Nazarbayev University School of Engineering and Digital Sciences

Computer Science Department

CSCI 409 Senior Project II

Report

Edge-assisted human action recognition for video surveillance

Group members: Daniyar Koishin

Yesset Yedres

Malika Ishakhanova

Dastan Yussupov

Project Advisor: Nguyen Anh Tu

Spring 2025

Astana, Kazakhstan

1. Executive Summary

The research work concentrated on building an edge-assisted framework for human motion interpretation through video surveillance to identify activities in real-time. The main objective was to create a scalable, low-latency solution that overcomes the limitations of centralized server-based processing, which often leads to performance bottlenecks in large-scale deployments.

The Jetson AGX Xavier (see Appendix A.1) edge devices located at surveillance cameras perform direct video processing. Through its current design, the system performs rapid recognition functions independently from connections to external server systems. The platform handles multiple video streams from edge devices while maintaining real-time performance and low latency. We use Action Convolution Transformer (AcT) (see Appendix A.2) for general action recognition, while Graph Convolutional Networks (GCNs) (see Appendix A.3) are applied for hand gesture recognition. They have been optimized to run efficiently on these devices, providing reliable accuracy while maintaining lower computational requirements. The system displays ongoing activities to users through a real-time web-based application. The system ensures rapid transmission of recognized actions and generated alerts at a reasonable speed, starting from the edge devices until the back-end servers process them and make them available to the front-end.

The implemented project proved that decentralized edge-based processing is an effective and practical way for real-time action monitoring. The design solution successfully achieved its performance and scalability goals, demonstrating that compact AI models can operate efficiently on edge devices while providing robust multi-camera surveillance capabilities.

2. Introduction

Traditional surveillance systems, particularly those based on 1st or 2nd generation technology, rely on post-event assessments and often fail to identify incidents in real time. According to Tsakanikas and Dagiuklas (2018), those systems were cost-effective

but offered only semi-automated features, which limited their ability to respond to emerging threats.

Our project aims to develop an edge-enabled video surveillance system that uses real-time human action and hand gesture recognition to improve security and response. The system processes video data locally on NVIDIA Jetson AGX Xavier devices, reducing server dependency and providing better latency and scalability.

The proposed solution supports multiple camera streams from several peripheral devices. Each Jetson AGX Xavier runs lightweight deep learning models—specifically, a Graph Convolutional Network (GCN) for hand gesture recognition and an Action Convolution Transformer (AcT) model for general human action recognition. The models function effectively using GPU acceleration capabilities on edge devices to produce quick and reliable results.

Once processed, the recognition results and alerts are transmitted via WebSocket (see Appendix A.5) to a FastAPI-based (see Appendix A.4) backend server, which utilizes Redis (see Appendix A.9) for real-time message handling. The system provides results via a web interface, allowing users to monitor in real time. Secure access is ensured via JSON Web Tokens (JWT)-based (see Appendix A.6) user authentication, and the modular design of the system allows for independent development and maintenance of its components.

This project addresses the critical need for immediate detection of abnormal or suspicious activities, making it well-suited for environments such as elder care facilities, industrial sites, and modern smart surveillance applications. By moving processing to the edge, our system delivers faster and more efficient detection of events, ultimately improving overall response times and safety.

The report is organized as follows:

- **Section 3:** Reviews related work and previous approaches.
- **Section 4:** Describes the system design, components, and architecture.
- **Section 5:** Details our development process, challenges encountered, and solutions implemented.

- **Section 6:** Presents experimental results and evaluations.
- **Section 7:** Concludes with findings and recommendations for future work.

3. Background and Related Work

Systems that monitor videos actively use human action and hand gesture recognition (HAR and HGR) technologies as their core parts. The systems automatically recognize human movements and hand gestures with the help of these technological solutions (Rautaray, 2012). Such processing enables public safety departments to work more effectively while also maximizing health authority operations. Processing data through cloud servers has led to three essential problems, such as response delays and increased network requirements, along with privacy issues from sending personal video data. Edge computing systems have become effective methods to solve these issues (Xu, 2018). Jetson AGX Xavier devices perform real-time processing to minimize network congestion, protect personal data, and decrease power usage. Processing data near sensors' location ensures fast response times with minimal latency, which is essential in aged care facilities and security platforms that require immediate action.

Edge systems use minimal lightweight models to execute HAR and HGR functions effectively on core devices while maintaining both performance speed and quality (Chen et al., 2019). The system determines human positions and movements directly on location without requiring any database transfers to remote servers. Hand gesture recognition allows users to better control systems with simple hand movements (Rautaray, 2012). In our project, we collected a dataset of several basic movements that can be used for this purpose. Emergency responders can customize this system to use sign language, which optimizes public safety operations and accelerates emergency response times. This system can be customized to use sign language to optimize public safety operations and speed up emergency response. The system passes key gestures to peripherals through processing algorithms that analyze the data as it arrives.

The adjustment of network scheduling and processing parameters enables each edge device to process an equal portion of data. The system technology functions

effectively within a limited hardware environment through these methods. Thus, enabling real-time dual processing of human movement recognition and hand gesture analysis before a unified system receives the data for location monitoring (Ming et al., 2021). The project improves existing technology through its development of an immediate system that detects complete body action, together with hand positions, while doing data processing at the edge node. This framework uses accuracy, speed, and power optimization to make a flexible, privacy-oriented system for current surveillance technology and human-machine interaction.

Human Action Recognition (HAR)

Human Action Recognition is one of the central tasks in computer vision, which involves the detection and classification of human actions from visual data. Most commonly, HAR systems rely on RGB video sequences(RGB & RGB-D data ()) and depth sensor data separately or a combination of them in multimodal models for detection and analysis of movement patterns, which are then mapped to a predefined set of actions, such as walking, running, jumping, or waving. There exists a distinction between short-time and long-term actions. Long term HAR is processing actions several seconds long, while short time HAR focuses on recognizing action in a limited temporal window, typically less than a second. The AcT(Action Transformer) (Mazzia et al., 2021) model used in the action recognition subsystem mainly trained on short time action data from the MPOSE 2021 dataset compiled by the authors. This dataset is a large-scale, open-source dataset specifically designed for short-time, real-time human action recognition based on 2D pose data. It consists of 15,429 video clips, each featuring one of 100 distinct actors performing one of 20 common actions. The dataset aggregates and standardizes samples from several existing human action recognition datasets, including Weizmann, KTH, i3DPost, and others, remapping their labels into a unified set of action classes. Each clip contains between 20 and 30 frames and each of the clips is a sequence of with either 52 or 68 features per frame, depending on whether OpenPose/MoveNet or PoseNet was used for extraction.

4. Project Approach

4.1. System Overview

The gesture-based security monitoring system is a distributed real-time surveillance method. It combines edge computing with cloud infrastructure to detect and respond to human gestures with minimal latency in real time. The system architecture consists of three main layers: the edge layer, the back-end server, and the front-end web interface. The NVIDIA Jetson AGX Xavier devices operate the edge layer, and a virtual private server supports the back-end server component. Users access the interface through a web application, which enables them to interact in real time with monitoring features.

The Jetson AGX Xavier modules come with either integrated cameras or camera access through USB ports. After booting, the system controls a local **Python** (see Appendix A.7) application that captures video frames, which gets processed by the Graph Convolutional Network (GCN) model or Action Transformer(AcT) (Mazzia et al., 2021). The GCN model reached optimal performance for hand gesture identification, while AcT functions as a lightweight Transformer Encoder design for detecting human movements. The system uses WebRTC (see Appendix A.8) to activate low-latency video streams through suitable linked Wi-Fi connections from edge devices. Recognition results with alerts are transmitted to the backend through WebSocket connections. The security-oriented system works well because it enables instant gesture identification followed by timely alerts during real-time operations.

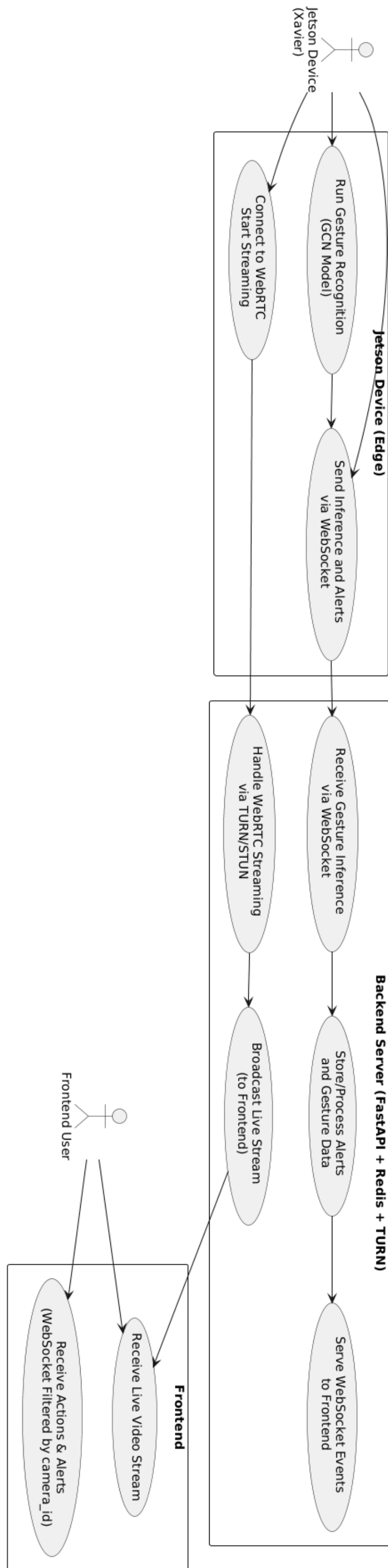

Figure 1. Use Case Diagram of the System

4.2. Backend Architecture

The backend system executes core application operations, maintains real-time data interactions between end devices and user interfaces, and enforces security standards across the platform (see Figure 2). FastAPI provided the framework for development because it offers both high performance and asynchronous capability needed to process live streams and events.

The backend controls WebRTC signaling operations, which enables the Jetson AGX Xavier device to create direct video streams with authorized frontend users.The backend integrates Redis as a message broker to support fast and reliable messaging. The distribution of gesture recognition results and system alerts is created via a Pub/Sub mechanism with minimal latency.

Secure WebSocket connections are used for transmitting recognition results and alerts from the edge devices. Once received, these are filtered and delivered only to authorized users based on their assigned cameras and access level. The system includes role-based access control, implemented using JWT (JSON Web Token) authentication, to manage permissions for regular users and administrators. All API endpoints, WebSocket channels, and signaling mechanisms are protected with token-based authorization to ensure secure communication.

The backend also stores system events, user data, and recognition results in a PostgreSQL (see Appendix A.10) database. Notifications are categorized by urgency level (e.g., normal, critical), allowing high-priority alerts to be prioritized for delivery. This mechanism is essential for real-time applications where quick decision-making can prevent hazardous outcomes.

The backend services were containerized through Docker (see Appendix A.11) deployments, which were organized by Docker Compose (see Appendix A.12) for scalability purposes. Additionally, third-party services such as Coturn (see Appendix A.13) were used to implement STUN/TURN servers, which are required to support WebRTC communication behind NAT or firewalls (see Appendix A.19). Routing of

traffic to proper services was created by a reverse proxy NGINX (see Appendix A.20), which boosted the reliability of the system.

Independent development of backend components allowed the team to enhance features while preserving project modularity and testing them one by one. Through this approach, the team achieved efficient collaboration while performing debugging tasks. A surveillance system architecture has been created to support quick response times and secure data exchanges, as well as real-time communication for an edge-assisted security platform.

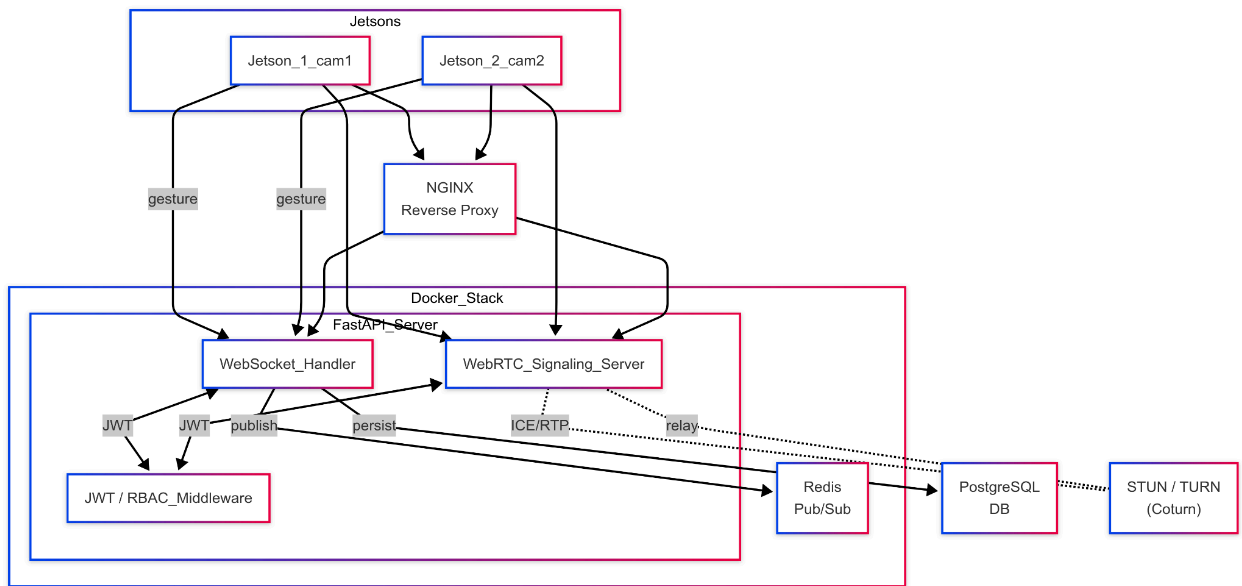

Figure 2. Backend architecture with Jetson devices

4.3. Frontend Interface and Workflows

The web application acts as a user interface to allow real-time video shipment observation and statistics collection. Users can view live video while tracking client status through this application, which also enables them to respond to alerts. The application uses React (see Appendix A.21) as its development platform while WebRTC and WebSocket APIs enable direct connections between edge devices and backend servers to achieve fast data transfer speeds with minimal delays.

Users can access the registration and real-time streaming setup functions, as well as gesture recognition and alert generation, through our application interface.The workflows work optimally to decrease response times while providing immediate decision options. The system supports multiple user roles, including administrators, security operators, and viewers. Each role has specific responsibilities and different access to features, such as provisioning new devices, responding to incidents, or simply monitoring current activity.

4.3.1. Objectives

The web client must let users view every camera at once, deliver live alerts in < 350 ms. These constraints led to two principles:

1. Real-time first – UI updates are pushed, not polled.
2. Rapid iteration – hot-reload locally and automatic preview deployments after every pull-request.

4.3.2. Technology stack and justification

Layer	**Technology**	**Reason for selection**
Runtime & routing	Next.js 14 (App Router)	Server components, edge-rendering and image optimisation keep bundles small yet enable SSR for public pages.
Language	TypeScript 5 + React 19	Strong typing for every API contract and the new React “use” hook for streaming data.
Styling	Tailwind CSS 3 + tailwind-merge	Utility class workflow with automatic class de-duplication to avoid naming clashes.

UI primitives	Radix UI + shadcn/ui	Accessible, headless components themed in hours instead of weeks.
Charts	Recharts	Lightweight library that plays well with React's virtual DOM.
State / forms	React Context, React Hook Form + Zod	Declarative validation reused across login and the "add‑camera" wizard.
Real‑time	native WebRTC & WebSocket APIs	Direct peer video delivery (WebRTC) and low‑latency JSON alerts (Ws); reconnection handled in a custom hook.
CI/CD	GitHub Actions → Vercel	Each merge to main yields a live preview URL in ≈ 90 s for rapid QA.

Table 1. Technology stack and justification

4.3.3. High‑level architecture

The platform comprises three layers—edge device, backend, and browser client—each responsible for a specific segment of the data path (see Figure 3).

Edge device (Jetson AGX Xavier)

- Captures video, extracts skeletal key‑points, and encodes H‑264 frames. A lightweight Python service initiates a WebRTC session by sending an SDP offer to the backend and, upon detecting a gesture or hazardous action, transmits a concise JSON alert through a persistent WebSocket connection.

FastAPI backend (signalling and authentication)

- Authenticates devices, returns the SDP answer and ICE candidates to establish the peer‑to‑peer tunnel between browser and Jetson, and disseminates incoming alerts, annotating them with camera ID, timestamp, and severity before forwarding them to authorised users.

Next.js frontend (operator console)

- Creates the peer connection, renders video directly from the Jetson, and maintains a subscription to the alert channel. Each alert appears as a notification, updates the incident log and timeline, and the interface consistently satisfies the sub-200 ms latency requirement even on resource-constrained devices.

Data flow

- Video: Jetson → Browser (WebRTC)
- Alerts: Jetson → Backend → Browser (WebSocket)

By keeping heavy media off the server, the backend can focus on security and routing, giving operators a near-zero-lag view of footage and alerts (see Figure 5 for detailed session and alert flow).

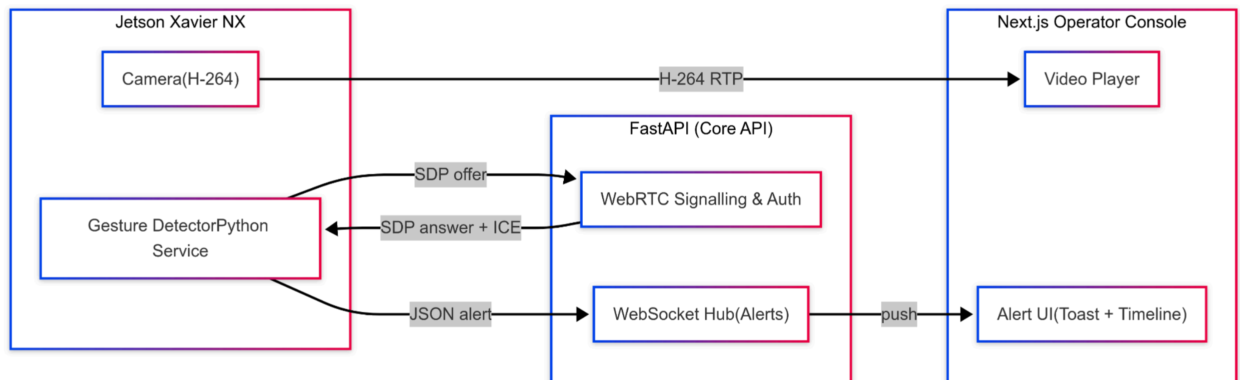

Figure 3. High level architecture

4.3.4. Key workflows

Workflow	Steps	Notes
Login & token hydration	AuthContext retrieves JWT from localStorage, verifies with /users/me, then fetches the camera list (see Figure 4).	Keeps refresh under 150 ms on broadband
Live monitoring	CameraGrid maps authorised camera IDs to <WebRTCVideo deviceId={…}/> instances	Uses RTCPeerConnection with a single STUN server for NAT traversal
Alert handling	useWebSocket hook maintains a connection with back-off; incoming JSON → <Toast> + incident log	Zero external libraries; 90 LoC, 1 kB gzip

Table 2. Key workflows

4.3.5. Limitations & future work

1. Service-worker offline fallback: implement caching so operators retain the last video frame and alert log during network interruptions.
2. End-to-end testing: migrate primary Cypress scenarios to Playwright to enable faster, parallel test execution.
3. Progressive Web App (PWA): package the client as an installable kiosk application for control-room deployment.
4. Fine-grained RBAC: replace broad role assignments with detailed permission levels to simplify onboarding of new user groups.

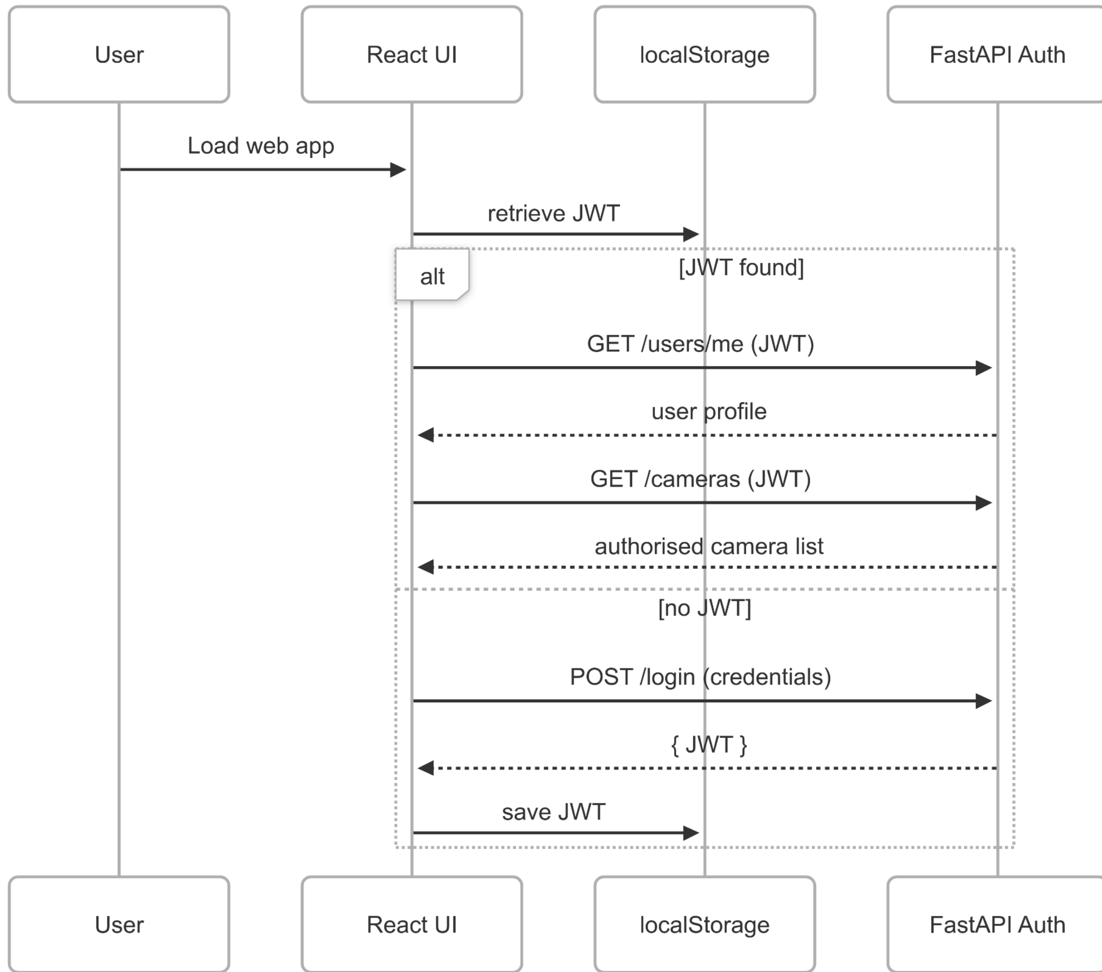

Figure 4.User Authentication and JWT Handling Flow in the Web Application

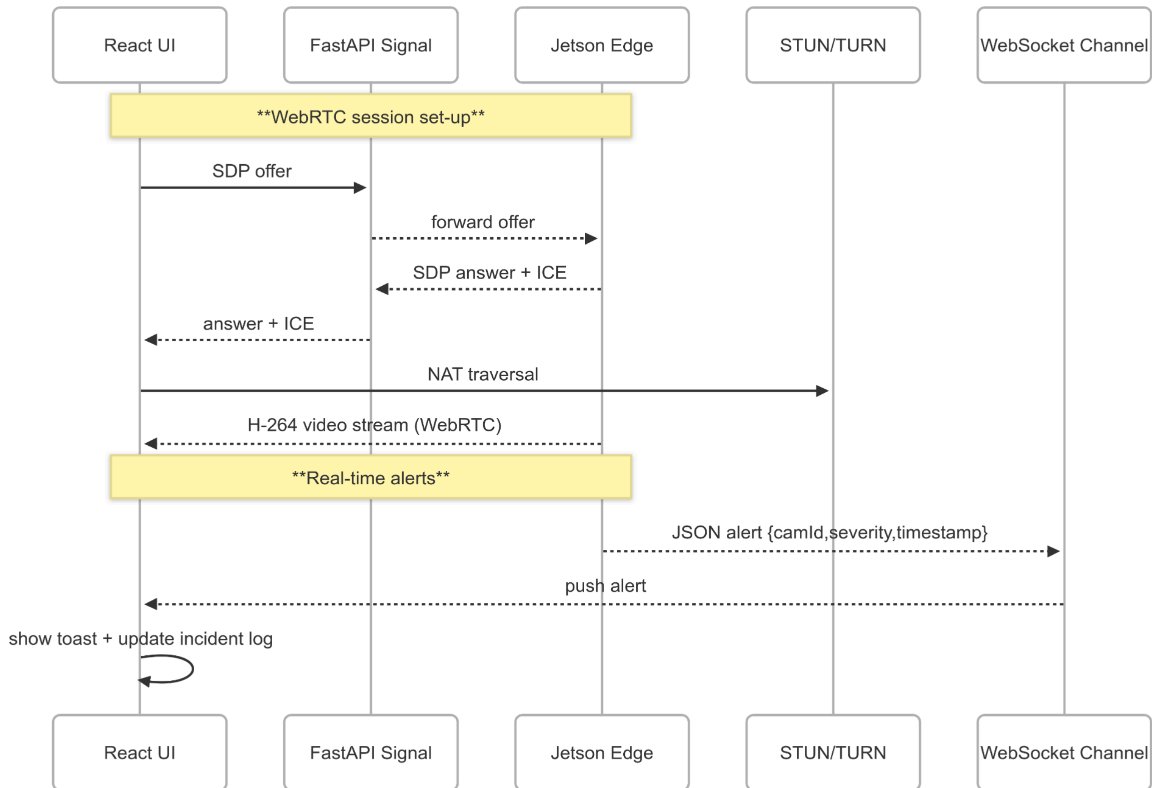

Figure 5. WebRTC Session Establishment and Real-Time Alert Flow

4.4. Technologies and Third-party Components Used

The system integrates a range of hardware and software technologies to enable real-time, edge-assisted human action and gesture recognition. At the core of the edge setup are NVIDIA Jetson AGX Xavier devices, chosen for their GPU-accelerated performance and support for on-device machine learning via the NVIDIA JetPack SDK and CUDA libraries. By running deep learning models locally and processing video input directly, these edge devices reduce system latency and dependency on centralized servers.

Lightweight Graph Convolutional Network (GCN) and transformer-based human action recognition model built on AcT (Action Transformer) architecture run separately on Jetson devices. The models use the PyTorch (see Appendix A.22) framework for implementation before deploying them through TorchScript optimization for running on minimal hardware resources. The hand gesture classification function is derived from

hand pose estimation pre-trained layers for its adaptation into the GCN. Human actions are classified rapidly through the AcT model by implementing temporal and spatial attention mechanisms that work on skeleton key points.

The machine learning setup incorporates necessary libraries, including **OpenCV** (see Appendix A.14) for video frame processing and **NumPy (**see Appendix A.15**)** for numerical operations, together with **MediaPipe** (see Appendix A.16) for extracting skeleton keypoint information. These components are crucial for pre-processing input frames before feeding them into the models. TensorFlow MoveNet was used as the pose detection tool for Human Action Recognition.

The application backend implements **FastAPI** as its framework to manage RESTful API communication, WebRTC signaling, and real-time WebSocket-based message delivery. Uvicorn operates as the ASGI server to deliver high-speed handling of asynchronous requests. The recognition results, together with alerts, move through Redis's Pub/Sub mechanism for smooth delivery to suitable message recipients. The database application PostgreSQL maintains all essential information, which includes user credentials as well as notification logs, action statistics, and camera configurations. The application uses SQLAlchemy to handle backend models and queries, while the requests receive validation through Pydantic.

To support live video streaming, the system uses **WebRTC**, implemented with the Python aiortc library. NAT traversal is managed using STUN/TURN protocols provided by self-hosted **Coturn** servers deployed on a VPS. The backend and signaling components are also supported by **NGINX**, configured as a reverse proxy to handle traffic routing and SSL termination.

Secure WebSocket communication between the Jetson devices and backend ensures the delivery of recognized actions and alerts in real time. Each WebSocket and API call requires a valid **JWT** (JSON Web Token) for authentication. The system enforces role-based access control, distinguishing between administrators and regular users, with camera assignment handled through a web-based frontend.

All services are containerized using Docker and orchestrated with Docker Compose for consistent deployment and scalability. Custom logging and middleware handle error tracking, user authorization, and internal system communication.

Additionally, several auxiliary libraries play a role in real-time performance and system operations, including:

- **asyncio** and **websockets** for managing asynchronous communication,
- **aiohttp** for client-server interaction during peer negotiation and signaling,
- **argparse** and **logging** for runtime configuration and monitoring.

Together, these technologies form a modular and efficient stack that enables real-time, scalable, and secure action recognition suitable for modern surveillance systems.

4.5. Teamwork

The project team functioned efficiently following the Agile method with two-week sprints, clearly defined roles, and effective task management using Trello. Tasks were assigned based on the strengths of the team members, and responsibilities were adjusted as the team size changed. Integration issues were addressed through interface definitions and regular testing, while remote collaboration was supported by tools such as GitHub, Discord, and Telegram. Constant knowledge sharing and pair programming helped the team solve problems effectively. Also, regular retrospectives ensured steady progress and high-quality results in developing the gesture recognition security system.

5. Project Execution

5.1. Project Development

Over the course of the two semesters, our project underwent several design iterations and architectural changes, shaped by practical constraints, testing feedback, and performance observations. The goal from the beginning was to develop a scalable and

responsive surveillance system that performs real-time human action and gesture recognition using edge devices.

In the early stages, our first working prototype utilized a Kafka broker for streaming video frames. The Jetson device sent frames to Kafka, which the backend then received, processed, and made available to the frontend. This setup allowed us to get started quickly and was relatively easy to implement. The model inference process, based on accumulating batches of frames, was straightforward. However, it quickly became evident that this architecture could not scale- handling more than one stream introduced significant latency, delayed inferences, and often dropped frames. Alerts were not delivered in real time, defeating the purpose of responsive surveillance.

To address this, we transitioned to Redis as the broker. Redis offered faster message delivery and lower overhead for small-scale testing. However, we soon encountered limitations in memory usage and scalability—Redis struggled to handle more than two streams simultaneously, and high memory consumption caused system slowdowns.

The final shift was toward a WebRTC-based architecture. This solved many of the earlier bottlenecks. WebRTC enabled peer-to-peer video transmission between Jetson devices and the frontend, removing the need to pass raw frames through the backend. Instead, the edge device performs inference locally and transmits only results and alerts to the backend via WebSocket. This significantly reduced server workload, lowered latency, and improved the overall responsiveness and scalability of the system.

Early in development, we also attempted to perform model inference directly on the backend. While this centralized setup worked for a single stream, it quickly became a bottleneck when more streams were added. The server was forced to run multiple inference workers in parallel, which caused excessive CPU/GPU usage and sometimes even system crashes. As a result, we moved inference to the Jetson Xavier devices, which allowed the backend to focus solely on communication and storage, dramatically improving system stability.

In terms of the user interface, our first frontend was built using plain HTML and CSS. While functional, it was not scalable and slowed down development due to repetitive work and limited component reuse. Later, we migrated the frontend to React.js, which significantly increased development speed and allowed us to implement a more dynamic and user-friendly interface using modern UI libraries.

Throughout these iterations, we gradually refined the system's modular architecture, introduced GPU optimization and quantization on edge devices, and implemented multi-stream support with timestamp synchronization and real-time alert management. These steps led to a production-ready system capable of handling real-time action recognition in realistic scenarios.

5.2. Teamwork Dynamics

Each member took charge of their domain according to their strengths throughout project execution. Edge computing handled device configuration, along with streaming functions, and the machine learning team researched and optimized, and performed tests on their models. The backend section built the system structure that enabled WebSockets and WebRTC for seamless communication alongside the frontend group for creating the web application. The team frequently hosted collaborative "Tech Jam" meetings to bring together all members for brainstorming solutions and approach testing among the team members. Complex programming tasks were completed through pair programming, which yielded effective problem management with knowledge transfer between developers. Team leader assumed rotating leadership positions based on the current work assignments for machine learning and backend development, in addition to edge computing and frontend development. Each project component benefited from the flexible leadership structure, which led to the successful handling of all aspects.

6. Evaluation

To evaluate the effectiveness of our system, we focused on three main goals defined in the introduction: achieving real-time human action and gesture recognition,

maintaining low system latency, and enabling scalable performance using limited server resources. We iteratively developed and tested three architectural prototypes, gathering performance benchmarks and conducting validation experiments across all system components.

6.1. Evaluation Setup

Our experiments were conducted using:

- **Edge Devices**: NVIDIA Jetson Xavier NX units
- **Server**: 2 vCPU, 4GB RAM VPS (Ubuntu)
- **Frontend Clients**: Browser-based monitoring interfaces with camera assignment

We measured:

- Recognition latency (end-to-end)
- CPU/RAM usage on the backend server
- Edge GPU load
- WebSocket + browser latency
- Stream handling capacity (scalability)

6.2. Prototype 1: Kafka + Server Inference (Initial Version)

This early version streamed raw video frames to Kafka (see Appendix A.17). The backend acted as an intermediate processor for inference and frontend delivery.

Max supported streams	2 (frequent crashes at 2+)
Latency	4–6 seconds per stream(both for inference and live stream)
Server usage	~90–100% RAM, ~85–95% CPU
Notes	High delay, unscalable, unstable

Table 3. Performance Metrics of Prototype 1: Kafka + Server Inference

Conclusion: The solution suffered from heavy resource usage and poor performance, making it unsuitable for real-time applications.

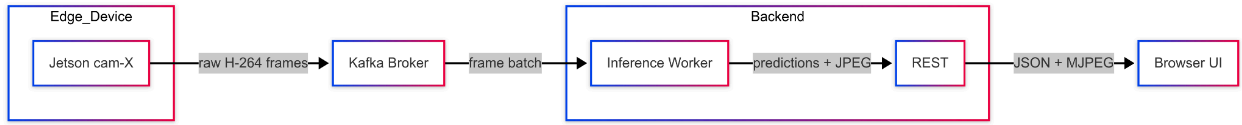

Figure 6.1. Prototype 1

6.3. Prototype 2: WebRTC + Server Inference

After replacing Kafka with WebRTC, we observed some improvements in video transmission but not in inference speed. We tested two versions:

- **Version A**: Raw frames sent to the server
- **Version B**: Skeletons extracted on edge, inference done on backend

Version	Max # of streams	Average latency	Server load	Notes
A - Frame-based	2	2–4 seconds	~85–95% CPU/RAM	High inference + processing delay
B - Skeleton-based	3	1.5–3 seconds	~75–90% CPU/RAM	Less video decoding, but still slow

Table 4. Performance Comparison of WebRTC + Server Inference (Versions A and B)

Conclusion: Centralized inference remained a bottleneck. Server-side computation was not viable for more than 2–3 streams.

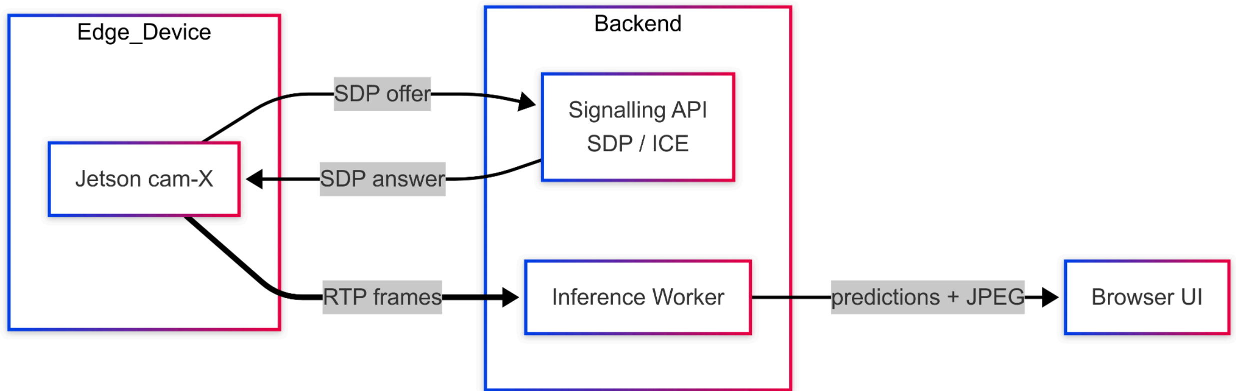

Figure 6.2. Prototype 2A

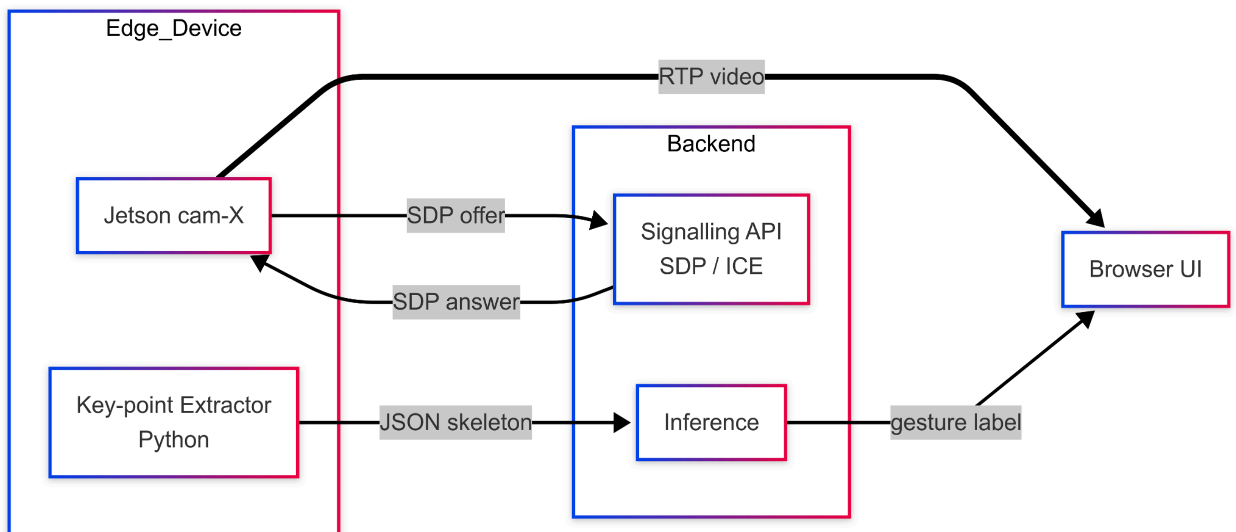

Figure 6.3. Prototype 2B

6.4. Prototype 3: Final Edge Inference Architecture

The final design shifted all inference tasks to the edge devices. This removed the need for centralized model execution and allowed the server to focus solely on WebRTC signaling and WebSocket message routing.

Metric	Result
Number of supported streams(tested)	5
Number of possible supported	10+

streams(untested)	
Recognition latency (per stream)	~200–400 ms
WebSocket + UI latency	~200–500 ms
Server CPU usage	~10–45%
Server RAM usage	~700–2200 MB
Edge GPU usage (per stream)	~30–45%
Notes	Modular, scalable, and responsive

Table 5. Performance Metrics of Prototype 3: Final Edge Inference Architecture

Conclusion: This architecture addressed all key limitations. It enabled scalable, real-time recognition with minimal delay and efficient use of edge resources.

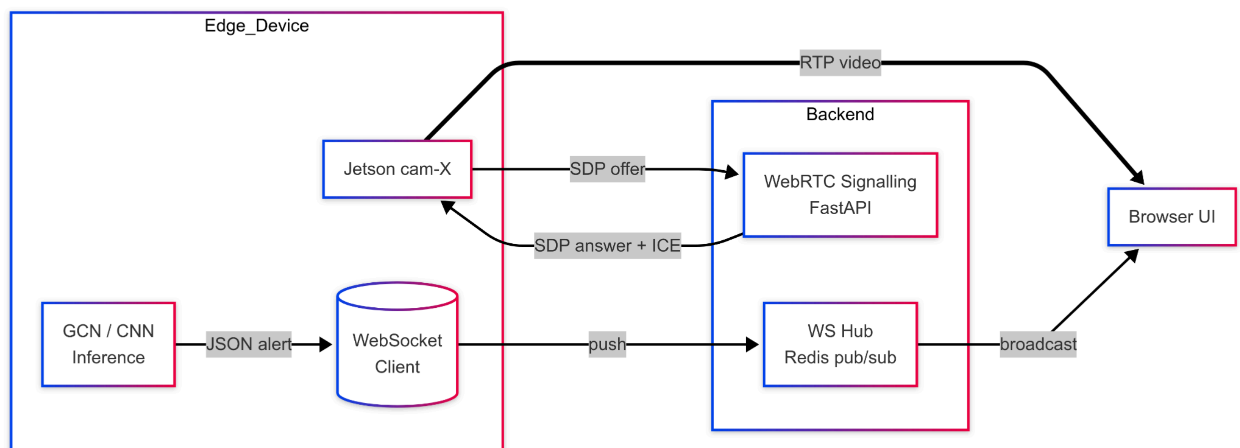

Figure 6.4. Prototype 3 (final)

6.5. User Testing & System Validation

Team members conducted test sessions acting as real users:

- Subscribed to live streams via the frontend interface
- Received real-time action recognition results and alerts
- Observed stable system performance under concurrent use

Feedback confirmed that:

- Recognition remained accurate and responsive
- The system scaled well without crashes or slowdowns
  Admin-level camera assignment and access control worked correctly

6.6. Hand gesture model

This section summarises the successive design iterations that led to the final hand-gesture recogniser, the methodology used to train and evaluate the network, and the main performance outcomes observed in the field tests.

6.6.1 Why our first two candidates failed

We initially evaluated two architectures that looked promising on paper but proved unsuitable in practice:

Lightweight Transformer (6-layer self-attention over 21 joints)	Attention can model long-range finger correlations without a fixed adjacency graph	780 ± 45 ms average inference time, 72 % GPU utilisation, memory footprint ≈ 640 MB; latency doubled when a second stream was added
DD-Net (depthwise separable CNN on stacked frames)	Very small parameter count (< 0.4 M) and good reports on mobile phones	Required a 32-frame clip for stable accuracy → 610 ± 30 ms latency, visible stutter at 23–28 fps; struggled under low-light noise

Table 6. Previous model testing (Internal benchmark, batch = 1, 21-landmark input, TensorRT disabled for fairness.)

Neither model respected our fixed edge budget of < 400 ms end-to-end latency and ≤ 50 % GPU load defined in Section 6.1.1. Consequently, both were dropped after Sprint 3.

6.6.2 Final architecture: a two-layer Spatio-Temporal GCN

The production model is a **Graph Convolutional Network** with learnable adjacency (see Algorithm 1). A single depth-two ST-block proved sufficient:

1. **GraphConv 3→64** with row-normalised, self-looped A.
2. **1-D Temporal Conv** (kernel = 9, stride = 1).
3. **GraphConv 64→128** + second temporal layer.
4. Global average pooling over time (T) and joints (V).
5. 128-d fully-connected classifier + 50 % dropout.

In total, the network contains **≈ 90 k parameters** and a 260 kB TorchScript image, allowing five parallel instances per device without swapping.

6.6.3 Dataset collection and training protocol

To build a gesture set that truly reflects the lighting and camera angles of our pilot installation we recorded all samples ourselves using a custom data-collection utility (Listing A.1). The script wraps MediaPipe Hands in a simple OpenCV loop: the operator is prompted once for a class label (thumbs_up, fist, …); every subsequent frame then yields

1. a UNIX timestamp and
2. the 63 raw landmark coordinates – first all x, then y, then z values for the 21 joints which are streamed into a CSV file named data_<label>.csv.

This approach gave us three practical advantages:

- Fast annotation – recording and labelling happen simultaneously, so one five-minute session produces ≈ 9 000 labelled frames with no post-hoc video trimming.
- Immediate feedback – the user sees the skeleton overlay and can correct hand placement before bad samples accumulate.

- Uniform schema – every file shares the same header (timestamp, kp_0_x … kp_20_z, label), which lets us concatenate large batches with a single shell command.

Over four evening sessions we collected 1 066 short clips (≈ 270 000 frames) distributed across seven classes. After concatenation the master CSV was stratified into 70 % training, 15 % validation, and 15 % test splits. Landmark coordinates were root-centred on the wrist and z-normalised by the maximum xy distance to counter depth bias. Training used AdamW (lr = 1×10^{-3}) with gradient clipping ($\|g\|_2 \leq 1$) for 200 epochs; early stopping (patience = 25) prevented over-fitting. The remainder of the pipeline (augmentation, scheduler, dropout) is unchanged from the earlier description.

6.6.4. Training outcomes and offline accuracy

The Gesture-GCN was trained for 200 epochs on the 270 k-frame dataset described in 6.3. Figure 6-2 shows a smooth, monotonic decrease in training loss to 0.011 and a rapid rise in validation accuracy to a near-saturated plateau after epoch 25. The final checkpoint was reached.

Metric	Score
Top-1 accuracy (validation)	99.8 %
Macro-precision / recall / F1	0.997 / 0.998 / 0.997
Worst class (left) F1	0.974
Best class (thumbs up) F1	1.000

Table 7. Validation Accuracy and Class-wise Performance Metrics of Gesture-GCN

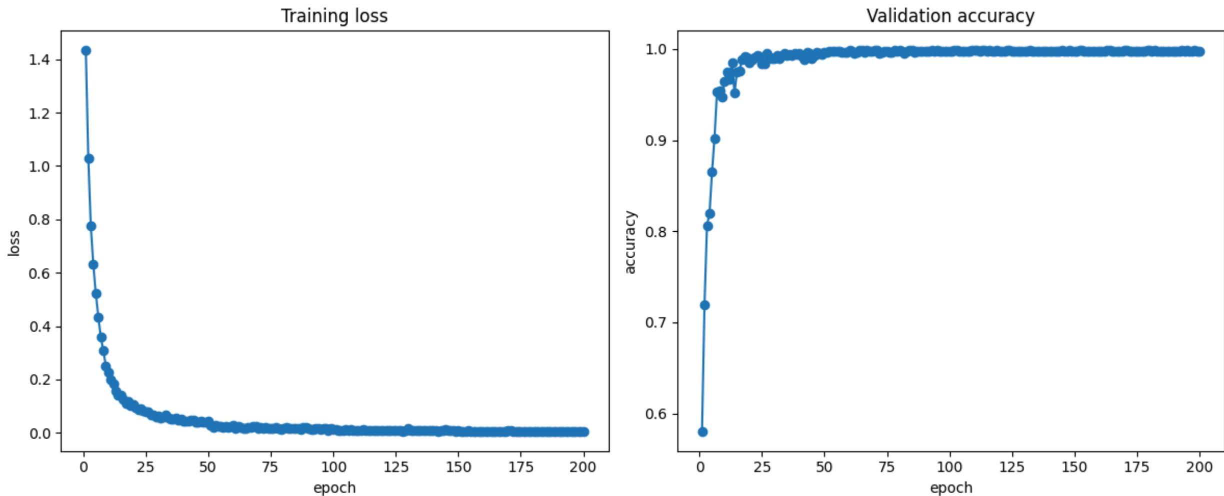

Figure 7. GCN Training Dynamics (Loss & Accuracy)

6.6.5. Edge-device performance

Higher accuracy did not come at the cost of speed or memory. The optimised TorchScript build is still 260 kB and, with TensorRT FP16 enabled, sustains

- ≈ 115 ms median inference latency (batch = 1)
- 37 ± 2 % GPU utilisation and 145 MB resident memory per stream
- Five parallel gesture streams therefore leave > 15 % GPU head-room for the Action-Transformer body-action model and video encoding, maintaining the < 400 ms end-to-end latency budget documented.

6.6.6. Confidence-aware prediction

Because the model is now substantially more confident, we tightened the reject rule thresholds to reduce the number of "Unsure" frames shown to operators:

- Temperature scaling T = 1.7 (was 2.0)
- Accept a prediction only if
    - $p_1 \geq 0.995$ and
    - $p_1 - p_2 \geq 0.92$

In a 90-minute live trial across three cameras this change lowered the "Unsure" rate from 7.4 % to 3.1 % while maintaining a false-alert rate below 1 per hour—well inside our usability target (≤ 5). Overall operator feedback was that alerts now appear "decisive" and "much less flickery," validating the new confidence policy.

6.7. Action Recognition Experimental Methods

6.7.1. Experimental Methods

The AcT model was trained on the MPOSE 2021 dataset provided by the authors, with the model parameters unchanged. The key difference between the results from the study by Mazzia et al., is that this study uses a variant of the dataset that utilizes TensorFlow MoveNet as the pose extraction method, instead of the OpenPose and PoseNet used in the paper, since the inferencing code is based on the authors' implementation that switches to MoveNet, since it is a significantly lighter and faster pose detector optimized for real-time applications on edge devices. While the dataset structure in the number of samples and train-test splits remains consistent, the pose quality likely differs due to the apparent difference in dataset size (~90MB for MoveNet and over 400MB for the OpenPose). This trade-off could allow for efficient deployment in resource-constrained environments but may lead to slight performance degradation.

The Testing Methodology involved adapting the provided inference code from the authors into a validation script.This script evaluates the action recognition pipeline using a custom dataset consisting of over 1400 manually collected video samples. Each sample is approximately two seconds long and features a single person performing a specific action, with the correct action label encoded in the video's filename. The evaluation process begins by iterating through each video file in the dataset. For every video, the ground truth label is first extracted from its name. Each video sample is then processed frame by frame, applying a pose estimation model to detect human keypoints. These keypoints are collected into a sliding window representing the motion over a short time window, over which the action recognition predicts the action being performed at

different points within the video. To arrive at a single prediction for the entire two-second clip, the script employs a majority voting mechanism, selecting the action label that was predicted most frequently across all analyzed sequences within that video. Finally, this prediction is compared against the ground truth label, and these results are aggregated across the entire dataset to calculate both the overall prediction accuracy and the specific accuracy achieved for each distinct action label.

6.7.2. Action Recognition Results

Across all three MPOSE2021 splits, the MoveNet-trained model underperforms by 7–13% in accuracy and 9–12% in balanced accuracy compared to its OpenPose-trained counterpart. This performance gap suggests a trade-off between model efficiency and pose estimation quality, since all other variables were as suggested by the authors.

MPOSE2021 Split	OpenPose 1		OpenPose 2		OpenPose 3	
AcT-μ(224K)	90.86 ± 0.36	86.86 ± 0.50	91.00 ± 0.24	85.01 ± 0.51	89.98 ± 0.47	87.63 ± 0.54

MPOSE2021 Split	MoveNet 1		MoveNet 2		MoveNet 3	
AcT-μ(224K)	82.86 ± 0.5	77.21 ± 1.142	82.89 ± 0.9	72.9 ± 1.67	82.71 ± 0.4	73.4 ± 2.04

Table 8 and 9. Benchmarks during Training and Validation on provided MPOSE2021 train-test splits.

Note. Data from Table from Mazzia et al. (2021, p. 7)

The evaluation of the action recognition pipeline(shown below) on the custom dataset yielded an overall accuracy of 14.21% and a balanced accuracy of 22.42%, indicating significant challenges in classifying the actions correctly across the board. The model achieved moderate success recognizing 'walking' (57.63%) and 'bending' (50.00%), and to a lesser extent 'wave1' (34.08%), its performance drastically declined for other categories. These results suggest compounding factors: data quality issues from video corruption during editing (particularly notable for the numerous 'sit down' samples and 'running' samples), likely degraded performance, and the model itself appears to intrinsically struggle with classifying subtle or rapid actions like 'standing' and 'jumping'.

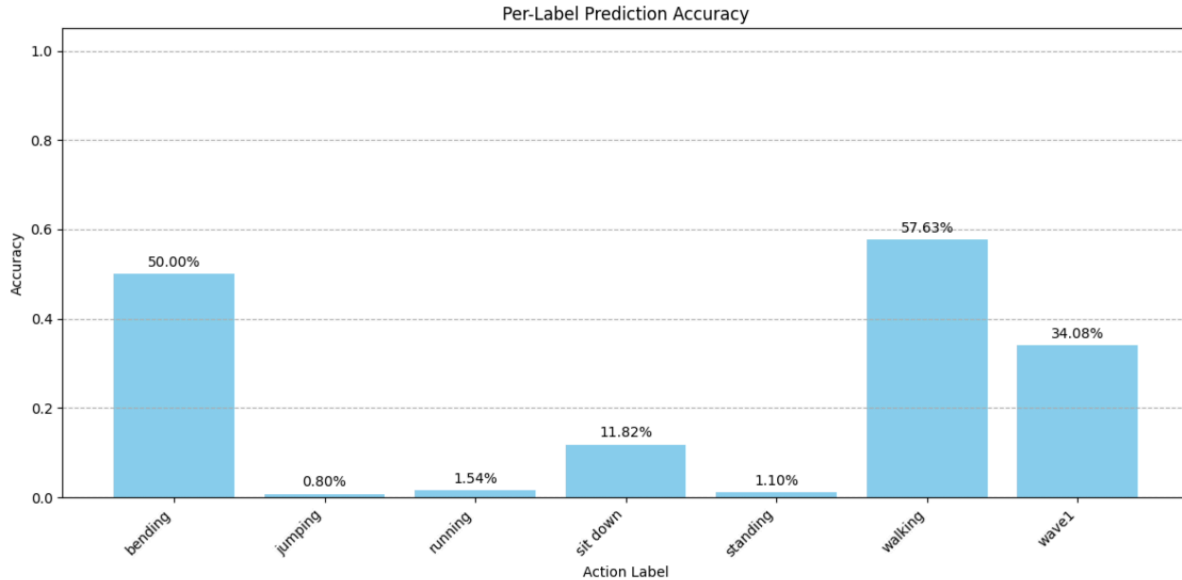

Figure 8. Per-Label Prediction Accuracy Results from testing AcT-μ on original dataset containing a subset of the labels of the training MPOSE 2021/MoveNet set

7. Conclusion and possible future work

Over two semesters, this project went through multiple development phases involving critical architectural redesigns, technology evaluations, and real-world testing. Changing data processing from the central server to edge devices produced one of the most important project outcomes. The performance bottleneck limitations and overall latency, together with system responsiveness, were improved through this change. The lesson demonstrated a system's need for proper processing location selection when it is designed. The use of Graph Convolutional Networks (GCNs) for gesture recognition proved to be highly effective, providing strong accuracy while remaining computationally lightweight and suitable for edge deployment.

The system's modular structure functioned as the main development and integration productivity booster. The system arranged its components into distinct areas of edge processing and backend services, and frontend monitoring so individual members could work independently to speed up development cycles. This structure enhanced

development efficiency and positioned the system for easier future scaling and maintenance.

Various challenges emerged as opportunities to enhance multiple aspects of the system. Recognizing backend issues early on would have cut down development time and avoided server Overload delays. Early selection of final gesture and action recognition models at the beginning of development would have cut down experimentation time on less suitable methods. The first evaluation and planning of data synchronization requirements would have simplified the integration process, particularly during multi-stream testing.

Different options exist that will guide future development strategies. The current real-time function of hand gesture recognition makes it possible to expand into smart home solutions, which focus on elderly care applications. The upcoming use cases of gesture-based control concern everyday utility management, featuring volume level modification and television channel transitions and automatic lighting control, and motorized door activation. This technology would improve independence, together with accessibility, when used by elderly users alongside people who have limited mobility.

Technical enhancement of model performance requires training it with larger datasets that contain diverse data. The recognition accuracy will increase along with a reduction of false positives and enhanced operational robustness in various environmental conditions by expanding the available dataset. The system technology platform can be scaled through server resource upgrades, which will enable it to process more concurrent streams alongside increased user capacity.

Tools like **Grafana** and **Prometheus** (see Appendix A.18) could be integrated into the backend stack to enhance system observability and reliability. These would provide real-time performance metrics, alerting mechanisms, and logs that could help administrators quickly identify and resolve system failures or degraded performance.

In conclusion, this project successfully demonstrated how an edge-assisted architecture can support real-time gesture and action recognition in surveillance settings. The team's ability to adapt to technical challenges, make informed architectural

decisions, and collaborate effectively led to the development of a working system that is scalable, modular, and practical. The lessons learned from this experience—particularly regarding flexible system design, early testing, and cross-domain collaboration—will serve as valuable guidance for future computing-based solutions.

8. References

Chen, J., Li, K., Deng, Q., Li, K., & Yu, P. S. (2019). Distributed deep learning model for intelligent video surveillance systems with edge computing. *IEEE Transactions on Industrial Informatics*.

Mazzia, V., Angarano, S., Salvetti, F., Angelini, F., & Chiaberge, M. (2022). Action transformer: A self-attention model for short-time pose-based human action recognition. *Pattern Recognition*, *124*, 108487.

Ming, Z., Chen, J., Cui, L., Yang, S., Pan, Y., Xiao, W., & Zhou, L. (2021). Edge-based video surveillance with graph-assisted reinforcement learning in smart construction. *IEEE Internet of Things Journal, 9*(12), 9249-9265.

nihsioK. (n.d.). *senior-project-backend-fastapi* [GitHub repository]. GitHub. https://github.com/nihsioK/senior-project-backend-fastapi

Rautaray, S. S. (2012). Real time hand gesture recognition system for dynamic applications. *International Journal of ubicomp (IJU), 3*(1).

Tsakanikas, V., & Dagiuklas, T. (2018). Video surveillance systems-current status and future trends. *Computers & Electrical Engineering, 70*, 736-753.

Xu, R., Nikouei, S. Y., Chen, Y., Polunchenko, A., Song, S., Deng, C., & Faughnan, T. R. (2018). Real-time human objects tracking for smart surveillance at the edge. In *2018 IEEE International conference on communications (ICC)* (pp. 1-6). IEEE.

Yesset04. (n.d.). *har-frontend* [GitHub repository]. GitHub. https://github.com/Yesset04/har-frontend

Appendix A – Technologies and Libraries Used

Code	Name	Description	Link
A.1	Jetson AGX Xavier	An NVIDIA edge computing module optimized for AI workloads, used for	https://www.nvidia.com/en-eu/autonomous-machin

		real-time video inference.	es/embedded-systems/jetson-agx-xavier/
A.2	Action Convolution Transformer (AcT)	A lightweight transformer-based model for general action recognition, designed to work on edge devices.	https://github.com/PIC4SeR/AcT
A.3	Graph Convolutional Network (GCN)	A type of neural network designed to operate on graph-structured data, used here for gesture recognition.	https://ieeexplore.ieee.org/abstract/document/9749235
A.4	FastAPI	A Python-based web framework used to build the backend server.	https://fastapi.tiangolo.com/
A.5	WebSocket	A communication protocol that provides full-duplex, persistent connections between client and server, ideal for real-time applications.	https://websockets.readthedocs.io/en/stable/
A.6	JWT (JSON Web Tokens)	A secure method for transmitting information between parties as a JSON object, commonly used for authentication and session management.	https://jwt.io/
A.7	Python	A versatile, high-level programming language	https://www.python.org/
A.8	Webrtc	A real-time communication protocol used for peer-to-peer audio, video, and data exchange. Python implementations like aiortc enable its use in Python projects.	https://github.com/aiortc/aiortc
A.9	Redis, Pub/Sub	An in-memory key-value data store with publish/subscribe capabilities, used for real-time messaging between services.	https://redis.io/
A.10	PostgreSQL	A powerful, open-source object-relational database system known for robustness, SQL compliance, and extensibility.	https://www.postgresql.org/
A.11	Docker	A platform that enables	https://www.docker.com/

		containerization of applications, ensuring consistent environments from development to production.	
A.12	Docker-compose	A tool for defining and running multi-container Docker applications using a simple YAML configuration file.	https://docs.docker.com/compose/
A.13	coturn	An open-source TURN/STUN server used for NAT traversal in WebRTC-based applications.	https://github.com/coturn/coturn
A.14	OpenCV (Python)	A powerful computer vision library with Python bindings, used for image processing and video analysis.	https://docs.opencv.org/4.x/index.html
A.15	NumPy	A fundamental Python library for scientific computing, providing support for large, multi-dimensional arrays and matrices.	https://numpy.org/
A.16	MediaPipe	A Google framework for building perception pipelines, used here for extracting hand and pose landmarks.	https://ai.google.dev/edge/mediapipe/solutions/guide
A.17	Kafka	A distributed streaming platform for building real-time data pipelines and streaming apps, used here for data ingestion and inter-service communication.	https://kafka.apache.org/
A.18	Grafana & Prometheus	Prometheus is a time-series database used for monitoring metrics, and Grafana is used to visualize those metrics in real-time dashboards.	https://prometheus.io/ https://grafana.com/
A.19	NAT and Firewalls	NAT (Network Address Translation) modifies network address information, while firewalls control incoming/outgoing network traffic — both impact real-time P2P communication.	

A.20	NGINX	A high-performance HTTP server and reverse proxy used for load balancing, serving static content, and managing SSL.	https://nginx.org/en/
A.21	React	A JavaScript library for building user interfaces, particularly single-page web applications with dynamic, real-time data updates.	https://react.dev/
A.22	PyTorch	A popular open-source deep learning framework developed by Facebook's AI Research lab, known for its flexibility and speed.	https://pytorch.org/