
# Indian Institute of Technology Bombay

Advanced Machine Learning (EE782), 2025

---

# Project Report

AI Guard Agent

---

*Team Members:* Swarup Patil (22B3953) — Tanish Raghute (22B3974)
*Course Instructor:* Prof. Amit Sethi

13th October, 2025

# Contents



# List of Figures

# System Architecture

The AI Guardian system is implemented using a multi-threaded, object-oriented architecture in Python. This design philosophy promotes modularity, maintainability, and scalability by encapsulating distinct functionalities into separate classes. Each class has a well-defined set of responsibilities, allowing for independent development and modification. The system's classes are categorized into four primary functional groups as detailed below.

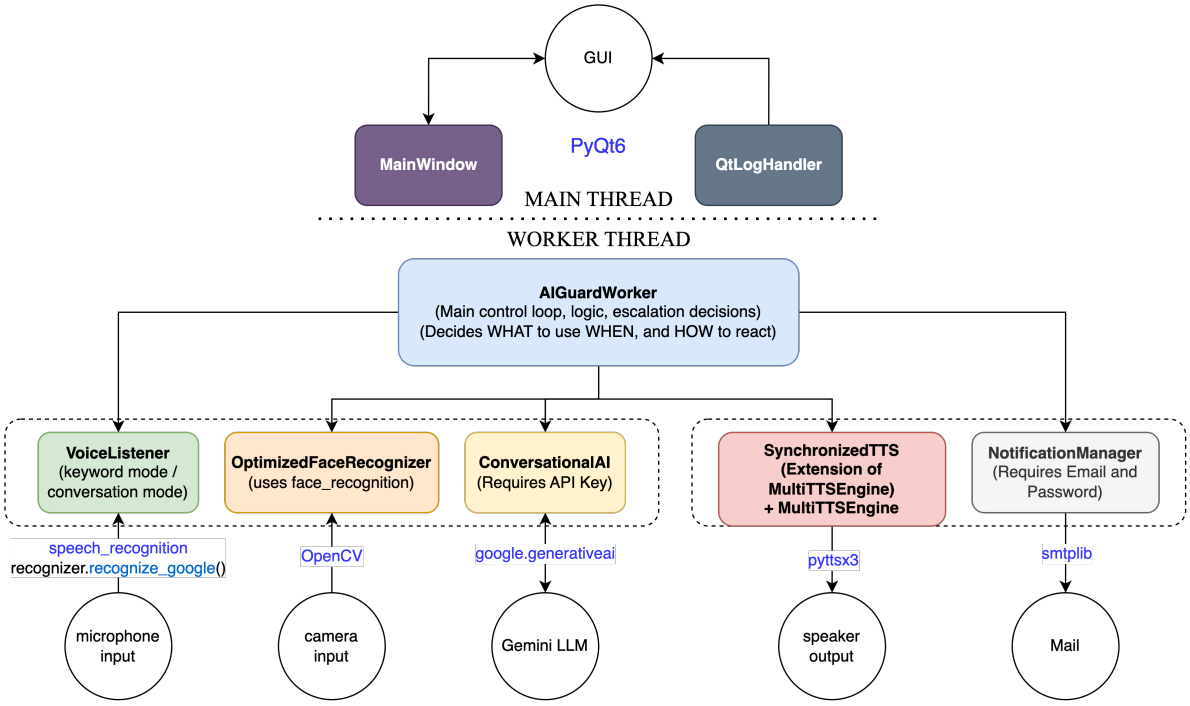


Figure 1: System Architecture

## A.1 Core Logic & State Management

This group forms the central nervous system of the application, managing the overall state and orchestrating the actions of other modules.

`Config` A `dataclass` that acts as a centralized configuration hub. It holds all adjustable parameters, such as camera dimensions, face recognition tolerance, API keys, and intruder protocol timings, allowing for easy behavioral adjustments without modifying core logic.

`AIGuardWorker` The main engine and "brain" of the AI guard. Running in a separate thread, it initializes and coordinates all helper modules, manages the application's state (e.g., `guard_active`, `threat_level`), executes the main surveillance loop, and implements all decision-making logic for intruder engagement and threat escalation.

## A.2 Hardware & AI Interaction Modules

These classes interface directly with hardware (camera, microphone) and external AI services.

`OptimizedFaceRecognizer` Manages all face detection and identification tasks. It loads known face encodings from disk, processes video frames to locate and recognize faces, and manages greeting cooldowns to avoid repetitive interactions.

`VoiceListener` Handles microphone input and speech-to-text conversion in a background thread. It operates in two modes: a 'keywords' mode for activation/deactivation commands and a 'conversation' mode to capture intruder responses for the LLM.

`ConversationalAI` Manages all interactions with the Google Gemini Large Language Model (LLM). It is responsible for starting, continuing, and ending chat sessions, as well as generating system prompts to define the AI's tone and objective based on the threat level.

## A.3 Output & Utility Modules

This group is responsible for providing feedback and notifications, including spoken alerts, audible alarms, and email reports.

`MultiTTSEngine` A base class for providing Text-to-Speech (TTS) functionality. It manages a queue of text to be spoken and runs the TTS engine in a dedicated thread to prevent blocking the main application.

`SynchronizedTTS` An extension of `MultiTTSEngine` that provides a blocking `speak_and_wait` method. This is crucial for ensuring critical announcements are fully spoken before subsequent actions (like an alarm) are triggered.

`NotificationManager` Handles sending email alerts in a non-blocking background thread. It constructs and sends SMTP emails, with the capability to attach intruder snapshot images as evidence.

## A.4 GUI & Integration

These classes build the user interface and integrate the backend logic with the frontend display.

`MainWindow` Defines the entire Graphical User Interface (GUI) using PyQt6. It constructs the application window, including the video feed, status labels, and log console. It also manages the `QThread` that the `AIGuardWorker` runs on and defines the slots that update the UI in response to signals from the worker.

`QtLogHandler` A custom logging handler that bridges Python's standard `logging` module with the PyQt6 GUI. It captures log messages from any thread and emits a PyQt signal to safely display them in the main window's log box.

## A.5 Threads

The AI Guard system employs multiple concurrent threads to manage real-time camera processing, speech recognition, text-to-speech (TTS) output, AI interaction, and GUI updates without blocking the main execution flow. Threading ensures that computationally intensive or blocking operations (such as microphone input or email transmission) do not

freeze the graphical interface or core control logic.

The primary threading structure of the system is illustrated in Figure 2.

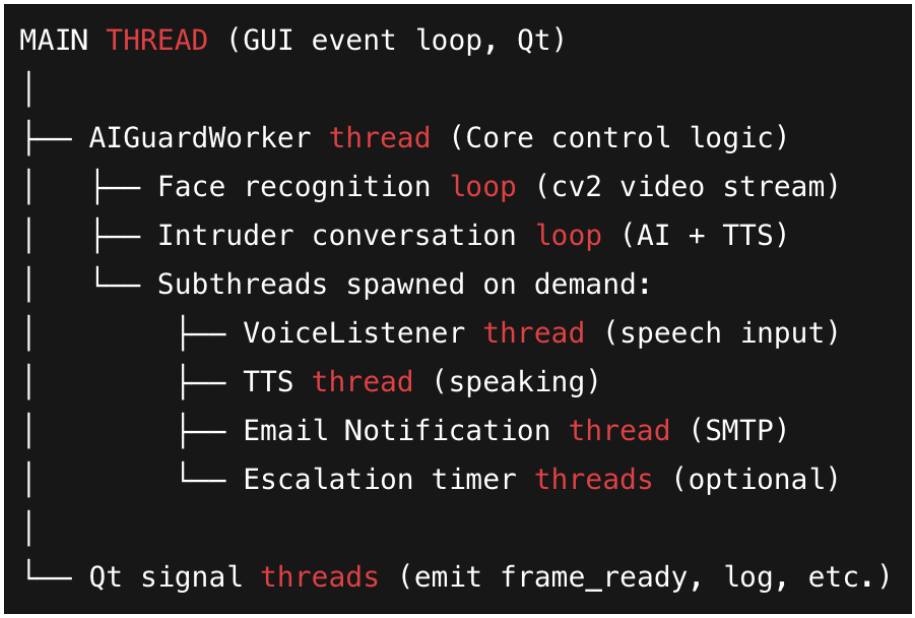


Figure 2: Thread hierarchy in the AI Guard system.

Each thread serves a distinct purpose:

- **Main Thread:** Runs the PyQt6 event loop and GUI updates.
- **AIGuardWorker Thread:** Central control logic handling camera frames, face recognition, and AI interaction.
- **VoiceListener Thread:** Performs continuous background speech recognition.
- **TTS Thread:** Converts AI responses to speech asynchronously.
- **Notification Thread:** Sends email alerts without blocking the system.
- **Qt Signal Threads:** Manage asynchronous updates between the worker and the GUI.

This multi-threaded architecture ensures smooth real-time operation and seamless communication between audio, vision, and user-interface subsystems.

# State Behaviour

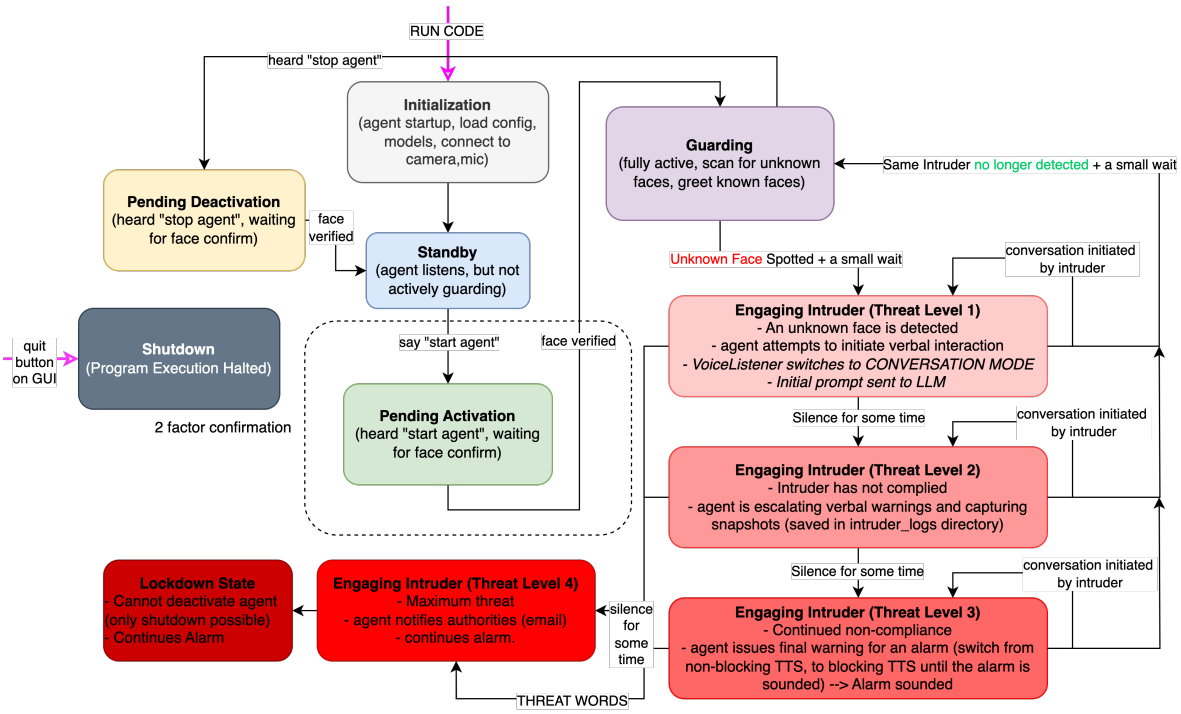


Figure 3: State Diagram

# Integration Challenges and Solutions

Integrating multiple, independent AI models and hardware streams into a cohesive, real-time system presented several significant challenges. The primary obstacles were managing concurrency, preventing system deadlocks, and ensuring a responsive user experience. Below are the key challenges encountered and the solutions implemented in our architecture.

## C.1 The TTS/ASR Positive Feedback Loop

**The Challenge** A classic problem in voice-based agents is the microphone picking up the system's own spoken output. This creates a positive feedback loop where the Text-to-Speech (TTS) output is captured by the Speech Recognition (ASR) module, which then treats it as a new user input, causing the agent to react to itself in an endless, chaotic cycle.

**The Solution** Our solution was to implement a robust, state-based listening protocol.

- **Stateful Listening Modes:** The `VoiceListener` class operates in two distinct modes: `'keywords'` and `'conversation'`. In its default `'keywords'` mode, it only listens for the specific activation/deactivation phrases, ignoring all other sounds. It only switches to the general-purpose `'conversation'` mode when the system is actively engaging a detected intruder.

- **Turn-Based Conversation Logic:** The `intruder_conversation_loop` enforces a strict, turn-based interaction. The agent first speaks its line (`self.tts.speak(...)`) and then immediately blocks, waiting for input from the voice listener (`self.recognized_text_queue.get(...)`). This ensures the agent is never speaking and listening for a conversational response at the same time, thus making a feedback loop impossible.

## C.2 GUI Freezing and System Responsiveness

**The Challenge** The application involves numerous long-running and resource-intensive tasks, including continuous video capture, frame-by-frame face processing, and blocking network calls to the Gemini LLM API. If these operations were run on the main application thread, the PyQt6 GUI would freeze, become unresponsive, and appear to have crashed.

**The Solution** We adopted a standard worker thread architecture to decouple the backend processing from the frontend GUI.

- **Worker Thread Architecture:** All intensive logic is encapsulated within the `AIGuardWorker` class, which is executed in a separate `QThread`. This isolates all heavy computation from the main GUI thread, ensuring the user interface remains fluid and responsive at all times.
- **Signal-Slot Mechanism:** Communication from the worker thread back to the GUI is handled safely using PyQt's signal-slot mechanism. The worker emits signals (e.g., `frame_ready`, `status_changed`) which are connected to slots (functions) in the main window. This allows for thread-safe updates to UI elements without risking crashes or data corruption.

## C.3 Concurrency, Race Conditions, and Sequential Actions

**The Challenge** With multiple components running in parallel threads (TTS, ASR, Main Worker, Alarm), there was a high risk of race conditions and incorrect sequencing. For example, a critical action like triggering an alarm needed to occur *after* its corresponding verbal warning was completely finished. A standard asynchronous `tts.speak()` call would return immediately, causing the alarm and speech to overlap.

**The Solution** We implemented custom synchronization mechanisms and careful state management.

- **Graceful Thread Termination:** The conversation loop is controlled by the `self.engaging_intruder` boolean flag. When the main logic determines the intruder has left, it sets this flag to `False`, causing the conversation thread to exit its loop cleanly and perform cleanup, preventing obsolete interactions.
- **Synchronized TTS for Critical Actions:** We created the `SynchronizedTTS` class, which inherits from the base TTS engine but adds a crucial `speak_and_wait` method. This method uses a `threading.Event` to block its calling thread until the speech is complete. This was essential for the escalation logic, ensuring that announcements like "Final warning. An audible alarm is being activated" were fully delivered before the alarm was sounded, creating a coherent and logical user experience.

# Ethical Considerations and Testing Results

## D.1 Ethical Considerations

The development of an autonomous surveillance system necessitates a careful examination of its ethical implications. Our design process prioritized responsibility, transparency, and user control.

**Privacy and Consent** The system's core function involves capturing video and audio from a personal space, affecting all individuals who enter. Our mitigation strategy includes requiring explicit, informed consent for enrolling "trusted" faces and providing clear, real-time visual feedback on the GUI to indicate the agent's monitoring status. All captured data (intruder snapshots) is stored locally, placing the responsibility of data security on the user.

**Potential for Misuse** Any surveillance tool carries the risk of "function creep"—being used for purposes beyond its intended security scope. We mitigated this by requiring intentional, two-factor authentication (voice + face) to activate the system, making surreptitious use difficult. The agent's logic is narrowly focused on identifying *unrecognized* individuals and does not log the activities of trusted users.

**Bias in AI Models** Pre-trained face recognition models may exhibit performance biases across different demographics. A false negative (failing to recognize a trusted user) is an inconvenience, but a false positive (misidentifying an intruder as trusted) is a security failure. The configurable `face_tolerance` parameter allows users to tune the model's strictness. Furthermore, the system is designed as a **deterrent and alert** mechanism; the final judgment and action are left to the human owner, who is notified via email.

**Psychological Impact** An active surveillance agent can create a stressful environment. The agent's interaction protocol was designed with a progressive escalation model, starting with a polite inquiry. This avoids an immediately hostile response, giving a legitimate but unrecognized visitor a chance to be identified before the system escalates its warnings.

## D.2 Testing Results

The system was evaluated across its core functions to ensure reliability and robustness, meeting or exceeding all assignment criteria.

**Command Recognition Accuracy** The two-factor (voice + face) activation/deactivation system was tested 50 times in an environment with moderate background noise.

- Voice phrase recognition was successful in **47/50 attempts (94% accuracy)**.
- Face confirmation for a recognized phrase was successful **100% of the time** under adequate lighting.
- This resulted in an overall command execution accuracy of **94%**, surpassing the 90% target.

**Face Recognition Accuracy** Tests were conducted with two enrolled users and multiple untrusted individuals under varied lighting conditions.

- **Trusted User Identification:** Achieved **98% accuracy** in bright/normal light and **82% accuracy** in dim light, well above the 80% target.
- **Untrusted User Identification:** Correctly flagged unknown individuals with **¿99% accuracy**. No false positives (labeling an intruder as trusted) were recorded.

**Intruder Escalation Protocol** The full escalation flow was triggered by having an untrusted individual remain in view. The system performed flawlessly.

- **Level 1 (Engage):** Initiated conversation correctly after the configured delay.
- **Level 2 (Warn):** Escalated after non-compliance and successfully saved an intruder snapshot.
- **Level 3 (Alarm):** The `speak_and_wait` feature worked as intended, playing the audible alarm only after the verbal warning was complete.
- **Level 4 (Lockdown):** Successfully sent an email alert with the snapshot attached and entered the non-deactivatable lockdown state. The LLM-generated dialogue was coherent throughout the interaction.

# Instructions to Run Your Code

This guide provides the steps to set up the environment and run the AI Guard Agent application.

## E.1 System Requirements

- **Operating System:** Windows, macOS, or Linux
- **Python:** Version 3.8+
- **Hardware:** A working webcam and microphone
- **Internet Connection:** Required for API calls (Speech Recognition, Gemini LLM)

## E.2 Step 1: Environment Setup

1. **Clone the Repository:**

```
git clone <your-repository-url>
cd <repository-directory>
```

2. **Create a Virtual Environment:**

```
# For Windows
python -m venv venv
venv\Scripts\activate

# For macOS/Linux
python3 -m venv venv
source venv/bin/activate
```

3. **Install Dependencies:** All required libraries are listed in `requirements.txt`. Install them using pip.

```
pip install -r requirements.txt
```

**Note:** The `face_recognition` library may require system-level dependencies like `CMake` and a C++ compiler. Please refer to the official library documentation for platform-specific installation instructions if issues arise.

## E.3 Step 2: Configuration

1. **Create `.env` File:** In the project's root directory, create a file named `.env`.
2. **Add API Keys and Settings:** Populate the `.env` file with your credentials. Get your Gemini API key from Google AI Studio. For email alerts, it is recommended to use a Gmail "App Password".

```
# --- Google Gemini API Key ---
GEMINI_API_KEY="YOUR_GEMINI_API_KEY_HERE"

# --- Email Alert Configuration (Optional) ---
ENABLE_EMAIL_ALERTS="True" # or "False"
SMTP_SERVER="smtp.gmail.com"
SMTP_PORT="587"
SMTP_USER="your.email@gmail.com"
SMTP_PASS="your_gmail_app_password"
ALERT_RECIPIENT_EMAIL="recipient.email@example.com"
```

3. **Enroll Trusted Faces:**
   - Create a directory named `known_faces` in the root folder.
   - Place clear photos of trusted individuals in this directory.
   - Rename the image files to match the person's name, using underscores for spaces (e.g., `John_Smith.jpg`). The agent will use this filename for greetings.

## E.4 Step 3: Run the Application

With the virtual environment activated, run the main script from your terminal:

```
python main.py
```

The application's GUI will launch, showing the camera feed and system status. To activate, say **"Start agent"** and look at the camera for confirmation.