Users

The domain of this project is “Camera Surveillance System,” specifically, a software component that enables home users to install and configure a self-hosted camera surveillance system on their local network, using a local server and connected cameras. The solution is supposed to be a viable alternative to commercial cloud services, offering the same or nearly the same set of features as the commercial counterparts.

The target users of this project are home users who want to avoid using cloud services for their surveillance systems, whether due to cost or privacy concerns. The users of this project are expected to have basic technical skills and be able to install and configure the system on their own.

Features

There is an abundance of free open-source software that can be used to build camera surveillance systems, such as Kerberos, Motion, Shinobi, ZoneMinder, and many others. Many of these solutions are quite mature and offer a wide range of features that are not feasible to implement in this project. Many of them are even more advanced than the commercial solutions as the latter often focus on the specific features that are most

demanded by the users, while the open-source solutions tend to be general-purpose and often end up being overwhelmingly complex.

Instead of trying to compete with the existing feature-rich solutions, this project will focus on the most common features that are expected from a modern camera surveillance system and will try to implement them in the most user-friendly way inspired by the commercial solutions. After reviewing the existing solutions, the following features are considered to be the most important and will be implemented in this project:

- Connect multiple cameras to the system and monitor the video stream from each camera simultaneously. The system should support a wide range of webcams available on the market.
- Detect motion in the video stream and record a video clip with the detected event. The motion detection algorithm should be able to distinguish between the events of interest and the background noise, changing lighting conditions, and other factors. The user should be able to configure the sensitivity of the motion detection algorithm.
- The nature of the moving object should be recognized, such as people, cars, or pets. This metadata should be stored along with the video clip, displayed in the user interface, and should be used for notifications.
- Send an alarm notification to a user about a detected motion, including the type of the moving object (a person, car, pet, etc.). The notification can be sent via different channels, such as email or SMS.
- Provide a user-friendly web interface to view the live stream and the recorded videos. Provide the ability to manage the recordings, such as deleting, downloading, and sharing them. The interface should be accessible from most modern web browsers and should be responsive to different screen sizes, such as mobile phones and tablets.
- A user should be able to secure the web interface with a password to prevent unauthorized access to the system.

Some of the existing solutions also provide AI-based object recognition of the type of the moving object (a person, car, pet, etc.) It enables the user to configure the system to detect only particular events and ignore others, such as pets. This feature aims to

reduce the number of false positive alarms when, for example, a pet is detected as a moving object. Only a few of the existing free open-source solutions provide such a feature while most of the commercial cloud services do. This feature is expected to be implemented in this project and is considered a distinguishing feature.

It is worth noting that the cloud-based solutions offer a number of features that are not feasible to implement in a self-hosted environment. The most notable is saving the recordings in the cloud, which ensures that the recordings can not be stolen or destroyed by an intruder. Another notable feature usually provided by the cloud services is the ability to view the live stream and the recordings from anywhere in the world, using a companion mobile application.
Cloud-based backup could be implemented in this project, but it would require the user to have a cloud storage account and to configure the system to upload the recordings to the cloud. This feature is a good candidate for future work, but it is not considered to be a part of the first version of the system. However, the design of the system allows for the implementation of this feature by third-party tools, such as rclone[1], which can be used to synchronize the recordings with the cloud storage. While it is not as convenient as the built-in cloud storage, it is a viable alternative for the users who require this feature.

The ability to view the live stream and the recordings from anywhere in the world is more challenging to implement, as it requires the web server to be accessible from the internet, which is not always possible due to the limitations of the home network and the internet service provider. Moreover, it effectively exposes the surveillance system to the same privacy and security risks that we aim to avoid by using the self-hosted solution. However, the design of the system allows for the implementation of this feature by third-party tools, such as VPN, which can be used to access the local network from anywhere in the world. While it is not as convenient as the built-in remote access, it is still an option for the users who require this feature.

[1] Rclone is a command-line program to manage files on cloud storage: https://rclone.org/

Design Principles

To fulfill the project requirements outlined in the previous section, the following design principles were established:

- The system should be designed as a single monolithic application, which encapsulates all the modules of the system with intent to ease the deployment and configuration process for the non-technical users, who are not expected to have advanced technical skills to deal with complex deployment scenarios, such as Docker or Kubernetes.
- Inside the monolithic application, the system should be designed to have a modular architecture, to manage the complexity of the system and to allow for easy extension and modification of the system.
- The system should be designed to have a clear separation of concerns, with each module responsible for a specific task, such as motion detection, object recognition, and video streaming.
- The system should require minimal configuration effort from the user, with sensible defaults provided for the configuration parameters.
- The system should be designed to have minimal external dependencies, requiring only software packages available in the standard OS installation. This will make the installation process as simple as possible.
- The system should follow a principle of least surprise, which means that the system should behave in a way that is consistent with the user's expectations. The system should provide meaningful error messages to the user in case of invalid configuration or other issues.

Given the above design principles, the architecture of the system was designed and iteratively updated during the development process. This document contains the final version of the architecture. The previous versions of the architecture are not included in this document, but were presented in the Preliminary and Draft reports.

Architectural Overview

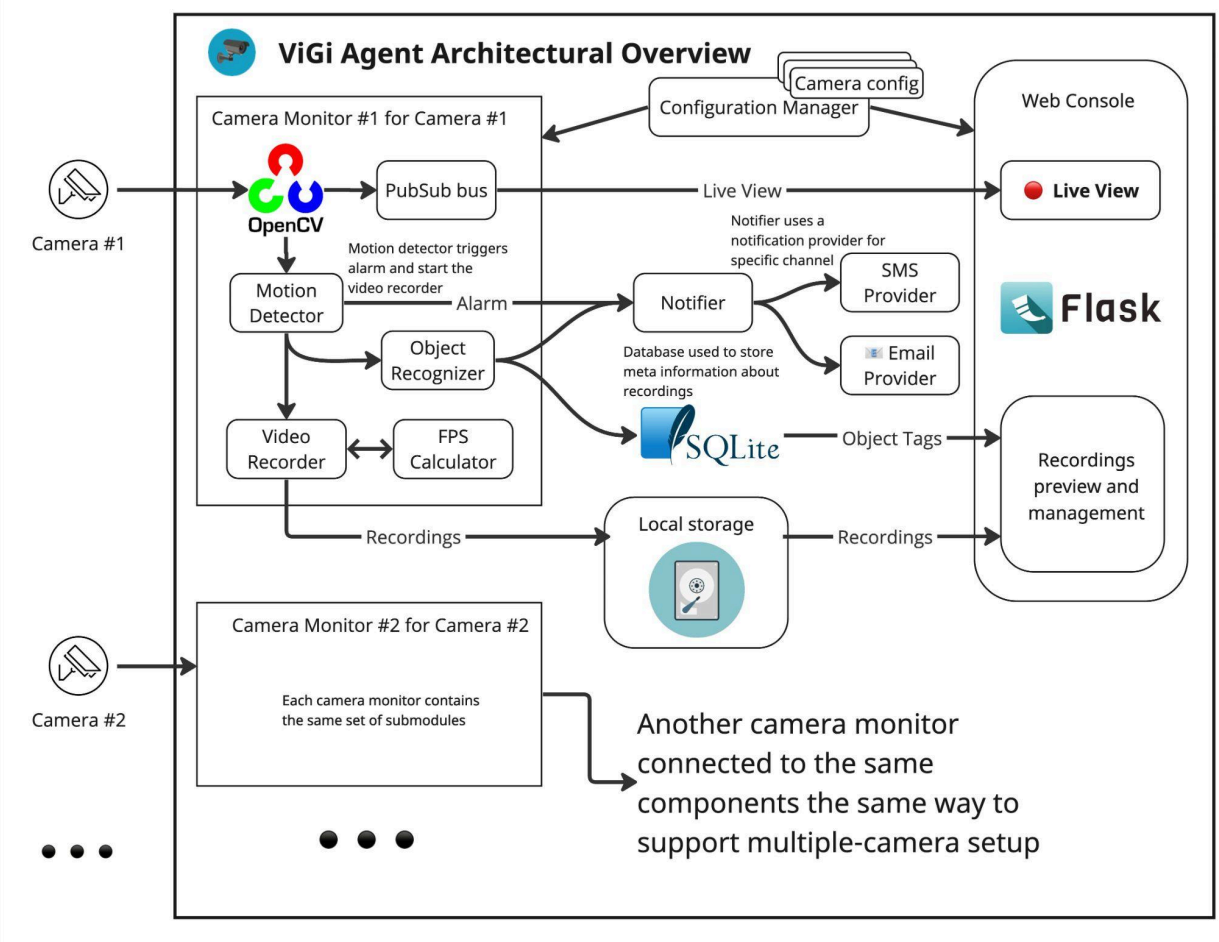

Figure 3: High-level architectural overview of the project.

The overall architecture of the project is shown in Figure 3. The arrows show the direction of the data flow between the modules, from the camera (left) to the web console (right). The system consists of the following modules:

Agent

The agent is a monolithic application that encapsulates all the modules of the system. From a user's perspective, the agent is the one software package that needs to be installed and run on a local server with connected cameras. When the agent is started, it initializes all the internal modules and provides access to the web console through the integrated HTTP server. The internal complexity of the system is hidden from the

user, but from the developer's perspective, the system is modular and each module can be developed and tested independently.

Camera Monitor

Camera Monitor is a module responsible for monitoring the video stream from one connected camera. Each connected camera is monitored by a separate instance of the Camera Monitor module that runs in a separate thread.. The Camera Monitor module consists of the following submodules:

- PubSub message broker is responsible for receiving the video frames from the camera and forwarding them to the other modules of the system that are interested in the video stream. Mainly, the web console listens to the video frames to display the live stream to the user. The PubSub pattern provides a decoupled way of communication between the modules, which allows for easy extension, and thread-safety.
- Motion Detector is responsible for detecting motion in the video stream and notifying other modules about the detected events. Details of the motion detection algorithm are described in the Implementation section.
- Object Recognizer is responsible for recognizing the nature of the moving object, such as a person, car, or a pet. These labels (tags) are stored along with the recording metadata and are used to specify the type of the moving object in the user interface and in the notifications.
- Video Recorder is responsible for recording the video clips (recordings) with the detected events. The video clips are stored on the Local storage and are accessible from the web console. The video recorder uses the FPS Calculator module to determine the frame rate of the video stream and to record the video clips with the correct frame rate. The details of the video recording process are described in the Implementation section.

Configuration Manager

Configuration Manager is a module responsible for reading, validating, and storing the configuration of the system. All modules of the system read the configuration from the Configuration Manager. The system has three sources of configuration in the following order of priority: the command-line arguments, the configuration file, and the default

configuration. The system has sensible defaults for the configuration parameters, which suit most of the users.

Notifier

Notifier is a high-level module responsible for sending notifications to the user about the detected events. The notifications could be sent using different channels, such as email, SMS, etc. Each channel is implemented as a separate submodule called Notification Provider. The user can configure each notification provider separately. This concept allows for easy extension of the system with new notification providers, such as a Web Push notification. Also, the Notifier could be configured to have any number of notification providers of the same type, such as SMS providers, to send the notifications to different phone numbers.

Storage

SQLite was chosen as a database solution for the metadata because it does not require DBMS software to be installed, which is in line with the design principle of minimal external dependencies. SQLite is included in the standard Python installation and is widely used in the industry. The video clips are stored in a file system, which is also a lightweight solution that does not require any configuration except for the path to the storage directory.

Web console

The web console is a user-friendly interface for the system, where the following features are implemented:

- The "Basic" HTTP authentication scheme to prevent unauthorized access to the console.
- Live stream of the video feed from the connected cameras (Figure 4 and 5).
- List of the recordings by date and time, with the detected events, including the metadata (Figure 6).
- Play a specific recording (Figure 7).
- Management of the recordings, such as deleting, downloading, and sharing them.

Wireframes

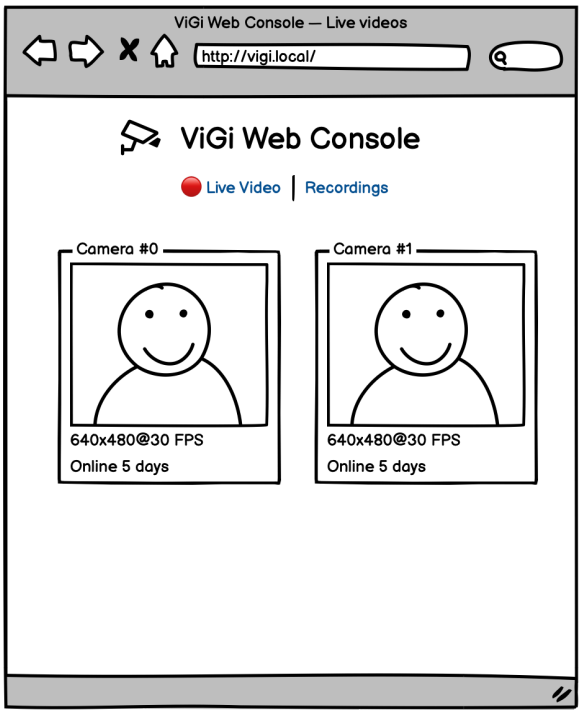

Figure 4: Live video preview wireframe.

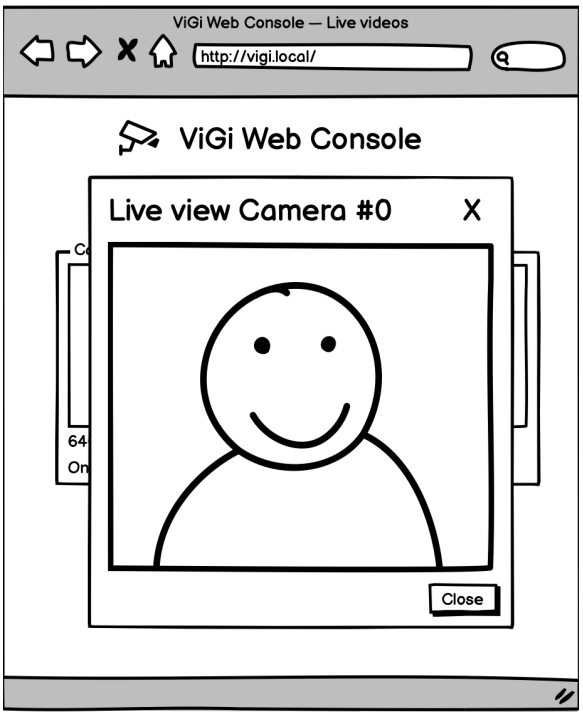

Figure 5: Live video popup.

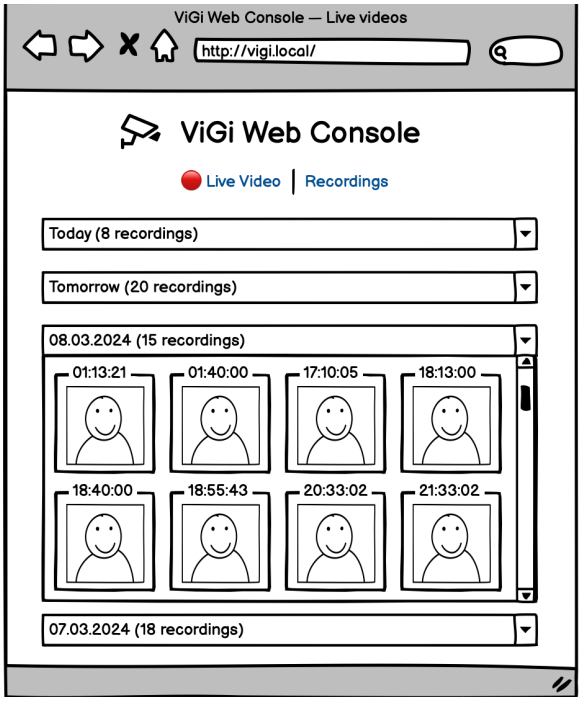

Figure 6: List or recording previews

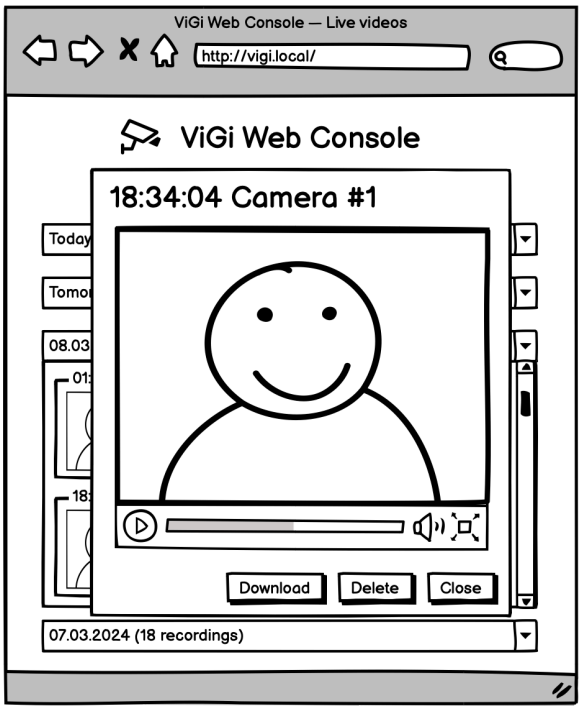

Figure 7: Play a recording

Implementation

Development environment

The source code of the project is managed using the Git version control system and is hosted on GitHub in a public repository: https://github.com/plashchynski/vigi/. The details on how to install and run the system are provided in the README.md file in the root directory of the project.

The development environment for the project was set up on an Apple ARM-based M1 laptop. The development environment consists of Python 3.12, Visual Studio Code text editor, and a number of third-party libraries, such as OpenCV, NumPy, and others that were installed using the pip package manager into a virtual environment (venv). The list of the third-party libraries and their versions is provided in the requirements.txt file.

Implementation Principles

To ensure the quality of the system the following principles were employed during the implementation of the system:

- The main programming language of the system should be Python, which is widely used in the industry and has a rich ecosystem of libraries and tools. The general guidelines for the Python code should be followed, such as PEP 8[2].
- App should be shipped as a pip package and published in PyPI[3], which will allow for easy installation and distribution of the system.
- Modules should be decoupled from each other, with the dependencies injected into the modules as constructor arguments. Decoupling the modules will allow for testing and extension of the system, as the dependencies can be easily mocked or replaced with mocks or alternative implementations.
- Defensive programming should be used to handle the invalid input, edge cases, I/O errors, and other issues.
- Generous use of Python type hints, docstrings, and comments.
- Logging should be used to diagnose possible issues.

[2] https://peps.python.org/pep-0008/

[3] The Python Package Index (PyPI) https://pypi.org/

Prototype

I have started the implementation of the project by creating a prototype of the system. The prototype was focused on the most challenging and crucial features of the system: motion detection, object recognition, and web streaming. The prototype was a part of the research process aimed to evaluate approaches to these problems, as seeing things in action provides insights that are not always possible to obtain from the literature.

The prototype was implemented as a Flask HTTP server providing a web page that streamed a live video feed from a camera (See Figure 8). The server used the OpenCV library to connect to the camera and fetch the video stream as a sequence of frames. These frames then were processed by the motion detection and object recognition algorithms and then sent to the client as a stream of JPEG images.

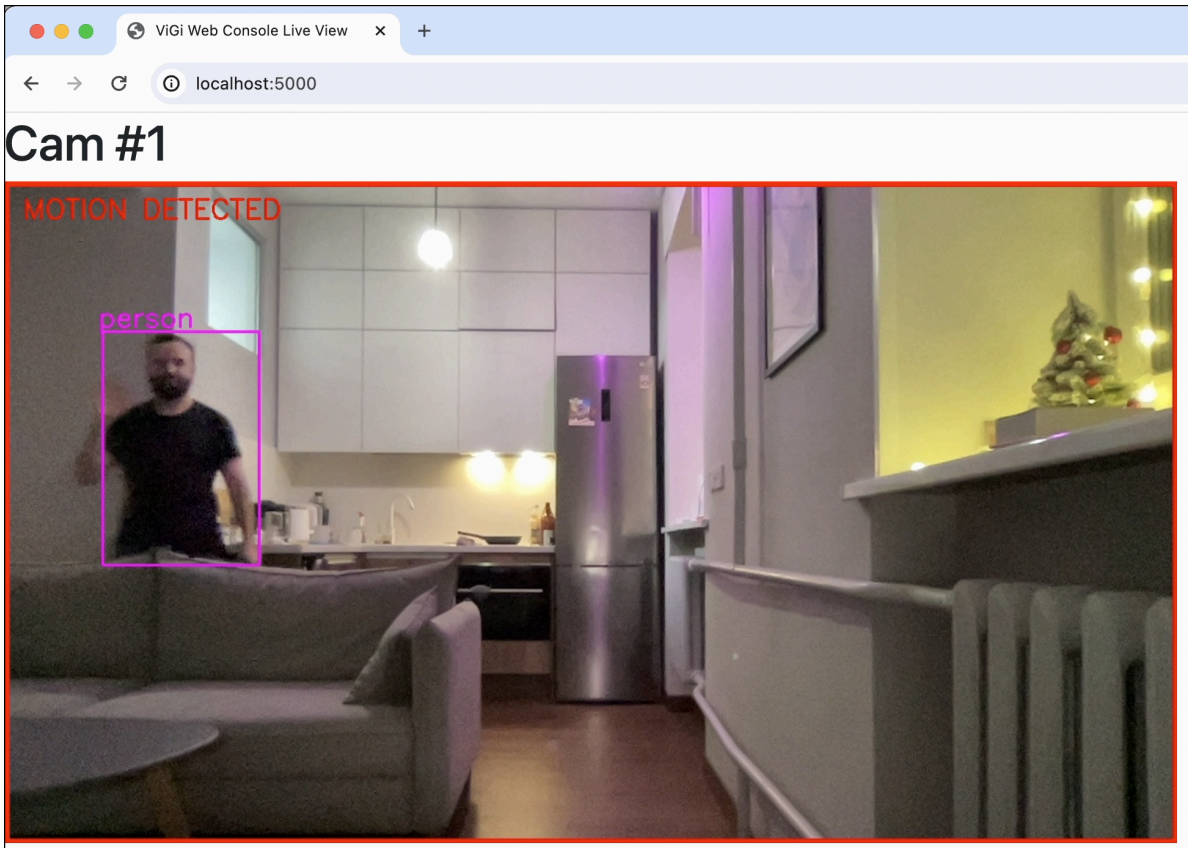

Figure 8: The screenshot of the prototype.

As the prototype was aimed to evaluate the approaches, the evaluation strategy, criteria, and specific results are presented in the Evaluation section of this report. This

section provides only a high-level overview of the findings that motivated the further development of the project:

1. As was found in the literature, there are a number of sophisticated approaches to stream video over the network, such as RTSP, RTMP, and HLS. All these approaches require a separate streaming server to convert the video stream from the camera to the desired format. Although these approaches provide a number of advantages, such as low latency and high quality, they have turned out to be computationally intensive and complex to implement.

   The prototype used a simpler approach, where the video stream was sent to the client as a sequence of JPEG images. Despite its simplicity, this approach was found to be surprisingly efficient. From the user's perspective, it is not distinguishable from real video streams, and it is much easier to implement and much less computationally intensive.

2. Related to the motion detection and object recognition algorithms, there is an expected trade-off between the accuracy and the computational complexity. The frame differencing algorithm was found to be the most computationally efficient, but it was not very accurate. Especially when the lighting conditions were changing, the algorithm was producing a lot of false positives.

   On the other end of the spectrum, the deep learning-based models, such as BSUV-Net 2.0, were found to be the most accurate, but they were also the most computationally intensive to the point that they were not able to run in real-time even on a laptop.

   The golden mean was found in the background subtraction algorithm MOG2 (Mixture of Gaussians). The algorithm was found to be accurate enough to distinguish between moving objects and the lighting changes, and it was also computationally efficient enough to run in real-time on a Raspberry Pi.

3. The object recognition algorithms have the same trade-off between the accuracy and the computational complexity with deep learning-based models

being the most accurate but also the most computationally intensive. However, in contrast to the motion detection, the object recognition deep learning-based models were found to be significantly more accurate than the traditional computer vision algorithms, such as Haar cascades and HOG. Also, the deep learning-based models were found to be able to run on a Raspberry Pi in real-time, which was not the case for the deep learning motion detection algorithms. So, the deep learning-based models were found to be the most suitable for the object recognition task in this project.

The above findings were used to inform the design of the system and to guide the development of the system. The prototype was used as a reference implementation for the motion detection and object recognition algorithms. The details of the implementation of these algorithms are described in the following sections.

Agent

The architecture of the agent is shown in Figure 3. This section provides a detailed description of the implementation of the agent and its internal modules. The entry point of the agent is the main.py file in the root directory of the project. The main.py file:

1. Initializes Configuration Manager, which manages the configuration of the system.
2. Initializes a logger to provide a detailed log of the system's operation. The logger is implemented using the standard Python logging library.
3. Initializes a Camera Monitor module for each connected camera.
4. Initializes a Camera Monitor module and its dependencies, such as a database connector, PubSub message broker, Motion Detector, Object Recognizer, and Video Recorder.
5. Starts the Camera Monitor in a separate thread. Each connected camera is monitored by a separate instance of the Camera Monitor module that runs in a separate thread.
6. Starts the integrated Flask HTTP server to provide access to the web console.

The main.py also manages the graceful shutdown of the system. The system can be stopped by pressing Ctrl+C in the terminal that triggers cleanup of the resources and the shutdown of the HTTP server.

Camera monitor

As was mentioned in the Design section, the `CameraMonitor` module is responsible for monitoring the video stream from one connected camera. This is the core module of the system, which encapsulates all the internal complexity of processing the video stream. Each connected camera is monitored by a separate instance of the Camera Monitor module that runs in a separate thread.

The `CameraMonitor` module is implemented in the `vigi_agent/camera_monitor.py` file. To decouple the Camera Monitor module from the dependencies, all the dependencies are injected into the Camera Monitor module as constructor arguments. It allows for easy testing and extension of the module, as the dependencies can be easily mocked or replaced with the alternative implementations.

The purpose of the Camera Monitor module is:

1. Run a thread that fetches frames from the camera using OpenCV's `VideoCapture()` instance.
2. Provides PubSub message broker to forward the video frames to the other modules of the system that are interested in the video stream. Flask web server subscribes to PubSub every time a user opens the web console to display the live stream from the camera.
3. Pass frames to the MotionDetector instance. If motion is detected, the `MotionDetector` module notifies the `CameraMonitor` instance via callback.
4. Maintain a temporal state of the system, "is the video currently being recorded?"
5. Call `Notifier` instance when the motion is detected to alert the users.
6. Start and stop video recording when the motion is detected using the `VideoRecorder` instance.

Motion Detector

MotionDetector is a module responsible for detecting motion in the video stream. It is implemented in the `vigi_agent/motion_detector.py` file. The motion detector calls `motion_callback` when the motion is detected. The `motion_callback` is implemented in the `CameraMonitor` module and is injected into the `MotionDetector` as a constructor argument. This way we abstract motion detection from the action that should be taken when the motion is detected.

Each frame from the camera is processed in the following order:

1. The original frame is copied to avoid modifying the original frame.
2. OpenCV's `cvtColor()` method is applied to convert a frame to grayscale to reduce the computational complexity, as the color information is not useful for future steps.
3. OpenCV's `createBackgroundSubtractorMOG2()` method is applied, which is an implementation of the MOG2 (Mixture of Gaussian) background subtraction algorithm. The MOG2 algorithm is used to distinguish between the foreground moving objects and static background. The result is an intensity mask.
4. OpenCV's `threshold()` filter is applied to convert MOG2's results to binary values, where the intensities above a certain threshold are considered as motion.
5. OpenCV's `cv2.medianBlur()` filter is applied to remove high-frequency noise.
6. The morphological operations `MORPH_OPEN` and `MORPH_CLOSE` are applied using OpenCV's `morphologyEx()` method to fill the holes in the thresholded mask and to remove the low-frequency noise.
7. The contours are detected using OpenCV's `findContours()` method. If any of the contours is larger than a certain threshold, it is considered as a detection.
8. The bounding boxes around the contours are outputted from the algorithm.

The source code of the motion detection algorithm with marked above-mentioned steps is shown in Figure 8. Intermediate results of each step are visualized in Figure 9. The visualizations are generated using a `scripts/motion_detection_demo.py` script.


```python
class MotionDetector():
    def update(self, frame):
        original_frame = frame.copy()

        # convert the current frame to grayscale
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # update the background model to get the foreground mask
        fg_mask = self.back_sub.apply(gray_frame)

        # Warming up the model by skipping the first few frames
        if self.skip_frames_count > 0:
            self.skip_frames_count -= 1

            draw_title(original_frame, 'WARMING UP')
            return (original_frame, set())

        # threshold the mask, the min_thresh value is set to 100 by default
        # this value roughly impacts the sensitivity of the motion detection
        min_thresh = 50 / self.sensitivity # the higher the sensitivity, the lower the threshold
        _, motion_mask = cv2.threshold(fg_mask, thresh = min_thresh, maxval = 255, type = cv2.THRESH_BINARY)

        # median blur to remove granular noise
        motion_mask = cv2.medianBlur(motion_mask, ksize = 3)

        # morphological operations to fill in holes
        kernel = np.array((15,15), dtype=np.uint8)

        # morphologyEx with MORPH_OPEN is the same as erode followed by dilate
        motion_mask = cv2.morphologyEx(motion_mask, op = cv2.MORPH_OPEN, kernel = kernel, iterations = 1)

        # morphologyEx with MORPH_CLOSE is the same as dilate followed by erode
        motion_mask = cv2.morphologyEx(motion_mask, op = cv2.MORPH_CLOSE, kernel = kernel, iterations = 1)

        # get contours of the moving objects in the frame
        contours, _ = cv2.findContours(motion_mask, mode = cv2.RETR_EXTERNAL, method = cv2.CHAIN_APPROX_SIMPLE)
        detections = []

        cnt_area_thresh = (2500 / self.sensitivity) # the higher the sensitivity, the lower the are threshold
        for cnt in contours:
            area = cv2.contourArea(cnt)
            if area > cnt_area_thresh:
                x, y, w, h = cv2.boundingRect(cnt)
                detections.append([x, y, x + w, y + h])

        detections = np.array(detections)
```

Figure 8: Source code of the motion detection algorithm with steps marked.

Figure 9: Intermediate results of each step of the motion detection algorithm generated by `scripts/motion_detection_demo.py` script.

Video Recorder

`VideoRecorder` (`vigi_agent/video_recorder.py`) is a module responsible for recording the video clips with the detected events. The video recorder uses OpenCV's `VideoWriter()` method to write the video clips to the local storage. Then the video clips are accessible from the web console and from the file system.

Two challenges were identified during the implementation of the video recorder. First, it was found that the frame rate of the video stream from the camera is not constant and can vary over time depending on the performance of the camera and the computer. If the video clips are recorded with the incorrect frame rate, they will be played back with the incorrect speed. To address this issue, the camera manager continuously monitors the frame rate of the video stream using a `FPSCalculator` module (`vigi_agent/utils/fps_calculator.py`) and passes the current frame rate to the video recorder. The video recorder uses the current frame rate to record the video clips with the correct frame rate.

Second challenge was to select the most suitable recording format that will satisfy the following requirements:

- A good balance between the quality and the file size of the video clips.
- Can be played in a HTML5 video player in a web browser without the need to install additional software.
- Can be recorded in real-time on a resource-constrained single-board ARM computer, such as Raspberry Pi.
- Encoder should be available in the standard OpenCV installation.

Unfortunately, HTML5 standard does not enforce a specific video format support. Different browsers support different video formats in different operating systems. After research and experimentation, the H.264 codec with the .mp4 container was chosen as the most widely supported format[4] with adequate quality and file size. Details on the evaluation of different recording formats are provided in the Evaluation section of this report.

[4] A list of browsers that support the H.264 codec: https://caniuse.com/?search=H.264

Web console

The web console is implemented using an integrated Flask HTTP server. The entry point of the web console is the vigi_agent/app.py file. Two different HTTP frameworks were evaluated for this purpose: FastAPI[5] and Flask[6]. Both frameworks are pretty similar in terms of features and performance, but Flask was chosen for its larger number of third-party extensions that can speed up the development process. For instance, Bootstrap-Flask[7] is used to provide a user-friendly interface for the web console using the Bootstrap css framework. Another third-party extension, flask-caching is used to cache the responses of the web console to improve the performance of the system.

Evaluation

The evaluation was conducted at different stages of the design and implementation process, and at different levels of the system. The objective of the evaluation phase was to verify that:

- The system meets the goals and requirements set in the Introduction section of this report.
- The system design is in line with the design principles set in the Design section of this report.
- The implementation of the system is in line with the implementation principles set in the Implementation section of this report.

Evaluation by Prototype

The objective of the prototyping phase was to evaluate the most suitable approaches to tackle the most challenging and crucial features of the system: motion detection, object recognition, and web streaming. Candidates were identified during the research phase and literature review.The evaluation was focused on the following aspects:

[5] https://fastapi.tiangolo.com/
[6] https://flask.palletsprojects.com/en/3.0.x/
[7] https://bootstrap-flask.readthedocs.io/en/stable/

- The accuracy of the motion detection algorithm in distinguishing between the events of interest and the background noise, changing lighting conditions, and other factors.
- The accuracy of the object recognition algorithm in distinguishing between different types of moving objects, such as people, cars, and pets.
- The performance of the motion detection and object recognition algorithms in terms of speed and resource consumption.
- The performance of the web streaming approach in terms of latency and quality of the video stream, as well as the difficulty of implementation.

The details of the implementation of the prototype are described in the Implementation section of this report. The evaluation of candidate algorithms was performed manually. I created a scene (Figure 10) with LED garlands that was constantly changing light modes. This setup was supposed to mimic different light conditions. The prototype showed a live video feed from the camera with bounding boxes around the detected moving objects.

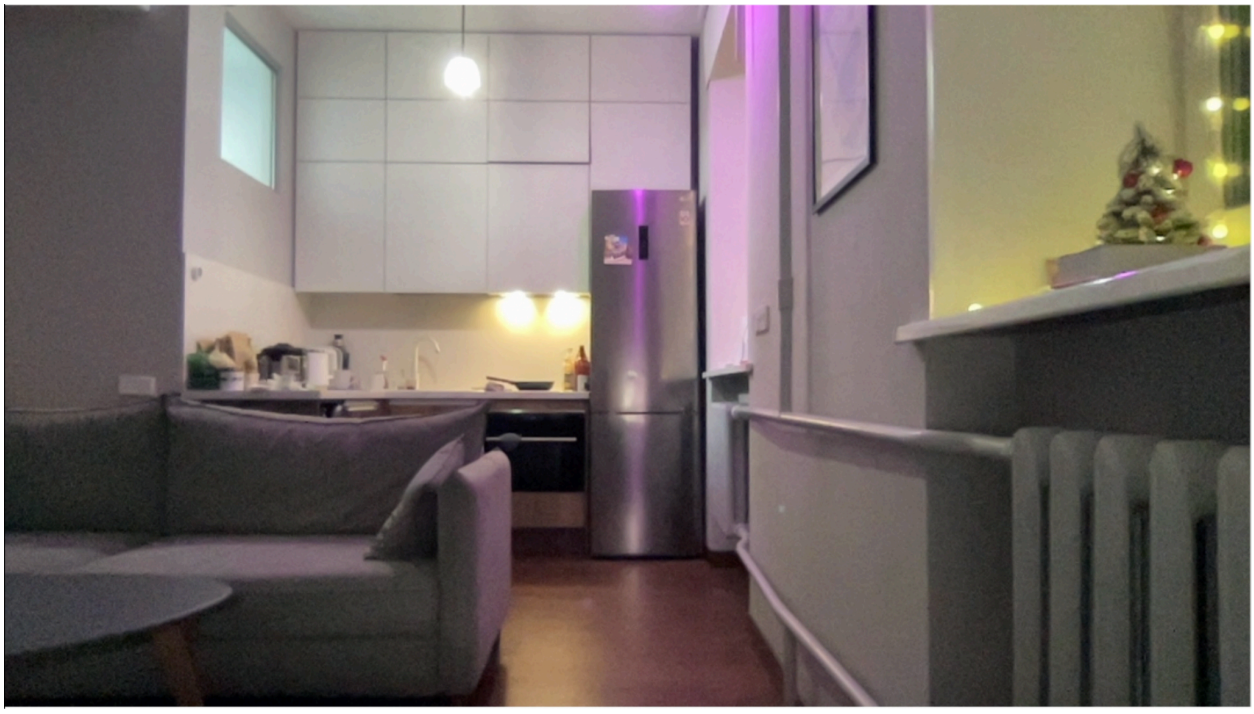

Figure 10: A scene with blinking LED garlands used to assess motion detection algorithms in the prototype.

The expected outcome was that the algorithm should distinguish the noise, such as changing light of a garland, from moving objects on the stage. The results of the evaluation of the motion detection methods are presented in Table 1.

Table 1: Motion detection methods evaluation results

Method	**Accuracy**	**Speed**	**Resource Consumption**	**Summary**
Frame Differencing	Low	High	Low	Computationally efficient, but with a lot of false positives, susceptible to changing lighting conditions.
Mixture of Gaussian[8]	High	High	Medium	It seems to be the golden mean in terms of accuracy and speed.
Farneback Optical Flow[9]	High	Low	High	The accuracy is highly dependent on post-processing and could be pretty accurate, but slow and computationally intensive.
K-nearest neighbors[10]	High	High	Medium	Pretty similar to MOG2 in terms of accuracy and speed, but slightly more susceptible to noise.
BSUV-Net 2.0[2]	High	Low	High	Highly accurate if trained on big datasets, but computationally intensive and not able to run in real-time on a Raspberry Pi

Along with the algorithms, different pre- and post-processing techniques were evaluated, such as thresholding, blurs, and morphological operations. While the deep learning-based models did not require pre- and post-processing, the traditional computer vision algorithms did show significant improvement in accuracy when the frame is preprocessed to reduce the noise.

[8] https://docs.opencv.org/4.9.0/d7/d7b/classcv_1_1BackgroundSubtractorMOG2.html
[9] https://docs.opencv.org/4.9.0/de/d9e/classcv_1_1FarnebackOpticalFlow.html
[10] https://docs.opencv.org/4.9.0/db/d88/classcv_1_1BackgroundSubtractorKNN.html

The prototype was also used to evaluate different object recognition algorithms, such as Haar cascades, HOG, and deep learning-based models (YOLOv8, RT-DETR, etc.). The main focus was on the performance of the algorithms, because the detection of the moving object is a computationally intensive task that needs to be performed in real-time.

While the traditional object recognition algorithms offer higher performance, the deep learning-based models were found to be significantly more accurate. I have tested a pre-trained YOLOv8.1[11] model in a nano variant (3.2M params) on a Raspberry Pi 3 B+ board (Figure 11) and found that they can run in real-time with a frame rate of 1-5 FPS, which is sufficient for the purpose of this project. RT-DETR[4] is a promising alternative to YOLO, but it is not available in a tiny variant so far.

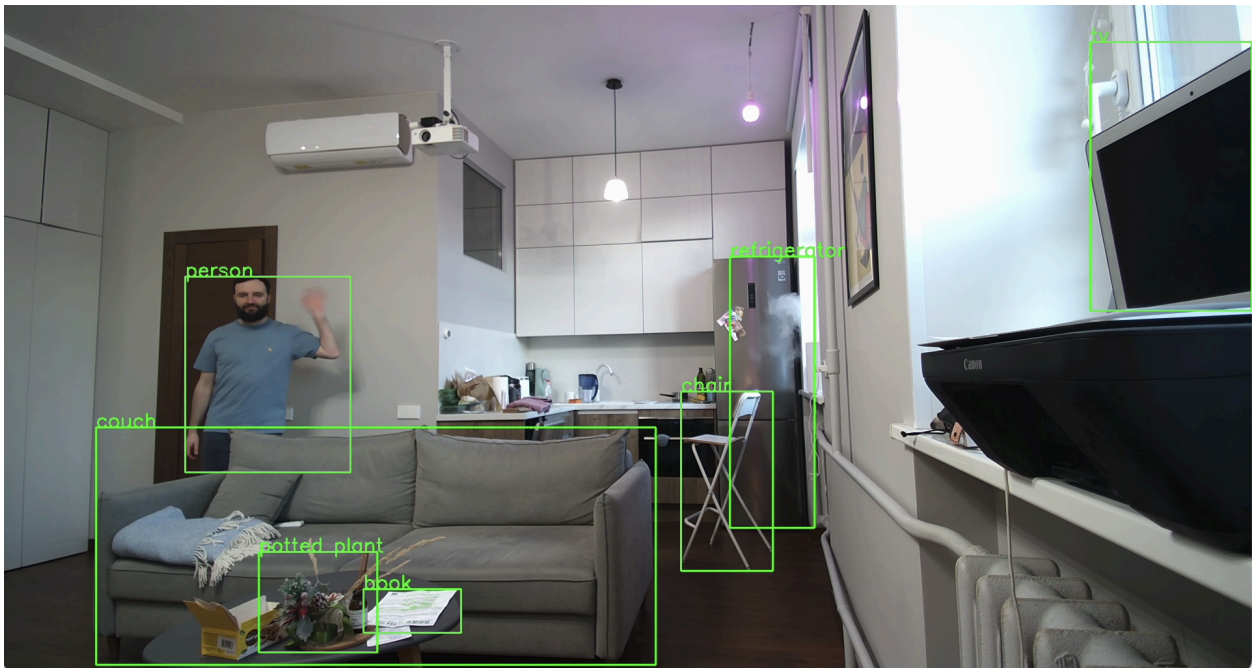

Figure 11: An example of object recognition performed by the YOLOv8.1 model in a nano variant (YOLOv8n) on Raspberry Pi 3 B+ (Source code: `scripts/yolo_demo.py`)

[11] https://docs.ultralytics.com/models/yolov8/

Unit tests and code quality

The methodology of TDD (Test Driven Development) in its relaxed form was used for the development of this project. In its pure form, TDD requires writing tests before writing the code and then writing the code to pass the tests. The downside of this approach is that it requires switching between writing tests and writing code, and overhead of writing tests for every single unit of code, which can be counterproductive.

The relaxed form of TDD, on the other hand, emphasizes the importance of writing tests when it speeds up the development process. For example, it will be beneficial to write tests for the motion detection module to have a set of sample video clips with and without motion to efficiently test the module and not have to manually record the video clips every time.

A Python standard library unittest was used to write the unit tests. The unit tests are located in the `tests/` directory of the project. The test coverage was measured using the Coverage.py[12] tool. The coverage report is presented in Figure 12.

Coverage report: 64%

coverage.py v7.4.3, created at 2024-03-10 21:29 +0300

Module	*statements*	*missing*	*excluded*	*coverage*
vigi_agent/__init__.py	0	0	0	100%
vigi_agent/context_processors.py	13	2	0	85%
vigi_agent/motion_detector.py	67	58	0	13%
vigi_agent/notification_providers/__init__.py	0	0	0	100%
vigi_agent/notification_providers/email_notification_provider.py	26	0	0	100%
vigi_agent/notification_providers/sms_notification_provider.py	14	0	0	100%
vigi_agent/routes/__init__.py	0	0	0	100%
vigi_agent/utils/drawing.py	14	9	0	36%
vigi_agent/utils/fps_calculator.py	15	0	0	100%
vigi_agent/utils/media.py	26	3	0	88%
vigi_agent/utils/pub_sub.py	18	0	0	100%
vigi_agent/utils/spatial.py	7	0	0	100%
Total	**200**	**72**	**0**	**64%**

coverage.py v7.4.3, created at 2024-03-10 21:29 +0300

Figure 12: Unit test coverage report generated by Coverage.py. Please note: It does not include untested files.

[12] https://coverage.readthedocs.io/en/7.4.3/

The unit tests were run every time the code was pushed to the repository by using GitHub Actions. Code quality was evaluated using the Pylint tool[13]. The Pylint tool provides a detailed report of the code quality with hints on how to improve the code. The code has been rated at **9.80/10** by Pylint suggesting that the code quality is high. The Pylint report is provided in the `pylint_report.txt` file in the root directory of the project.

Motion Detection

To evaluate the accuracy and performance of the selected motion detection algorithm, I have created a big dataset of video clips with and without moving objects. The clips contained different scenes with different lighting conditions, different moving objects, and different types of noise, like camera vibration or sensor noise. To create the dataset, I have used the following cameras:

- iPhone 13 back camera 1920x1080@30 FPS
- Hikvision DS-U02 web camera 1920x1080@30 FPS
- Apple Macbook Air M1 FaceTime HD camera 1080×720@30 FPS
- Apple Macbook Air 13-inch (Mid 2014) Webcam 1080x720@30 FPS
- Samsung Galaxy A14 back camera 1920x1080@25 FPS

The dataset[14] contains more than 20 video clips of 10-20 seconds filmed with different lighting conditions, different moving objects, and different types of noise, like camera vibration or sensor noise (Figure 13).

I have created a unit test `tests/test_motion_detector.py` that runs the motion detection algorithm against this dataset and compares the results with the expected results. All tests passed, which means that the motion detection algorithm is accurate and reliable in distinguishing between the events of interest and the background noise, changing lighting conditions, and other factors. It also reliably works with different types of cameras, formats, resolutions, and FPS. Moderate camera vibration does not trigger false positives.

[13] https://pylint.readthedocs.io/en/stable/

[14] The dataset can be downloaded from here: https://vigi-statics.s3.us-west-2.amazonaws.com/samples.zip

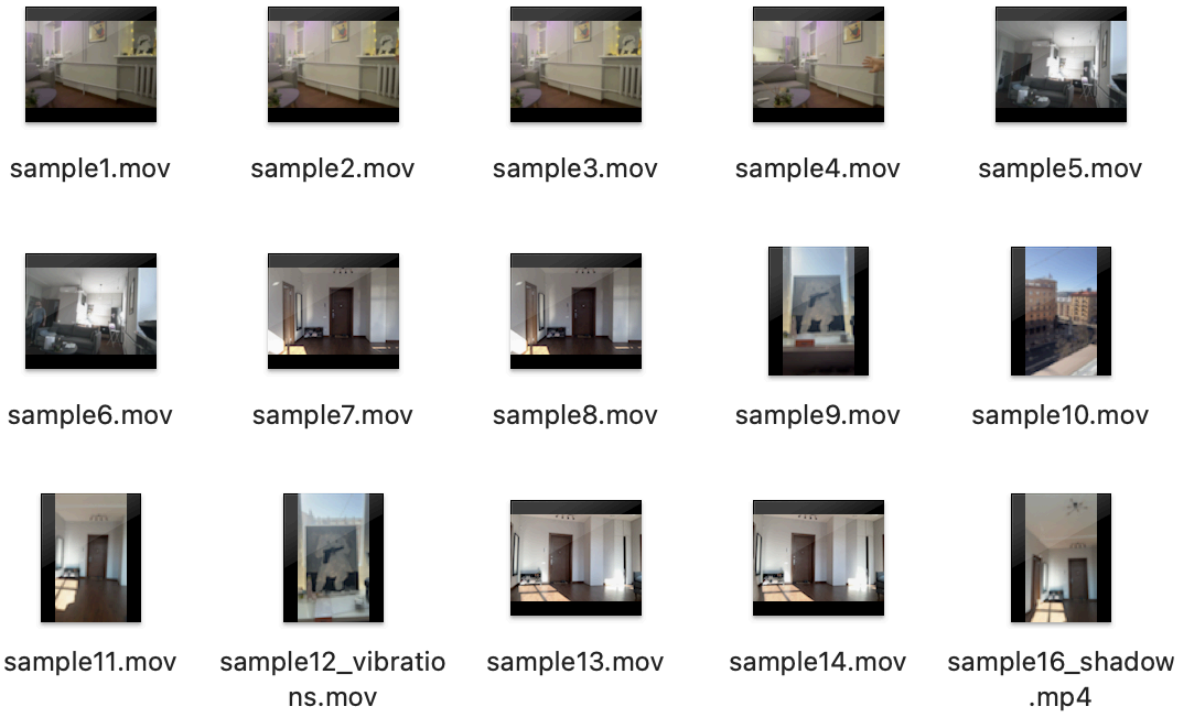

Figure 13: Dataset of video clips with and without moving objects used to evaluate the motion detection algorithm.

Recording formats

The format of the recorded video clips was evaluated manually by recording and playing video clips in HTML5 video players in different web browsers and operating systems. H.264/MP4 was recorded and played in all the tested web browsers and operating systems while other test formats (VP8/WebM, MPEG-4, etc.) showed compatibility issues.

Another useful tool for the evaluation was the "Can I use..." website[15], which provides a detailed compatibility table of the video formats and codecs across different web browsers and operating systems (Figure 14).

[15] https://caniuse.com/

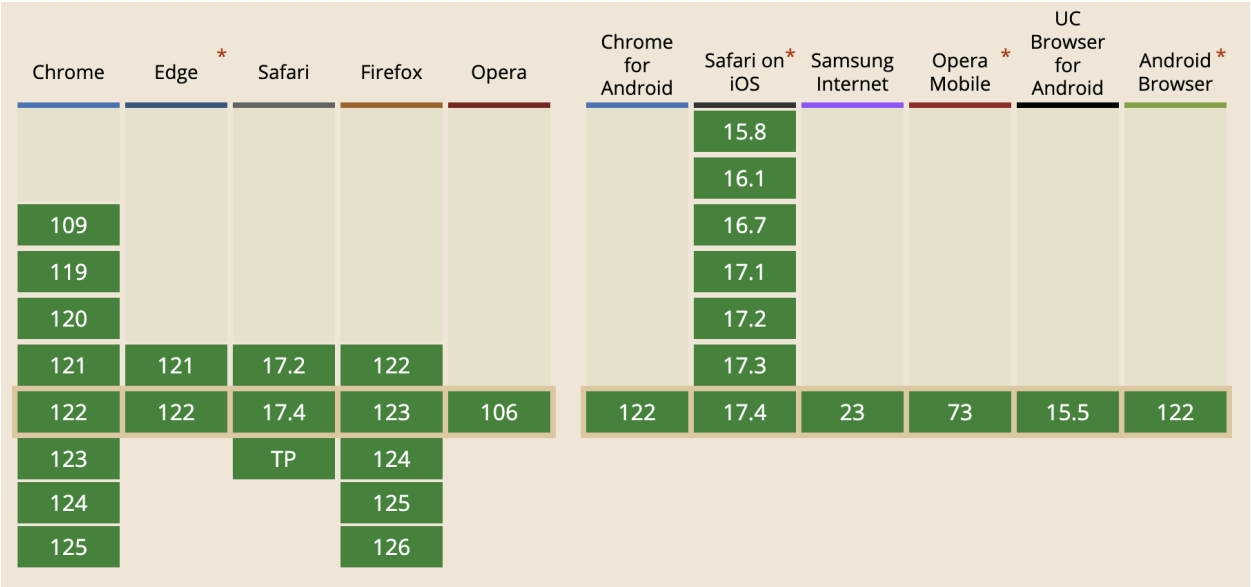

Figure 14: Compatibility of the H.264 codec with different web browsers and operating systems as per the "Can I use..." website[16].

Performance

The performance was evaluated by running the system on available hardware:

- Apple MacBook Air M1 16GB RAM
- Apple MacBook Air (Early 2014) 8GB RAM
- Raspberry Pi 3 B V1.2 (Figure 15 and 17).

The performance was evaluated in terms of subjective user experience, such as the latency and quality of the video stream, as well as by objective measurements, such as the FPS of the video stream and the CPU and memory usage of the system. To collect the objective measurements, I have created a script `scripts/cpu_mem_profiler.py` that uses the psutil[17] library to collect the CPU and memory usage.

The script was run on the Raspberry Pi to collect the measurements of the system while it was running the motion detection and object recognition algorithms in real-time (Figure 16). The performance was found to be satisfactory on all the tested

[16] https://caniuse.com/?search=H.264
[17] https://psutil.readthedocs.io/en/latest/

hardware, however, on the Raspberry Pi, FPS of the system is a bit low (around 2 FPS) when the object recognition module is enabled.

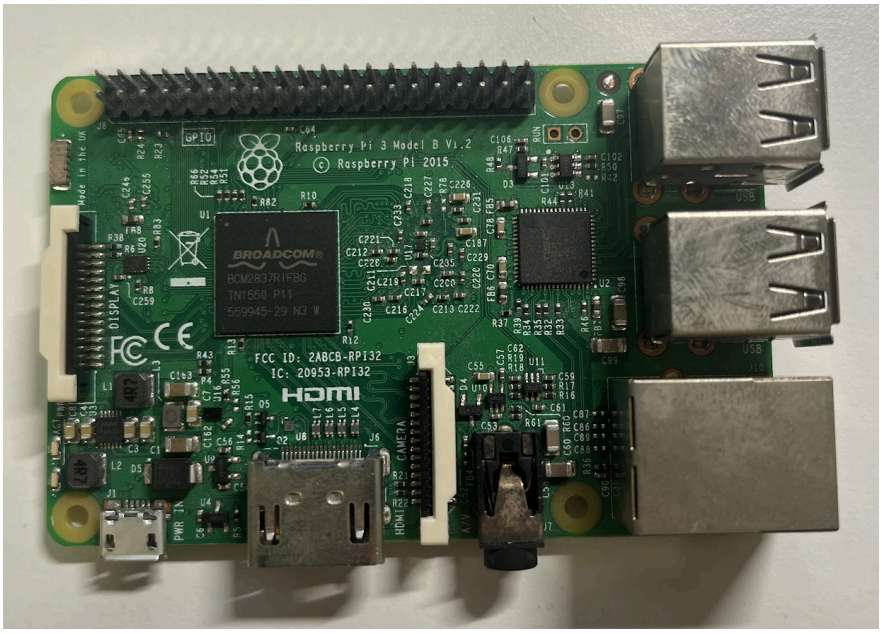

Figure 15: Raspberry Pi board used to test the system.

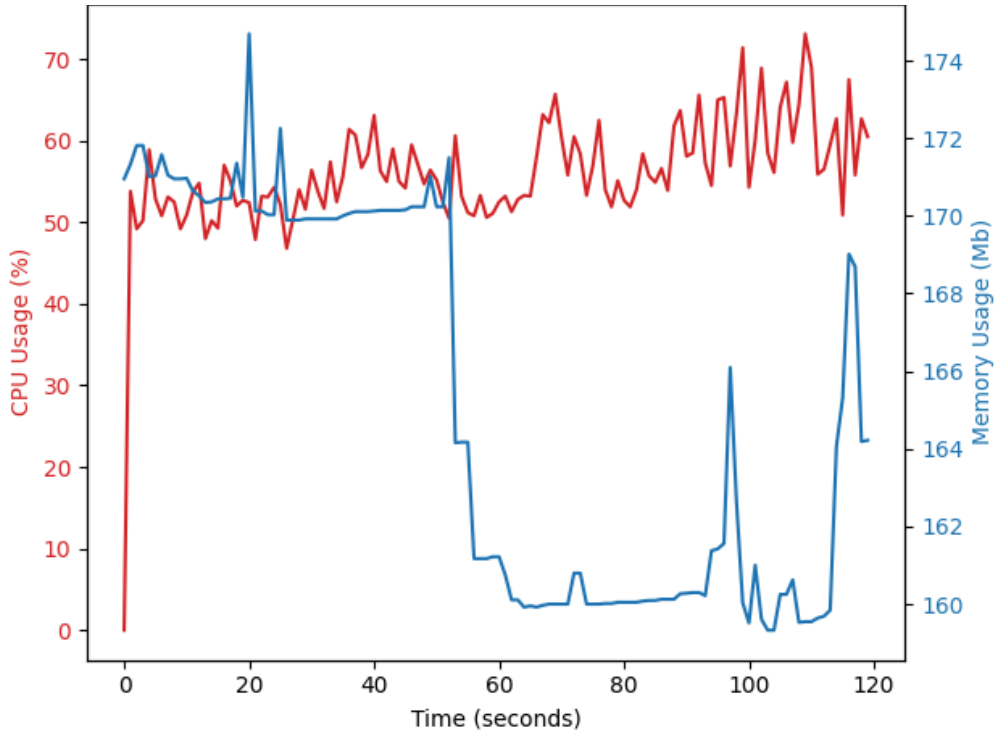

Figure 16: CPU and memory usage measured on Raspberry Pi with one camera.

Figure 17: A Raspberry Pi 3 B with connected Hikvision DS-U02 web camera.

Web console

Security of the web console was evaluated by setting up a HTTP Basic Authentication by command-line arguments `--http-basic-username` and `--http-basic-password` or `--http-basic-hashed-password`. After setting up the authentication, the web console required the user to enter the username and password to access any page of the web console. The authentication was found to be secure and reliable.

Accessibility of the web console was evaluated on the following devices and web browsers:

- Apple MacBook Air with the Safari and Google Chrome web browsers (Figure 18 and 19).
- A Windows laptop with Edge and Google Chrome web browsers.
- Apple iPhone 13 with Safari web browser.
- Android smartphone Samsung Galaxy A14 with the Google Chrome web browser.

As a result, the web console was found to be accessible from most modern web browsers and was responsive to different screen sizes, such as mobile phones and tablets.

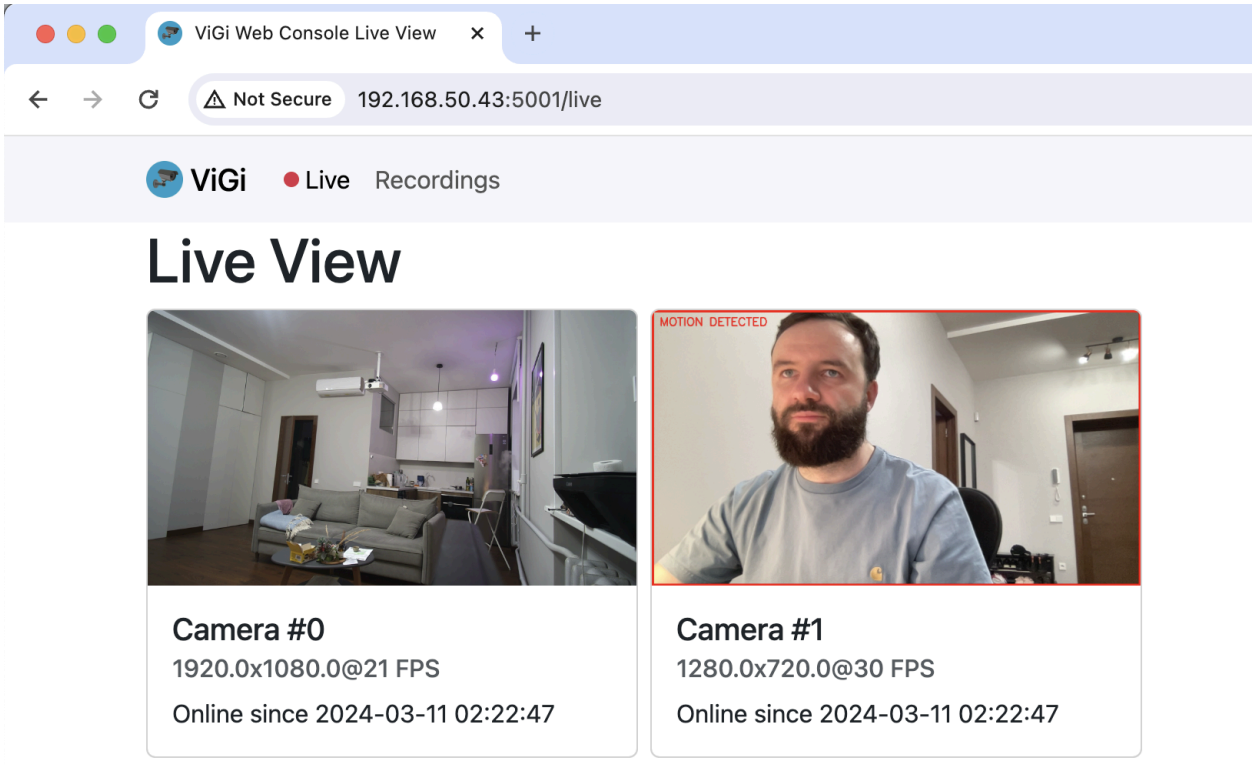

Figure 18: Live view page of the web console

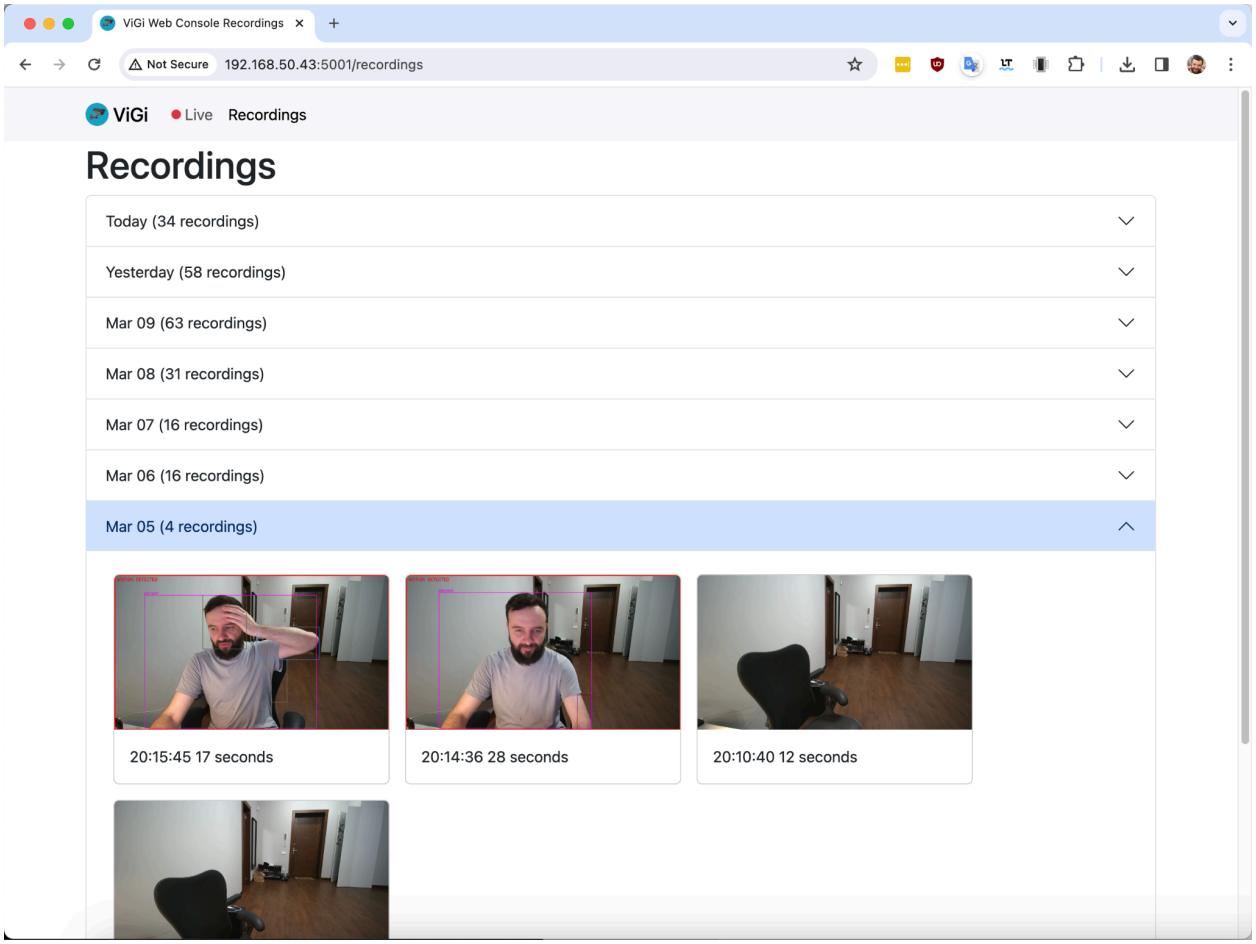

Figure 19: Recordings page of the web console.

Acceptance Testing

Acceptance testing was performed manually by following the testing scenarios provided in the Appendix section of this report. These scenarios were designed to cover the most important features of the system, such as the live video feed, motion detection, object recognition, notifications, recordings preview, and management.

The testing scenarios were successfully performed on the aforementioned devices and web browsers. Some of the scenarios were also recorded and are provided as part of the attached presentation video.

Conclusion

I believe that the difference between a prototype and a mature version is the degree in which details are polished. A mature version is expected to have production-grade security, error handling, logging, caching, security, and graceful shutdown. Often, the amount of work required to finalize a solution is much larger than the amount of work required to prototype a new fancy feature.

In this project, my goal was to create a mature version with a reasonable set of features. I aimed at a robust and reliable system rather than a feature-rich, but fragile. For this reason and due to the time constraints, I decided to focus on improving and polishing the core features and sacrificing some of the advanced features, such as a mobile application, I initially planned to implement.

Due to the time constraints, I had no chance to conduct a formal user testing, but I personally used this project for my own needs for the last few weeks andI saw the system in the wild. I believe that it is already a useful complete solution for the home needs with enough features to be a substitute for the commercial cloud services I used before.

I believe the most distinctive feature of this system is the ability to run multiple cameras simultaneously and to provide a live video feed from all the cameras in the web console. This feature allows the user to monitor multiple areas of the house at the same time, which makes the system more useful and versatile.

The system was designed with modularity and extensibility in mind, so it is possible to extend the system with the advanced features in the future. For instance, it is trivial to add a new notification method, such as Web Push. It requires only one new module and a few lines of code. The same applies to object recognition. It is possible to replace the YOLO model with a more advanced model, such as RT-DETR, it requires only a few lines of code to replace the model.

The original work plan is presented in the Appendix section of this report. The work plan was followed closely, with the exception of the mobile application, which was not

implemented due to the time constraints. However, due to the agile nature of the project, it was possible to omit some of the features without affecting the overall quality of the system.

Things I would do differently if I had more time:

- I would conduct a formal user testing to get feedback from the real users. I am aware that user testing and feedback are crucial for the success of any project targeting a wide audience.
- The recording page in the web console is simplistic and lacks necessary features, such as search, sorting, and pagination. I would implement these features to make the recording management more convenient.
- Configuration is currently stored in an INI file, which is not very user-friendly. I would implement a web page in the web console to manage the configuration.
- I would implement an automatic cleanup of old recordings to prevent the storage from being filled up with the old recordings.

Overall, I am satisfied with the result of the project. It gave me a chance to learn a lot of new things and to apply the knowledge I gained during the course. It also pushed me to an unprecedented level of self-discipline, as I had to work and study two other courses at the same time.

References

[1] Hussien, H.M., Meko, S.F., Teshale, N.B. 2018. *Comparative Analysis of Moving Object Detection Algorithms*. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 244. Springer, Cham. Retrieved Jan 7, 2024 from https://doi.org/10.1007/978-3-319-95153-9_16.

[2] M. Ozan Tezcan, Prakash Ishwar, Janusz Konrad. 2021. BSUV-Net 2.0: *Spatio-Temporal Data Augmentations for Video-Agnostic Supervised Background Subtraction*. arXiv:2101.09585. Retrieved Jan 7, 2024 from https://arxiv.org/abs/2101.09585.

[3] MotionPlus developers. 2023. MotionPlus, a software motion detector. Retrieved Jan 7, 2024 from https://motion-project.github.io/.

[4] Wenyu Lv, Yian Zhao, Shangliang Xu, Jinman Wei, Guanzhong Wang, Cheng Cui, Yuning Du, Qingqing Dang, Yi Liu. 2023. *DETRs Beat YOLOs on Real-time Object Detection*. arXiv:2304.08069. Retrieved Jan 7, 2024 from https://arxiv.org/abs/2304.08069.

[5] Max Wilbert. 2023. *Video Streaming Protocols: 6 Preferred Formats for Professional Broadcasting*. Retrieved Jan 7, 2024 from https://www.dacast.com/blog/video-streaming-protocol/.

[6] ZoneMinder developers. 2023. *ZoneMinder, an open-source CCTV software*. Retrieved Jan 7, 2024 from https://zoneminder.com/.

[7] Jocher, G., Chaurasia, A., & Qiu, J. 2023. *YOLO by Ultralytics (Version 8.0.0)*. Retrieved Jan 7, 2024 from: https://github.com/ultralytics/ultralytics.

Appendix

Credits

Visual elements used in the report:

- 2024. CCTV icon by Creative Stall Premiumm. 2024. Retrieved Jan 7, 2024 from: https://www.flaticon.com/free-icon/cctv_695308?term=cctv&page=1&position=25&origin=search&related_id=695308.
- Icons by the Miro team. 2024. Retrieved Jan 10, 2024 from https://miro.com/.
- A FFmpeg icon by the FFmpeg team. 2024. Retrieved Jan 10, 2024 from: https://en.wikipedia.org/wiki/FFmpeg#/media/File:FFmpeg_Logo_new.svg
- An OpenCV icon by the OpenCV tram. 2024. Retrieved Jan 10, 2024 from: https://en.wikipedia.org/wiki/OpenCV#/media/File:OpenCV_Logo_with_text_svg_version.svg
- A SQLite icon by the SQLite team. 2024. Retrieved Jan 10, 2024 from: https://en.wikipedia.org/wiki/SQLite#/media/File:SQLite370.svg
- A FastAPI icon by the FastAPI team. 2024 Retrieved Jan 10, 2024 from: https://fastapi.tiangolo.com/.
- Wireframe visuals by https://balsamiq.com/.

Third party software used

- GitHub — for hosting code.
- https://miro.com/ — For drawing diagrams.
- Balsamiq — for wireframes.
- https://www.figma.com/ — For artworks, icon customization.
- Apple iMovie — for preparing the video presentation.
- Visual Studio Code — for writing code.

Acceptance Testing Scenarios

Scenario 1: One camera, live video feed

1. Install the system on a local server using the installation instructions provided in the README.md file.
2. Run the system with one connected camera.
3. Open the web console in a browser.
4. The expected outcome is that the web console displays the live video feed from the connected camera.
5. Click on the small card with the live video feed.
6. The expected outcome is that the video feed is displayed in a pop-up with a larger size.

Scenario 2: Multiple cameras, live video feed

1. Run the system with multiple connected cameras.
2. Open the web console in a browser.
3. The expected outcome is that the web console displays the live video feeds from all the connected cameras.
4. Click on each small card with the live video feed.
5. The expected outcome is that the video feed is displayed in a pop-up with a larger size for each camera.

Scenario 3: Motion detection

1. Run the system with one connected camera.
2. Open the web console in a browser.
3. Move any object in front of the camera to trigger the motion detection.
4. The expected outcome is that the web console displays the live video feed from the connected camera with a bounding box around the moving object.
5. Open the recordings tab in the web console.
6. The expected outcome is that the web console displays the recording with the detected event.
7. Click on the recording with the detected event.

8. The expected outcome is that the web console displays a pop-up with a player that plays the recording with the detected event.

Scenario 4: SMS Notifications

You will need a Twilio account to perform this scenario.

1. Configure SMS notification provider using the instructions provided in the README.md file using the Twilio API keys.
2. Run the system with one connected camera.
3. Move any object in front of the camera to trigger the motion detection.
4. The expected outcome is that the system sends an SMS notification to the configured phone number.

Scenario 5: Object recognition

1. Run the system with one connected camera and the YOLOv8 model for object recognition.
2. Move any recognizable object in front of the camera to trigger the object recognition.
3. Open the web console in a browser with the recordings tab.
4. Find the recording with the detected event.
5. Click on the recording with the detected event.
6. The expected outcome is that the web console displays a pop-up with a player and under the player, it displays a list of the recognized objects in the recording.

Work Plan

This is the original work plan that was presented in the beginning of the project. Agile methodology will be used for the development of this project. The work will be divided into a number of sprints, each lasting two weeks. By the end of each sprint, a deliverable piece of software will be produced. This approach will allow for a more flexible development process and will make it easier to adapt to the changing requirements, less subject to the risks of unforeseen circumstances. For example, if a particular feature turns out to be more complex than expected, it can be omitted or simplified without affecting the rest of the project. Even by the end of the first sprint, a working version of the system will be ready to use, albeit with limited functionality.

The work will be divided into the following sprints:

Sprint #	**Tasks (Backlog)**
1. Dates: 11.12.23 — 26.12.23	Setup the development environment, create a prototype of a basic HTTP server that streams a video from a camera: 1. Setup the development environment 2. Create a basic HTTP server using the FastAPI framework 3. Implement a basic streaming module using the FFmpeg library 4. Create a HTML page that displays a video stream from the connected camera
2. Dates: 26.12.23 — 08.01.24	Implement a prototype of the motion detection module: 1. Connect to a camera using the OpenCV library. 2. Implement a basic motion detection algorithm that detects motion in the video stream 3. Save video recording with the detected motion to the disk
3. Dates: 08.01.24 — 22.01.24	Web console: 1. Implement a web server using the FastAPI framework that provides access to the live video feeds ("Live" page). 2. Implement a web console that provides access to the video recordings ("Events" page).
4. Dates: 22.01.24 —	Mobile application: 1. Implement a mobile application that provides access to the live video feeds ("Live" screen).

05.02.24	2. Implement a list of events with video clips ("Events" screen).
5. Dates: 05.02.24 — 19.02.24	Object recognition module, a draft report: 1. Implement a basic object recognition module using a pre-trained deep learning model that recognizes objects in the video stream and labels the video recordings accordingly. 2. Implement a push notification service that sends push notifications to the user's mobile device when motion is detected with the type of the object. 3. Create and file the draft report.
6. Dates: 05.02.24 — 19.02.24	Support for multiple cameras: 1. Support for multiple cameras for the streaming module. 2. Support for multiple cameras for the motion detection module. 3. Support for multiple cameras for the web console. 4. Support for multiple cameras for the mobile application.
7. Dates: 19.02.24 — 04.03.24	Evaluation, testing, and report: 1. Evaluation and testing 2. Create and file the final version of the report.