Health Monitoring and Emergency Report System in smart homes using Foundation Models

Alim Tleuliyev
dept. of Computer Science
Nazarbayev University
Astana, Kazakhstan
alim.tleuliyev@nu.edu.kz

Alisher Kalabayev
dept. of Computer Science
Nazarbayev University
Astana, Kazakhstan
alisher.kalabayev@nu.edu.kz

Batyr Bodaubay
dept. of Computer Science
Nazarbayev University
Astana, Kazakhstan
batyr.bodaubay@nu.edu.kz

Diana Ulaskhanova
dept. of Computer Science
Nazarbayev University
Astana, Kazakhstan
diana.ulaskhanova@nu.edu.kz

Madi Turgunov
dept. of Computer Science
Nazarbayev University
Astana, Kazakhstan
madi.turgunov@nu.edu.kz

***Abstract*—The primary goal of this project was to present an Artificial Intelligence (AI) powered healthcare solution that is able to reduce the strain on hospitals by effectively replacing existing at-home monitoring systems with a system that can autonomously monitor patient telemetry at their own home using accessible IoT devices, detect anomalies, process them using foundation models and provide a detailed report to a healthcare professional. The project achieves this by integrating a complex pipeline of devices, ranging from on-body sensors, which capture relevant information from the patient, to at-home edge computers, which process this data and look for any anomalies, to a remote high-performance processing server, which acts as both an interface for the healthcare professional, and as a host for more powerful AI models that observe the data and generate a comprehensive report, all while providing an intuitive interface for both the patient and the healthcare professional to comfortably interact with the system. Our solution aligns with the modern trend of the integration of AI into every facet of human existence. By harboring the power of various foundation models, we are able to replace what would usually be hours, if not days of grueling work for the healthcare professional, as well as privacy issues for the patient.**

***Index Terms*—Health monitoring, Internet of Things (IoT), edge computing, foundation models, smart homes, anomaly detection, remote patient monitoring, artificial intelligence, large language models (LLMs), multimodal data fusion**

I. Introduction

With the ever-increasing population and the growing strain on healthcare infrastructure, modern hospitals are struggling to effectively and efficiently care for all the people who require their assistance. We believe that the introduction of Artificial Intelligence into the scene will drastically change, just as it has with any other environment, the sphere of healthcare. Our project intends to tackle the strain applied to hospitals by allowing individuals whose conditions are either not critical enough to be hospitalized or are otherwise unable to attend proper hospitalization to be monitored by healthcare specialists in the comfort of their homes. Modern solutions that aim to tackle the same problem are inefficient: they require constant monitoring from healthcare staff, they are prone to privacy issues, and rely on outdated systems, most of the time [1].

This is exactly the problem that our project will attempt to solve: using the power of edge computing and artificial intelligence, we can monitor the condition of the patient without having a doctor assigned to them 24/7. With the rapid growth in the amount of IoT devices around the world, our system is aimed at being easy-to-implement with existing technology.

The system that we have developed will constantly monitor the state of the patient autonomously, without human intervention using various sensors, including, but not limited to oxymeters, thermometers, heart rate sensors, ECG apparatus as well as video and audio footage from conventional smart home systems. Using this data, the system aims to detect anomalies in the patient's state, process it, and inform both the patient and the assigned healthcare representative of their condition, followed by a detailed report from a medically-trained large language model (LLM).

This report will cover the various layers of the system, including the *Existing Research*, our planned *Project Approach*, the detailed *Project Execution*, as well as the *Evaluation and Conclusion*, with references to the related work provided at the end. All of the necessary graphs and figures are provided in Appendices.

II. Background and Related Work

Utilization of at-home monitoring systems is not new to the world of Artificial Intelligence, and many such systems were or are in both the research and execution stages. However, our system is unique in its ability to minimize the human factor in this problem. A solution similar to ours is described in the IoT-based system for elderly monitoring proposed by Jayant et al., (2024) [2] which utilized heart rate, ECG, and temperature sensors. Compared to our solution, it is relatively basic, utilizing PCA for anomaly detection, and being used

only for anomaly detection, without any further processing. In short, their system integrates an array of sensors collecting physiological data, and transmitting it to a cloud platform via a NodeMCU ESP8266. The data is then visualized on a Blynk dashboard and analyzed using a combination of Principal Component Analysis (PCA), Z-score, Isolation Forest, and XGBoost for anomaly detection. The healthcare professional utilizing the system might not have all the necessary context to provide a competent analysis. Many other works in this category exist, including research done by [3] and [4], however, all of these systems effectively only solve the anomaly detection part of our project. The paper by [5] successfully points out the underutilization of AI in the sphere of healthcare as a whole. A proposed system by [6] describes a system similar to ours, albeit only on a high-level. It points out the benefits of a fully-automated Remote Patient Monitoring system and proposes its usage in scenarios when the patient does not need urgent care. Most importantly, it notes Technical Costs as one of the main drawback of such systems, which is something that our project aims to mitigate by utilizing either widely-available devices, or devices on a budget. Throughout the Fall semester, we have worked on a detailed review paper detailing most scientific work covered by papers on this topic. The paper can be accessed via this link.

III. Project Approach

A. Project Overview

The project is effectively split into 5 layers, as showcased in Figure 1. The overall pipeline starts at the sensor level, more specifically the Arduino board recording the vitals of the observed person (from this point referred to as Patient), which sends their vitals to the Patient's Edge Computer, where the system, in the case of a detection of any kind of anomaly, sends a signal to the main high-performance server. This server then analyzes the sensor, as well as the video and audio data collected by the Edge Computer in order to detail a comprehensive report by utilizing the Foundation Layer, which utilizes a Dynamo server. The report is saved in the database and is accessible to the person assigned to monitor the Patient (from this point referred to as the Doctor). Both the Patient and the Doctor can receive notifications regarding this situation via a Telegram Bot interface. All information is stored in a MongoDB object database in order to preserve speed and performance.

B. Sensor Data

For our experiments, we used two publicly available ECG datasets:

- **MIT-BIH Arrhythmia Dataset**
  - Number of Samples: 109,446
  - Number of Categories: 5
  - Sampling Frequency: 125 Hz
  - Data Source: PhysioNet's MIT-BIH Arrhythmia Database
  - Classes: `['N': 0, 'S': 1, 'V': 2, 'F': 3, 'Q': 4]`
- **PTB Diagnostic ECG Database**
  - Number of Samples: 14,552
  - Number of Categories: 2
  - Sampling Frequency: 125 Hz
  - Data Source: PhysioNet's PTB Diagnostic Database

These datasets were combined for the training data.

Model Architecture: We used a 1D Convolutional Neural Network for classification. The architecture includes three convolutional layers, each followed by batch normalization and max pooling to reduce the feature dimensions. The extracted features are flattened and passed through two dense (fully connected) layers, and the final output layer uses a softmax activation for multi-class classification.

Initial Training: In the initial version of the model:

- Each convolutional layer used 64 filters.
- Gaussian noise with standard deviation 0.5 was added to the input signals as data augmentation.
- No dropout or regularization was used.

This model was trained for up to 40 epochs and achieved a validation accuracy of **72.45%**. Results are shown Figure 2.

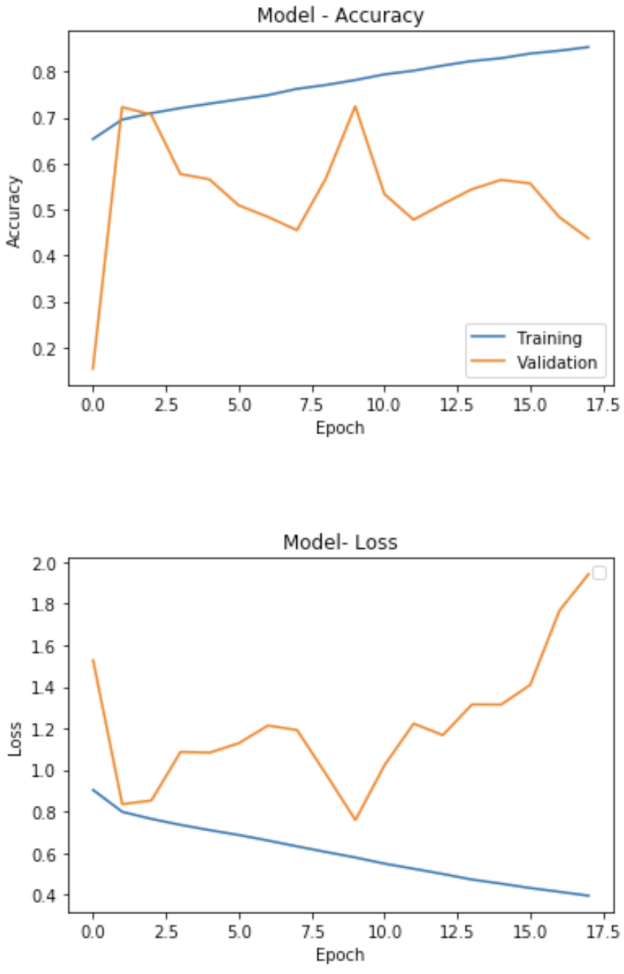

Fig. 2. Initial ECG training

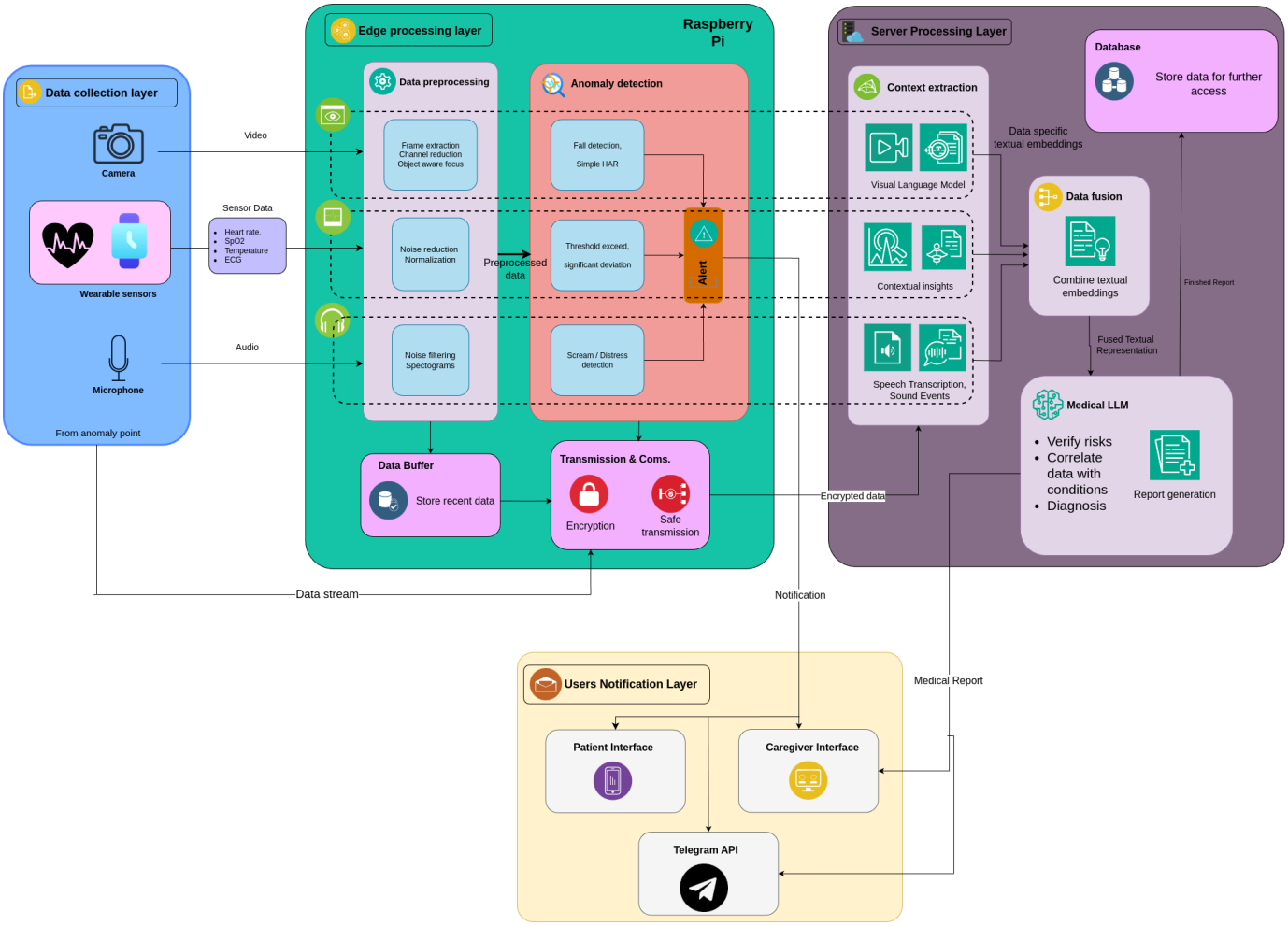

Fig. 1. Final Architecture

Refined Training: To improve performance, we made several changes:

- Reduced the number of filters in each convolutional layer from 64 to 32.
- Added dropout layers after each convolutional and dense layer to reduce overfitting.
- Introduced L2 regularization in the dense layers.
- Reduced the noise level added to the input (standard deviation 0.1 instead of 0.5).
- Reduced training epochs to 20 based on early stopping.

These modifications resulted in a significant improvement, reaching a validation accuracy of **88.76%**. Results are shown Figure 3.

Initially, we planned to collect ECG, temperature, oxygen, and heart rate data using sensors. However, due to technical issues, the ECG sensor did not work, and we also did not have functioning temperature, oxygen, and heart rate sensors.

As a solution, we used four buttons connected to an Arduino. The Arduino was connected to a Raspberry Pi via USB. The Arduino simulates sensor data: when a button is pressed, it starts sending simulated anomaly data.

The Raspberry Pi receives the raw data from the Arduino and runs the trained model locally. It decides whether the incoming data represents an anomaly or not.

The setup is shown on Figure 4.

C. Edge Layer System

The edge layer represents the embedded device responsible for direct data acquisition from sensors, local processing for anomaly detection, and subsequent reporting to the remote server. Designed to operate autonomously in diverse environments, this layer prioritizes real-time performance, resource efficiency, and robust data handling, especially in scenarios with intermittent network connectivity. This section details the hardware, software architecture, data pipelines, and operational aspects of the edge layer implementation.

1) Hardware Components: The edge layer is built upon a compact, low-power computing platform, specifically a **Raspberry Pi 4 Model B**. This single-board computer provides the necessary processing power for data acquisition, executing anomaly detection algorithms, managing system state, and

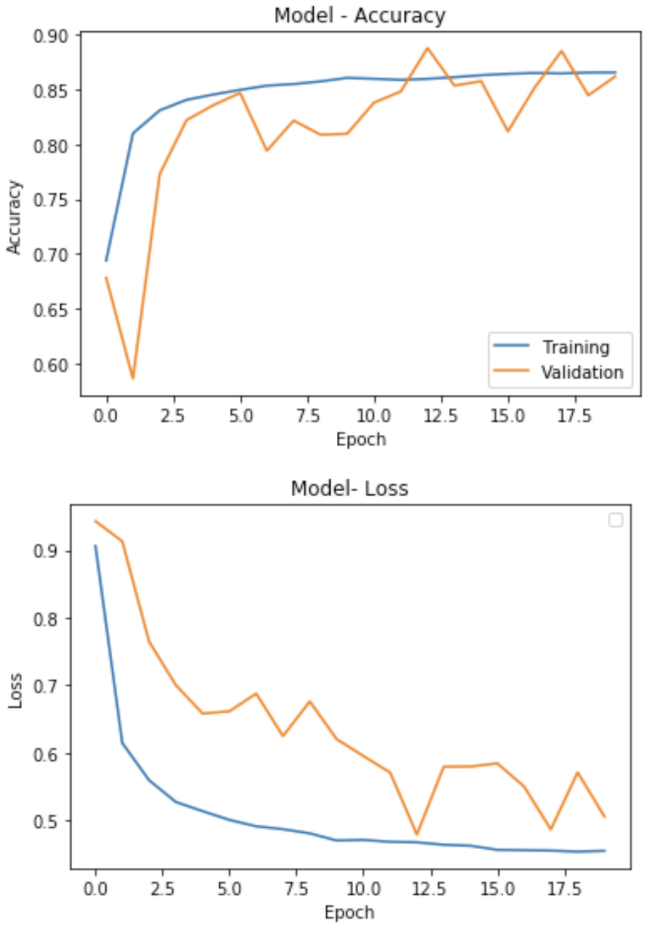

Fig. 3. Initial ECG training

handling network communication. Attached peripherals facilitate multimodal data capture:

- **Computing Unit:** Raspberry Pi 4 Model B, running a Linux-based operating system, serving as the central processing hub.
- **Sensor Interface:** An **Arduino** board connected via a serial interface (e.g., USB-to-Serial) to the Raspberry Pi. This microcontroller acts as an intermediary, collecting analog and digital data from various medical-grade sensors, performing necessary analog-to-digital conversion, formatting the data (as JSON strings), and transmitting it serially to the Raspberry Pi.
- **Vital Signs Sensors:** Specific sensors connected to the Arduino include:
  - ECG Sensor: Captures electrocardiogram signals (often transmitted as a list of readings).
  - Heartbeat Sensor: Provides heart rate readings (e.g., in BPM).
  - Oxygen Saturation (SpO2) Sensor: Measures blood oxygen levels (e.g., in percentage).
  - Temperature Sensor: Measures body temperature (e.g., in Celsius).

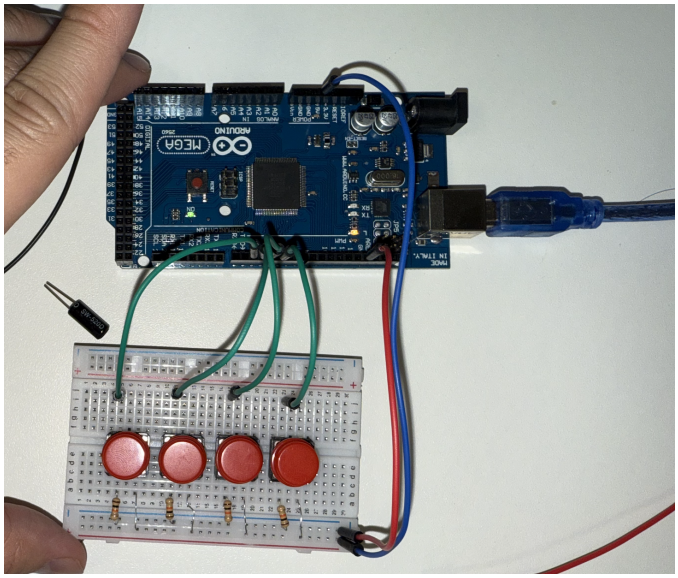

Fig. 4. Initial ECG training

- **Video Capture Device:** An **Intel® RealSense™ D435** camera. This camera provides the RGB video stream used for fall detection via object detection. The system interfaces with the camera using standard libraries like OpenCV, configured for specific resolution and frame rate (`CAPTURE_CONFIG`).
- **Audio Capture Device:** A **Razer Seiren** Bluetooth microphone. Audio streams are captured from this device using standard audio input libraries (like PyAudio), providing data for distress sound detection.
- **Network Interface:** The Raspberry Pi's integrated Wi-Fi or Ethernet connectivity for communication with the remote server API (`SERVER_CONFIG`).
- **Local Display (Optional):** A connected monitor for displaying the local visualization dashboard during development or deployment, offering real-time insight into vital signs and system status.

A photograph of the assembled hardware system is shown in Figure 5.

2) Sensor Data Acquisition and Anomaly Modeling: This section details the process of acquiring data from the vital signs sensors, the methodologies used for simulating sensor data via the Arduino interface, and the specifics of how anomaly detection models or rules for these vital signs were developed and implemented for the edge device.

3) Software Architecture: The software on the edge layer is designed as a multi-threaded application written in Python, emphasizing modularity and concurrency to handle simultaneous data streams and processing tasks efficiently on the limited resources of the Raspberry Pi. The core components and their interactions are managed using Python's `threading` module.

- **Core Management (`core/`):** This package contains critical modules for maintaining system

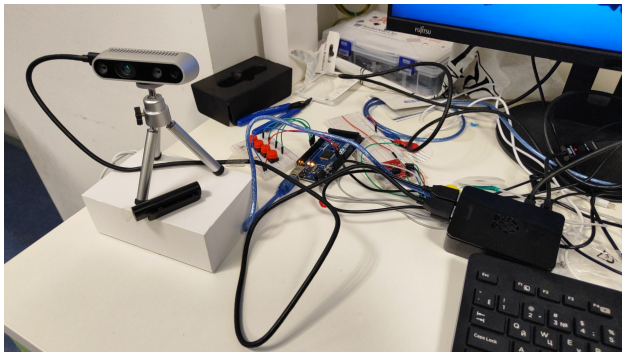

Fig. 5. Physical setup of the Edge Layer system, showing the Raspberry Pi 4 Model B, Arduino microcontroller, Intel® RealSense™ D435 camera, Razer Seiren microphone, and connected vital signs sensors.

state (`state_manager.py`) and coordinating event responses (`event_manager.py`). The `state_manager` uses thread-safe mechanisms like `threading.Lock` to manage shared data structures, including sensor readings, anomaly statuses, data buffers (video and audio), and event flags. The `event_manager` is responsible for triggering comprehensive actions when anomalies occur, such as collecting buffered data, encoding, and initiating network transmission.

- **Sensor Interfaces (`sensors/`):** Modules in this package abstract the hardware interactions. `sensor_reader.py` communicates with the serial interface for vital signs from the Arduino, `video_capture.py` handles the camera interface using OpenCV for the Intel RealSense camera, and `audio_capture.py` manages the microphone interface using PyAudio for the Razer Seiren. These modules continuously acquire raw data and push it into shared buffers or update sensor states.
- **Anomaly Analysis (`analysis/`):** This package houses the logic for detecting anomalies. `anomaly_detection.py` provides generic threshold-based checks used by the sensor reader for vital signs. `fall_detection.py` implements vision-based fall detection using YOLO models on video frames from the RealSense camera. `audio_inference.py` applies heuristic analysis to audio streams from the Razer Seiren to detect distress sounds. These modules process data from the capture components and update anomaly statuses in the `state_manager`.
- **Network Communication (`network/`):** Responsible for interacting with the remote server. `auth.py` handles the login process to obtain an authentication token. `data_sender.py` manages sending regular vital updates and event payloads to the server APIs, incorporating the authentication token and handling potential network issues, including token expiration.
- **Utilities (`utils/`):** Provides supporting functionalities such as encoding video frames to MP4 and audio chunks to WAV (`encoding.py`), monitoring system resources like memory usage (`monitoring.py`), and persistently storing event payloads locally when server communication fails (`local_payload_storage.py`).
- **Visualization (`visualization/`):** Contains the `vitals_visualization.py` module, which runs a lightweight Flask and Socket.IO server. This server provides a web-based dashboard accessible on the local network, displaying real-time vital signs read from the `state_manager` and status information.

The application's entry point, `main.py`, initializes and starts the threads for each of these core functionalities, allowing them to run concurrently on the Raspberry Pi.

4) Data Acquisition and Processing Pipelines: The system processes data through distinct, concurrent pipelines for each modality, integrated through the central state manager and the event handling system.

a) Vital Signs Pipeline: `sensor_reader.py` continuously reads data packets from the serial connection established with the Arduino. Each packet, expected to be a JSON object containing readings for ECG, heartbeat, oxygen, and temperature, is parsed. For each sensor type, the raw reading is passed to `update_sensor_reading` in `analysis.anomaly_detection`. This function updates the sensor's value and timestamp in the global `sensor_state` and performs a threshold-based anomaly check (e.g., heartbeat $>$ max BPM, oxygen $<$ min SpO2) based on configurations in `settings.py`. The anomaly status for that sensor is updated in the `anomaly_present` map within the `state_manager`. If a *new* anomaly status (transition from not-anomalous to anomalous) is detected, it triggers the main event handling process via a lazy import call to `event_manager.handle_anomaly_event`. Additionally, a separate `regular_data_sender_thread` periodically reads the latest vital signs from the `state_manager` and sends them to the server's vitals endpoint via `data_sender.send_regular_vitals` in `network.data_sender`. The `vitals_visualization` also reads this state for the local dashboard display.

b) Audio Pipeline: The `audio_capture_thread` uses PyAudio to capture continuous audio streams from the Razer Seiren microphone, segmenting them into fixed-size chunks (`CHUNK`). These chunks are appended to a fixed-size circular buffer, `audio_buffer`, within the `state_manager` to provide a history of recent audio. The `audio_inference_thread` periodically wakes up, reads a recent segment of audio from the buffer, and performs heuristic analysis using `audio_inference.detect_audio_anomaly`.
This analysis calculates acoustic features from the audio samples using libraries like Librosa, such as RMS volume, zero-crossing rate, and spectral properties (centroid, bandwidth, contrast). A confidence score is derived by weighting these features against empirically determined thresholds (`DETECTION_THRESHOLD`). To

reduce false positives, a confirmed audio anomaly requires multiple consecutive chunks to exceed the confidence threshold, tracked by a `collections.deque` history (`detection_history`). If a confirmed anomaly is detected, the status in `anomaly_present["audio"]` is updated via `check_and_update_anomaly_status` in `analysis.anomaly_detection`, triggering the event manager. The thread incorporates a cooldown period (`COOLDOWN_PERIOD`) after a detection is triggered to prevent spamming events. When an event is processed by the `event_manager`, it retrieves the relevant audio chunks from the `audio_buffer`, encodes them into a WAV format, converts to base64 using `utils.encoding`, and includes this in the payload sent to the server.

c) Video Pipeline (Fall Detection): The `video_capture_thread` captures frames from the Intel RealSense D435 camera using OpenCV at a target frame rate (`TARGET_FPS`). It maintains a local fixed-size circular buffer (`_frame_buffer`) storing recent frames and stores the latest captured frame (`_latest_frame`) within the `video_capture` module itself, using a dedicated reentrant lock (`_buffer_lock`) to minimize contention and provide low-latency access. It also appends frames to a temporary `_post_event_frames` buffer when the `event_recording` flag is set in the `state_manager`. The `video_inference_thread` periodically retrieves the most recent frame (via `get_latest_frame`), downscales it (`CAPTURE_CONFIG["DOWNSCALED_SIZE"]`) to reduce processing load, and runs a lightweight YOLO model (`yolov8n.pt`) for object detection. The inference results are processed to identify bounding boxes for persons (COCO class ID 0). Fall detection logic tracks the vertical centroid of detected persons across frames. A fall is flagged if a significant drop in the centroid position occurs between frames, defined as a fraction of the frame height (`CAPTURE_CONFIG["FALL_THRESHOLD_FRACTION"]`). The detected fall status updates `anomaly_present["video"]` via `check_and_update_anomaly_status` in `analysis.anomaly_detection`, triggering the event manager. An optional `display_preview_thread` can show the latest frame with status overlays using OpenCV's GUI functions for local monitoring. When an event is processed by the `event_manager`, it retrieves a window of pre-event frames from the `_frame_buffer` and the collected post-event frames from `_post_event_frames` (via `get_frame_buffers`), encodes them into an MP4 video format using `utils.encoding`, converts to base64, and includes this in the payload sent to the server.

5) Event Handling and Reporting: The `event_manager` coordinates the system's response to detected anomalies. When `handle_anomaly_event(trigger_source)` is called by any analysis pipeline indicating a *new* anomaly:

1) It first checks if the system is within an event cooldown period (`THRESHOLDS["EVENT_COOLDOWN_SECONDS"]`) or if another event is already being processed, using the `send_lock`. This prevents rapid-fire event triggers.
2) If an event can be triggered, the cooldown timer is reset, and the `event_in_progress` flag is set.
3) A separate thread, `process_anomaly_event`, is spawned to handle the event processing asynchronously, preventing the anomaly detection threads from blocking.
4) The `process_anomaly_event` thread sets the `event_recording` flag in the `state_manager` to signal the video capture thread to save post-event frames.
5) For video-triggered events, it pauses briefly (`CAPTURE_CONFIG["POST_EVENT_SECONDS"]`) to allow capture of post-event context frames.
6) It retrieves the relevant buffered data:
   - Video frames: A segment of pre-event frames from the video capture buffer (`_frame_buffer`) and the collected post-event frames (`_post_event_frames`).
   - Audio chunks: The recent history from the `audio_buffer`.
7) It retrieves the current state of all other sensors (`sensor_state`) and the overall anomaly status (`anomaly_present`) from the `state_manager` as contextual data. NumPy types are sanitized for JSON serialization (`sanitize_for_json`) using a utility function within `event_manager`.
8) Video frames are encoded into an MP4 format and audio chunks into WAV format using `utils.encoding`. These binary data streams are then Base64 encoded. Console progress bars are implemented in the encoding process to provide feedback.
9) A comprehensive payload is constructed, including device ID, sensor context (values and statuses), encoded audio data, and encoded video data.
10) If configured (`STORE_PAYLOADS_LOCALLY` in `event_manager.py`), the payload is saved to local storage regardless of network success. This is useful for debugging and backup.
11) The payload is sent to the remote server's data endpoint via a lazy import call to `network.data_sender.send_data`.
12) The `send_data` function handles network communication, including authenticating with the server (obtaining/using a JWT token via `network.auth.login`), retrying on token expiration, and setting timeouts. The JWT token is stored and managed within the `state_manager`.
13) If sending fails and `STORE_ON_SERVER_FAILURE` is true (and the payload wasn't already stored locally), `send_data` saves the payload using `utils.local_payload_storage`.
14) Finally, the `event_in_progress` flag is cleared within the `state_manager`, allowing new events to be triggered after the cooldown period.

In addition to event-triggered data, a dedicated

`regular_data_sender_thread` periodically fetches the latest vital signs from the `sensor_state` and sends them to the server's vitals endpoint via `send_regular_vitals` in `network.data_sender`. This provides a continuous stream of health data to the server even when no anomalies are detected.

The system's logging (`logging` module) provides crucial insights into the operation of each pipeline, thread activity, anomaly detection events, and network communication status. Figure 6 shows typical output from the system logs.

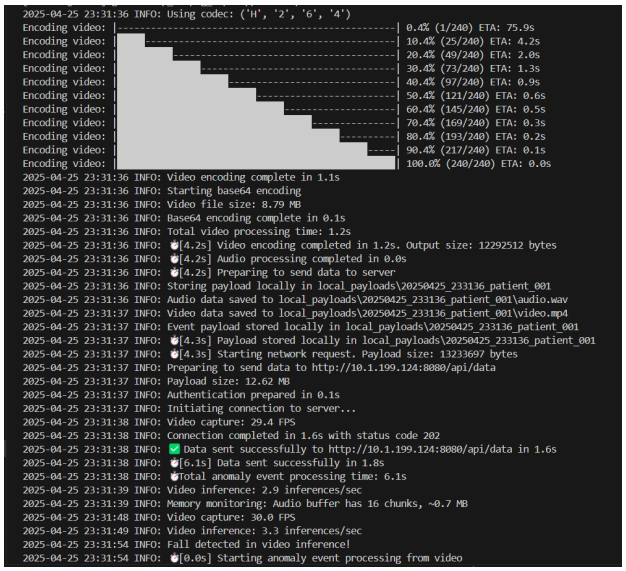

Fig. 6. Screenshot illustrating typical log output from the Edge Layer system, showing sensor readings, detection events, and communication status.

6) State Management and Thread Synchronization: Given the multi-threaded architecture on the Raspberry Pi, careful state management and thread synchronization are critical to prevent race conditions and data corruption. The `state_manager.py` module centralizes access to shared resources and defines thread-safe access patterns.

- **Global State Dictionaries (`sensor_state`, `anomaly_present`):** Store the latest sensor values, timestamps, and current anomaly status for each modality.
- **Circular Buffers (`audio_buffer`):** `collections.deque` is used for audio, providing efficient fixed-size buffering. The video buffer (`_frame_buffer`) is managed locally within `video_capture` using a similar concept, but specifically tuned for high-frequency video capture.
- **Event Flags (`last_event_time`, `event_in_progress`, `event_recording`):** Control the event handling flow and trigger post-event data capture.
- **Authentication Token (`jwt_token`):** Stores the current JWT token received from the server for authenticated API calls.
- **Thread Locks (`state_lock`, `send_lock`, `_buffer_lock`):** `threading.Lock` and `threading.RLock` are used to ensure that only one thread can modify shared state or access buffers at a time. The design minimizes the time threads spend holding locks to reduce blocking and maximize concurrency. Specifically, `state_lock` protects general sensor state and anomaly flags. `send_lock` protects event triggering logic and prevents concurrent event processing. `_buffer_lock` within `video_capture` protects the video frame buffers, crucial for maintaining high capture frame rates without interference from inference or event threads.

Accessing and modifying shared data structures *must* be done within 'with lock_name:' blocks as appropriate to ensure thread safety. Functions like `update_sensor_reading` and `check_and_update_anomaly_status` are designed to acquire and release the necessary locks efficiently.

7) Local Visualization: A local web-based dashboard is provided by the `vitals_visualization.py` module. This module runs a separate lightweight Flask application integrated with Socket.IO, which operates concurrently with the main monitoring threads. It serves simple HTML, CSS, and JavaScript files (`templates/`, `static/`). The JavaScript client connects to the Flask server via Socket.IO and receives real-time vital sign updates streamed from the edge device. The server runs a background thread (`background_thread`) that periodically reads the latest sensor data and anomaly statuses from the `state_manager` (with minimal locking) and emits them to connected clients using 'socketio.emit'. The dashboard uses Chart.js to plot the vital signs over time and visually indicates anomaly statuses (normal, warning, critical) based on thresholds or the explicit anomaly flags from the backend. This local interface is valuable for verifying sensor data flow, monitoring system health during development and deployment, and demonstrating the system's real-time capabilities without relying on the remote server connection. Figure 7 shows a screenshot of the local vitals dashboard.

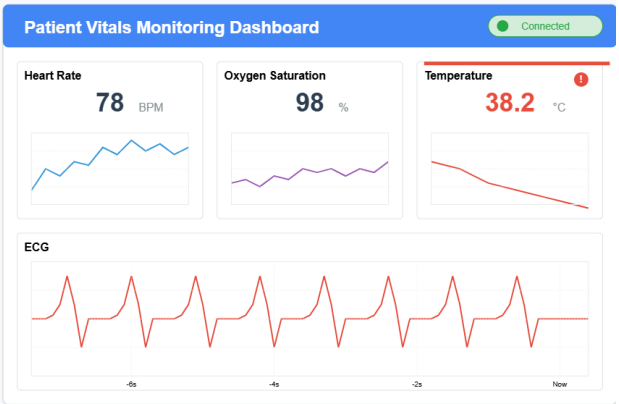

Fig. 7. Screenshot of the local web-based dashboard displaying real-time patient vital signs and anomaly statuses.

D. Backend Server

In our solution, the Backend Server is the endoskeleton holding the entire deliverable together. In order to achieve maximum performance, we have decided to utilize the Go programming language, containerized using Docker to ensure modularity of the system, combined with a MongoDB database for non-relational data storage, which has proven to be crucial to the completion of the deliverable. Using the Go Programming Language, unlike our previous plan of using Python's FastAPI, gives us the full benefit of a pre-compiled, error-free, and, most importantly, predictable server behaviour. The functionality of the backend system (codenamed as the *Oreo Server*) varies greatly and achieves many goals of the project. The file structure is show below in Figure 8 Just like the

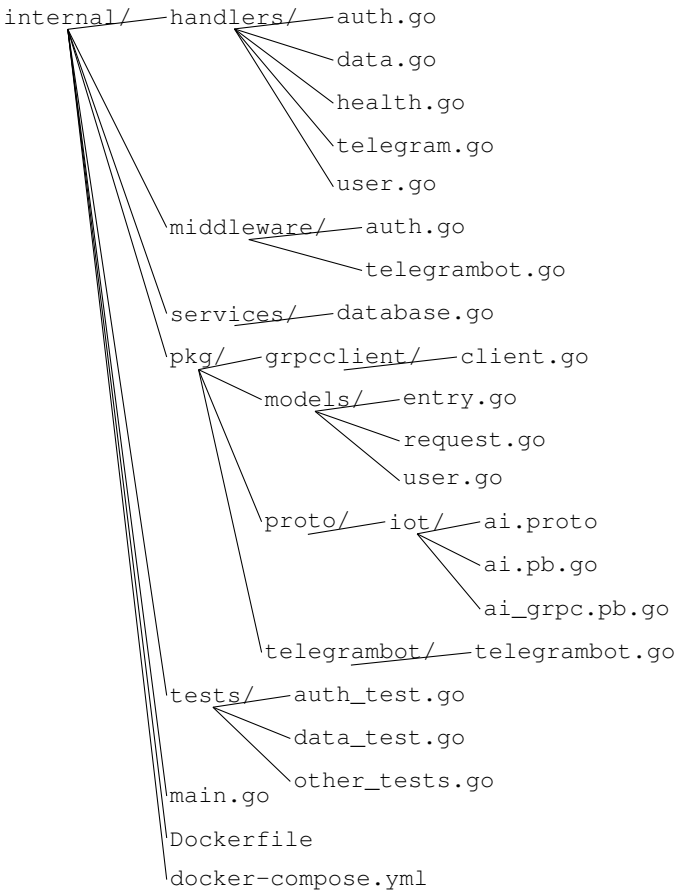

Fig. 8. Project file structure of the IoT server backend

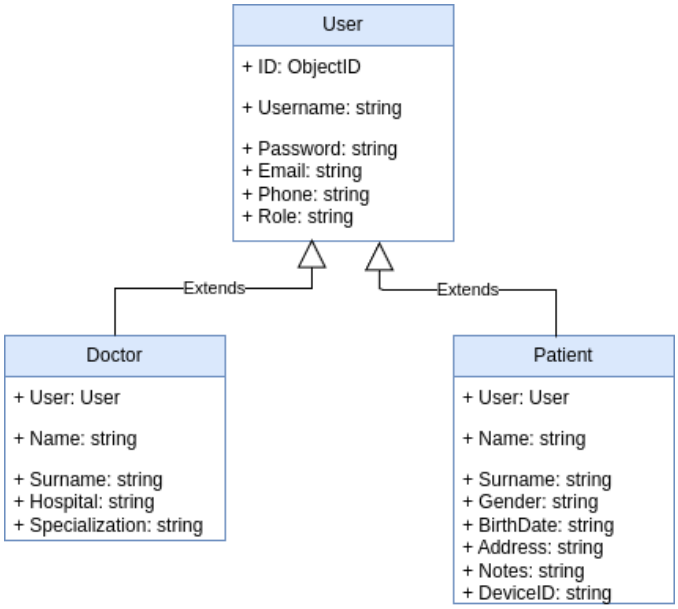

Fig. 9. Models for User Storage

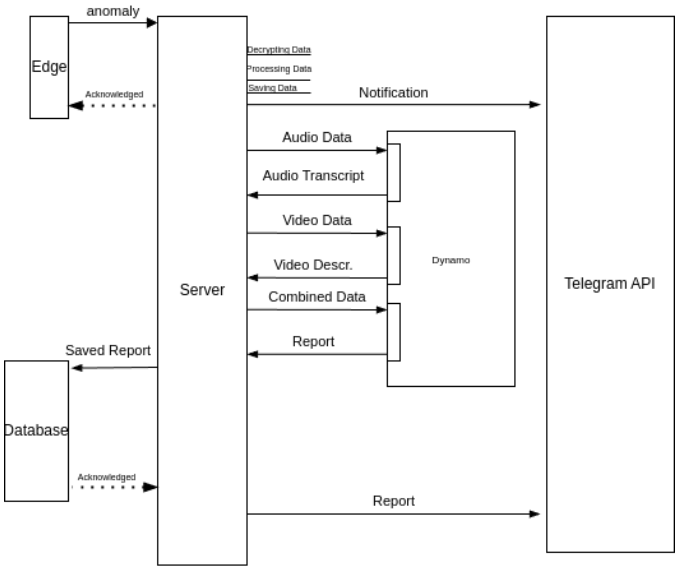

Fig. 10. Pipeline of the Anomaly Event

project itself, it is divided into four main "pipelines": User Handling, Report Handling, Vitals Handling, and Middleware such as the Telegram Handler, Dynamo handler and MongoDB Handler. The exact functionality of each of these layers will be described in the following sections.

1) User Handling: The user handling layer is composed of modules that handle user authentication, registration, login, creation, assignments, information. It utilizes the User, Patient, Doctor, PatientDoctor models (Fig 9) Authentication is done using the JWT Token [7] system. The tokens are set to expire after a week. The Patient token is used both by the Edge Computer and the User. Each Doctor can be assigned to many Patients, while only one Doctor can be assigned to one Patient.

2) Report Handling: The reports are also handled on the back-end server. The pipeline of reports can be seen in 10. The anomaly report from the Edge Server, alongside the relevant data, is received by the server. The report can only come from an authenticated edge computer. The appropriate user is located, the audio, video and sensor data is processed, saved, and the edge computer receives an acknowledgement. Afterwards, a notification is sent to the Patient's *Telegram*

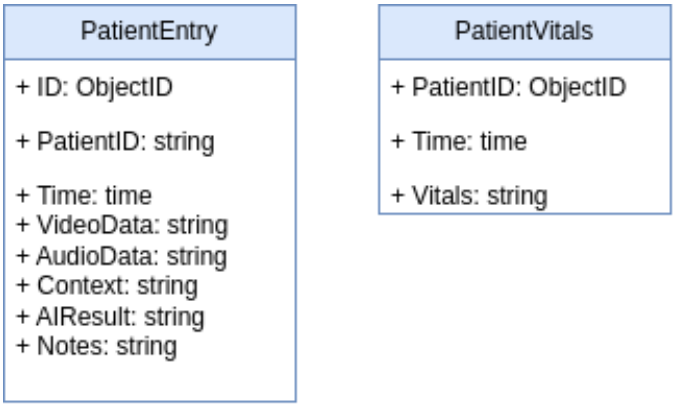

Fig. 11. Vitals and Patient Report data structures

about the abnormal situation, and the data is sent to the appropriate foundation models. More about Data Processing is mentioned in *Data Fusion and Foundation Model Processing*. The server saves the report in the MongoDB database in the form described by Figure 11, and, after receiving the acknowledgement from the database, sends the report notification to the assigned Doctor. The data is received in base64 string format, and in case either video or audio is unavailable, the handler will still run the pipeline. Upon request, the reports are shown to the Doctor, if and only if they are assigned to the report-owning patient.

3) Vitals Handling: Similar to report handling, a separate, less complicated pipeline is active for Vitals handling. Vital data are received from the Edge Computer once every few seconds, and they are assigned to the patient. The vitals data include (but are not limited to) heartbeat, ECG, blood oxygen level, and body temperature. However, because vitals data are stored in a json-like file structure, they are not limited to these sensors, and can be easily interchanged. The full structure is shown on Figure 11. Unlike reports, due to the frequency of vitals data, and the fact that our system is designed to handle heavy loads, only one instance of vitals is stored at the same time for the same Patient. This is enough for both the Doctor and Patient to see their vitals data in real time.

4) Middleware: Telegram Handler: Our system utilizes Telegram's API in order to create a Telegram notification bot. This bot is able to be linked to a certain user, both Patient and Doctor, in order to receive their respective notifications. The linking process is handled in two steps: first, the user starts the chat, which sends a unique ChatID to the user, who then has to enter that code into their frontend interface (Fig 15). Afterwards, the Telegram bot sends yet another code, this time a randomly-generated verification code, which is active for 15 minutes. If the user fails to enter the information in 15 minutes, the process is to be repeated again. Otherwise, the User is linked to the Telegram chat instance. The Telegram Chat is stored in the database in a single model, which is visible in Figure 14. The Doctor receives full reports from the Patient 12, while the Patient receives notifications about alerts

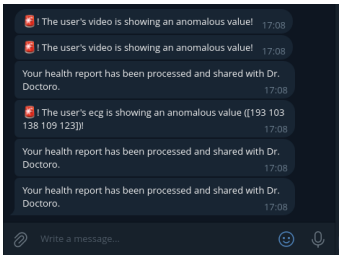

Fig. 12. Example of a Patient's Telegram Notification

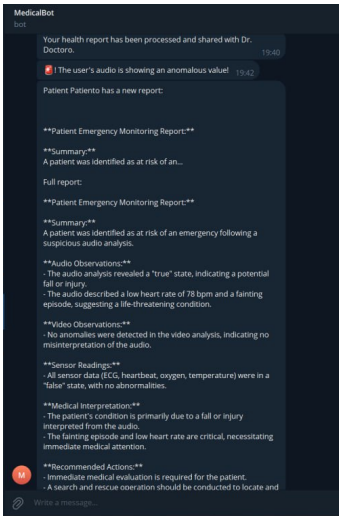

Fig. 13. Example of a Doctor's Telegram Notification

and the fact their report was sent to their Doctor 13.

5) Middleware: Dynamo Handler: The Dynamo handler interacts with the Dynamo server, which hosts the backend's foundation models. The Dynamo Handler is unique in the fact that it utilizes gRPC technology [8]. The benefits of

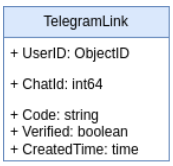

Fig. 14. Telegram Linking Model

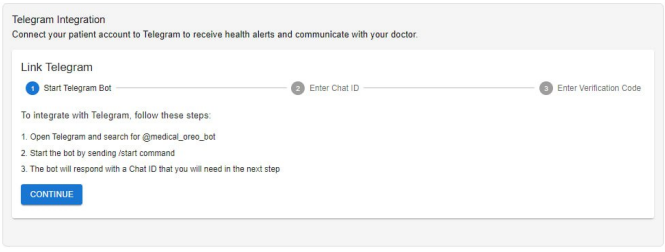

Fig. 15. User Interface for linking the Telegram Bot

using gRPC instead of a standard REST configuration is in the fact that all data is strongly typed, and that it utilizes a more efficient data transfer method called "Protobuf", which runs on HTTP/2, rather than HTTP/1.1. As described in the section about report handling, the backend will send the data received from the edge computer to the VLM, AudioLM, and a specially trained Medical Large Language Model, which will finalize the report to be saved. More details about the foundation models can be found in the *Data Fusion and Foundation Model Processing* segment of this report.

6) Middleware: MongoDB Handler: The MongoDB handler is the most simplistic handler of the entire backend. Its sole purpose is to authenticate with our database, send and receive data. It is largely unmodified from the default Go implementation of a MongoDB handler, only containing the necessary configuration information, such as ports or addresses for the database.

E. Data Fusion and Foundation Model Processing

Data Fusion and Foundation Model Processing layer is responsible for audio-to-text, video-to-text and report generation. The audio obtained from edge device passes to Audio Language Model (ALM) and audio classification with the description is returned to Backend Server. The same procedure, but with Video Language Model (VLM), is performed with video and returned to Backend Server. After describing audio and video modalities, Backend Server will combine descriptions to one prompt, add edge sensor data and send to Dynamo Server where LLM model with given system prompt for detection abnormalities will generate medical report.

Models that are used for audio-to-text, video-to-text and medical report generation:

- Audio Language Model: Qwen/Qwen2-Audio-7B-Instruct
- Video Language Model: llava-hf/llava-onevision-qwen2-7b-ov-hf
- LLM Model: deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B

Data Fusion is happening in LLM Model when medical report generation. Audio and Video descriptions are fused to one comprehensive prompt which passes to LLM Model for medical report generation.

The deployment of the models are performed by Open Source framework: AI-Dynamo by Nvidia. AI-Dynamo is cloud scaleable, resource shared framework for Language Models deployment. This framework was adopted for ALM and VLM models deployment with automatic resource sharing for better performance.

F. Frontend Implementation

The frontend of the Health Monitoring System is based on a React web application to ensure a responsive, user-friendly, and role-based interface. It allows users to monitor their conditions in real time and facilitates fast action in case any abnormalities are detected. Main functionalities of the frontend include access based on the role, admin, patient, or doctor role, authentication system, vital signs monitoring in real-time, patient and doctor data management, generation of medical reports on patients' condition and telegram integration.

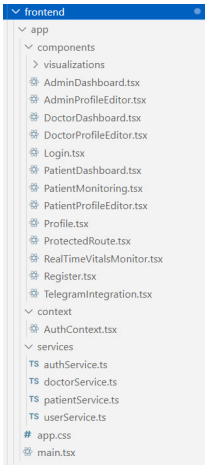

Fig. 16. Frontend folder structure

For the development, React with TypeScript was used. The technology stack also includes build tool Vite for fast development, styling Tailwind CSS for ready made components and Router DOM dynamic routing. Axios library handles requests to backend. Tailwind and Material UI enable the interface to adapt across different desktops.

The folder structure is shown in Figure 16. The entry point, main.tsx file, initializes the application and sets up providers and routing. The AuthContext.tsx file manages login state, token storage, and user role checking. Service module handles communication with backend. Components for Login/Registration manage data validation and submission. Protected Routes ensure users can access only permitted pages. Auth data is stored in local storage to maintain the session.

User roles supported are doctor, admin and patient roles. Respective dashboards allow different functionality based on the role. Admin Dashboard allows to see lists of all patients and doctors and assign doctors to patients, Figure 17. Doctor Dashboard allows to access patients' health data, report generation and telegram notifications, Figure 19. Patient Dashboard is needed to visualize patients' personal and health data, Figure 18.

A doctor has the functionality to sort reports if he needs, by oldest or newest or by date, and view past reports moving between pages, Figure 20. In the doctor dashboard doctor can view the multimodal version of the report with the sensor data, video, audio and LLM analysis. He can also add his notes.

Report structure is shown in several figures, Figure 21, Figure 22, Figure 23, Figure 24. In the downloadable version of the report, it has only patients' sensor data and LLM analysis.

The authService manages login and session persistence. Role-based dashboards fetch respective data using service modules. Data is passed to UI components which also handle different user interactions with the system.

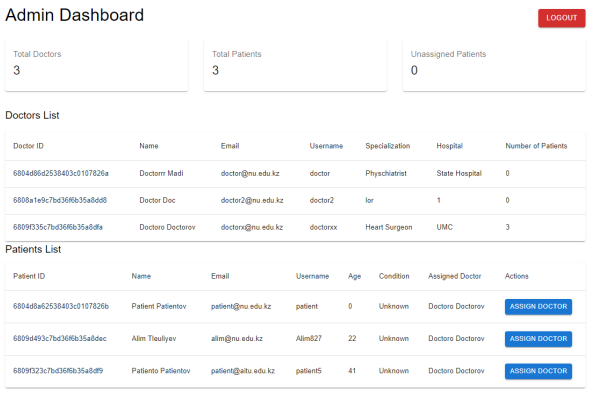

Fig. 17. Admin dashboard

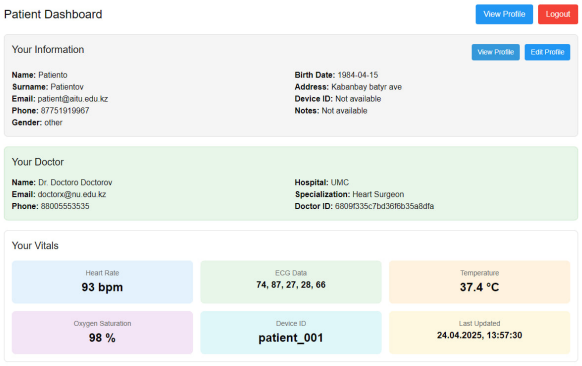

Fig. 18. Patient dashboard

Frontend is integrated with Backend module to handle authorization, vitals data fetching and report generation, the Telegram module to receive immediate alerts and Real-Time Vitals Monitoring module to get current vitals data continuously.

IV. Project Execution

A. Fall Semester

During the Fall semester, our main objective was to collect existing literature in order to formulate a *Review Paper*.

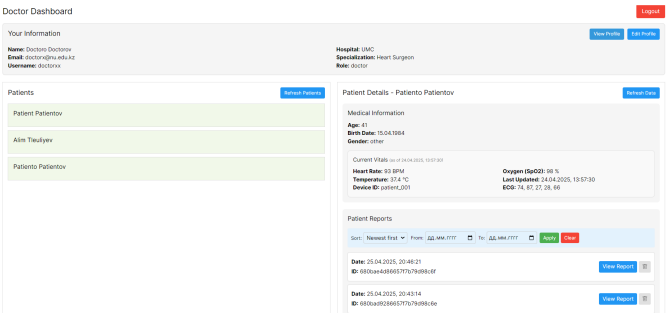

Fig. 19. Doctor dashboard

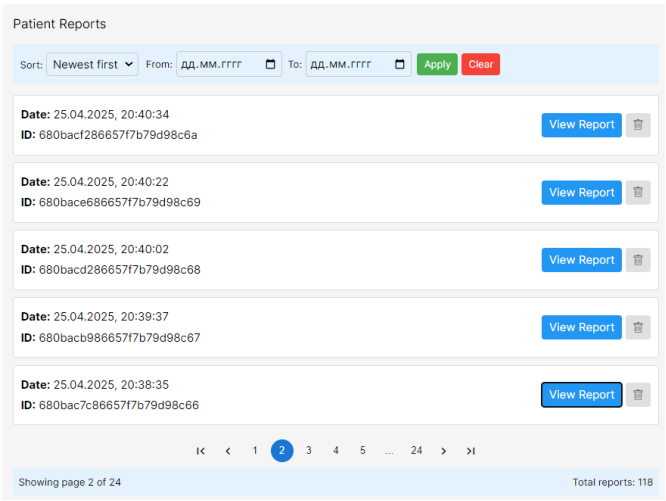

Fig. 20. Report sorting in doctor dashboard

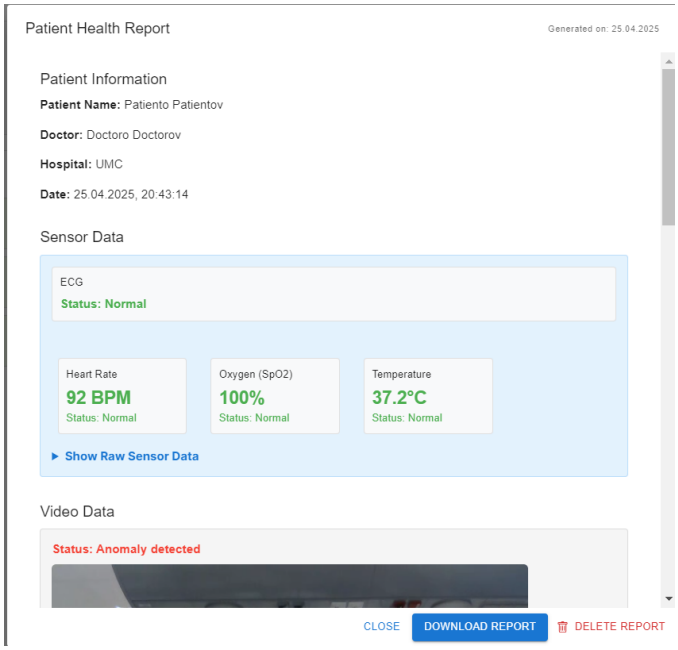

Fig. 21. Report: sensor data

Throughout the semester we were busy planning out the system, envisioning the pipeline, as well as collecting up-to-date research on the topics at hand. We held weekly meetings in both between the teams as well as with our project supervisor. Each member was designated a certain part of the system, for which they would both collect the relevant research, as well as work on rigorous design of the system at hand. Due to time constraints, and the fact that we were unable to start practical work on the project until we received the needed hardware, we were unable to start the execution of the project at hand until the Spring Semester. Our initial system was,

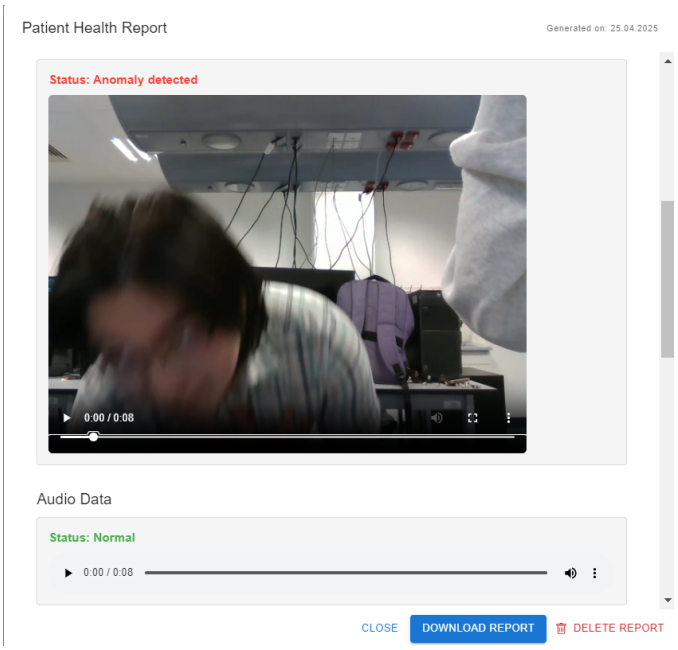

Fig. 22. Report: video/audio data

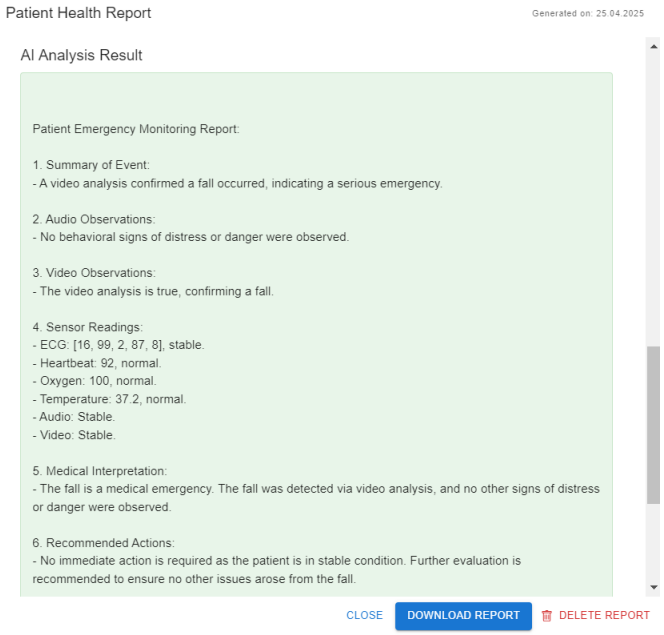

Fig. 23. Report: llm analysis

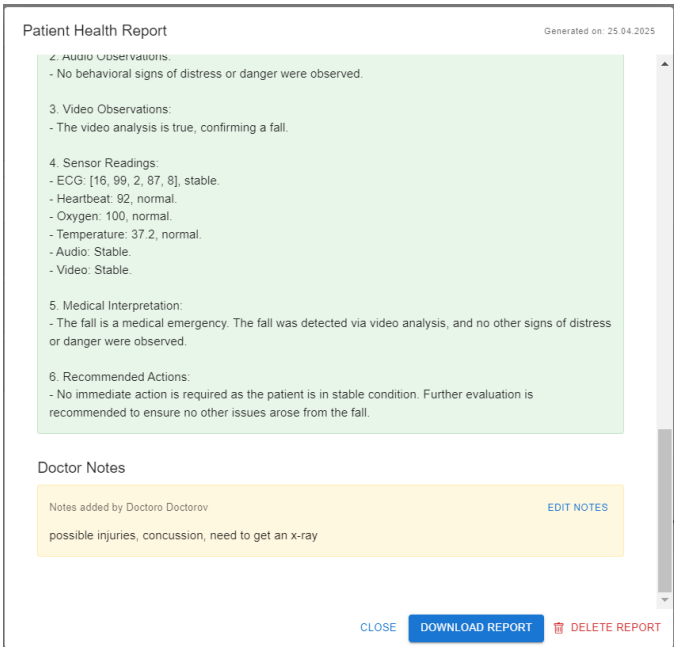

Fig. 24. Report: doctor notes

as most projects tend to be, overambitious. Our initial plan outline can be seen in Fig 25. As can be seen, our original plan was to have the Edge Computer serve the purpose of purely the data sender, with the High-Performance server analyzing all three modalities and inferring anomalies. This has proven to be unfeasable, as the near-constant analysis of data would not only overload the system but would under-utilize the edge computer. By the end of the Fall Semester, we had refined our model in order to alleviate this error. The refined model has correctly assumed most of the functionality of our final deliverable, albeit with some critical changes made in the Spring Semester. The model can be seen in Fig 26.

Our plan was to train or fine-tune a complex model, something that was interchanged to a pre-trained Medical LLM, due to both time constraints and the fact that we were unable to collect all the data necessary. Our main goal of the semester was to complete our Review Paper, which we have done.

B. Spring Semester

During the Spring semester, we have begun working on the practical part of the project. The following hardware components were used in the project:

- Raspberry Pi 4 Model B
- DHT22 Temperature and Humidity Sensor
- AD8232 ECG Sensor
- MAX30102 Integrated pulse oximeter and heart-rate monitor sensor
- Intel RealSence
- Razer Bluetooth Microphone
- Arduino Nano Board
- Minor wiring components (breadboard, switch, buttons)

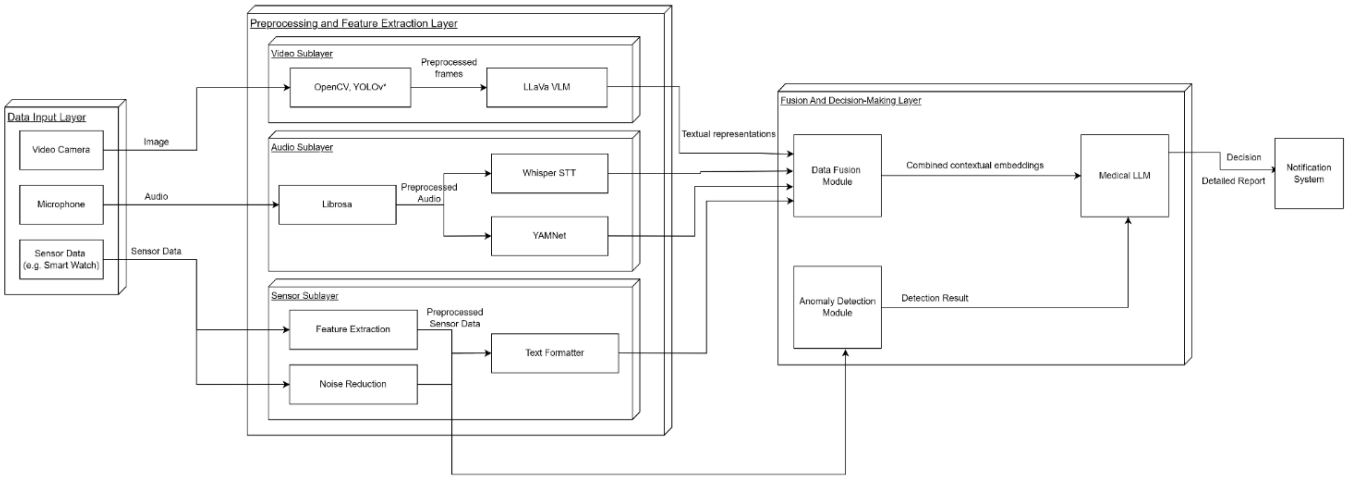

Fig. 25. Initial Design of our system as planned in Fall of 2024

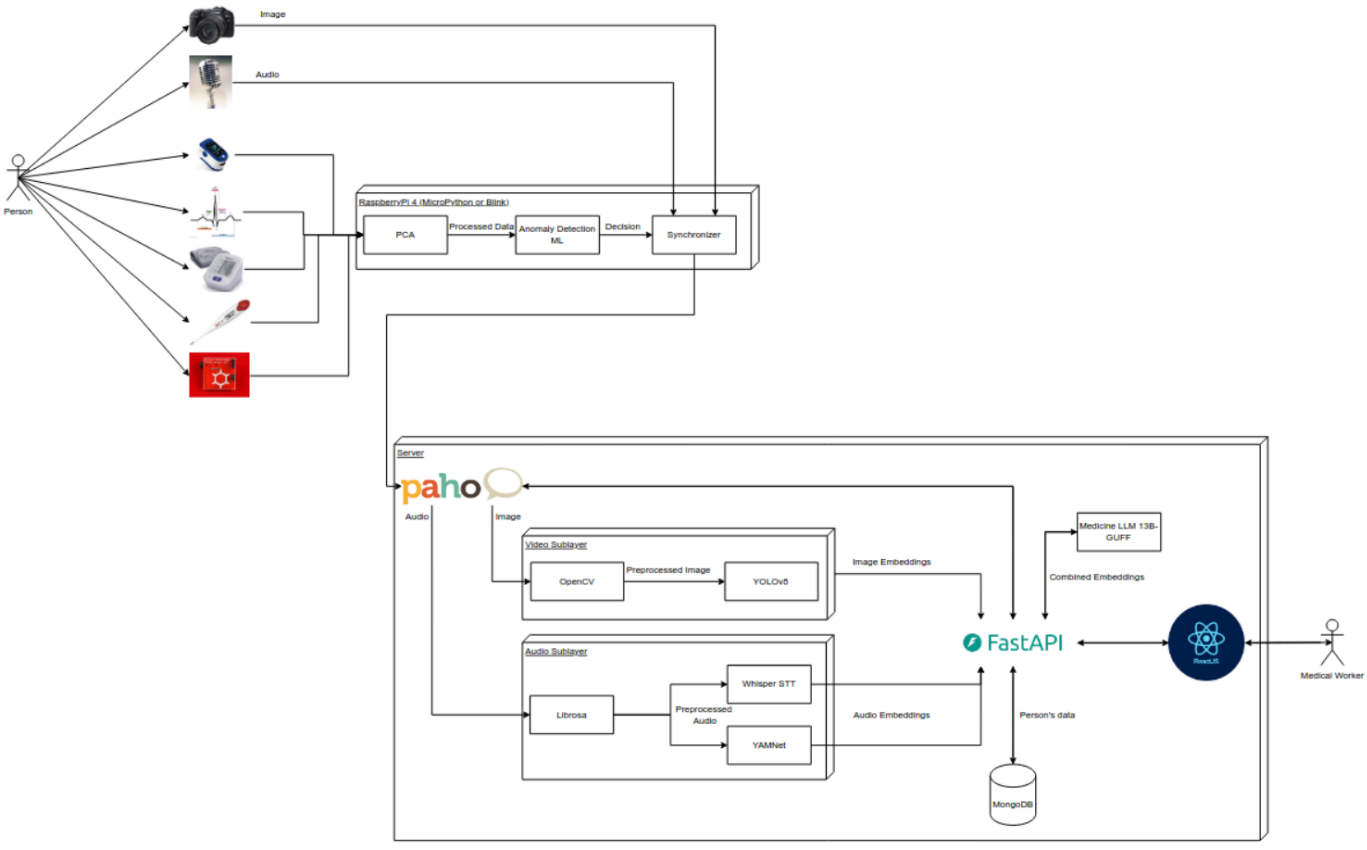

Fig. 26. Final Design of our system as planned in Fall of 2024

- 16GB microSD Card
- High-Performance Computer (Ryzen 9 7950X3D, 2x RTX 4090, 128GB RAM)

After receiving the necessary hardware, we have finalized our deliverable requirements, and proceeded to work. The work was split into 5 parts of equal importance: The Sensor readings, managed by Alim, Edge Computing, managed by Batyr, Backend Go Server, managed by Madi, Dynamo Foundation Model server, managed by Alisher and the React Frontend interface, managed by Diana. Throughout our work during the semester we have faced a lot of issues and had to decrease the grandiosity of our project. The planned Chatbot interaction was scrapped, as it would require the deployment of an additional model to the system. While sensor data was receivable, some of the hardware (namely, the ECG sensor) was faulty, which prompted us to instead opt for simulated anomalous data triggered by a press of one of the four buttons, which would still be detected by the Edge Computer as if it was real data, and processed for anomalies. While, originally, a Telegram bot was not in our plan, we needed a way to send emergency notifications to both users and doctors. Instead of spending more time developing a native application with push notification functionality, we opted to use the Telegram API, since it executed a similar role with minimal setup, and was just as effective and available as push notifications. The majority of the issues that we have encountered during the execution of the project, however, were physical. At first, the models that we envisioned to use in our high-performance server would not work well, as they would be far too heavy and take up too much video memory on our device. To account for that, we have switched to less advanced models that offer better performance in a high-traffic environment.

V. Evaluation

A. Automated Evaluation

The backend was undergoing rigorous automated testing, including a wide range of unit tests for every single function to be able to handle different types of data, go under loads and cope with the unavailability of certain modules (such as the Dynamo server and the MongoDB database). Every update of the server first required to pass these automated unit tests in order to be approved and deployed on our production server.

B. Model Inference Evaluation

The ALM, VLM and LLM models were evaluated in term of inference time. The main reason for such metric is main idea of the system - fast response time in critical situations for the patients. To perform evaluation, test audio and video datasets were collected. UCF101 dataset was used for evaluation of VLM model, Problema-Especial dataset was used for evaluation of ALM model.

As we can see in 27, UCF101 does not contain very large video samples which is suitable for our case because inference will be performed on relatively short video samples with distress. Figure 28 shows us that model inference time is very fast and does not depends much on video duration.

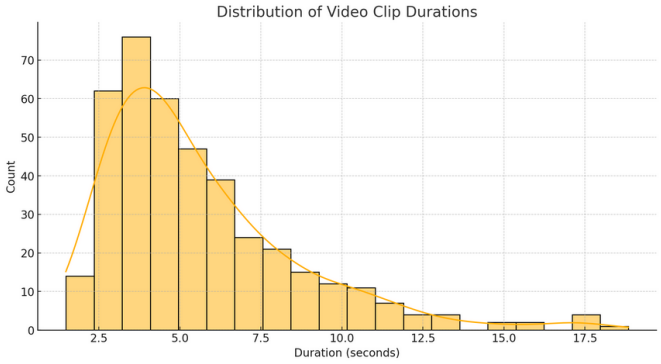

Fig. 27. UCF101 duration distribution

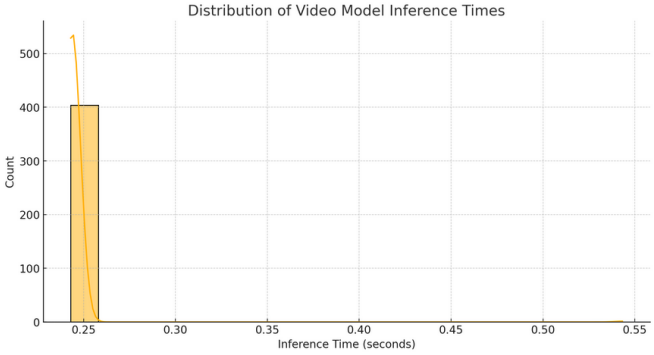

Fig. 28. VLM Model inference time distribution

Problema-Especial dataset contains various audio samples in terms of duration which we can see from Fig. 29, duration varies form 10 seconds to 2 minutes. Figure 30 shows us that model inference time is not such fast as VLM model inference time. However, this can be explained by various audio duration in test dataset.

Evaluation in terms of models' inference time showed that these models are suitable for our case because inference time

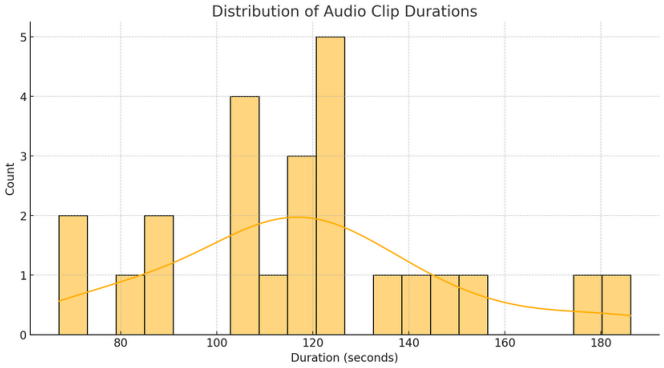

Fig. 29. Problema-Especial duration distribution

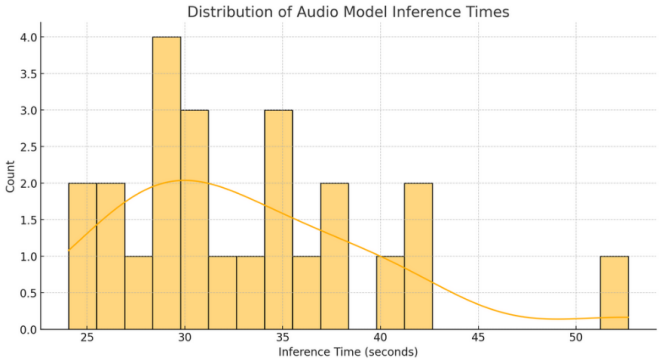

Fig. 30. ALM Model inference time distribution

on GPU is fast enough.

C. Real-Life testing

The entire pipeline was tested on live subjects in a real environment in our laboratories. While some of the hardware was not working, the proof-of-concept data from Arduino successfully functioned no matter the environmental conditions. In addition, Audio and Video data was tested on real subjects, and anomalous events (such as falling, distress, choking, etc.) was successfully detected and analyzed by the server pipeline.

VI. Conclusion and Possible Future Work

In conclusion, we can safely say that the main deliverable of our Senior Project has been completed successfully. Despite the frequent plan changes, time constraints, and the novelty of this project, we were able to complete our goal of creating an autonomous system that observes a patient and generates appropriate reports based on any anomalous event, which could later be viewed by a healthcare professional. We believe that with further work the project can be further improved to include many of the original features that we had missed or decided not to implement. One of the main and obvious areas of improvement will be the fine-tuning of the Medical LLM, which, in its current state, is prone to hallucinations. With the acquisition of proper sensors, we can greatly extend the functionality of our system due to its reliance on modularity. A big improvement would be, theoretically, if we could collect real data on the exact same sensor set up instead of relying on online datasets to detect anomalies. Although this would not offer perfect accuracy, it would greatly improve our system's performance. Finalizing the project in a state where it can be presented in a research paper would be our desired goal.

References

[1] L. P. Serrano, K. C. Maita, F. R. Avila, R. A. Torres-Guzman, J. P. Garcia, A. S. Eldaly, C. R. Haider, C. L. Felton, M. R. Paulson, M. J. Maniaci, and A. J. Forte, "Benefits and challenges of remote patient monitoring as perceived by health care practitioners: A systematic review," *The Permanente Journal*, vol. 27, no. 4, pp. 100–111, 2023.

[2] P. Jayant, E. Vincent, Mohana, M. Moharir, and A. K. A R, "Smart health monitoring and anomaly detection using internet of things (iot) and artificial intelligence (ai)," in *2024 Second International Conference on Intelligent Cyber Physical Systems and Internet of Things (ICoICI)*, p. 479–485, IEEE, Aug. 2024.

[3] A. P, "Artificial intelligence based anomaly detection in patient health monitoring using ensemble learning methods," in *Proceedings of the 1st International Conference on Artificial Intelligence, Communication, IoT, Data Engineering and Security, IACIDS 2023, 23-25 November 2023, Lavasa, Pune, India*, IACIDS, EAI, 2024.

[4] E. A. Kadir, L. D. Putri, S. L. Rosa, A. Siswanto, F. Assidiqi, and M. F. Evizal, "Anomaly detection of patient data in public hospital used internet of things sensors," in *2024 International Conference on Inventive Computation Technologies (ICICT)*, p. 1734–1739, IEEE, Apr. 2024.

[5] U. Pagallo, S. O'Sullivan, N. Nevejans, A. Holzinger, M. Friebe, F. Jeanquartier, C. Jean-Quartier, and A. Miernik, "The underuse of ai in the health sector: Opportunity costs, success stories, risks and recommendations," *Health Technology (Berlin)*, vol. 14, no. 1, pp. 1–14, 2024. Epub 2023 Dec 12.

[6] N. Nigar, "Ai in remote patient monitoring," 2024.

[7] M. B. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)." RFC 7519, May 2015.

[8] A. Sangwai, S. Sapale, S. Ghodake, and R. Jadhav, "Barricading system - system communication using grpc and protocol buffers," in *2023 5th Biennial International Conference on Nascent Technologies in Engineering (ICNTE)*, pp. 1–5, 2023.