DOI: 10.12700/AIS.2025.004

Development of AI-Based Surveillance Software

Kornél Szabó
Alba Regia Faculty
Obuda University
Székesfehérvár, Hungary
szkornel005@gmail.com

Éva Hajnal
Alba Regia Faculty
Obuda University
Székesfehérvár, Hungary
hajnal.eva@amk.uni-obuda.hu

***Abstract*—This paper presents the development of an intelligent video surveillance software designed to perform real-time processing of security camera footage. The system combines motion detection, object detection, and tracking to separate relevant activities from background variations. A web-based interface was designed to support camera management and recording access. Results from a 30-day test with four cameras showed that the software reduced storage usage by approximately 98.4% compared to continuous recording. The system was also able to process video in real-time at 10 FPS per camera. This makes multi-camera monitoring both efficient and user-friendly by recording only when activity (e.g., a person) is detected within user-defined zones and allowing easy event filtering.**

Keywords—artificial intelligence, video surveillance, real-time processing, motion detection, object detection, object tracking

I. Introduction

Video surveillance has long been an important part of security, helping to keep lives and property safe. These systems are now widely used in both public and private areas, and the resulting video data can pose challenges for storage, processing, and management. Continuous recording from numerous cameras consumes a large amount of storage and can be difficult to review afterward. Manual video monitoring relies on human operators, which is expensive, time-consuming, prone to error, and difficult to oversee when there are many cameras [1].

Advances in machine vision and artificial intelligence enable automatic filtering and highlight relevant details. This reduces storage usage, as only important activities are recorded, and makes it easier for operators to review footage, even with multiple cameras. Intelligent video surveillance systems are capable of detecting motion in real time, recognizing and classifying objects, and tracking their movements. This reduces the possibility of human error, speeds up response times in emergencies, and makes the system easily scalable, meaning that larger areas and more cameras can be monitored effectively [2], [3].

These advancements motivate the development of software capable of efficiently processing multiple video streams and extracting information relevant to security monitoring. The main problem addressed in this paper is how a video surveillance system could be implemented that is capable of processing video streams from multiple cameras in real time, detecting and tracking relevant objects, and selectively recording events within user-defined zones, while minimizing computational requirements.

To address this problem, the objectives of the work are as follows:

- Ensure real-time processing of multiple video streams.
- Detect and track objects (e.g., people or vehicles) in monitored zones.
- Implement selective recording based on motion and zone definitions.
- Provide a user interface for monitoring video streams, managing recorded events, and playback.

A. Overview of Relevant Technologies

1) Motion Detection Methods

Motion detection is a fundamental area of machine vision, which involves identifying changes in the position of objects relative to their background. It plays an important role in many areas of application (e.g., intelligent video surveillance, traffic monitoring). In motion detection, it is necessary to distinguish between what is static (background) and what is moving (foreground) [4].

a) Frame Differencing

This is traditional and one of the simplest solutions. It takes the difference between two consecutive frames. Among the discussed methods, it requires the least computational resources. Its disadvantage is that it cannot accurately identify the entire contour of a moving object [4]. Results are noisy. Blurring and various morphological operations (e.g., closing) can be used to remove noise and improve the quality of motion detection [5].

b) Temporal Differencing

This is based on the temporal difference between consecutive frames, pixel by pixel, focusing on changes over time. It does not give good results if the object to be detected moves too slowly or too quickly, because in these cases the difference between frames is very small. It is capable of handling sudden changes in lighting conditions in indoor environments [4].

c) Optical Flow

One of the most computationally intensive methods among traditional solutions. In this case, motion detection is based on estimating the optical flow field of the image and then clustering based on the optical flow distribution of the video frames. It is memory-intensive, requires complex calculations, and is sensitive to noise [4]. Similar to Frame Differencing, a significant part of the noise can be removed with blurring and morphological operations, thus improving the quality of the motion detection [6].

d) Background Subtraction

One of the simplest and most reliable motion detection methods. A reference frame (background model) is given. The difference between the current frame and the reference frame is calculated. The resulting image shows the moving object. It is not suitable for handling dynamically changing backgrounds [4]. The reference frame can be updated over time, allowing changes in the background to be tracked. In such cases, the noise can remain minimal [7].

2) Object Detection Methods

Object detection is a key area of computer vision that is essential to many applications (e.g., self-driving vehicles, pedestrian and face recognition). Object detection techniques can be divided into two main groups: traditional methods, which use various machine learning techniques, and deep learning-based approaches [8].

a) Traditional Solutions

Before the advent of deep learning, object detection relied on manually extracted features with classical machine learning algorithms. Human expertise was key in selecting relevant features. Main methods include General Hough Transform (Ballard D, 1981), Harris Corner Detector (1988), and SIFT (Scale-Invariant Feature Transform, Lowe, 2004) [8].

b) Deep Learning-Based Methods

Deep learning techniques require less manual feature extraction and perform better with large datasets. They can independently learn features, adapt to environmental conditions, and detect multiple objects in real time. R-CNN (Region-based Convolutional Neural Networks) architectures (e.g., R-CNN, Fast/Faster/Mask/Cascade R-CNN) follow a two-stage process: region proposal and then object detection on the proposed regions [8], [9]. YOLO (You Only Look Once) and SSD (Single Shot Detector) architectures perform detection in a single step eliminating the need for a separate regional proposal stage [10].

Based on a comparative analysis conducted by *Subbiah et al.* [11], the YOLO architecture performs well in real-time surveillance, as it can detect objects quickly without much loss in accuracy. Although certain R-CNN models can achieve higher accuracy, their processing times are longer, making them less suitable for real-time applications.

B. Test Environment Overview

Four cameras were installed on a suburban residence to monitor critical areas of the property: Entrance Gate, Terrace, Rear Garage, and Front Garage. Fig. 1 shows the location and coverage areas of the surveillance cameras.

Fig. 1. Camera placement on the property. Top-left: Front Garage; Top-right: Entrance Gate; Bottom-left: Rear Garage; Bottom-right: Terrace. (Public areas masked in black.)

II. Methods

The system consists of four Hikvision DS-2CE16H0T-IT3F analog cameras connected by coaxial cables to a central Hikvision DS-7204HTHI-K1 DVR. The DVR streams video over the network at a resolution of 5 MP (2560×1944 px), 20 FPS, with a constant bitrate of 12,288 Kbps and H.264 encoding. Processing is performed by a high-performance computer running Windows 11, with an Intel Core i7-13700K CPU, Nvidia GeForce RTX 4090 GPU, and 32 GB of RAM.

The software was implemented in C# using the .NET Core framework. Motion detection, object detection, and tracking algorithms run on the backend, and the processed data is transmitted via a JSON API to a web-based Blazor interface, where it is displayed and user interactions occur.

Live video data is streamed from the installed cameras using the RTSP protocol. Several NuGet packages were used. For computer vision tasks and camera access, the OpenCV .NET wrapper EmguCV [12] was utilized. YOLO object detection was implemented using YoloDotNet [13]. The Jonker-Volgenant algorithm for object tracking was applied via the LapjvCSharp package [14]. For image manipulation and rendering, SkiaSharp was employed [15].

The base structure of the system, which is executed for each frame that needs to be processed, is illustrated in Fig. 2.

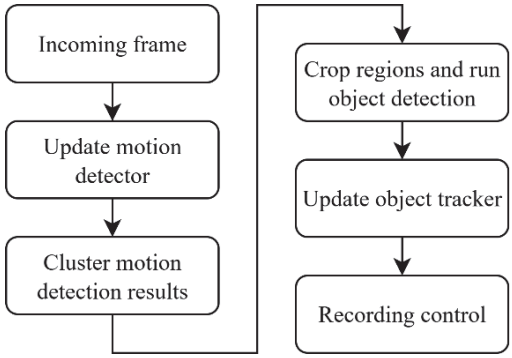

Fig. 2. Block diagram of the base process, which executes on every frame to be processed. Video frames update the motion detector, generating motion boxes that are clustered into detection regions. These regions are processed by the object detector, producing bounding boxes, labels, and confidence scores, which update the centroid-based tracker. Recording is triggered when an active object with the target label is detected and stopped after a set period of inactivity.

Fig. 3 presents a frame with motion boxes (blue), detection regions (green), and detected object bounding boxes (red).

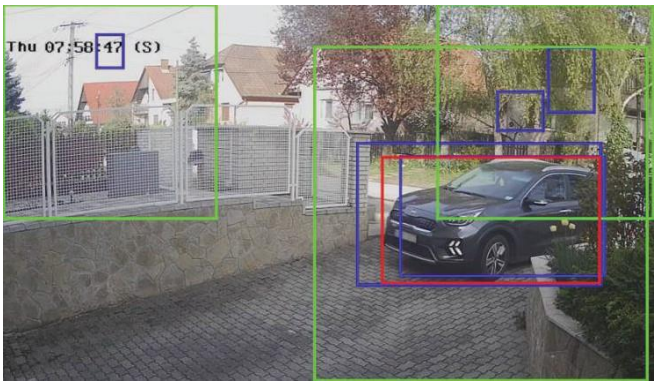

Fig. 3. Frame illustrating the motion detection and object detection process. Blue boxes indicate areas of detected motion, green boxes represent clustered detection regions derived from nearby motion, and red boxes show the bounding boxes of objects identified by the object detection algorithm.

Fig. 4 shows the image sections corresponding to the three detection regions shown in Fig. 3 used for object detection. In Fig. 4(a), no objects were detected. In Fig. 4(b) and Fig. 4(c), the same vehicle was detected in both regions. The object tracker associated these detections with the same vehicle, resulting in a single tracked object.

Fig. 4. Image sections of the detection regions shown in Fig. 3, cropped from the original frame for object detection. (a) No objects detected; (b) and (c) the same vehicle detected.

A. *Zones and Motion Masks*

Users can define motion masks and zones on the cameras by specifying polygon vertices, as illustrated in Fig. 5. Motion detection is disabled in areas covered by motion masks, thereby reducing resource usage. If at least one zone is defined, the system records only events occurring within that zone. Each zone can be named, and the object types of interest (e.g., *person* and *car* labels) can be specified.

Fig. 5. Example of a motion mask and a zone named “test_zone.” Zone vertices (blue) and motion mask vertices (red) are specified by their (x, y) coordinates. The black area formed by the red vertices represents the motion mask, while the blue area formed by the blue vertices represents the zone. The image coordinate system has its origin at the top-left corner (0, 0) and extends to the bottom-right corner (2560, 1944).

An object is considered within a zone if the bottom-center of its bounding box lies inside or on the edge of the zone, determined using EmguCV's [12] *PointPolygonTest* function. It returns +1 (inside), -1 (outside), or 0 (on an edge).

B. *Motion Detection*

From the relevant technologies introduced previously, considering the requirements of real-time processing for motion detection, the Background Subtraction method was selected, specifically using the Running Average [16] technique. This means that newly arriving frames are not compared to a pre-recorded static reference image (e.g., the first frame received), but to a continuously updated moving average image that is adjusted to the incoming frames with a small weight. This approach allows the background to dynamically follow slower changes in the environment while detecting objects that appear suddenly (e.g., moving people or objects) as foreground.

During the preprocessing of the incoming video frames, the images undergo grayscale conversion and are then resized to 100 pixels high while maintaining the aspect ratio. Optionally, contrast enhancement is performed, followed by a motion mask to filter out uninteresting areas, and Gaussian filtering to reduce noise. The background for motion detection is modeled using a moving average image, which is updated with exponential weighting (a = 0.01, during calibration a = 0.2). The difference between the current frame and the background model is computed. This difference is then thresholded and dilated. Contours that are too small are discarded, keeping only relevant movements, from which motion boxes are formed. The size of the boxes is scaled back to the original image size. Calibration starts when motion affects a significant part of the frame; during calibration, the background model is updated at an accelerated rate until a stable state is restored. The process is illustrated in Fig. 6.

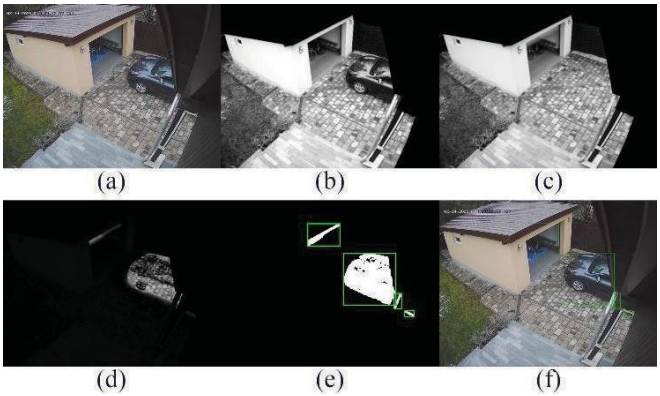

Fig. 6. Steps of motion detection from incoming video frames: (a) original frame, (b) grayscale conversion, resizing, optional contrast enhancement, motion mask application, and Gaussian filtering, (c) background model (moving average), (d) absolute difference between the current frame and the background model, (e) binary mask after thresholding and dilation with significant contours highlighted, (f) final motion boxes scaled to the original frame size.

Detected motion boxes are clustered to reduce redundancy and to prepare for subsequent object detection. Each region is expanded by a scaling factor around the boxes, and overlapping boxes are merged. Each cluster is ensured to be at least as large as the input size of the detection model. The system output comprises detection regions, each containing one or more motion boxes. The process is illustrated in Fig. 7.

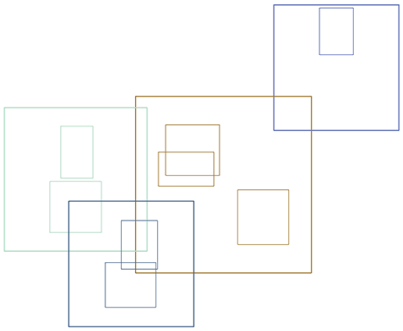

Fig. 7. Motion boxes are grouped into clusters, each forming a detection region. Different colors indicate clusters; boxes in the same cluster share the same color.

C. Object Detection

Object detection is performed on the detection regions obtained from motion detection. Each region is cropped from the original frame and processed using the YoloDotNet package [13] with the pre-trained YOLOv12s model [17]. Detections are filtered by class labels and confidence thresholds, and bounding boxes are mapped back to the original frame coordinates. If a detected object lies near the edge of the detection region in which it was detected, the region is expanded, and detection is repeated. Overlapping detections are resolved using Non-Maximum Suppression (NMS), and highly overlapping duplicates are removed. The output includes the object class, confidence score, bounding box, and the corresponding detection region, which is then used for object tracking based on bounding box centroids.

D. Object Tracking

The main purpose of object tracking in this system is to determine whether an object becomes stationary, such as a parked vehicle. Several solutions can be used to track recognized objects. Due to real-time processing, a simpler technique was chosen, which tracks objects based on the centroids of the bounding boxes surrounding them (centroid object tracking [18], [19]). The tracking result for a passing vehicle is illustrated in Fig. 8.

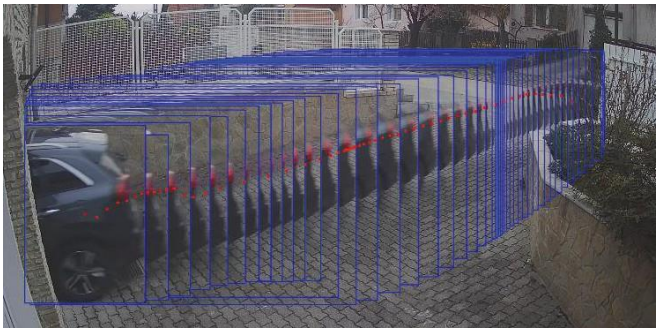

Fig. 8. Tracking of a vehicle over 56 consecutive frames. Blue rectangle: detected object; red dot: centroid used for tracking. The tracker updates object positions frame by frame by matching centroids of detected objects to previously tracked objects, handling entries, exits, and temporary occlusions.

The object tracker receives the object detection results (bounding boxes, labels, and confidence scores) for each processed frame. Let the set of currently tracked objects be

$$T = \{ T_1, T_2, ..., T_m \} \quad (1)$$

and the set of detected objects in the current frame be

$$D = \{ D_1, D_2, ..., D_n \} \quad (2)$$

where m and n are the numbers of tracked and detected objects, respectively. Each tracked object T_i and detected object D_j has a centroid $c_{T_i}, c_{D_j} \in R^2$. A cost matrix $C \in R^{m \times n}$ is constructed using the Euclidean distances between the centroids:

$$C_{ij} = ||c_{T_i} - c_{D_j}||_2, i=1, ..., m, j=1, ...n \quad (3)$$

The assignment problem aims to match detected objects to tracked objects to minimize total cost. This is formulated as a Linear Assignment Problem (LAP) and can be efficiently solved using the Jonker-Volgenant algorithm [20] applied via the LapjvCSharp package [14]. The solver outputs an assignment array:

$$x[i], i=1, ..., m \quad (4)$$

where $x[i]$ is the index of the detected object assigned to tracked object T_i. If no detection is assigned to T_i, then $x[i] = -1$. Using these assignments, each tracked object is updated with information from the corresponding detected object.

In practical scenarios, the number of detected objects may differ from the number of tracked objects ($n \neq m$):

1. More detected objects than tracked objects ($n > m$): New objects enter the scene. Unassigned detected objects are initialized as new tracks.
2. Fewer detected objects than tracked objects ($n < m$): Objects leave the scene or become occluded. Unassigned tracked objects are temporarily retained and later deleted if they do not reappear after a certain number of frames.

E. Database and Storage Management

Data related to recordings and detected objects are stored in an SQLite database. The database was implemented using the Entity Framework Core Object-Relational Mapping (ORM) framework with a Code-First approach. The resulting database structure is illustrated in Fig. 9.

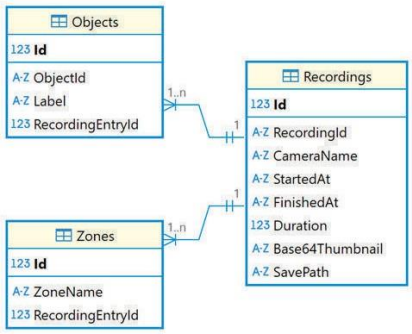

Fig. 9. Entity-Relationship diagram of the SQLite database showing the structure of the Objects, Zones, and Recordings tables, including primary keys, attributes, and relationships.

The storage manager maintains available disk space and enforces retention policies. It periodically deletes recordings older than the configured retention period and can perform emergency cleanup when free disk space falls below a defined threshold.

The *appsettings.json* file enables users to configure system parameters, including camera properties, motion detection thresholds, YOLO model parameters, recording and storage settings.

F. Web-based User Interface

A web-based user interface was developed using the Blazor framework and Bootstrap for UI design, as illustrated in Fig. 10, enabling users to view live camera feeds, camera metadata, motion masks and zones, active motion boxes, detection regions, object detections, and recorded videos. Recordings can be filtered by camera, zones, object labels, and recording start times. Basic recording information is also available. Additionally, recordings can be played back, downloaded, or deleted.

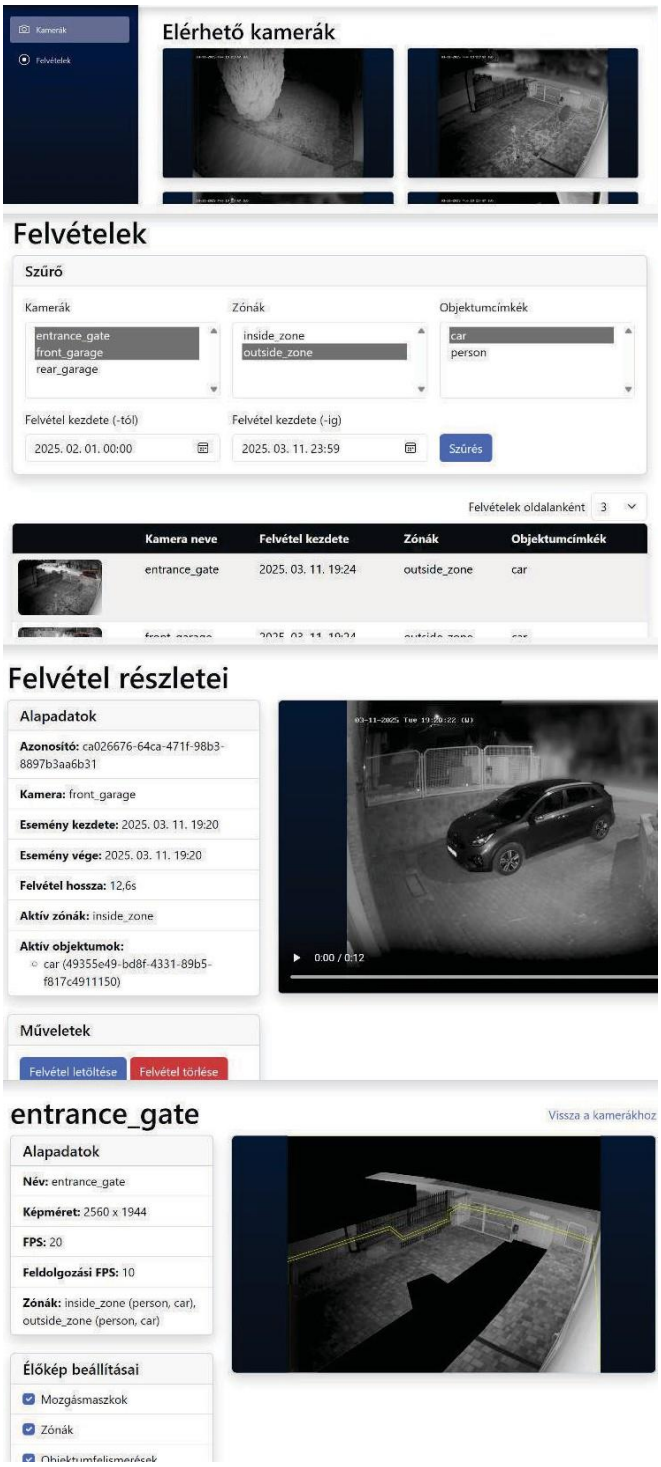

Fig. 10. Web-based user interface of the application, showing pages for available cameras, recordings, recording details, and camera details, as well as features for viewing live camera feeds, motion detection areas, object tracking, and managing recordings with playback, download, and deletion options.

If there are any, other recordings related to a given recording will appear on the same interface. A recording is linked to another if they have at least one active object in common (for example, if a recording stops because an arriving car has parked, and then the same car drives away in another recording, the system links them together due to object tracking and displays them as related recordings). However, due to the use of simple centroid-based tracking, objects of the same type passing in front of each other may occasionally be incorrectly associated.

III. Results and Discussion

A. Performance Evaluation

The performance of the created system in terms of memory usage is primarily determined by the characteristics of the incoming video streams (frame rate, image size, bit rate). Higher resolution, frame rate, and bit rate require more memory to store raw frames in the pre-recording buffer. In the current setup, with four cameras and a 3-second pre-recording, the average memory usage is 5380 MB.

Processor usage is mainly determined by processing FPS and the video stream properties, as encoding is required for recording video when an event occurs. During video recording, CPU usage increases by about 3% per camera. On the hardware used, with four cameras processing at 10 FPS (recording at 20 FPS), average CPU usage is 27%. CPU load can be reduced by applying motion masks, damping motion detector parameters, or lowering the processing FPS.

Since cameras are accessed via a network, bandwidth also limits the maximum number of cameras; average network traffic is 47.8 Mb/s in the current setup.

The key performance metric is the number of skipped frames, which occur when resources are insufficient, typically due to object detection overload. The YOLOv12s model [17] processes an image in under 10 ms, allowing up to 100 image sections to be processed per second across all cameras. The number of image sections that need to be processed per frame and the number of skipped frames depend on environmental activity (e.g., wind), motion masks, processing FPS, and image size. Currently, skipped frames are negligible; additional cameras or busier scenes may require fine-tuning.

The detection accuracy is primarily determined by the pre-trained YOLOv12s model [17]. A low number of false activations was observed during the test period. While larger models may offer higher detection accuracy, they typically require more computational resources and are less suitable for real-time multi-camera processing.

B. Storage Usage Evaluation

To evaluate storage usage, the system was tested for 30 days with the four-camera setup described previously. No zones were defined for testing; the entire image was used for detection, and only areas unlikely to contain relevant objects (e.g., sky, trees, rooftops) were masked for motion detection. The configuration was set so that recording would start only when an object labeled as *person*, *car*, or *cat* appeared with at least 65% confidence. The storage usage for each camera is summarized in Table I, along with the recording count and median recording duration. Over the 30-day period, a total of 10,413 recordings were captured across the four cameras.

TABLE I. Recording Count, Median Duration, and Storage Usage per Camera (Aug. 17 – Sep. 15, 2025)

Camera Name	Recording Count	Median Duration (s)	Storage Used (GB)
Entrance Gate	5,503	9.97	138.78
Front Garage	4,293	4.50	89.24
Rear Garage	319	12.80	8.35
Terrace	298	15.93	12.23
Total	**10,413**	–	**248.60**

Without intelligent processing, approximately 15,206.4 GB of storage would be required for 30 days of continuous recording (5.28 GB/hour per camera). With the proposed software, this was reduced to 248.60 GB, representing a 98.4% reduction. This demonstrates the system's effectiveness in recording only relevant activity.

Table II presents the distribution of object labels that triggered recordings. The Entrance Gate and Front Garage cameras had a more balanced mix of persons and cars. The Rear Garage was predominantly triggered by persons, with cars appearing infrequently, while the Terrace recordings were triggered mainly by persons, with a few instances of cats, as no cars can access this area.

TABLE II. OBJECT LABELS PER CAMERA

Camera Name	Label	Count	Percentage
Entrance Gate	person	3,134	56.95%
Entrance Gate	car	2,342	42.56%
Entrance Gate	cat	27	0.49%
Front Garage	car	2,308	53.76%
Front Garage	person	1,948	45.38%
Front Garage	cat	37	0.86%
Rear Garage	person	290	90.91%
Rear Garage	cat	26	8.15%
Rear Garage	car	3	0.94%
Terrace	person	277	92.95%
Terrace	cat	21	7.05%

IV. CONCLUSION

The developed system demonstrates effective real-time processing of security camera footage with integrated motion detection, object detection, and tracking. The web-based interface provides efficient access to camera feeds and recorded data. Performance evaluation shows that the system maintains 10 FPS per camera with negligible skipped frames, while CPU and memory usage remain low and scalable. Storage usage evaluation indicates that intelligent, selective recording reduces storage usage from approximately 15,206 GB (for continuous recording for 30 days at 5.28 GB/hour per camera) to 248.60 GB, representing a 98.4% reduction, confirming that only relevant activity is captured. Overall, the system successfully achieves the objectives outlined in the Introduction, including real-time multi-camera processing, selective event recording, and user-friendly monitoring.

Future improvements may include searching for a more accurate object tracking algorithm to better handle overlapping objects of the same type and exploring faster and more efficient object detection methods. It would also be possible to integrate a notification mechanism (e.g., email, SMS, push) when a certain type of object appears in a defined zone. Additional functionality could include identifying people using face or gait recognition.

REFERENCES

[1] N. Sulman, T. Sanocki, D. Goldgof, and R. Kasturi, “How effective is human video surveillance performance?,” in 19th International Conference on Pattern Recognition (ICPR 2008), Tampa, FL, USA, IEEE, Dec. 2008, pp. 1–3, doi: 10.1109/ICPR.2008.4761655.

[2] S. W. Ibrahim, “A comprehensive review on intelligent surveillance systems,” in Communications in Science and Technology, vol. 1, no. 1, pp. 7–14, May 2016, doi: 10.21924/cst.1.1.2016.7.

[3] O. M. Olaniyi and S. Ganiyu, “Intelligent video surveillance systems: a survey,” in Balkan Journal of Electrical and Computer Engineering (BAJECE), vol. 11, 2023, doi: 10.17694/bajece.1223050.

[4] S. Manchanda and S. Sharma, “Analysis of computer vision based techniques for motion detection,” in 2016 6th International Conference - Cloud System and Big Data Engineering (Confluence), Noida, India, IEEE, 2016, pp. 445–450, doi: 10.1109/CONFLUENCE.2016.7508161.

[5] I. Berrios, “Introduction to motion detection: part 1,” Medium, [Online]. Available: https://medium.com/@itberrios6/introduction-to-motion-detection-part-1-e031b0bb9bb2. [Accessed: Aug. 18, 2025].

[6] I. Berrios, “Introduction to motion detection: part 2,” Medium, [Online]. Available: https://medium.com/@itberrios6/introduction-to-motion-detection-part-2-6ec3d6b385d4. [Accessed: Aug. 19, 2025].

[7] I. Berrios, “Introduction to motion detection: part 3,” Medium, [Online]. Available: https://medium.com/@itberrios6/introduction-to-motion-detection-part-3-025271f66ef9. [Accessed: Aug. 20, 2025].

[8] X. Zou, “A review of object detection techniques,” in 2019 International Conference on Smart Grid and Electrical Automation (ICSGEA), Xiangtan, China, IEEE, 2019, pp. 251–254, doi: 10.1109/ICSGEA.2019.00065.

[9] Z.-Q. Zhao, P. Zheng, S.-T. Xu, and X. Wu, “Object detection with deep learning: a review,” IEEE Trans. Neural Netw. Learn. Syst., 2019, pp. 3212–3232, doi: 10.1109/TNNLS.2018.2876865.

[10] D. D. Aboyomi and C. Daniel, “A comparative analysis of modern object detection algorithms: YOLO vs. SSD vs. Faster R-CNN,” ITEJ (Information Technology Engineering Journals), vol. 8, no. 2, pp. 96–106, 2023, doi: 10.24235/itej.v8i2.123.

[11] U. Subbiah, D. K. Kumar, S. K. Thangavel, and L. Parameswaran, “An extensive study and comparison of the various approaches to object detection using deep learning,” in 2020 International Conference on Smart Electronics and Communication (ICOSEC), Trichy, India, IEEE, 2020, pp. 183–194, doi: 10.1109/ICOSEC49089.2020.9215321.

[12] EmguCV, “Emgu CV: OpenCV in .NET,” [Online]. Available: https://www.emgu.com/. [Accessed: Aug. 29, 2025].

[13] YoloDotNet, “YoloDotNet: YOLO Object Detection for .NET,” [Online]. Available: https://www.nuget.org/packages/YoloDotNet/. [Accessed: Aug. 29, 2025].

[14] LapjvCSharp, “LapjvCSharp: Jonker–Volgenant Algorithm for .NET,” [Online]. Available: https://www.nuget.org/packages/LapjvCSharp/. [Accessed: Aug. 29, 2025].

[15] SkiaSharp, “SkiaSharp: Cross-Platform 2D Graphics API for .NET,” [Online]. Available: https://github.com/mono/SkiaSharp. [Accessed: Aug. 29, 2025].

[16] cvexplained, “Running average model – background subtraction,” [Online]. Available: https://cvexplained.wordpress.com/2020/04/17/running-average-model-background-subtraction/. [Accessed: Aug. 21, 2025].

[17] Ultralytics, “YOLOv12: You Only Look Once Version 12,” 2024. [Online]. Available: https://docs.ultralytics.com/models/yolo12. [Accessed: Aug. 22, 2025].

[18] S. Roy, “Centroid object tracking,” Medium, [Online]. Available: https://medium.com/@subhajeet.roy/centroid-object-tracking-835d75e4a75a. [Accessed: Aug. 22, 2025].

[19] A. Jaiswal, “A tutorial on centroid tracker & counter system,” [Online]. Available: https://www.analyticsvidhya.com/blog/2022/05/a-tutorial-on-centroid-tracker-counter-system/. [Accessed: Aug. 23, 2025].

[20] R. Jonker and A. Volgenant, “A shortest augmenting path algorithm for dense and sparse linear assignment problems,” Computing, vol. 38, pp. 325–340, 1987, doi: 10.1007/BF02278710.