

## Project architecture report

שמות הסטודנטים	ת.ז.
Gil Matzafi	
Daniel Michaelshvili	
קוד פרויקט	038
מנחה	Dr. Ariel Roth

## Abstract

*This project develops an innovative system for monitoring infants near hazardous objects using smart cameras integrated with machine learning. The system sends real-time alerts to parents and enables monitoring and customization of hazardous objects. It provides an easy-to-use, personalized solution with a strong focus on safety and peace of mind.*

## 1. Introduction

*The rapid advancement of machine learning and computer vision technologies has opened new possibilities for enhancing child safety. This project explores the design and development of a dual-camera monitoring system for infants, equipped with real-time object detection capabilities. The system combines a **wearable head-mounted camera**, capturing the infant's unique point of view, with a **static camera** positioned in the environment to provide a broader context of the surroundings.*

*By integrating these two complementary perspectives, the system improves the accuracy of hazard detection, provides timely alerts to caregivers, and ensures a safer environment through an intuitive and reliable solution. This multi-angle approach allows for more comprehensive supervision, especially during dynamic activities or when the infant moves out of the wearable camera's field of view.*

## 2. Problem Statement

*Infants are naturally curious and often interact with their surroundings in unpredictable ways. However, a momentary lack of supervision—especially in environments containing*

*potentially hazardous objects—can result in severe injuries. Traditional baby monitors provide only passive audio or video feeds, lacking the intelligence to recognize and warn about imminent threats. They do not identify dangerous objects, offer no context-aware alerts, and rely entirely on the caregiver's constant attention.*

*This gap creates a critical vulnerability: during routine tasks such as working from home or household chores, caregivers may miss warning signs of danger. There is a clear need for a smarter, real-time monitoring solution that not only sees but understands the infant's environment—and acts accordingly.*

### **3. Objectives**

1. ***Design and implement an intelligent dual-camera monitoring system,*** combining a wearable head-mounted camera and a static room camera to enable real-time detection of hazardous objects from multiple perspectives.
2. ***Develop and integrate machine learning algorithms*** capable of accurately identifying dangerous objects and scenarios in the infant's environment.
3. ***Build an intuitive, user-friendly mobile application*** that allows caregivers to configure settings, manage hazard classes, and receive alerts with minimal effort.
4. ***Utilize cloud-based infrastructure*** for efficient data processing, centralized storage, and long-term alert history tracking.
5. ***Ensure user data privacy and system security*** through robust encryption methods and access control mechanisms.

## 4. User Stories

### ***User Account Management:***

**1. As a user,** I want to easily create an account in the app,

**so that** I can start using the monitoring system and receive personalized alerts.

subtasks:

- Design and implement the user registration screen in the app.
- Develop backend API endpoints for account creation.
- Integrate the registration screen with the server endpoints and test the flow.

**2. As a user,** I want a simple interface in the app to log in and log out of my account,

**so that** I can securely access and manage my baby monitoring data.

subtasks:

- Implement a login screen and logout button in the app.
- Create server-side authentication endpoints (login, logout, tokens creation and validation).
- Ensure secure token storage on the client side.

**3. As a user,** I want an easy way to delete my account and all its data from the app,

**so that** I can have full control over my personal information and privacy.

subtasks:

- Create user account deletion confirmation flow in the app.
- Implement secure server-side endpoint for deleting a user and related data.

**4. As a user,** I want the ability to edit my account details through a user-friendly interface in the app,

**so that** I can keep my information accurate and up to date.

subtasks:

- Design profile editing screen for account details.
- Implement an 'update account' endpoint on the server.

## ***Baby Profiles Management:***

*5. As a user, I want a convenient interface in the app to create, edit, delete, and view my baby profiles,*

*so that I can easily manage monitoring settings for each of my children.*

### *subtasks:*

- *Implement UI for managing baby profiles (create/edit/delete/view).*
- *Develop backend CRUD endpoints for baby profiles.*
- *Wire up the frontend to support profile operations with the backend.*

## ***Machine Learning Model Training Interface for Danger Detection System:***

*6. As a user, I want the ability to train two different models for each baby profile, a "head camera" model and a "static camera" model,*

*so that the system can accurately detect hazards from both the infant's perspective and the room's overall view.*

### *subtasks:*

- *Allow selection of model type (head/static) before starting to work in the training model screen.*
- *Build backend logic to distinguish between the two model types per profile.*
- *Test model training separately for each camera type.*

**7. As a user,** I want a simple interface to add, edit, delete, and view classes in any model I wish to train, with the ability to upload images/videos and label them,

**so that** the model can be trained properly.

subtasks:

- Create UI for managing model classes (add/edit/delete/view).
- Implement image/video upload for class examples and enable labeling.

**8. As a user,** I want to define the risk level for each class in each model,

**so that** the detection alerts during system runtime reflect the severity of the identified hazard and help me prioritize my response accordingly.

subtasks:

- Allow risk level selection in class creation/edit forms.
- Update model metadata structure to store class risk levels.
- Reflect risk level in detection alert UI.

**9. As a user,** I want to see the current training status of a model, the date of its last training, and receive a notification when training is complete,

**so that** I can stay informed and know exactly when the model is ready to be used.

subtasks:

- Add training status indicator and last trained date to UI per model.
- Track training progress on backend and expose via endpoint.
- Implement push notification to the user when model training is done.

**10. As a system administrator,** I want model training to run on Google Cloud and have the trained model automatically downloaded and organized on the server,

**so that** system performance is maintained.

subtasks:

- Trigger model training on Google Cloud using cloud function.
- Automatically download the resulting .pt model to the server when training completes.
- Organize model files in the server file system by profile and type.

## ***Camera Integration with the App:***

*11. As a user, I want a simple interface in the app to connect or disconnect cameras to or from specific models,*

*so that I can control which camera feeds are used for danger detection.*

### *subtasks:*

- *Add interface to associate/disconnect camera to/from model in app.*
- *Build server logic to wait for and store camera IP addresses.*

*12. As a user, I want to watch a live video stream from each connected camera (head or static) for any baby profile and model,*

*so that I can monitor the environment before, during, or after the danger detection system is running.*

### *subtasks:*

- *Implement live video streaming screen for both head and static cameras for each baby profile.*
- *Enable viewing during and outside detection runtime.*
- *enable stream switching per baby profile.*

## ***Running the Danger Detection System:***

*13. As a user, I want to easily start and stop the danger detection system for any baby profile,*

*so that I can control when monitoring is active based on my needs.*

### *subtasks:*

- *Add button to start/stop danger detection for a selected baby profile.*
- *Connect button to backend monitoring control.*
- *Display current system state clearly in the UI.*

*14. As a user, I want to receive an alert on my connected devices when a hazardous object is detected by the model during system runtime,*

*so that I can respond quickly to potential dangers.*

subtasks:

- *Set up real-time alert delivery (e.g., via WebSocket or push notification).*
- *Ensure alerts are delivered across the user devices.*
- *Show detected class and context in alert content.*

*15. As a user, I want a clean screen displaying all detected classes, organized by camera, baby profile, and time,*

*so that I can easily review and understand past detections.*

subtasks:

- *Build a live alerts list on the home screen with sorting by baby, camera, and time.*

*16. As a user, I want to receive an alert if a camera disconnects unexpectedly while the system is running,*

*so that I can take immediate action to restore monitoring.*

subtasks:

- *Detect camera disconnection on the server during system runtime.*
- *Send a disconnection alert to the user immediately.*

## ***Alert History Screen:***

*17. As a user, I want to view a history screen that displays all alerts I've received, organized by baby profile,*

*so that I can track past hazards and monitor trends over time.*

subtasks:

- *Create an alert history screen filtered by baby profile.*
- *Display alert data in reverse chronological order.*
- *Include filters/search to help users analyze past alerts.*

## Sample Test Code for User Stories

### # User Story 1 : user creation

```
def test_register_user():  
  
    url = "http://localhost:8000/auth/register"  
  
    data = {"username": "test123", "email": "test123@example.com", "password": "test123"}  
  
    response = requests.post(url, json=data)  
  
    print(response)  
  
    assert response.status_code == 201
```

### # User Story 2: user login and logout

```
def test_login_logout_user():  
  
    login_url = "http://localhost:8000/auth/login"  
  
    logout_url = "http://localhost:8000/auth/logout"  
  
    credentials = {"username": "test123", "password": "test123"}  
  
    login_res = requests.post(login_url, json=credentials)  
  
    print(login_res.json())  
  
    assert login_res.status_code == 200  
  
    token = login_res.json()["access_token"]  
  
    headers = {"Authorization": f"Bearer {token}"}
```

```
    logout_res = requests.post(logout_url, json={"baby_profile_ids": [], "fcm_token":  
"mock_token"}, headers=headers)  
  
    print(logout_res.json())  
  
    assert logout_res.status_code == 200
```



### **# User Story 3: delete a user**

```
def test_delete_account():  
    login_url = "http://localhost:8000/auth/login"  
    login_credentials = {"username": "test123", "password": "test123"}  
    login_res = requests.post(login_url, json=login_credentials)  
    print(login_res.json())  
    assert login_res.status_code == 200  
    token = login_res.json()["access_token"]  
    url = "http://localhost:8000/user/delete"  
    headers = {"Authorization": f"Bearer {token}"}  
    res = requests.delete(url, headers=headers)  
    assert res.status_code == 200
```

## **# User Story 5: baby profile creation**

```
def test_create_baby_profile():  
  
    login_url = "http://localhost:8000/auth/login"  
  
    login_credentials = {"username": "test123", "password": "test123"}  
  
    login_res = requests.post(login_url, json=login_credentials)  
  
    print(login_res.json())  
  
    assert login_res.status_code == 200  
  
    token = login_res.json()["access_token"]  
  
    url = "http://localhost:8000/baby_profiles/"  
  
    headers = {"Authorization": f"Bearer {token}"}  
  
    data = {"name": "baby"}  
  
    res = requests.post(url, headers=headers, json=data)  
  
    print(res.json())  
  
    assert res.status_code == 200
```

## **# User Story 17: GET detection results of a user**

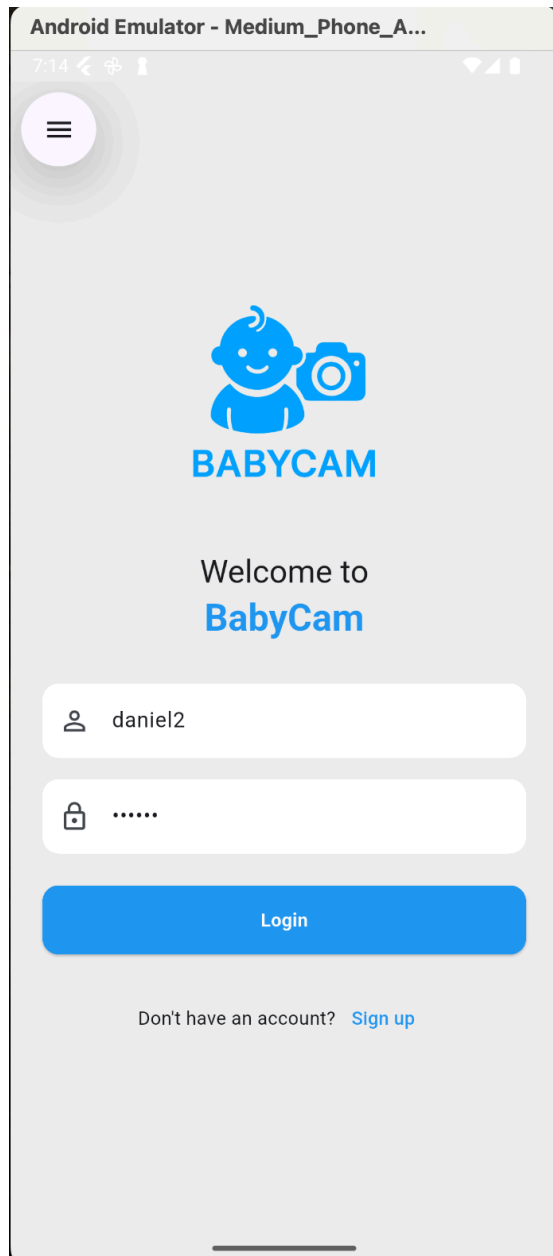
```
def test_alert_history():  
  
    login_url = "http://localhost:8000/auth/login"  
  
    login_credentials = {"username": "test123", "password": "test123"}  
  
    login_res = requests.post(login_url, json=login_credentials)  
  
    print(login_res.json())  
  
    assert login_res.status_code == 200  
  
    token = login_res.json()["access_token"]  
  
    url = f"http://localhost:8000/detection_results/my"  
  
    headers = {"Authorization": f"Bearer {token}"}  
  
    res = requests.get(url, headers=headers)
```

```
assert res.status_code == 200
```

```
assert isinstance(res.json(), list)
```

## 5. Application Screenshots

### 1. Login Screen




## 2. Registration Screen

Android Emulator - Medium\_Phone\_A...

7:13

☰



**BABYCAM**

Welcome to  
**BabyCam**

**Create Account**

✉ gil@gmail.com

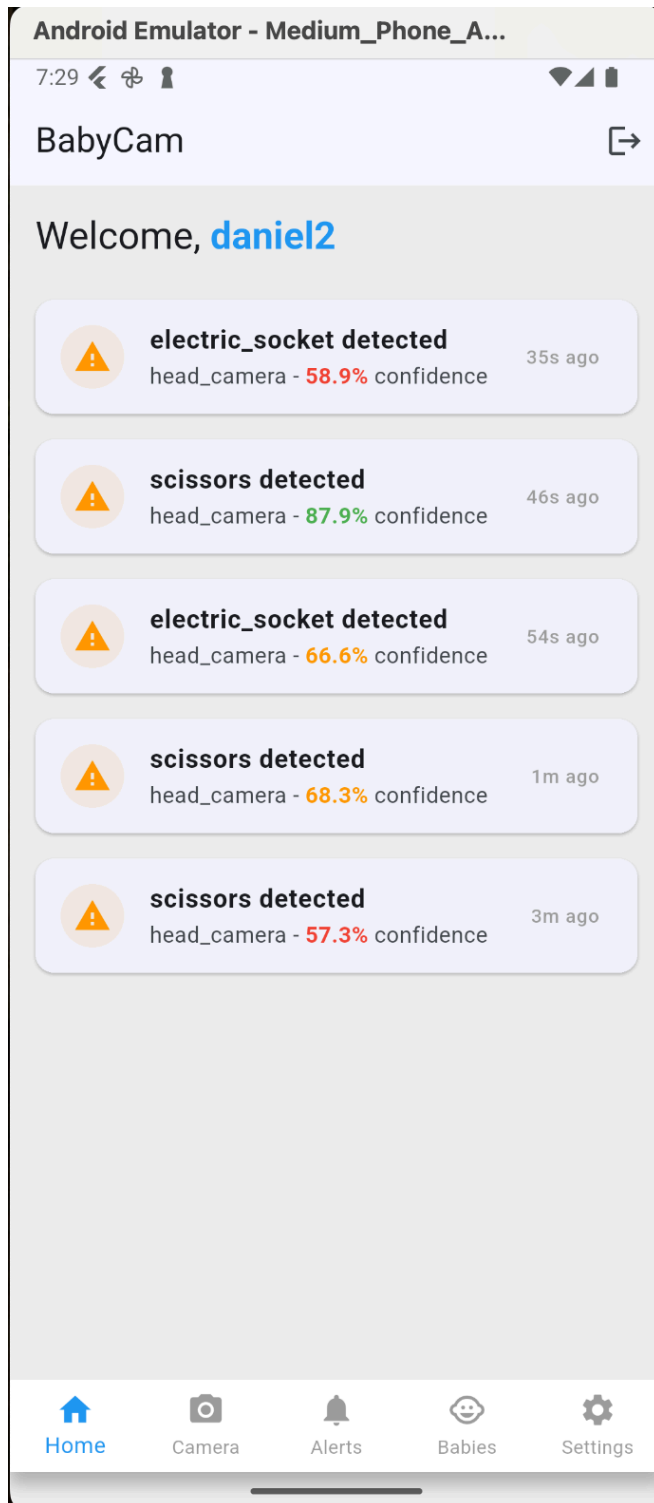
👤 gil

🔒 ..... 🔒

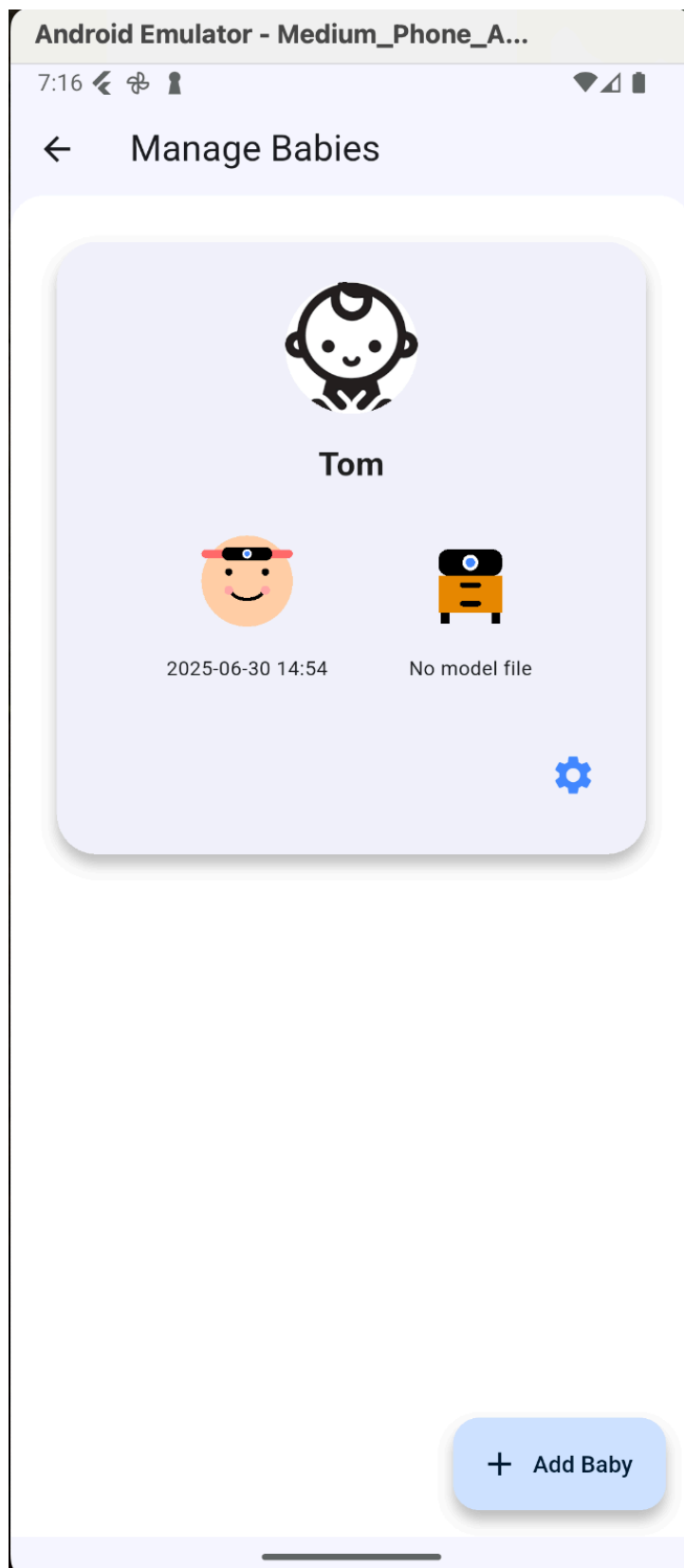
🔒 ..... 🔒

**Sign Up**

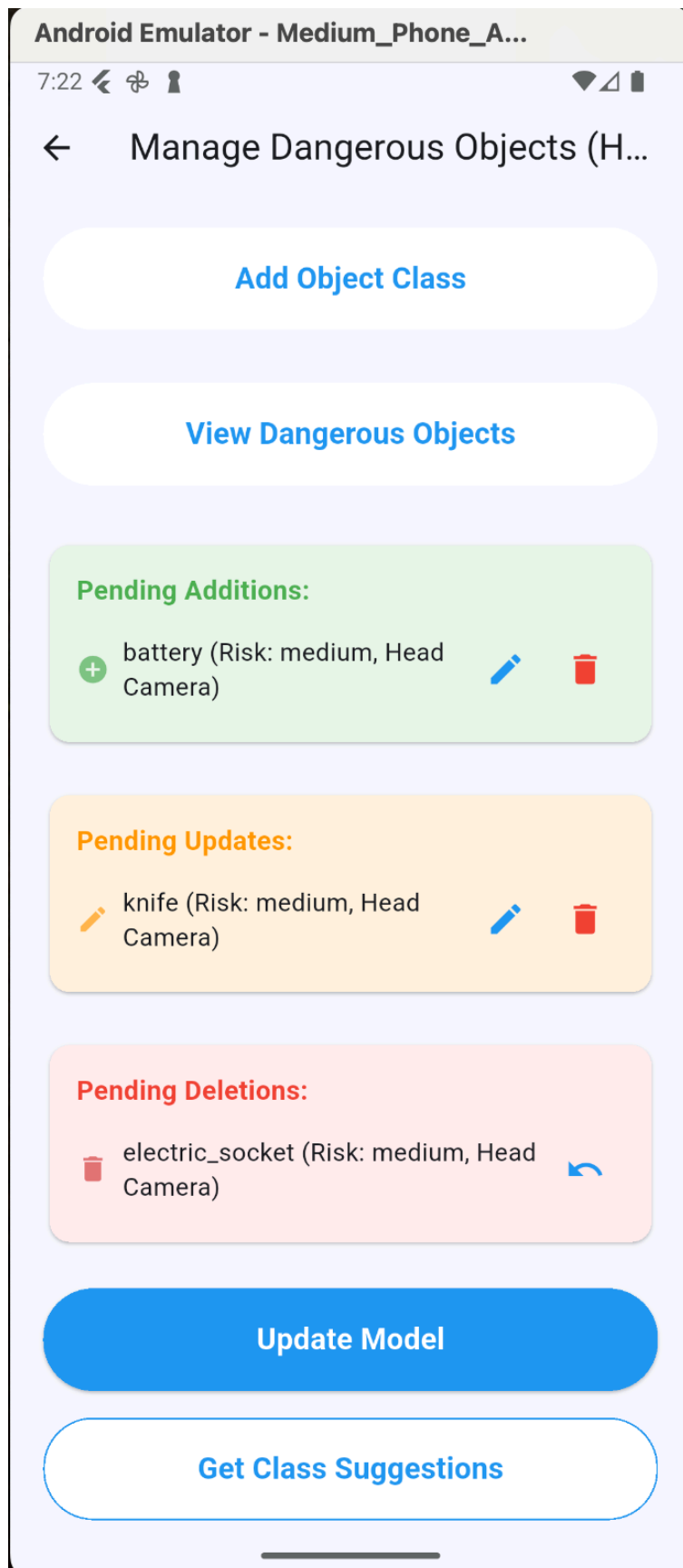
### 3. Home Screen



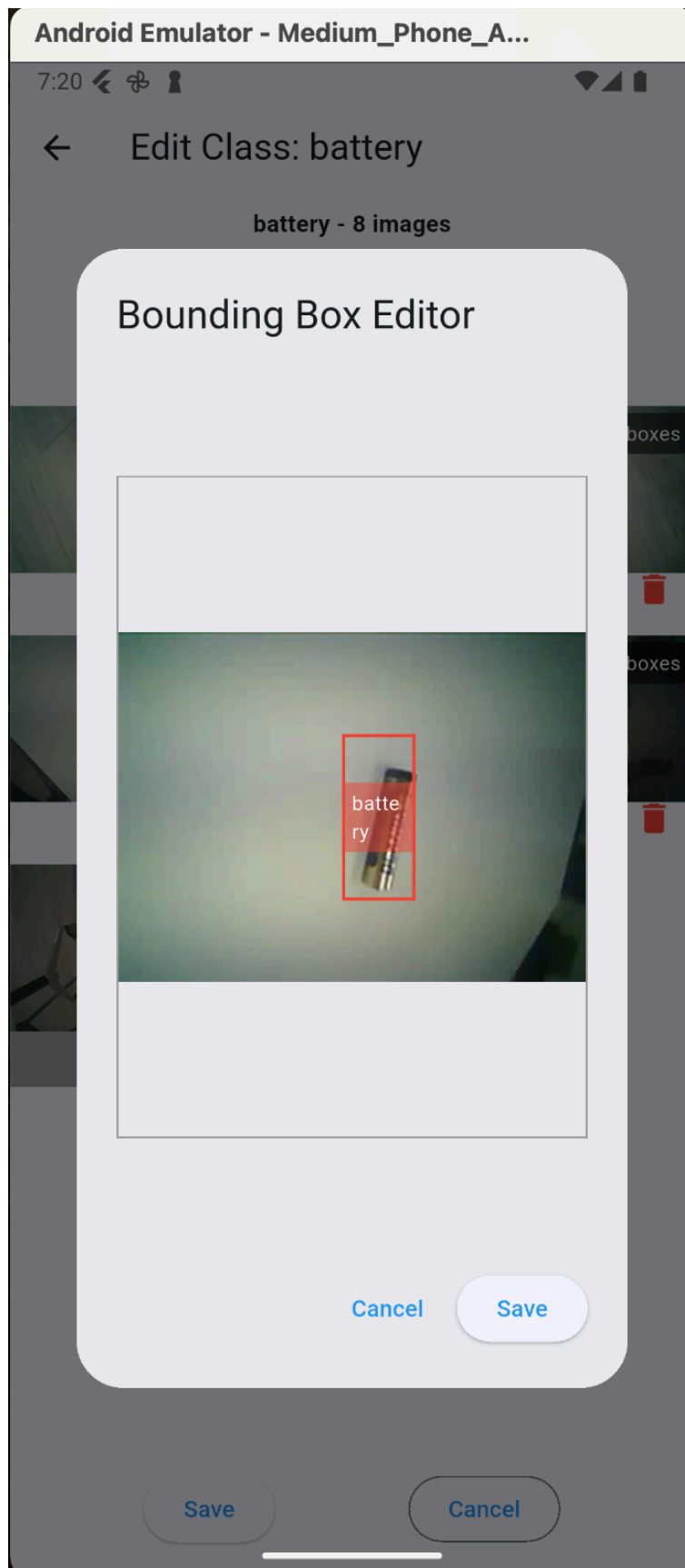
#### 4. Baby Profiles Screen



## 5. Model Edit Screen

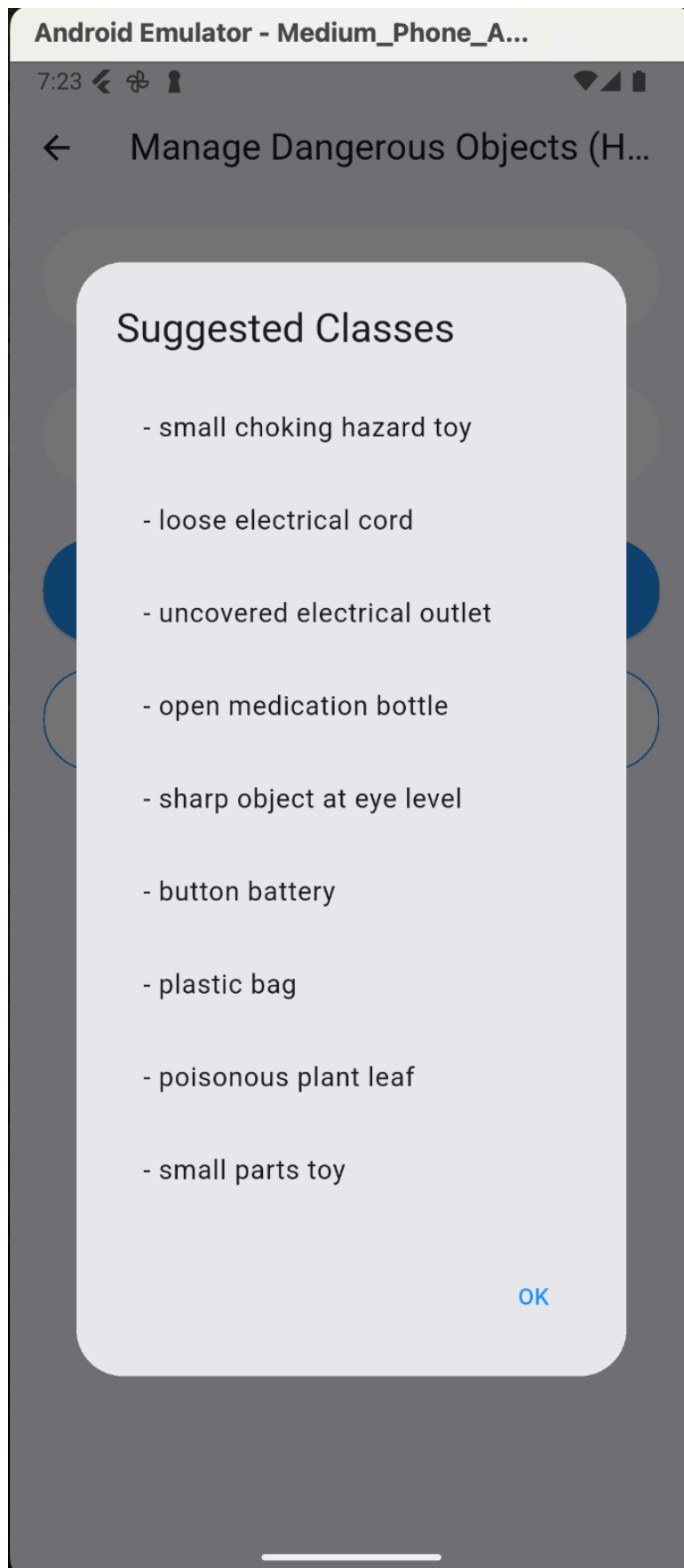


## 6. Picture Labeling Screen

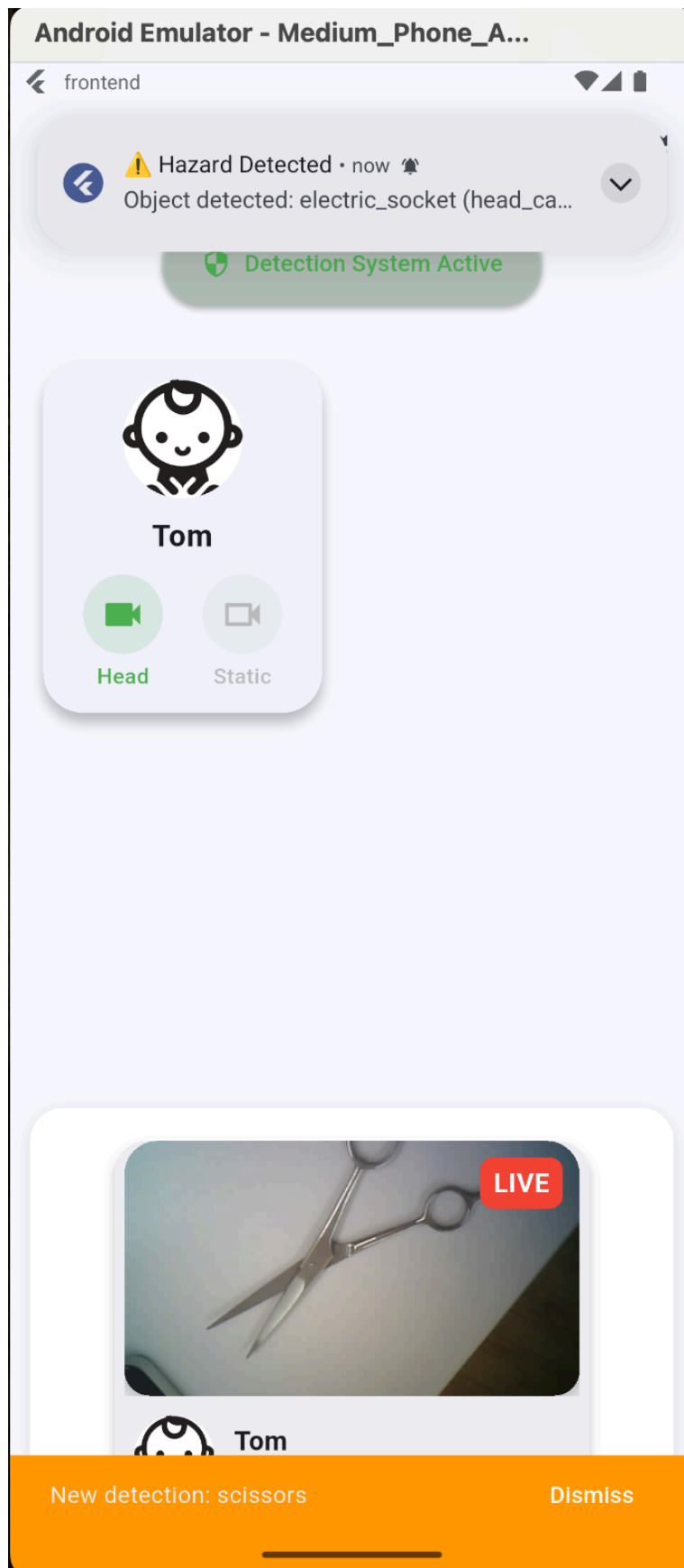




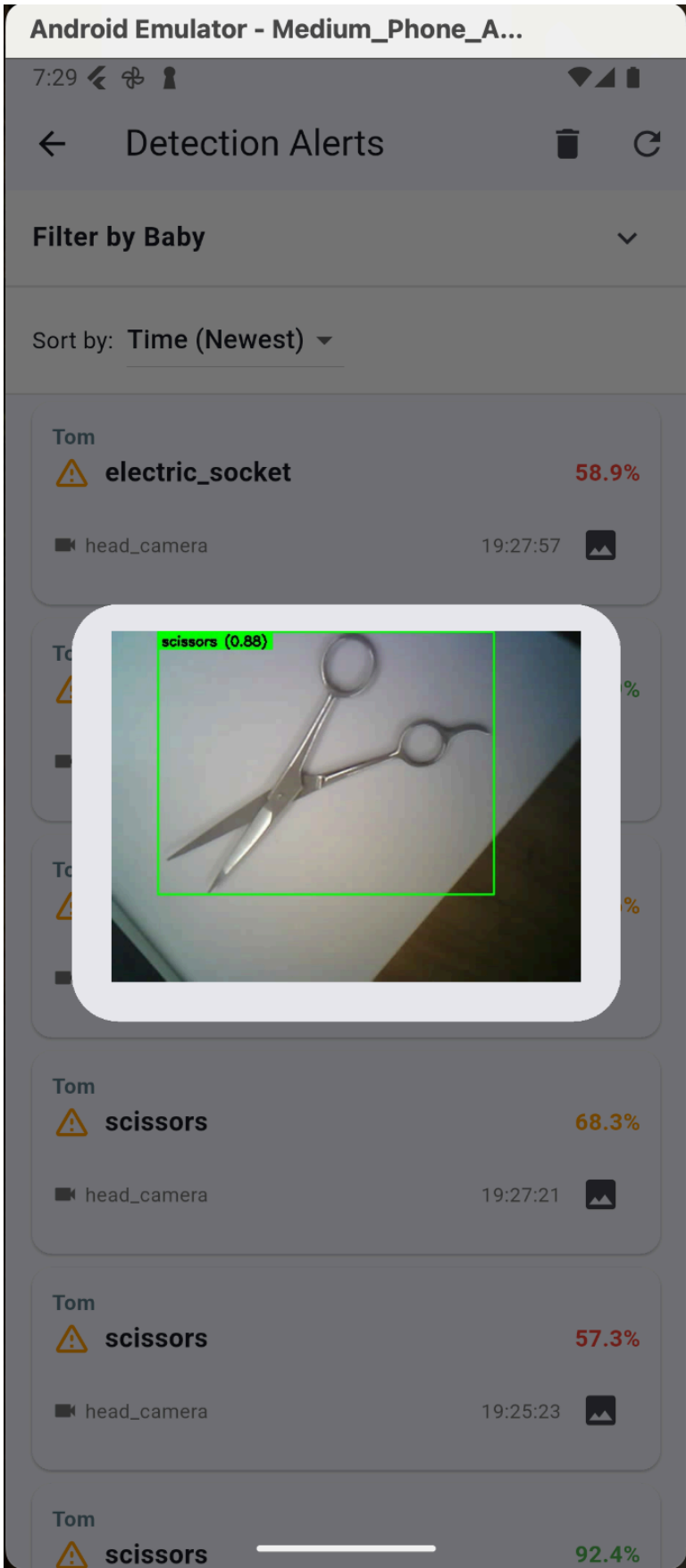
## 7. Classes Suggestion Screen



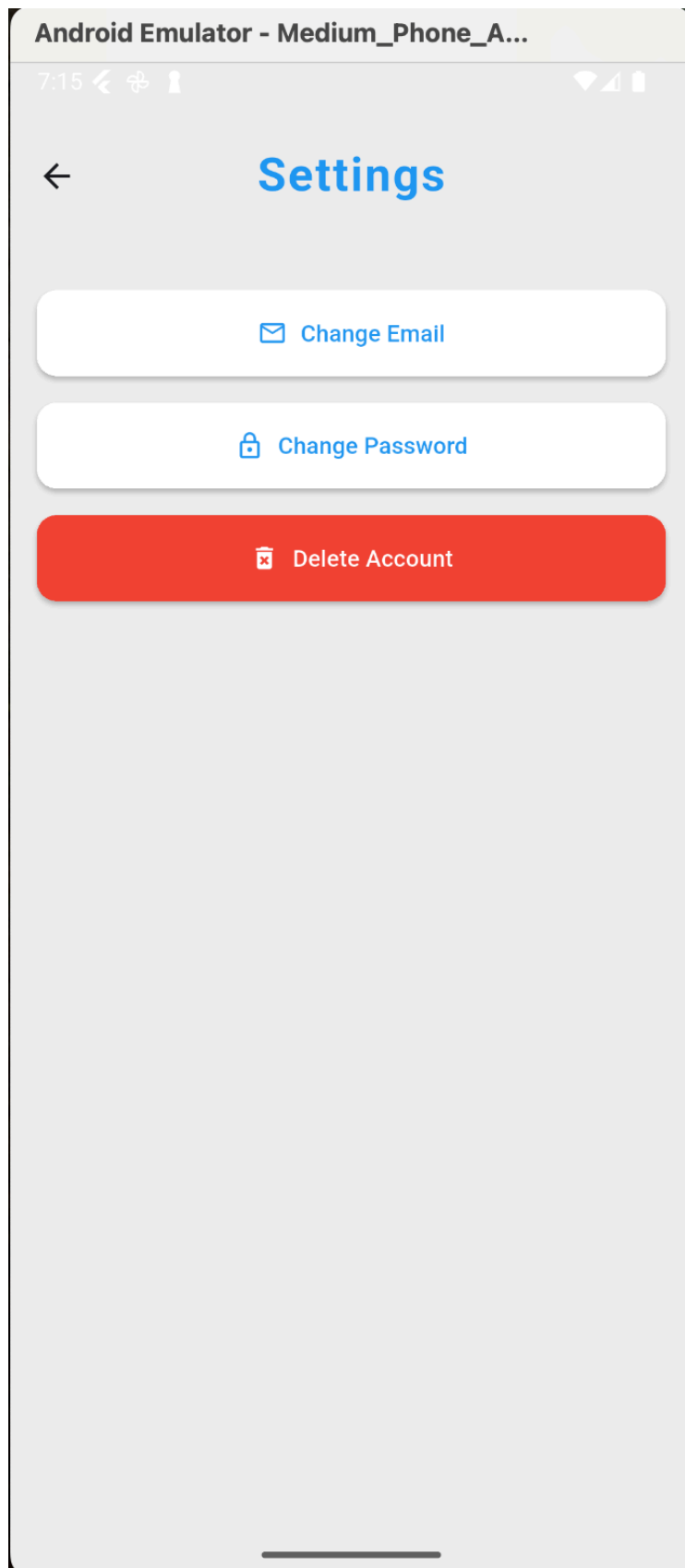
## 8. Camera Screen



9. Detection Alerts History Screen



## 10. Detection Alerts History Screen



## 6. Typical User Flows

### **1. User Flow: Configuring Model Classes (Hazardous Objects)**

1. The user navigates to the baby profiles screen.
2. He taps on the head/static camera button for the baby whose model he wants to update.
3. He taps the "Add Object Class" button, enters the name of the new class, and selects its risk level.
4. He uploads images on the screen and labels them by drawing bounding boxes.
5. He saves the changes.
6. He taps on "Update Model".

### **2. User Flow: Connecting a Camera**

1. The user taps on his baby head/static camera button in the camera screen
2. The user powers on an ESP32-CAM device.
3. The app sends a connection request to the backend to wait for a camera IP.
4. The ESP32-CAM sends its IP address to the server.
5. The server verifies camera availability and saves the IP under the correct baby profile and camera type.
6. The app updates the status of the camera as "Connected".

### **3. User Flow: Activating the Danger Detection System**

1. The user opens the app and logs in.
2. The user connects the camera (as described in User Flow 2)
3. The video stream gets displayed on the screen.
4. The user taps on the "Activate Detection System" button.
5. The app sends a request to the server to activate the danger detection system for the selected baby profile and camera type(s).
6. The server verifies that a camera is connected and begins processing video feed(s) with the appropriate model.
7. Once activated, the app indicates the system is monitoring.
8. If a hazardous object is detected, the user receives a real-time alert on all connected devices.

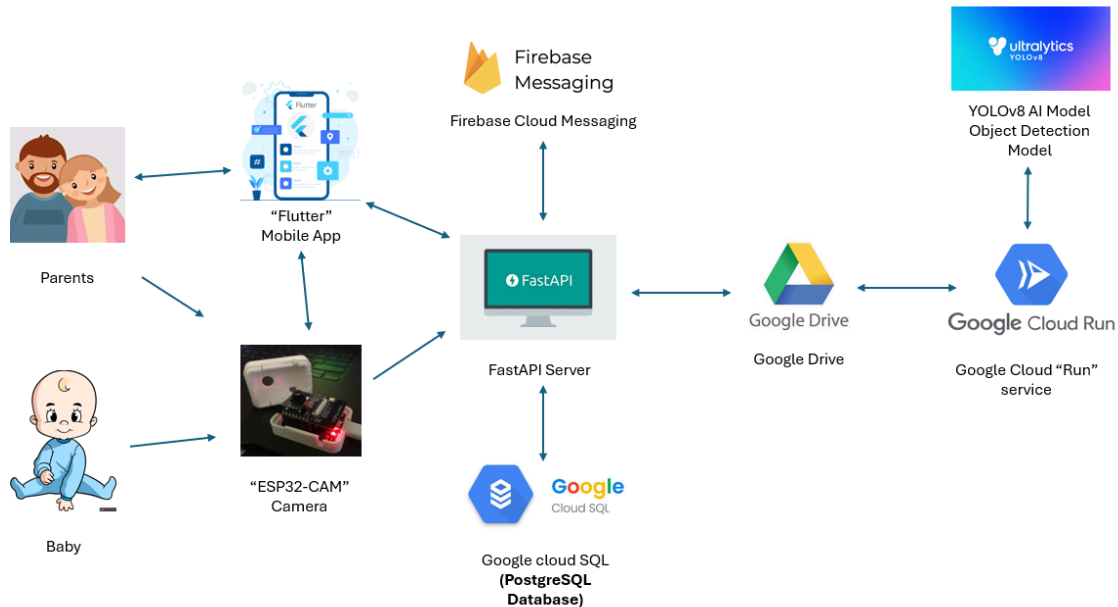
#### **4. User Flow: Stopping the Danger Detection System**

1. *The user taps the "Activate Detection System" button in the camera screen (to turn it off).*
2. *The app sends a request to the server to stop detection for the selected camera(s).*
3. *The system stops processing frames and updates the UI to reflect that monitoring has ended.*

#### **5. User Flow: Viewing Alert History**

1. *The user navigates to the "Alert History" section in the app.*
2. *The screen displays a list of all past alerts grouped by baby profile.*
3. *The user can filter alerts by baby profiles.*
4. *Tapping an alert entry shows full details including timestamp, object class, and severity.*
5. *The user can view the detection picture by tapping on the picture button of the alert.*

## 7. Project Entity Diagram



## System Components and Interactions

This system architecture diagram illustrates the core components of the BabyCam project and how they interact to provide real-time infant monitoring, alert delivery, and AI-powered hazard detection.

### Components Description

- **Parents:** The end users who receive alerts, view live streams, and manage baby profiles through the mobile app.
- **Baby:** The monitored subject, equipped with a wearable camera (ESP32-CAM).
- **ESP32-CAM Camera:** A low-cost microcontroller with a camera module, responsible for streaming real-time video to the FastAPI server.
- **Flutter Mobile App:** A cross-platform client application that allows parents to interact with the system. It provides:
  - Baby profile management

- *Class labeling and dataset upload*
- *Viewing live streams*
- *Receiving alerts via push notifications*
- ***FastAPI Server:*** *The central backend component handling:*
  - *User authentication and baby profile management*
  - *Receiving camera streams*
  - *Initiating detection services*
  - *Managing training triggers and model status*
  - *Communicating with external services like Google Cloud and Firebase*
- ***Google Cloud SQL:*** *Stores structured backend data, such as users, baby profiles, detected hazards, and class definitions using a PostgreSQL database.*
- ***Google Drive:*** *Used as cloud storage for dataset ZIPs and trained model files during model training workflows.*
- ***Google Cloud Run:*** *Executes the training script in the cloud environment, triggered by the backend via HTTP.*
- ***YOLOv8 AI Model:*** *The object detection model used to identify hazardous objects based on the trained dataset.*
- ***Firebase Cloud Messaging (FCM):*** *Sends real-time push notifications to the mobile app when hazards are detected or system events occur.*

---

### **Main Component Interactions**

1. ***Mobile App ↔ FastAPI Server:*** *The app sends requests (e.g., login, update classes, start detection), while the server responds with data or actions.*
2. ***ESP32-CAM → FastAPI Server:*** *Streams video via HTTP; server processes it using OpenCV and YOLOv8.*



3. **FastAPI Server** ↔ **Cloud SQL**: All persistent data (users, baby profiles, detection results and etc..) are stored and queried here.
4. **FastAPI Server** ↔ **Google Drive**: Used for uploading datasets and downloading trained model files during the model training pipeline.
5. **FastAPI Server** → **Cloud Run**: Triggers model training job with references to the uploaded dataset.
6. **Cloud Run** → **YOLOv8**: Executes the training process and saves the **.pt** model file to Google Drive.
7. **FastAPI Server** → **Firebase Messaging**: Sends real-time alerts (e.g., hazard detection or camera disconnection) to the mobile app via FCM.
8. **Mobile App** ← **Firebase**: Receives and displays alerts to users.

## 8. Implementation

### 8.1 Technologies Used

- **Backend:**  
*Python, FastAPI – provides RESTful APIs for authentication, baby profile management, model training triggers, real-time detection control, and integration with external services.*
- **Frontend:**  
*Flutter – cross-platform mobile application used by parents and caregivers to manage the system, configure baby profiles, label training data, view live video streams, and receive alerts.*
- **Database:**  
*Google Cloud SQL (PostgreSQL) – stores structured data such as users, baby profiles, object classes, detection events, and FCM tokens.*
- **Cloud Services:**
  - **Google Cloud Run** – runs the training pipeline for YOLOv8 models in response to backend triggers.

- **Google Drive API** – used to exchange training datasets and model files between the server and cloud jobs.
  - **Firestore Cloud Messaging (FCM)** – delivers real-time alerts (e.g., hazard detection or camera disconnection) to all connected user devices.
  - **AI Model:**  
YOLOv8 (Ultralytics) – custom-trained object detection model used to identify hazardous objects in live camera feeds.
  - **Smart Cameras:**  
ESP32-CAM – low-cost Wi-Fi camera modules used in both head-mounted and static configurations.  
The implementation was based on the open-source project [esp32-cam-mjpeg-multiclient](https://github.com/arkhipenko/esp32-cam-mjpeg-multiclient), which supports stable MJPEG streaming to multiple clients simultaneously.  
  
“<https://github.com/arkhipenko/esp32-cam-mjpeg-multiclient>”
- 

## **Development Tools & AI Assistance**

Throughout the development process, we made extensive use of the following tools and platforms:

- **Git & GitHub** – for version control, collaboration, and managing project repositories.
- **GitHub Copilot** – to accelerate development with AI-assisted code suggestions.
- **ChatGPT** – used as an AI assistant to plan architectural decisions, debug errors, and generate code examples and documentation.
- **Cursor – The AI Code Editor** – provided an AI-enhanced coding environment, integrated with ChatGPT, to streamline development and refactoring directly within the editor.