AI Guard Agent: A Multi-Modal Surveillance System using Pre-Trained Models

Department of Electrical Engineering
Advanced Topics in Machine Learning Assignment 2

Submitted by: Dhruv Meena and Madhava Sri Ram
B.Tech, Electrical Engineering

October 2025

Contents

1 Introduction **2**
1.1 Goal . . . 2

2 System Architecture **3**
2.1 Overview . . . 3
2.2 Major Modules . . . 3

3 Tools and Frameworks **4**
3.1 Core Libraries . . . 4
3.2 Why These Choices . . . 4

4 Implementation Details **5**
4.1 Threading and Synchronization . . . 5
4.2 ASRListener Thread . . . 5
4.3 Face Recognition Flow . . . 6
4.4 Enrollment Process . . . 6
4.5 Guard Loop Logic . . . 6

5 Key Code Snippets Explained **7**
5.1 Voice Activity Detection (VAD) . . . 7
5.2 Cosine Distance for Embedding Matching . . . 7
5.3 Thread Coordination During TTS . . . 7

6 Testing and Debugging **8**
6.1 Initial Issues . . . 8
6.2 Thread Race Conditions . . . 8
6.3 Camera Deadlocks . . . 8
6.4 Debugging Process . . . 8

7 Appendix **9**
7.1 Project Structure . . . 9
7.2 Dependencies . . . 9
7.3 Acknowledgements . . . 9

Chapter 1

Introduction

1.1 Goal

The goal of this project was to design an intelligent, privacy-conscious, and cost-effective **AI Guard Agent** as an integrated multimodal system that can:

- Monitor a room using webcam, microphone, and speakers.
- Activate or deactivate using spoken commands.
- Identify authorized individuals using face recognition.
- Engage in verbal interaction with unrecognized persons to deter intrusion.

The focus of the project was not on training new models from scratch, but on **integration** of pre-trained AI models (vision, speech, and language) into a cohesive, robust, real-time system.

Chapter 2

System Architecture

2.1 Overview

Flowchart for our project is given in our README.md using mermaid code.

2.2 Major Modules

- **Speech Recognition (ASR):** Records short audio bursts and transcribes them into text commands.
- **Text-to-Speech (TTS):** Converts system responses into spoken audio.
- **Face Detection:** Uses MediaPipe for fast face localization.
- **Face Recognition:** Uses DeepFace (ArcFace model) to generate embeddings and match against the local database.
- **Main Guard Loop:** Controls mode switching, monitors the camera, and triggers escalation logic.
- **Enrollment System:** Guides the user through capturing multiple face images for registration.

Chapter 3

Tools and Frameworks

3.1 Core Libraries

- **OpenCV:** Video capture and real-time frame display.
- **MediaPipe:** Face detection backend (lightweight, real-time performance).
- **DeepFace:** For computing embeddings via pre-trained face models.
- **SoundDevice + WebRTC VAD:** For audio capture and speech activity detection.
- **SpeechRecognition:** To perform transcription using Google's speech API.
- **gTTS / pyttsx3 / Coqui TTS:** For natural voice synthesis.
- **Threading + Queue:** To run asynchronous ASR without blocking main video loop.

3.2 Why These Choices

- **MediaPipe** was chosen over Haar Cascades due to its GPU acceleration and robustness in varying lighting conditions.
- **DeepFace (ArcFace backend)** provides accurate face embeddings without requiring training.
- **WebRTC VAD** offers low-latency voice activity detection.

Chapter 4

Implementation Details

4.1 Threading and Synchronization

The project uses a multi-threaded design:

- **Main thread:** Handles camera capture, display, and guard logic.
- **ASRListener thread:** Continuously listens for commands.

To ensure safe concurrency:

- **Locks** protect shared audio resources.
- **Events** coordinate between ASR and TTS:
  - `tts_active`: Set when TTS is speaking.
  - `asr_pause`: Set during enrollment to stop ASR.

This design prevents the system from transcribing its own voice or interfering during enrollment.

4.2 ASRListener Thread

Listing 4.1: Simplified ASR Listener Loop

```
class ASRListener(threading.Thread):
    def run(self):
        while not self.stop_event.is_set():
            if tts_active.is_set() or asr_pause.is_set():
                time.sleep(0.1); continue
            fname, ratio, _ = record_wav_with_vad(
               INPUT_DEVICE_INDEX, 2)
            if not fname: continue
            text = transcribe_wav_file(fname)
            if text: self.q.put(text)
```

This thread continuously records short clips, filters silence via WebRTC VAD, transcribes using Google ASR, and pushes recognized text into a shared queue for the main loop.

4.3 Face Recognition Flow

1. Detect faces via MediaPipe:

```
faces = detect_faces_mediapipe(frame, confidence=0.5)
```

2. For each detected face, crop the bounding box.

3. Generate embeddings:

```
embedding = get_embedding_from_image(face_crop)
```

4. Compare embeddings against enrolled database using cosine distance.

4.4 Enrollment Process

To add a trusted user, the system pauses ASR, prompts the user for a typed name, and captures multiple images from different angles.

Listing 4.2: Core structure of enrollment logic

```
say("Please type the person's name.")
name = input("Name to enroll: ").strip()
for pose in poses:
    show_prompt(pose)
    face = capture_best_face(cam)
    embeddings.append(get_embedding_from_image(face))
save_enrollment(name, embeddings)
```

Each image is validated for face area and resolution. The resulting embeddings are stored as a compressed NumPy file.

4.5 Guard Loop Logic

The main loop integrates all components:

- Waits for ASR commands.
- When armed:
  - Continuously checks for faces.
  - Matches against enrolled faces.
  - If unknown face persists, initiates escalating audio warnings.

Chapter 5

Key Code Snippets Explained

5.1 Voice Activity Detection (VAD)

VAD ensures we record only speech segments:

```
vad = webrtcvad.Vad(2)
for frame in frames:
    if vad.is_speech(frame, rate):
        voiced_frames += 1
speech_ratio = voiced_frames / total_frames
```

Only if the ratio exceeds a threshold (e.g., 0.6) is the segment saved for transcription.

5.2 Cosine Distance for Embedding Matching

```
def cosine_distance(a, b):
    sim = np.dot(a,b) / (np.linalg.norm(a)*np.linalg.norm(b))
    return 1 - sim
```

A lower cosine distance indicates higher similarity between embeddings.

5.3 Thread Coordination During TTS

```
def say(text):
    with audio_lock:
        tts_active.set()
        tts_engine.say(text)
        tts_engine.runAndWait()
        tts_active.clear()
```

Setting `tts_active` prevents the ASR thread from recording during speech output.

Chapter 6

Testing and Debugging

6.1 Initial Issues

- **Key Misfires in ASR:** Google ASR often misheard "guard" as "god" or "card". Fixed by fuzzy string matching using a list of allowed variants.
- **TTS Voice Quality:** `pyttsx3` sounded robotic; replaced with `gTTS` for smoother voice output.
- **DeepFace API mismatch:** Different versions use `model_name` or `model`. Fixed by trying both.

6.2 Thread Race Conditions

At times, ASR and TTS overlapped, causing feedback. Introducing `tts_active` and `asr_pause` events solved the synchronization issue.

6.3 Camera Deadlocks

When the camera failed to open or return frames, OpenCV froze. Added frame validity checks and fallback sleeps to prevent crashes.

6.4 Debugging Process

Debugging was primarily done through structured logging using timestamped messages like:

```
[ASR] >>> HEARD: "enroll"
[GUARD] All faces recognized - resetting
```

This made it easier to trace the flow between ASR, recognition, and TTS responses.

Chapter 7

Appendix

7.1 Project Structure

```
AI_Guard_Agent/

 main.py                 # main guard loop
 test.py                 # tests devices, tools, dependencies
 enrolled/               # saved face embeddings
 README.md               # Instructions to Run + Flowchart
 activation_phrases.txt # phrases that sound similar to actual commands
 requirements.txt        # dependencies
```

7.2 Dependencies

```
opencv-python
mediapipe
deepface
sounddevice
webrtcvad
SpeechRecognition
gtts
numpy
```

7.3 Acknowledgements

Majority of code was written using LLMs.