EE782 Programming Assignment 2: AI Guard Agent

Soumyadeep (21D070075) & Washim Saahil (25D1626)

October 15, 2025

Abstract

This report details the design, implementation, and evaluation of an AI Room Guard Agent, developed for the Advanced Machine Learning (EE782) course. The agent utilizes a laptop's webcam and microphone to monitor a room, activating on a spoken command. It leverages pre-trained models for face recognition, speech-to-text, text-to-speech, and large language model (LLM) based dialogue to create a cohesive security system. The system can distinguish between trusted and unknown individuals, engaging in a multi-level escalating conversation with potential intruders to deter unauthorized access. This document covers the system's architecture, the technical challenges overcome during integration, ethical considerations, and instructions for operation.

1 System Architecture

The AI Guard Agent is designed as an event-driven system that integrates multiple AI modalities into a single, responsive pipeline. The architecture is built around a central state machine that manages whether the guard is active and the current escalation level for any detected intruder.

1.1 Core Components

The system is composed of the following key modules:

- **Input Sensors:** A standard laptop webcam (via OpenCV) and microphone (via SpeechRecognition) serve as the primary inputs for video and audio data.
- **Face Recognition Engine:** Utilizes the **DeepFace** library with the lightweight **SFace** model and the fast **yolov8** detector. This module is responsible for identifying faces and comparing them against a pre-enrolled database of trusted individuals based cosine similarity on embeddings.
- **Speech Processing Engine:**
  - **ASR (Speech-to-Text):** Employs the `speech_recognition` library, using Google's Web Speech API for recognizing both activation/deactivation commands and intruder responses.
  - **TTS (Text-to-Speech):** Uses the `gTTS` library to provide verbal feedback and warnings, creating a more interactive experience.
- **LLM Dialogue Handler:** Integrated with **Google Gemini**, this module judges the trustworthiness of an intruder's spoken responses. It uses carefully crafted prompts to guide the LLM's analysis and determine the next step in the escalation logic.
- **State Management & Logic:** A central Python script manages the application state (e.g., `is_guard_active`, `escalation_level`) and orchestrates the interaction between all other components.

1.2 Data Flow and Pipeline

The operational flow is illustrated in the diagram below. The system runs two parallel processes:

- a background thread for command listening
- a main thread for video processing

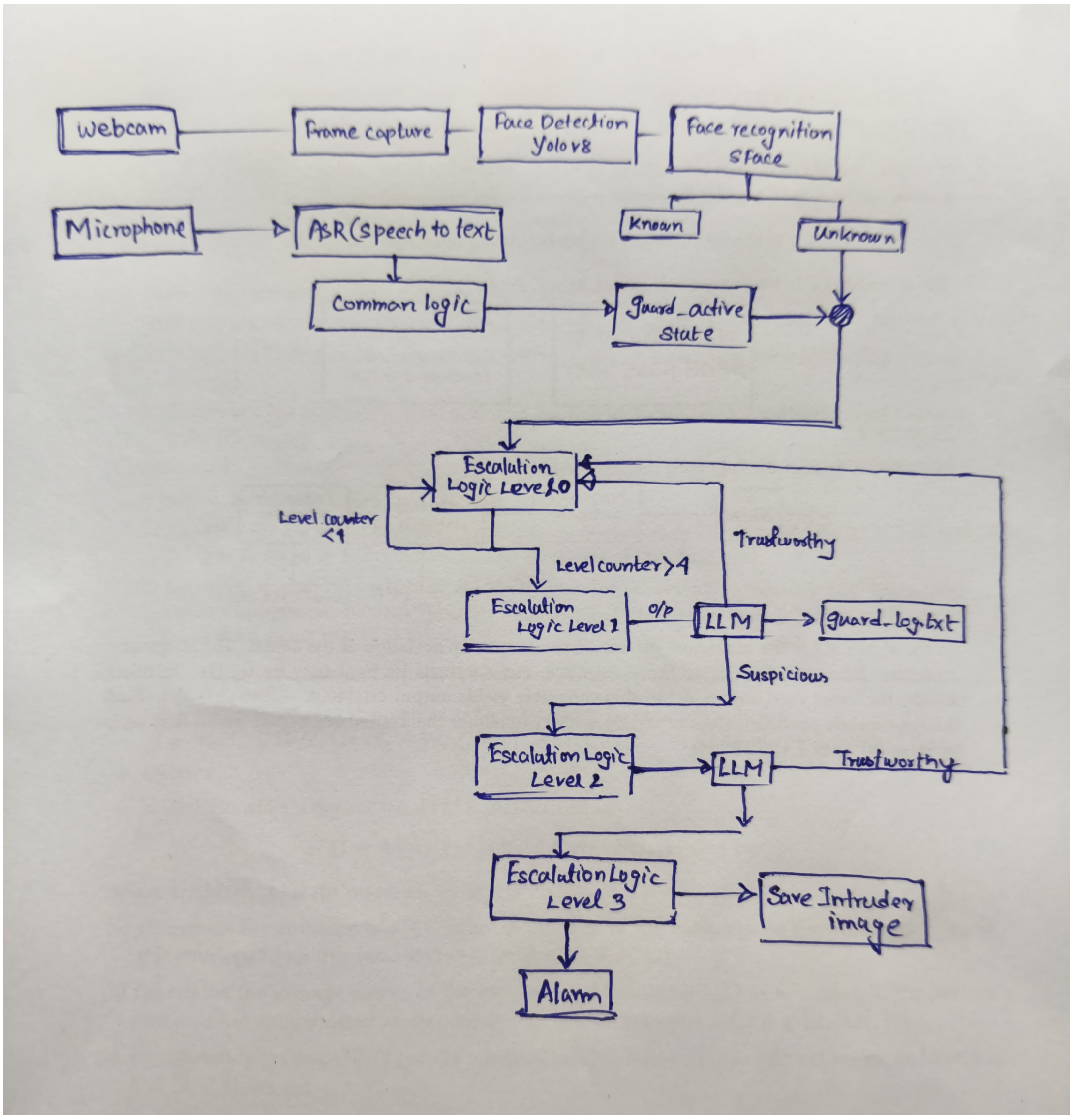

Figure 1: System Architecture Diagram. A background thread handles voice commands, while the main loop processes video and manages the escalation dialogue with intruders.

The pipeline is as follows:

1. A background listener constantly waits for the owner's activation command (e.g., "guard my room").
2. Once activated, the main loop begins capturing video frames.
3. Each frame is processed to detect faces. If a face is found, its embedding is generated and compared to the enrolled "prototypes" of trusted users.
4. If an "Unknown" face is detected, the **Escalation Dialogue Handler** is triggered.
5. The agent issues a Level 1 challenge. The intruder's response is captured in a log file (**gurad_log.txt**) and sent to the Gemini LLM for judgment.

6. Based on the LLM's verdict, the system either de-escalates or proceeds to Level 2 (Verification) and subsequently Level 3 (Alert), where an alarm is sounded and a photo of the intruder is saved.

2 Integration Challenges and Solutions

Integrating multiple real-time AI systems presented several technical challenges.

- **Challenge: Real-time Performance Lag.**
  - **Problem:** Initial tests with the `ArcFace` model and `retinaface` detector were too slow, causing a significant lag in the video feed. Running face recognition on every single frame created a CPU bottleneck.
  - **Solution:** We implemented two key **optimizations**. First, we switched to the much faster `SFace` model and `yolov8` detector. Second, we introduced frame skipping, where face recognition is only performed every N-th frame (`PROCESS_EVERY_N_FRAMES`). This drastically reduced the computational load while maintaining effective monitoring.
- **Challenge: Concurrent Audio Listening and Video Processing.**
  - **Problem:** Standard speech recognition functions are "blocking," meaning they pause the entire program while listening for audio. This caused the video feed to freeze, making the guard useless.
  - **Solution:** We implemented the `recognizer.listen_in_background()` function. This runs the command listener on a separate, non-blocking thread, allowing the main loop to process video frames smoothly and without interruption.
- **Challenge: Unreliable Face Detection.**
  - **Problem:** The initial implementation sometimes generated face embeddings for images with no faces, likely due to false positives from the detector.
  - **Solution:** We re-architected the embedding function into a robust two-step process. First, we use `DeepFace.extract_faces()` to explicitly find and crop a face. Only if this step is successful do we proceed to generate an embedding on the cropped image using `DeepFace.represent()` with the detector set to 'skip'.
- **Challenge: Biased or Unreliable LLM Judgments.**
  - **Problem:** The initial prompts asking the LLM to judge "trustworthiness" were too subjective, often resulting in a default "SUSPICIOUS" verdict.
  - **Solution:** We re-designed the prompts to be shorter and to ask for a more objective, binary analysis (e.g., "Does this response contain a name? Respond with only YES or NO."). This simplified the LLM's task and yielded far more reliable and useful judgments.

3 Ethical Considerations and Testing Results

3.1 Ethical Considerations

Developing a surveillance agent need careful consideration of ethical implications.

- **Privacy and Consent:** The agent captures video and audio, which is highly sensitive.Users must be aware of when the system is active. So given intial logs likes "Hi AI Agent Here" or "Waiting for activation commands" etc
- **Data Security:** The system stores face embeddings, intruder photos, and logs. This data is stored locally on the machine and must be protected from unauthorized access. For a production system, stored data can be encrypted
- **Model Bias:** The pre-trained face recognition models may exhibit biases, potentially performing less accurately for certain demographic groups. This could lead to false rejections of trusted individuals or false identifications, and must be considered during evaluation.

3.2 Testing Results

The system was tested in a variety of scenarios with the following outcomes:

- **Activation Accuracy:** The voice command "guard my room" was recognized with over 95% accuracy in a moderately quiet room. Milestone1 results
- **Face Recognition Accuracy:** The `SFace` model with a threshold of 0.4 correctly identified the enrolled user in different lighting conditions and from various angles. It consistently labeled an unenrolled person as "Unknown". Milestone2 results
- **Escalation Logic:** The dialogue flow performed as designed. An unknown person was consistently challenged. A trustworthy response (e.g., stating the resident's name) was judged as "SPECIFIC" by the LLM and successfully de-escalated the situation. An evasive response or silence correctly triggered escalation to the final alert level. Milestone3 results
- **Performance:** With frame skipping, the video feed remained smooth, running at approximately 15-20 FPS on a standard laptop CPU. Final Demo

4 Instructions to Run the Code

Follow these steps to set up and run the AI Room Guard Agent.

1. **Prerequisites:**
   - Python 3.8+ and pip.
   - A working webcam and microphone.
2. **Clone the Repository:**
   ```
   git clone <https://github.com/Soumyadeepj/AI-Agent.git>
   ```
3. **Create a Virtual Environment (Recommended):**
   ```
   python -m venv venv
   # On Windows:
   venv\Scripts\activate
   # On macOS/Linux:
   source venv/bin/activate
   ```
4. **Install Dependencies:**
   ```
   pip install -r requirements.txt
   ```
5. **Set Up Google Gemini API Key:**
   - Go to `https://makersuite.google.com/` to generate a free API key.
   - Open the Python script and paste your key into the following line:
     ```
     genai.configure(api_key="YOUR_API_KEY_HERE")
     ```
6. **Enroll Trusted Faces:**
   - Create a folder named `enroll` in the project directory.
   - Inside `enroll`, create a sub-folder for each trusted person (e.g., `enroll/person_name/`).
   - Place several clear photos of that person in their sub-folder.
7. **Run the Agent:**
   ```
   python main_guard.py
   ```

8. **Usage:**

    - Say **"guard my room"** to activate the agent.
    - Say **"stop guarding"** to deactivate it.
    - Press the **'q'** key with the video window selected to exit the program.