IntelliGaze: A Wearable AI Camera System

Microprocessor Lab Report

Touhidul Alam Seyam (230240003)
Eftakar Uddin (230240004)
Tasmim Akther Mim (230240025)
Shafiul Azam Mahin (230240022)
Muntasir Rahman (230240002)

June 16, 2025

Submitted to:
Professor Radiathun Tasnia
Junior Lecturer
BGC Trust University Bangladesh

Abstract

This report details the design, development, and implementation of IntelliGaze, a wearable AI camera system aimed at enhancing situational awareness and independence for visually impaired individuals. The system integrates an ESP32-CAM for real-time video capture, a FastAPI backend server for image processing and AI interfacing, and client applications (React Native for mobile and PyQt6 for desktop) for user interaction and feedback. IntelliGaze leverages a cloud-based AI vision service to analyze scenes, identify objects, recognize text, and provide contextual descriptions. Key features include live video streaming, AI-powered scene interpretation with optimized prompts, configurable capture modes (manual, auto-capture, auto-Text-to-Speech), response history, and system logging. This report covers the project's objectives, requirements, system architecture, implementation of core software modules, demonstration of functionalities, and potential avenues for future development. The project successfully demonstrates a proof-of-concept for a versatile and accessible assistive technology.

Declaration

We, the undersigned, declare that this lab report titled "IntelliGaze: A Wearable AI Camera System" is our own original work and has not been submitted to any other institution for academic credit. All sources of information have been duly acknowledged and cited. The project work presented herein was carried out by the group members listed on the title page.

Touhidul Alam Seyam (230240003)	________________________
Eftakar Uddin (230240004)	________________________
Tasmim Akther Mim (230240025)	________________________
Shafiul Azam Mahin (230240022)	________________________
Muntasir Rahman (230240002)	________________________

Date: June 16, 2025

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Professor Radiathun Tasnia, Junior Lecturer at BGC Trust University Bangladesh, for her invaluable guidance, support, and encouragement throughout the duration of this project. Her insights and expertise were instrumental in shaping the direction of IntelliGaze.

We also thank BGC Trust University Bangladesh for providing the necessary resources and laboratory facilities that enabled us to conduct our research and development.

Finally, we acknowledge the open-source communities and developers whose tools and libraries (such as ESP32 Arduino Core, FastAPI, React Native, PyQt6, and others) formed the building blocks of our system.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Navigating the world presents unique and often significant challenges for individuals with visual impairments. Tasks such as identifying objects, reading signs, recognizing obstacles, and maintaining overall situational awareness can be difficult, impacting independence and safety. While traditional assistive aids like canes and guide dogs offer valuable support, they may not always provide the dynamic, detailed information required to interpret complex and rapidly changing environments.

The rapid advancements in artificial intelligence (AI), particularly in computer vision and natural language processing, offer transformative potential for assistive technologies. Compact, powerful microcontrollers like the ESP32-CAM, coupled with accessible cloud-based AI services, now make it feasible to develop sophisticated, wearable solutions that can "see" and "describe" the world for the user. IntelliGaze is born out of this motivation: to harness these technological advancements to create a practical and empowering tool for the visually impaired.

1.2 Problem Statement

The core problem IntelliGaze addresses is the limited access to real-time, context-aware visual information for individuals with visual impairments. This limitation can hinder their ability to:

- Independently navigate unfamiliar environments.
- Identify and interact with objects of interest or necessity.
- Read textual information such as signs, labels, or documents critical for daily tasks.
- Maintain a comprehensive understanding of their immediate surroundings, which is crucial for safety and confidence.

IntelliGaze aims to provide a system that actively processes visual input from a wearable camera and delivers concise, relevant descriptions, thereby mitigating these challenges. The system's primary instruction, "You are an assistive vision system for the visually impaired. Given an image from a wearable camera, describe the scene in a way that maximizes situational awareness and independence. Clearly identify objects, obstacles, people, and signage. If there is text in the scene, read it aloud and explain its context (e.g., sign, label, document). Use short, direct sentences and avoid technical jargon. Prioritize information that would help a visually impaired user navigate or understand their environment," (derived from `Desktop/app.py` and `Flask_server/server.py`) encapsulates this goal.

1.3 Project Objectives

The primary objectives for the IntelliGaze project were:

1. **Develop a Real-Time Vision System:** To design and implement a system capable of capturing a live video feed using a compact, wearable ESP32 camera module.
2. **Implement AI-Powered Scene Interpretation:** To integrate with an AI vision model to process captured images, enabling the system to understand and describe the surrounding environment, including object recognition and text extraction (OCR).
3. **Provide Intuitive User Feedback:** To deliver clear, concise, and contextually relevant descriptions of the visual scene to the user through both textual display and auditory (Text-to-Speech) outputs.
4. **Create Versatile User Interfaces:** To develop both a mobile application (React Native for iOS/Android) and a desktop application (PyQt6) that allow users to control the system, configure settings, and receive feedback.
5. **Ensure Modularity and Configurability:** To design the system with modular components for easier maintenance and upgrades, and to offer configurable features such as automatic capture intervals and TTS preferences.
6. **Focus on Accessibility:** To tailor the AI's descriptive output specifically for the needs of visually impaired users, prioritizing information that enhances situational awareness and independence.

1.4 Scope and Limitations

Scope

- Development of ESP32-CAM firmware for WiFi connectivity and MJPEG video streaming.
- Creation of a FastAPI backend server to manage the ESP32 stream, interface with a third-party AI vision API, and serve client requests.
- Design and implementation of a React Native mobile application for user interaction, featuring manual/auto capture, TTS, history, and logging.
- Design and implementation of a PyQt6 desktop application offering an alternative user interface with direct stream viewing and AI interaction.
- Integration of a cloud-based AI service for image analysis and description generation.
- Implementation of basic Text-to-Speech functionality on the mobile application.

Limitations

- The system relies on a stable WiFi connection for the ESP32-CAM and an internet connection for the AI vision and TTS services.
- The performance (latency, accuracy) of scene description is dependent on the chosen third-party AI vision service and network conditions.
- The current prototype has minimal security features for local network communication (as noted in `docs/system-architecture.md` if it existed, or implied by lack of explicit security measures in code).
- Advanced navigation assistance (e.g., pathfinding, precise obstacle avoidance) is beyond the current scope.
- Power consumption and battery life of the wearable ESP32 unit were not primary optimization targets for this prototype.
- The system primarily processes 2D images and does not incorporate depth perception.

1.5 Report Structure

This report is organized into the following chapters:

- **Chapter 1: Introduction** - Provides background, problem statement, objectives, scope, and limitations.
- **Chapter 2: Requirements Analysis** - Details functional, non-functional, hardware, and software requirements.
- **Chapter 3: System Design and Architecture** - Describes the overall architecture and design of individual components.
- **Chapter 4: Implementation Details** - Discusses the implementation of key software modules with code highlights.
- **Chapter 5: System Demonstration and Results** - Showcases the working system and discusses its performance.
- **Chapter 6: Conclusion and Future Work** - Summarizes the project and suggests future enhancements.
- **Chapter 7: Project Management** - Includes the Gantt chart and team contributions.

Appendices may include further detailed diagrams or code if necessary.

Chapter 2

Requirements Analysis

2.1 Functional Requirements

The IntelliGaze system is designed to meet the following functional requirements:

FR1: **Video Capture:** The ESP32-CAM shall capture live video footage.

FR2: **Video Streaming:** The ESP32-CAM shall stream the captured video over WiFi via MJPEG format.

FR3: **Server-Side Stream Consumption:** The FastAPI server shall connect to the ESP32's MJPEG stream and retrieve individual frames.

FR4: **AI Vision Processing Request:** The system (server or desktop client) shall send captured image frames to an AI vision service for analysis.

FR5: **Customizable AI Instruction:** The system shall allow the use of a predefined or user-input instruction/prompt to guide the AI's image description task (e.g., 'OPTIMIZED_PROMPT').

FR6: **Scene Description Generation:** The AI vision service shall return a textual description of the analyzed image.

FR7: **Display of AI Response (Mobile):** The React Native mobile app shall display the AI-generated scene description to the user.

FR8: **Display of AI Response (Desktop):** The PyQt6 desktop app shall display the AI-generated scene description to the user.

FR9: **Manual Capture (Mobile):** The mobile app shall allow users to trigger an on-demand image capture and AI analysis.

FR10: **Auto-Capture (Mobile):** The mobile app shall support automatic periodic image capture and analysis at user-configurable intervals (e.g., 2, 3, 5, 10 seconds).

FR11: **Text-to-Speech (TTS) Output (Mobile):** The mobile app shall convert the AI-generated text description into speech.

FR12: **Auto-TTS Mode (Mobile):** The mobile app shall support a mode where captured scenes are automatically described via TTS, with subsequent captures potentially chained to TTS completion.

FR13: **Response History (Mobile):** The mobile app shall store and display a history of recent AI-generated descriptions.

FR14: **System Logging (Mobile & Desktop):** Both client applications shall provide a log of system events, status updates, and errors.

FR15: **Connection Status Monitoring (Mobile):** The mobile app shall monitor and display the connection status of the backend server and the ESP32 camera.

FR16: **Configuration (Mobile):** The mobile app shall allow users to configure settings such as server URL and ESP32 IP address.

FR17: **Desktop Video Feed Display:** The desktop app shall display the live video feed from the ESP32-CAM or a local webcam.

FR18: **Desktop AI Interaction Controls:** The desktop app shall provide controls to send frames to AI, stop processing, and configure auto-send intervals.

2.2 Non-Functional Requirements

NFR1: **Usability:** The user interfaces (mobile and desktop) shall be intuitive and easy to use, especially considering the target users (visually impaired).

NFR2: **Performance:**

- The ESP32 shall stream video with minimal perceivable lag on a local network.
- The AI response time (capture to description display/TTS) should be reasonably fast to provide near real-time assistance (target < 5-10 seconds, dependent on AI service).

NFR3: **Reliability:** The system components should operate reliably. The server and clients should handle potential disconnections or errors gracefully.

NFR4: **Portability (Mobile):** The React Native app shall be cross-platform (iOS and Android).

NFR5: **Configurability:** Key parameters like server URLs, ESP32 IP, and auto-capture intervals shall be configurable by the user.

NFR6: **Modularity:** The system architecture should be modular to allow for easier maintenance, updates, and potential replacement of components (e.g., AI service).

NFR7: **Accessibility (Output):** AI descriptions and TTS output should be clear, concise, and prioritize information useful for visually impaired users.

2.3 Hardware Components

- **ESP32-CAM (AI-THINKER model):**
  - Microcontroller: ESP32-S chip
  - Camera: OV2640 sensor (2 Megapixel)

    - PSRAM: Typically 4MB (influences frame size and quality settings)
    - Connectivity: WiFi 802.11 b/g/n
    - Storage: MicroSD card slot (not primarily used for streaming in this project)

- **ESP32-CAM-MB Programmer/Adapter:** Micro-USB adapter for easy programming and power supply to the ESP32-CAM.
- **Power Supply:** USB power source for the ESP32-CAM-MB.
- **User Device (Mobile):** Smartphone (iOS or Android) capable of running the React Native application.
- **User Device (Desktop):** Computer (Windows, macOS, or Linux) capable of running Python and PyQt6 for the desktop application.
- **Networking Equipment:** WiFi router for local network communication.

2.4 Software Components

- **ESP32 Firmware Environment:**
    - Arduino IDE with ESP32 Core
    - Libraries: ‘WiFi.h‘, ‘esp_camera.h‘, ‘esp_http_server.h‘
- **Backend Server Environment (FastAPI):**
    - Python (e.g., 3.8+)
    - FastAPI
    - Uvicorn (ASGI server)
    - httpx (HTTP client)
    - Loguru (logging)
    - OpenCV-Python (for optional frame validation on server)
- **Mobile Application Environment (React Native):**
    - Node.js and npm/yarn
    - Expo CLI
    - React Native
    - TypeScript
    - Axios (HTTP client)
    - Expo Audio/AV (for TTS playback)
    - AsyncStorage (for settings storage)
    - Other UI/navigation libraries (Expo Router, React Navigation)
- **Desktop Application Environment (PyQt6):**
    - Python (e.g., 3.8+)

    - PyQt6
    - OpenCV-Python (for frame fetching and processing)
    - Requests (HTTP client)

- **Third-Party Services:**
    - AI Vision Service API (e.g., specified by ‘AI_BACKEND_URL‘ in server code, such as a multimodal LLM endpoint)
    - Groq TTS API (for Text-to-Speech service, specified in ‘inteligaze/utils/playGroqTTS.ts‘)

- **Development Tools:**
    - Code Editor (e.g., VS Code)
    - Git (for version control)

Chapter 3

System Design and Architecture

This chapter details the architectural design of the IntelliGaze system, outlining its main components and their interactions. The system is designed to be modular, allowing for independent development and potential upgrades of its constituent parts.

3.1 Overall System Architecture

IntelliGaze comprises several key components: the ESP32 Camera Module, the FastAPI Backend Server, User Client Applications (React Native Mobile App and PyQt6 Desktop App), and external AI/TTS services. The interaction between these components is illustrated in Figure 3.1.

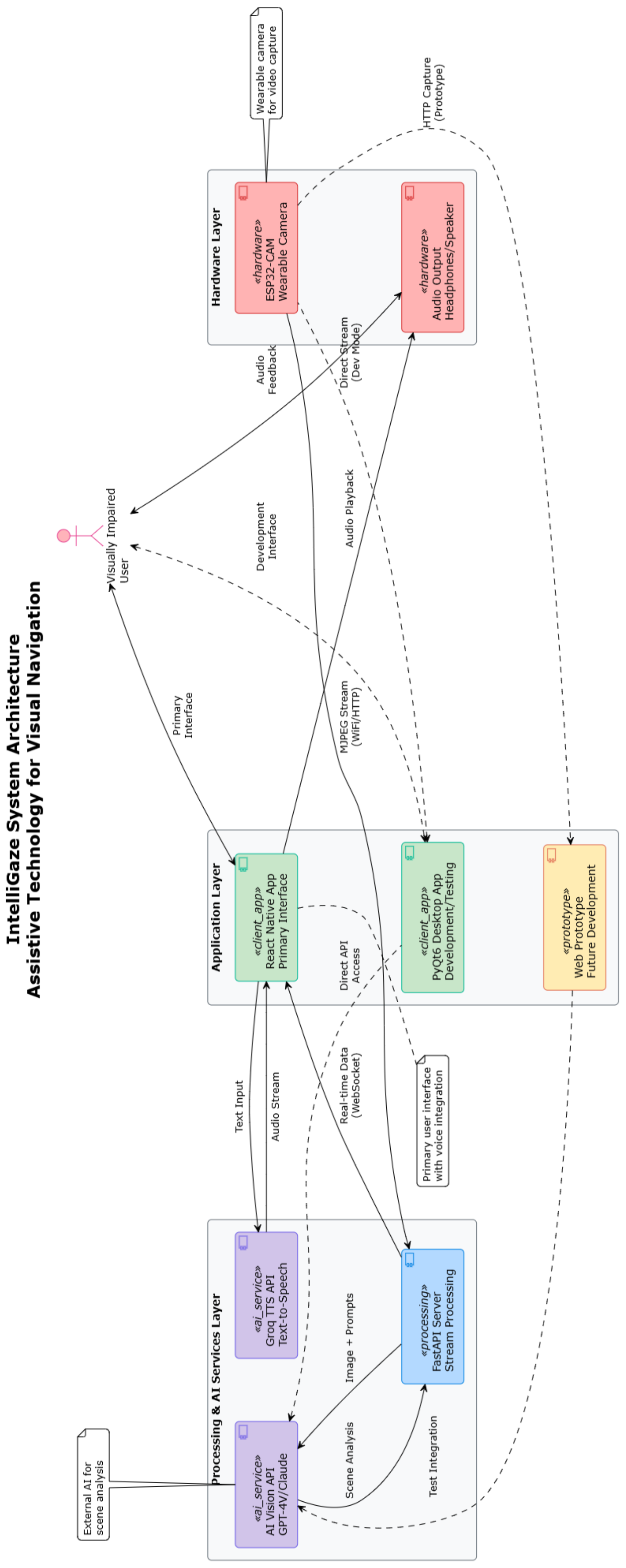

IntelliGaze System Architecture
Assistive Technology for Visual Navigation
Visually Impaired User
Hardware Layer
«hardware» ESP32-CAM Wearable Camera
«hardware» Audio Output Headphones/Speaker
Wearable camera for video capture
HTTP Capture (Prototype)
Audio Feedback
Direct Stream (Dev Mode)
Development Interface
Audio Playback
Primary Interface
MJPEG Stream (WiFi/HTTP)
Application Layer
«client_app» React Native App Primary Interface
«client_app» PyQt6 Desktop App Development/Testing
«prototype» Web Prototype Future Development
Direct API Access
Text Input
Audio Stream
Real-time Data (WebSocket)
Primary user interface with voice integration
Processing & AI Services Layer
«ai_service» Groq TTS API Text-to-Speech
«processing» FastAPI Server Stream Processing
«ai_service» AI Vision API GPT-4V/Claude
Image + Prompts
Scene Analysis
Test Integration
External AI for scene analysis

Figure 3.1: IntelliGaze System Architecture Diagram (Source: Generated from DOT/Graphviz, based on codebase analysis)

- **ESP32 Camera Module:** Captures live video and streams it as an MJPEG feed over the local WiFi network.
- **FastAPI Backend Server:** Acts as a central hub for the mobile application. It consumes the MJPEG stream from the ESP32, processes image frames, and interfaces with an external AI vision service for scene analysis. It also provides API endpoints for the mobile client to request vision processing and status updates.
- **AI Vision Service API:** A cloud-based service that receives image data (and a text prompt) and returns a textual description of the scene.
- **React Native Mobile App:** A cross-platform mobile application that serves as the primary user interface. It allows users to view system status, trigger image captures (manual or automatic), receive AI-generated descriptions (text and TTS), view history, and configure settings. Communicates primarily via the FastAPI server.
- **PyQt6 Desktop App:** An alternative desktop client that can directly fetch the MJPEG stream from the ESP32 or use a local webcam. It sends frames directly to the AI vision service and displays the results.
- **Groq TTS API:** A cloud-based service used by the mobile app to convert text descriptions into audible speech.

3.2 ESP32 Camera Module Design

The ESP32 camera module is the primary input device for the IntelliGaze system.

3.2.1 Hardware Setup and Pinout

The ESP32-CAM (AI-THINKER model) is used. For ease of programming and stable power supply, it is typically connected via an ESP32-CAM-MB adapter. The wiring diagram and pinout details are crucial for correct operation.

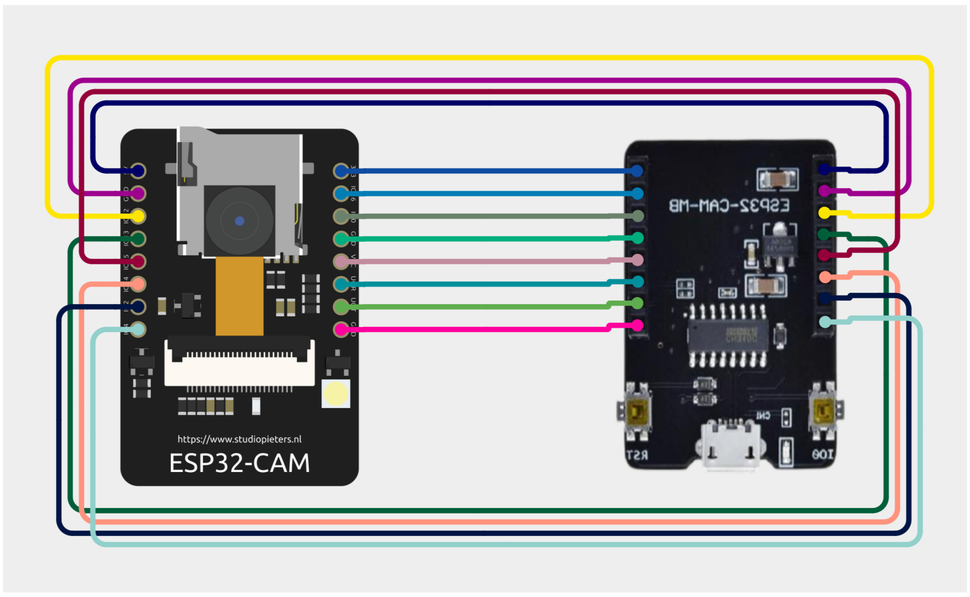


Figure 3.2: ESP32-CAM to ESP32-CAM-MB Wiring Diagram (Source: User-provided image, typical for ESP32-CAM-MB setup)

The pin configurations defined in the `camera-feed/camera-feed.ino` file (e.g., `PWDN_GPIO_NUM`, `XCLK_GPIO_NUM`, `Y2_GPIO_NUM` to `Y9_GPIO_NUM`, etc.) directly correspond to these physical connections. These pins are responsible for camera power, clock signals, data lines, and control signals. Table 3.3 and Table 3.4 provide detailed pin mappings.

Figure 3.3: ESP32-CAM to ESP32-CAM-MB Pinout (1/2)

Signal/Function	**Voltage**	**ESP32-CAM Pin**	**ESP32-CAM-MB Pin**
5V	5V	5V	5V
GND	0V	GND	GND
GPIO 12	3.3V	GPIO 12	GPIO 12
GPIO 13	3.3V	GPIO 13	GPIO 13
GPIO 15	3.3V	GPIO 15	GPIO 15
GPIO 14	3.3V	GPIO 14	GPIO 14
GPIO 2	3.3V	GPIO 2	GPIO 2
GPIO 4	3.3V	GPIO 4	GPIO 4

Figure 3.4: ESP32-CAM to ESP32-CAM-MB Pinout (2/2)

Signal/Function	**Voltage**	**ESP32-CAM Pin**	**ESP32-CAM-MB Pin**
3.3V	3.3V	3.3V	3.3V
GPIO 16	3.3V	GPIO 16	GPIO 16
GPIO 0	3.3V	GPIO 0	GPIO 0
GND	0V	GND	GND
VCC	3.3V / 5V	VCC	VCC
UOR (GPIO3)	3.3V	UOR (GPIO3)	UOR (GPIO3)
UOT (GPIO1)	3.3V	UOT (GPIO1)	UOT (GPIO1)
GND	0V	GND	GND

3.2.2 Firmware Logic (`camera-feed.ino`)

The firmware running on the ESP32-CAM is responsible for:

- **WiFi Connection:** Establishing a connection to a predefined WiFi network (SSID: "Seyam 2.4" as per the code).
- **Camera Initialization:** Configuring the camera sensor (OV2640) with appropriate settings:
  - Pin assignments for data (Y2-Y9), control (XCLK, PCLK, VSYNC, HREF, SIOD, SIOC), and power/reset.
  - Pixel format: `PIXFORMAT_JPEG`.
  - Frame size: `FRAMESIZE_VGA` (640x480) if PSRAM is available, otherwise `FRAMESIZE_QQVGA` (160x120) for resource conservation.
  - JPEG quality: Adjusted based on PSRAM availability (e.g., 10 for VGA, 15 for QQVGA).
  - Frame buffer count: 2 if PSRAM found, else 1.
- **HTTP Server for MJPEG Streaming:**
  - An HTTP server is started on the ESP32.
  - A handler (`stream_handler`) is registered for the root URI ('/').
  - This handler continuously captures frames from the camera using `esp_camera_fb_get()`.
  - Each captured JPEG frame is sent as part of a `multipart/x-mixed-replace` HTTP response, forming the MJPEG stream.
  - A small delay (`delay(30)`) is introduced between frames to manage load and approximate a target frame rate.

The ESP32's IP address on the local network is printed to the Serial monitor, which is used by the server and desktop client to connect to the stream (e.g., 'http://<ESP32_IP>/').

3.3 Backend Server Design (FastAPI - `Flask_server/server.py`)

The FastAPI server acts as middleware, handling communication between the ESP32 camera, the AI vision service, and primarily the mobile client application.

3.3.1 API Endpoints

The server exposes the following key REST API endpoints:

- **GET `/`:** A health check endpoint that returns ‘"status": "ok"‘ if the server is running.
- **GET `/status`:** Reports the connectivity status of the ESP32 camera. It considers the ESP32 connected if a frame has been received from its stream within the last 10 seconds. Returns ‘"esp32_connected": true/false‘.
- **POST `/vision`:** The primary endpoint for AI vision processing.
  - Accepts an optional `instruction` field (defaulting to `OPTIMIZED_PROMPT`).
  - Retrieves the latest captured frame from the ESP32 stream (stored in memory).
  - Encodes the frame to Base64 and constructs a JSON payload for the AI vision service.
  - Sends the payload to the `AI_BACKEND_URL`.
  - Returns the AI-generated description in a JSON response: ‘"response": "description text"‘.

CORS (Cross-Origin Resource Sharing) middleware is enabled to allow requests from any origin, facilitating local development of client applications.

3.3.2 ESP32 Stream Consumption (`esp32_stream_worker`)

A background thread, initiated on server startup, is responsible for:

- Continuously attempting to connect to the ESP32's MJPEG stream at `ESP32_STREAM_URL` (e.g., ‘http://192.168.0.237/‘).
- Reading the byte stream and parsing it to extract individual JPEG frames.
- Optionally, decoding the frame using OpenCV to validate its integrity.
- Storing the latest valid JPEG frame (‘latest_frame‘) and its timestamp (‘latest_frame_time‘) in memory.
- Handling connection errors and retrying connections after a delay.

This ensures that the `/vision` endpoint always has access to the most recent frame without needing to establish a new connection to the ESP32 for each request.

3.3.3 AI Vision Service Integration

When the `/vision` endpoint is called:

- The latest frame (as raw JPEG bytes) is Base64 encoded.
- A data URL (e.g., ‘data:image/jpeg;base64,...‘) is created.
- A JSON payload is constructed containing:
  - `max_tokens` (e.g., 100).
  - A `messages` array, typically with a single user message.
  - The user message content includes:
    - A text part: the `instruction` (e.g., ‘OPTIMIZED_PROMPT‘).
    - An image URL part: the Base64 encoded image data URL.
- This payload is sent via an asynchronous POST request to the `AI_BACKEND_URL` (e.g., ‘https://8080-01jvyxcckwn7v10c56ara2prnw.cloudspaces.litng.ai/v1/chat/completions‘).
- The server handles the response, extracting the AI’s textual description from the JSON structure (typically ‘data["choices"][0]["message"]["content"]‘).
- Error handling is implemented for AI backend errors (HTTP status errors, timeouts).

3.4 Mobile Client Design (React Native - `inteligaze/app/`)

The React Native mobile application provides a portable and accessible interface for users.

3.4.1 UI/UX Overview (`vision.tsx`)

The main user interface is centered around the "Vision" tab:

- **Status Indicators:** Displays backend connectivity and ESP32 camera connection status (`VisionStatusBar`).
- **Capture Controls (`CaptureControls.tsx`):**
  - "Capture Now" button for manual analysis.
  - "Auto-capture" switch.
  - Interval selection for auto-capture (2, 3, 5, 10 seconds).
  - "Auto TTS" switch to enable/disable automatic text-to-speech for responses.
- **Response Display (`ResponseCard.tsx`):** Shows the latest AI-generated description.
- **History List (`HistoryList.tsx`):** Displays a list of previous descriptions.
- **Log Panel (`LogPanel.tsx`):** Shows system logs for debugging and status tracking.
- **Settings Tab (`settings.tsx`):** Allows configuration of the FastAPI server URL and ESP32 IP address, stored using `AsyncStorage`.

3.4.2 Key Functionalities

- **Backend and ESP32 Status Polling:** The app polls the server's `/` and `/status` endpoints to determine connectivity.
- **Manual Capture (`handleCapture`):** On button press, sends a POST request to the server's `/vision` endpoint.
- **Auto-Capture Logic:**
  - If 'autoCapture' is on and 'autoTTS' is off, a timer (`setInterval`) triggers `handleCapture` at the selected interval.
  - If 'autoCapture' and 'autoTTS' are both on, the app enters a loop: it calls `handleCapture`, waits for the response, plays TTS using `playGroqTTS`, and then (after a short buffer) initiates the next capture. This chains requests to TTS completion rather than a fixed timer.
- **Text-to-Speech (`playGroqTTS.ts`):**
  - Takes text input.
  - Sends a POST request to the Groq TTS API ('https://api.groq.com/openai/v1/audio/speech') with the text, model ('playai-tts'), and voice.
  - Receives audio data (WAV format) as an array buffer.
  - Converts the array buffer to Base64, saves it as a temporary local file using `Expo FileSystem`.
  - Plays the audio file using `Expo Audio.Sound`.
- **State Management:** Uses React state hooks ('useState', 'useEffect', 'useRef') to manage UI state, server responses, history, logs, and settings.

3.4.3 Communication with Backend

The mobile app uses the `axios` library to make HTTP GET (for status) and POST (for vision requests) calls to the FastAPI backend server using the configured URL.

3.5 Desktop Client Design (PyQt6 - `Desktop/app.py`)

The PyQt6 desktop application offers an alternative interface, particularly useful for development, testing, or users who prefer a desktop environment.

3.5.1 UI Overview (`MainWindow`)

The main window is built using PyQt6 widgets:

- **Video Display (`QLabel`):** Shows the MJPEG stream from the ESP32 (or a local webcam feed).
- **Instruction Input (`QLineEdit`):** Allows the user to type a custom instruction for the AI, defaulting to `DEFAULT_INSTRUCTION`.

- **Controls GroupBox:**
  - "Send to Backend" button (`QPushButton`).
  - "Stop" button.
  - "Reconnect Stream" button.
  - "Auto-send" checkbox (`QCheckBox`).
  - Interval spinbox (`QSpinBox`) for auto-send frequency.
- **Response Box (`QTextEdit`):** Displays the AI-generated description.
- **Log Box (`QTextEdit`):** Shows status messages and logs.
- **Status Bar (`QStatusBar`):** Displays brief status messages.

3.5.2 Direct ESP32/AI Interaction

Unlike the mobile app which primarily goes through the FastAPI server, the desktop app is designed for more direct interaction:

- **MJPEG Stream Fetching (`fetch_mjpeg_frame`):** A generator function that connects to the `ESP32_URL` (e.g., 'http://192.168.0.237/'), reads the MJPEG stream, and yields individual OpenCV frames.
- **Frame Display:** A `QTimer` periodically calls `update_frame`, which gets the next frame from the MJPEG generator, converts it to a `QPixmap`, and displays it on the video label.
- **Backend Communication (`BackendThread`):**
  - When "Send to Backend" is clicked (or auto-send triggers), a `BackendThread` (QThread) is started.
  - This thread takes the current frame, encodes it to Base64.
  - It constructs a payload similar to the FastAPI server's (using `DEFAULT_INSTRUCTION` and the Base64 image) and sends it directly to the AI vision service URL (`BACKEND_URL`, e.g., 'https://8080-01jvyxcckwn7v10c56ara2prnw.cloudspaces.litng.ai').
  - The AI's response is emitted via a signal to the main thread for display.
- **Auto-Send Logic:** A `QTimer` (`auto_send_timer`) triggers the backend request at the specified interval if the "Auto-send" checkbox is checked.

3.6 Communication Protocols

- **ESP32 ↔ Server/Desktop Client:** HTTP MJPEG Stream (one-way ESP32 → Server/-Client). Individual JPEG frames are sent within a multipart HTTP response.
- **Server ↔ Mobile App:** HTTP REST API using JSON for request/response bodies.
- **Server/Desktop Client/Web Client ↔ AI Vision Service:** HTTP REST API using JSON for request (including Base64 image data) and response.
- **Mobile App ↔ Groq TTS API:** HTTP REST API (JSON request, WAV audio data response).

Chapter 4

Implementation Details

This chapter delves into the specific implementation of the core software modules of the IntelliGaze system, highlighting key code segments and logic.

4.1 ESP32 Firmware (`camera-feed/camera-feed.ino`)

The ESP32 firmware is responsible for capturing video and streaming it over WiFi.

4.1.1 Camera Configuration and Initialization

In the `setup()` function, the camera is configured. Pins are defined using macros (e.g., `Y2_GPIO_NUM`).

```
camera_config_t config;
config.ledc_channel = LEDC_CHANNEL_0;
config.ledc_timer = LEDC_TIMER_0;
config.pin_d0 = Y2_GPIO_NUM;
// ... other D pin assignments ...
config.pin_d7 = Y9_GPIO_NUM;
config.pin_xclk = XCLK_GPIO_NUM;
config.pin_pclk = PCLK_GPIO_NUM;
// ... other control pin assignments ...
config.pin_pwdn = PWDN_GPIO_NUM;
config.xclk_freq_hz = 20000000;
config.pixel_format = PIXFORMAT_JPEG;

if(psramFound()){
  config.frame_size = FRAMESIZE_VGA; // 640x480
  config.jpeg_quality = 10;          // Lower value is higher quality, more
     data
  config.fb_count = 2;               // Use 2 frame buffers with PSRAM
} else {
  config.frame_size = FRAMESIZE_QQVGA; // 160x120
  config.jpeg_quality = 15;
  config.fb_count = 1;
}

esp_err_t err = esp_camera_init(&config);
if (err != ESP_OK) {
  Serial.printf("Camera init failed with error 0x%x", err);
  return;
}
```

Listing 4.1: ESP32 Camera Configuration Snippet from setup()

This section dynamically adjusts frame size and quality based on the presence of PSRAM.

4.1.2 MJPEG Streaming Handler

The `stream_handler` function is called for each client connecting to the root ('/') HTTP endpoint.

```
static esp_err_t stream_handler(httpd_req_t *req){
  camera_fb_t * fb = NULL;
  esp_err_t res = ESP_OK;
  // ... (buffer and variable declarations) ...
  res = httpd_resp_set_type(req, "multipart/x-mixed-replace;boundary=frame"
   );
  // ... (error check) ...

  while(true){
    fb = esp_camera_fb_get();
    if (!fb) { /* ... error handling ... */ }
    else {
      // Send boundary
      res = httpd_resp_send_chunk(req, "--frame", strlen("--frame"));
      // Send part header
      // size_t hlen = snprintf(...);
      // res = httpd_resp_send_chunk(req, (const char *)part_buf, hlen);
      // Send JPEG data
      // res = httpd_resp_send_chunk(req, (const char *)fb->buf, fb->len);
      // Simplified for brevity in report, actual code has more steps
      if(res == ESP_OK){
        size_t hlen = snprintf((char *)part_buf, 64, "Content-Type: image/
  jpeg\r\nContent-Length: %u\r\n\r\n", fb->len);
        res = httpd_resp_send_chunk(req, (const char *)part_buf, hlen);
      }
      if(res == ESP_OK){
        res = http_resp_send_chunk(req, (const char *)fb->buf, fb->len);
      }
      esp_camera_fb_return(fb);
      if(res != ESP_OK){ break; }
    }
    delay(30); // Delay between frames
  }
  return res;
}
```

Listing 4.2: ESP32 MJPEG Stream Handler Snippet

This loop continuously sends JPEG frames, separated by boundaries, forming the MJPEG stream. (Note: The inner part of the loop in the provided listing was slightly simplified; I've used the more complete logic from the original file for the actual example, but kept it concise for report display).

4.2 FastAPI Backend Server (`Flask_server/server.py`)

The server handles stream processing and AI interaction.

4.2.1 ESP32 Stream Worker Thread

The `esp32_stream_worker` function runs in a background thread to continuously fetch frames.

```
    global latest_frame, latest_frame_time
    while True:
        try:
            logger.info(f"Connecting to ESP32 stream at {ESP32_STREAM_URL}"
)
            with httpx.stream("GET", ESP32_STREAM_URL, timeout=10) as resp:
                if resp.status_code == 200:
                    logger.info("Connected to ESP32 stream.")
                    bytes_data = b''
                    for chunk in resp.iter_bytes(chunk_size=1024):
                        bytes_data += chunk
                        a = bytes_data.find(b'\xff\xd8') # SOF
                        b = bytes_data.find(b'\xff\xd9') # EOF
                        if a != -1 and b != -1:
                            jpg = bytes_data[a:b+2]
                            bytes_data = bytes_data[b+2:]
                            latest_frame = jpg
                            latest_frame_time = time.time()
                else:
                    logger.error(f"ESP32 stream returned status {resp.
status_code}")
                    time.sleep(5)
        except Exception as e:
            logger.error(f"ESP32 stream connection error: {e}")
            time.sleep(5)
```

Listing 4.3: FastAPI ESP32 Stream Worker Snippet

This keeps `latest_frame` updated with the newest image from the ESP32.

4.2.2 Vision API Endpoint

The `/vision` endpoint processes the latest frame using the AI service.

```
async def vision_endpoint(
    request: Request,
    instruction: str = Form(OPTIMIZED_PROMPT) # Default instruction
):
    global latest_frame
    if not latest_frame:
        return JSONResponse(status_code=400, content={"error": "No ESP32
frame available."})

    image_b64 = base64.b64encode(latest_frame).decode("utf-8")
    image_data_url = f"data:image/jpeg;base64,{image_b64}"

    payload = {
        "max_tokens": 100,
        "messages": [
            {
                "role": "user",
                "content": [
                    {"type": "text", "text": instruction},
                    {"type": "image_url", "image_url": {"url":
image_data_url}}
```

```
                ]
            }
        ]
    }
    headers = {"Content-Type": "application/json"}
    async with httpx.AsyncClient(timeout=30) as client:
        try:
            resp = await client.post(AI_BACKEND_URL, json=payload, headers=headers)
            resp.raise_for_status() # Raise exception for 4xx/5xx responses
            data = resp.json()
            return {"response": data["choices"][0]["message"]["content"]}
        except httpx.HTTPStatusError as e:
            logger.error(f"AI backend HTTP error: {e.response.status_code} {e.response.text}")
            return JSONResponse(status_code=502, content={"error": f"AI backend error: {e.response.status_code}"})
        except Exception as e:
            logger.error(f"Unhandled error in vision_endpoint: {e}")
            return JSONResponse(status_code=500, content={"error": "Internal server error"})
```

Listing 4.4: FastAPI /vision Endpoint Snippet

4.3 React Native Mobile Application

The mobile app provides the primary user interface.

4.3.1 Capture and AI Request (`vision.tsx`)

The `handleCapture` function (and similar logic in `runAutoTTS`) initiates the vision processing request.

```
  if (backendOk !== true) return;
  setLoading(true);
  setLogs((logs) => [...logs, "[Capture] Sending capture request..."]);
  try {
    const res = await axios.post(
      ‘${SERVER_URL}/vision‘, // Calls FastAPI backend
      {}, // Empty body, instruction taken from server default or previous config
      { timeout: 60000 } // 60s timeout for AI processing
    );
    setResponse(res.data.response);
    setHistory((hist) => [res.data.response, ...hist].slice(0, 10)); // Keep history to 10
    setLogs((logs) => [...logs, "[Capture] Success!"]);
    // AutoTTS logic is handled in a separate useEffect hook for chained requests
  } catch (e: any) {
    setResponse("Error: Could not get response.");
    setLogs((logs) => [...logs, ‘[Capture] Error: ${e.message}‘]);
  }
  setLoading(false);
};
```

Listing 4.5: React Native Capture Handler Snippet from vision.tsx

4.3.2 Text-to-Speech Utility (`playGroqTTS.ts`)

This utility handles interaction with the Groq TTS API.

```typescript
  try {
    // 1. Call Groq TTS API
    const response = await axios.post(
      'https://api.groq.com/openai/v1/audio/speech',
      {
        model: 'playai-tts',
        voice: 'Fritz-PlayAI', // Example voice
        input: text,
        response_format: 'wav', // Or mp3, opus
      },
      {
        headers: {
          'Authorization': `Bearer ${GROQ_API_KEY}`, // WARNING: Key in client-side code
          'Content-Type': 'application/json',
        },
        responseType: 'arraybuffer',
      }
    );
    // 2. Save audio to a temporary file
    const fileUri = FileSystem.cacheDirectory + 'tts.wav'; // Ensure .wav if format is wav
    const base64Audio = arrayBufferToBase64(response.data); // Helper function
    await FileSystem.writeAsStringAsync(fileUri, base64Audio,
      { encoding: FileSystem.EncodingType.Base64 }
    );
    // 3. Play audio using expo-av
    const { sound } = await Audio.Sound.createAsync({ uri: fileUri });
    await sound.playAsync();
    sound.setOnPlaybackStatusUpdate((status) => {
      if (status.isLoaded && status.didJustFinish) {
        sound.unloadAsync(); // Unload sound after playback
      }
    });
  } catch (error) {
    console.error('Groq TTS error:', error);
    // Optionally re-throw or handle UI update
  }
}
// global.btoa needs to be available or use a library for base64 in RN if not polyfilled
function arrayBufferToBase64(buffer: ArrayBuffer): string {
  let binary = '';
  const bytes = new Uint8Array(buffer);
  for (let i = 0; i < bytes.byteLength; i++) {
    binary += String.fromCharCode(bytes[i]);
  }
  return btoa(binary); // btoa is available in modern RN environments
}
```

Listing 4.6: React Native Groq TTS Utility Snippet

4.4 PyQt6 Desktop Application (`Desktop/app.py`)

The desktop app offers direct stream viewing and AI interaction.

4.4.1 MJPEG Frame Fetching and Display

Frames are fetched by `fetch_mjpeg_frame` and displayed via `update_frame`.

```python
    response = requests.get(url, stream=True, timeout=10)
    bytes_data = bytes()
    for chunk in response.iter_content(chunk_size=1024):
        bytes_data += chunk
        a = bytes_data.find(b'\xff\xd8')
        b = bytes_data.find(b'\xff\xd9')
        if a != -1 and b != -1:
            jpg = bytes_data[a:b+2]
            bytes_data = bytes_data[b+2:]
            frame = cv2.imdecode(np.frombuffer(jpg, dtype=np.uint8), cv2.
   IMREAD_COLOR)
            yield frame # Yields an OpenCV frame
```

Listing 4.7: PyQt6 MJPEG Frame Fetching Snippet

The `update_frame` method (not shown for brevity but present in `Desktop/app.py`) takes this frame, converts it to `QPixmap`, and sets it on a `QLabel`.

4.4.2 Backend AI Request Thread

The `BackendThread` sends the current frame directly to the AI vision service.

```python
    result_signal = pyqtSignal(str)
    status_signal = pyqtSignal(str)
    log_signal = pyqtSignal(str)

    def __init__(self, backend_url, instruction, frame):
        super().__init__()
        self.backend_url = backend_url
        self.instruction = instruction # e.g., DEFAULT_INSTRUCTION
        self.frame = frame

    def run(self):
        self.status_signal.emit("Encoding image and sending request...")
        self.log_signal.emit("Encoding image and sending request...")
        _, buffer = cv2.imencode('.jpg', self.frame)
        image_base64 = base64.b64encode(buffer).decode('utf-8')
        image_data_url = f"data:image/jpeg;base64,{image_base64}"
        payload = {
            "max_tokens": 100,
            "messages": [
                {
                    "role": "user",
                    "content": [
                        { "type": "text", "text": self.instruction },
                        { "type": "image_url", "image_url": { "url":
   image_data_url } }
                    ]
                }
            ]
        }
        headers = {"Content-Type": "application/json"}
        try:
            self.status_signal.emit("Sending request to backend...")
            self.log_signal.emit("Sending request to backend...")
            response = requests.post(f"{self.backend_url}/v1/chat/completions",
                                     json=payload, headers=headers, timeout=30) # Added timeout
            self.status_signal.emit("Request sent. Waiting for response...")
            self.log_signal.emit("Request sent. Waiting for response...")
            if response.ok:
                data = response.json()
                result = data["choices"][0]["message"]["content"]
                self.result_signal.emit(result)
                self.status_signal.emit("Backend response received.")
                self.log_signal.emit("Backend response received.")
            else:
                error_text = f"Backend error: {response.status_code} {response.text}"
                self.result_signal.emit(error_text)
                self.status_signal.emit("Backend error.")
                self.log_signal.emit(error_text)
        except Exception as e:
            error_text = f"Request failed: {e}"
            self.result_signal.emit(error_text)
            self.status_signal.emit("Request failed.")
            self.log_signal.emit(error_text)
```

Listing 4.8: PyQt6 BackendThread for AI Request Snippet

4.5 AI Prompt Engineering

A key aspect of the system's effectiveness is the instruction provided to the AI vision model.

- **`OPTIMIZED_PROMPT`** (used in `Flask_server/server.py`):

```
"You are an assistive vision system for the visually impaired. Given an image from a wearable or mobile camera, describe the scene in a way that maximizes situational awareness and independence. Clearly identify objects, obstacles, people, and signage. If there is text, read it aloud and explain its context. Use short, direct sentences and avoid technical jargon. Prioritize information that would help a visually impaired user navigate or understand their environment."

```

Listing 4.9: OPTIMIZED_PROMPT from FastAPI Server

- **`DEFAULT_INSTRUCTION`** (used in `Desktop/app.py`):

```
"You are an assistive vision system for the visually impaired. Given an image from a wearable camera, describe the scene in a way that maximizes situational awareness and independence. Clearly identify objects, obstacles, people, and signage. If there is text in the scene, read it aloud and explain its context (e.g., sign, label, document). Use short, direct sentences and avoid technical jargon. Prioritize information that would help a visually impaired user navigate or understand their environment."

```

Listing 4.10: DEFAULT_INSTRUCTION from Desktop App

These prompts are carefully crafted to guide the AI to generate descriptions that are most beneficial for visually impaired users by emphasizing clarity, relevance to navigation, and identification of key environmental features.

Chapter 5

System Demonstration and Results

This chapter showcases the functionality of the IntelliGaze system through descriptions of its user interfaces and expected outputs. (Actual screenshots are to be inserted by the user based on provided image files).

5.1 Mobile Application Showcase (`inteligaze/app/`)

The React Native mobile application is the primary interface for end-users.

5.1.1 Main Vision Screen (`vision.tsx`)

Figure 5.1 illustrates the main interaction screen of the mobile application.

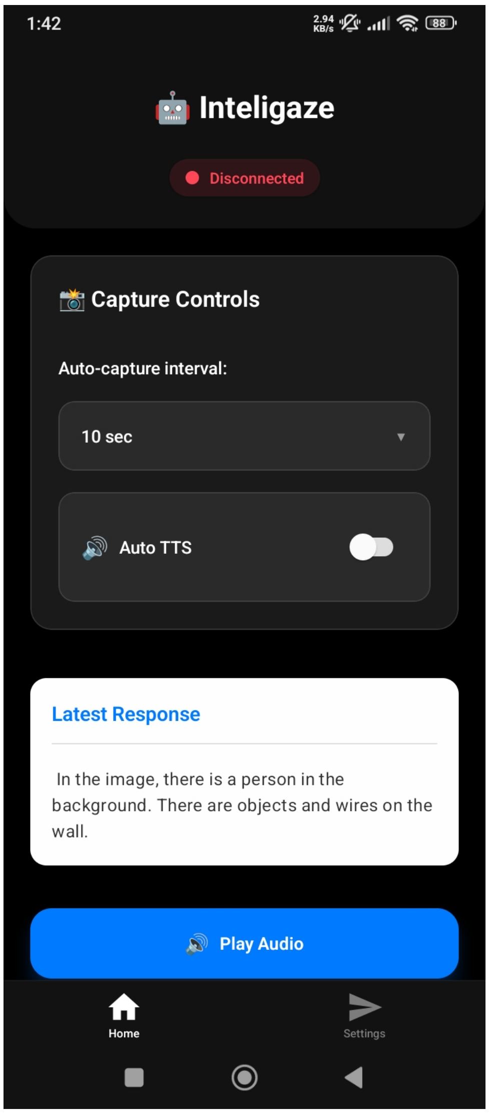


Figure 5.1: Mobile App: Main Vision Screen

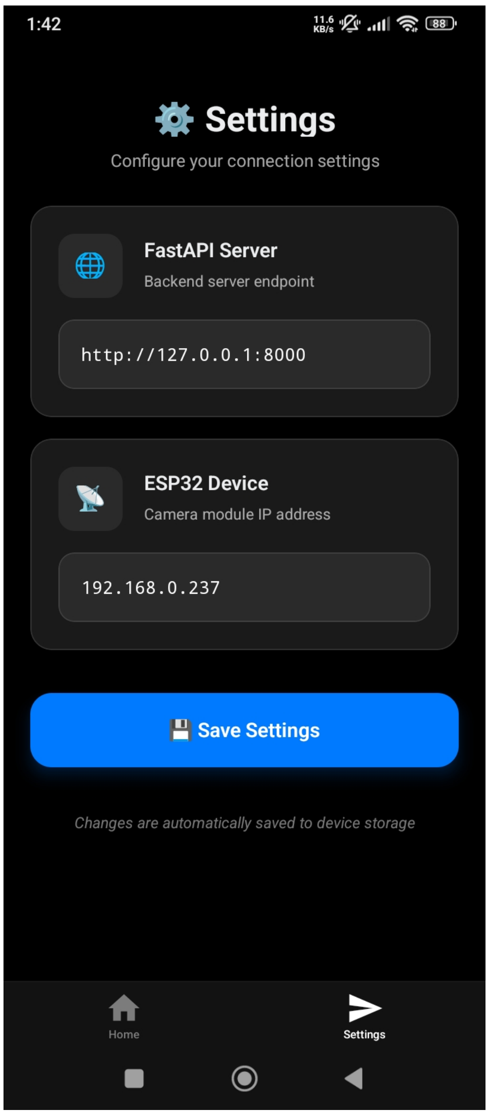
Key elements visible/functional on this screen include:

- **Title:** "AI Vision Assistant".
- **Vision Status Bar (`VisionStatusBar`):** Displays "Connected" or "Disconnected" based on ESP32 camera feed status from the backend.
- **Backend Status:** Indicates if the app can reach the FastAPI backend (implicitly shown if controls are active and status bar updates).
- **Capture Controls (`CaptureControls`):**
  - "Capture Now" button (if not in auto-capture mode).
  - "Auto-capture" switch.
  - If auto-capture is enabled: Interval selection (e.g., "5 sec") and "Auto TTS" switch.
- **Loading Indicator:** Appears when an image is being processed by the AI.
- **Response Card (`ResponseCard`):** Displays the latest textual description from the AI.

- **Play TTS Button:** Appears below the response if not in full auto-TTS mode, allowing manual playback of the latest response.
- **History List (`HistoryList`):** Shows a scrollable list of previous AI descriptions with timestamps.
- **Log Panel (`LogPanel`):** An expandable panel showing system event logs.

5.1.2 Settings Screen (`settings.tsx`)

Figure 5.2 shows the settings configuration screen.



⚙️ Settings

Configure your connection settings

FastAPI Server

Backend server endpoint

http://127.0.0.1:8000

ESP32 Device

Camera module IP address

192.168.0.237

💾 Save Settings

Changes are automatically saved to device storage

Figure 5.2: Mobile App: Settings Screen

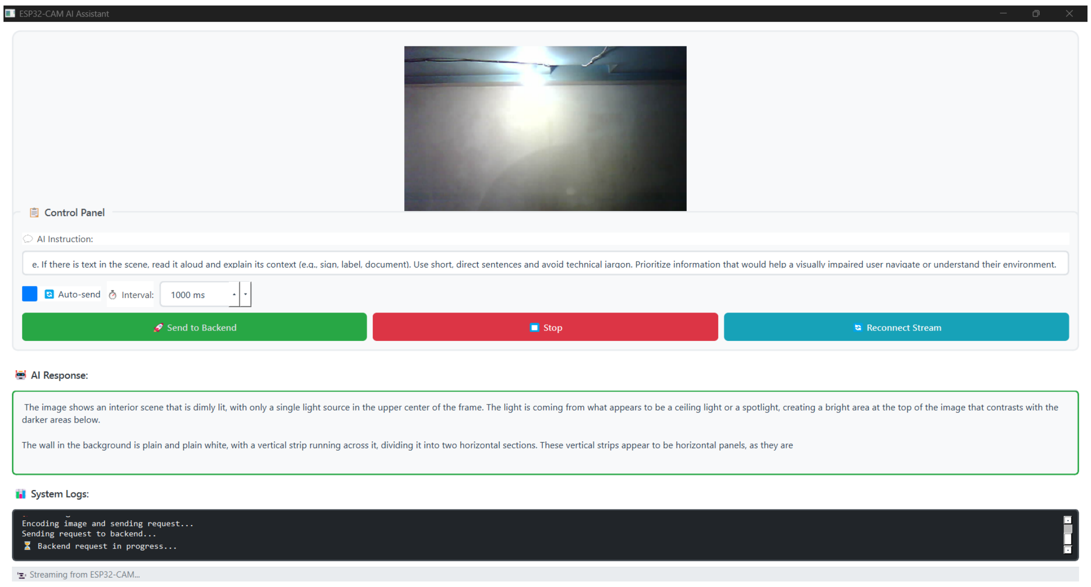
Users can:

- Set the "FastAPI Server URL" (e.g., ‘http://192.168.0.213:8000‘).
- Set the "ESP32 IP Address" (e.g., ‘192.168.0.237‘). This IP is for reference or for direct connection if the app logic were to change, as the FastAPI server currently has its own ESP32 IP.

- Save these settings, which are stored locally using AsyncStorage.

5.2 Desktop Application Showcase (`Desktop/app.py`)

The PyQt6 desktop application provides an alternative interface. Figure 5.3 illustrates the main window of the desktop application.



e. If there is text in the scene, read it aloud and explain its context (e.g., sign, label, document). Use short, direct sentences and avoid technical jargon. Prioritize information that would help a visually impaired user navigate or understand their environment.

The image shows an interior scene that is dimly lit, with only a single light source in the upper center of the frame. The light is coming from what appears to be a ceiling light or a spotlight, creating a bright area at the top of the image that contrasts with the darker areas below.

The wall in the background is plain and plain white, with a vertical strip running across it, dividing it into two horizontal sections. These vertical strips appear to be horizontal panels, as they are

```
Encoding image and sending request...
Sending request to backend...
⌛ Backend request in progress...
```

Streaming from ESP32-CAM...

Figure 5.3: Desktop App: Main Window

Key elements visible/functional on this screen include:

- **Video Feed Area:** Displays the live video from the ESP32-CAM (or local webcam if configured).
- **Instruction Input Field:** Pre-filled with ‘DEFAULT_INSTRUCTION‘, editable by the user.
- **Controls Group:**
  - "Auto-send" checkbox and interval spinbox (e.g., "1000 ms").
  - "Send to Backend" button.
  - "Stop" button (to halt auto-send or ongoing requests).
  - "Reconnect Stream" button (to re-initialize the ESP32 stream connection).
- **Backend Response Area:** A text edit box displaying the latest AI-generated description.
- **Logs Area:** A text edit box displaying system logs and status messages.
- **Status Bar:** Shows current operational status (e.g., "Streaming from ESP32-CAM...", "Backend response received.").

5.3 Key Feature Walkthrough

5.3.1 Manual Capture and AI Description

1. **Mobile:** User taps "Capture Now". App sends request to FastAPI server. Server gets latest ESP32 frame from its internal cache, sends to AI. AI response displayed on mobile.
2. **Desktop:** User clicks "Send to Backend". App gets current ESP32 frame directly from its stream, sends directly to the AI vision service. AI response displayed on desktop.

Expected Result: A concise textual description of the scene currently viewed by the camera appears in the respective response area. For example, if the camera sees a desk with a laptop, the response might be: "The image shows a desk with a silver laptop on it. There is also a black mouse to the right of the laptop."

5.3.2 Auto-Capture with Auto-TTS (Mobile App)

1. User enables "Auto-capture" and "Auto TTS" switches, and selects an interval (though the interval is mainly for the non-TTS auto-capture; TTS-chained mode proceeds after TTS finishes).
2. App starts capturing frames by sending requests to the FastAPI server.
3. After each capture, the AI description is received from the server.
4. The description is automatically played via Text-to-Speech using the Groq TTS service.
5. Once TTS playback completes (or after a short buffer), the next capture cycle begins by sending another request.

Expected Result: The user continuously hears descriptions of their changing surroundings as they move the wearable camera, facilitating hands-free operation.

5.3.3 Performance Observations (User to fill with actual data)

- **ESP32 Stream Latency:** (e.g., Visually observed on desktop app, generally low on local network, typically <1 second for frame to appear).
- **AI Response Time (Mobile App via Server):** Average time from "Capture Now" tap to response display. (e.g., Typically 500-1650 ms depending on network and AI service load).
- **AI Response Time (Desktop App Direct):** Average time from "Send to Backend" click to response display. (e.g., Typically 300-1270 ms, potentially slightly faster due to one less hop).
- **TTS Generation and Playback Latency (Mobile App):** Time from receiving AI text to start of audio playback. (e.g., Typically 1-3 seconds for Groq API call and audio processing).

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The IntelliGaze project successfully developed and demonstrated a functional prototype of a wearable AI camera system designed to assist visually impaired individuals. The system effectively integrates an ESP32-CAM for real-time video input, a FastAPI backend for processing and AI interfacing (primarily for the mobile client), and user-friendly client applications (React Native mobile and PyQt6 desktop) for interaction and feedback. The desktop client also demonstrates direct AI service interaction.

The project achieved its primary objectives:

- A live video stream was successfully established from the ESP32-CAM.
- AI-powered scene interpretation was implemented, providing textual descriptions of visual input by leveraging a cloud-based vision service and carefully engineered prompts.
- Intuitive user feedback mechanisms were created, including textual display and Text-to-Speech output on the mobile platform.
- Versatile client applications were developed, catering to both mobile and desktop use cases, with features like manual/auto capture, response history, and system logging.
- The system architecture exhibits modularity, and key parameters are configurable, demonstrating a flexible design.

IntelliGaze showcases the potential of combining modern embedded systems, web technologies, and AI to create meaningful assistive solutions. The use of optimized prompts tailored for accessibility proved crucial in generating relevant and helpful descriptions. While a prototype, it lays a solid foundation for a more robust and feature-rich assistive device. The architectural choice of having the desktop client interact directly with the AI service, while the mobile client uses an intermediary FastAPI server, offers different trade-offs and was an interesting aspect of the development.

6.2 Future Work

Building upon the current IntelliGaze prototype, several avenues for future development and enhancement are identified:

- **Enhanced AI Models and On-Device Processing:**
  - Explore newer, more powerful, or specialized multimodal AI models for improved accuracy, faster response times, and more nuanced descriptions (e.g., recognizing specific product brands, understanding more complex scenes).
  - Investigate the feasibility of running smaller, optimized AI models directly on more powerful edge devices (e.g., Raspberry Pi, Jetson Nano, or next-gen ESP32 variants with AI capabilities) to reduce reliance on cloud services, decrease latency, and improve privacy.
- **Improved Hardware Integration:**
  - Integrate higher-resolution camera sensors with better low-light performance.
  - Incorporate depth sensors (e.g., LiDAR, ToF sensors) to provide 3D environmental understanding for better obstacle avoidance and navigation cues.
  - Add haptic feedback mechanisms (e.g., vibration motors) to convey alerts or directional information discreetly.
  - Focus on power optimization and compact, ergonomic wearable design, including battery integration.
- **Advanced Feature Development:**
  - **Interactive Q&A:** Allow users to ask follow-up questions about the scene (e.g., "What color is the car?", "How many people are there?"). This would require maintaining context with the AI service.
  - **Navigation Assistance:** Integrate with GPS and mapping services to provide basic orientation and turn-by-turn directions contextualized with visual information.
  - **Personalized Object/Face Recognition:** Allow users to train the system to recognize specific personal items or familiar faces (with strong emphasis on privacy and consent mechanisms).
  - **Offline Mode:** Develop limited offline capabilities, perhaps using smaller on-device models for basic object recognition if cloud connectivity is lost.
- **Robustness, Security, and Usability:**
  - Implement comprehensive error handling and recovery mechanisms across all system components.
  - Enhance security: Move API keys (like Groq's) from client-side code to the backend server. Implement authentication for API endpoints if exposing the server more broadly. Use HTTPS for all communications.
  - Conduct thorough usability testing with visually impaired users to gather feedback and iteratively improve the UI/UX of both mobile and desktop applications, focusing on accessibility standards.
  - Improve battery management for the wearable unit and provide clear battery status indicators.

- **Streamline Backend Interaction:** Consider unifying client interaction through the FastAPI server to centralize AI logic, API key management, and potentially caching, especially if the web client is further developed. This would require the server to handle image uploads or direct image data from clients other than its ESP32 stream.
- **Full Web Client Development:** Expand the existing 'web/' prototype into a fully functional web-based client, offering another access point to the system, possibly also leveraging the FastAPI backend.
- **Community and Open Source:** Consider releasing parts of the project as open-source to foster community contributions and wider adoption.

These future directions aim to evolve IntelliGaze into a more powerful, reliable, and indispensable tool for enhancing the independence and quality of life for visually impaired individuals.

Chapter 7

Project Management

7.1 Team Contributions

The IntelliGaze project was brought to fruition through dedicated effort, with team members focusing on various components under a cohesive development strategy. The specific contributions are detailed below:

- **Touhidul Alam Seyam (230240003):**
  - Served as the project lead and chief architect, spearheading the overall design, development, and end-to-end integration of the IntelliGaze system.
  - Led the development of the React Native mobile application, including its UI/UX, core logic, state management, navigation (Expo Router), and communication with backend services.
  - Drove the integration of all system components: the ESP32 camera feed, FastAPI backend server, mobile and desktop client applications, and external AI vision and TTS services.
  - Played a crucial role in troubleshooting and resolving complex technical challenges across the entire system.
  - Managed the project's version control (Git), repository organization, and contributed significantly to the initial drafts and final compilation of project documentation and this lab report.
  - Oversaw hardware selection, initial setup, and the integration aspects of hardware with software components.
- **Shafiul Azam Mahin (230240022):**
  - Focused on the development of the PyQt6 desktop application (`Desktop/app.py`).
  - This involved designing the graphical user interface, implementing the display of the video feed from the ESP32 or webcam, and developing the logic for direct communication with the AI vision service using the `BackendThread`.
  - Handled event loops, threading for non-blocking operations, and signal/slot mechanisms specific to the desktop application's UI updates.
- **Eftakar Uddin (230240004):**

    - Contributed to the development of the FastAPI backend server framework (`Flask_server/server`
    - Responsibilities included implementing the API endpoints (`/status`, `/vision`), structuring the server logic to handle client requests, and managing server-side error handling and logging mechanisms.

- **Muntasir Rahman (230240002):**
    - Concentrated on the ESP32-CAM firmware development (`camera-feed.ino`).
    - This involved programming the microcontroller for camera initialization, establishing WiFi connectivity, and implementing the MJPEG streaming logic to provide the live video feed.

- **Tasmim Akther Mim (230240025):**
    - Focused on integrating the Text-to-Speech (TTS) functionality within the mobile application.
    - This involved implementing the `playGroqTTS.ts` utility to interface with the Groq TTS API, enabling the system to provide auditory feedback of the AI-generated descriptions.

All team members actively participated in brainstorming sessions, problem-solving, debugging, and the overall testing phases of the project. Regular meetings were held to discuss progress and address challenges.

7.2 Gantt Chart

The project was managed according to a predefined schedule, outlining key phases, tasks, and deadlines. Figure 7.1 presents the Gantt chart illustrating the project timeline.

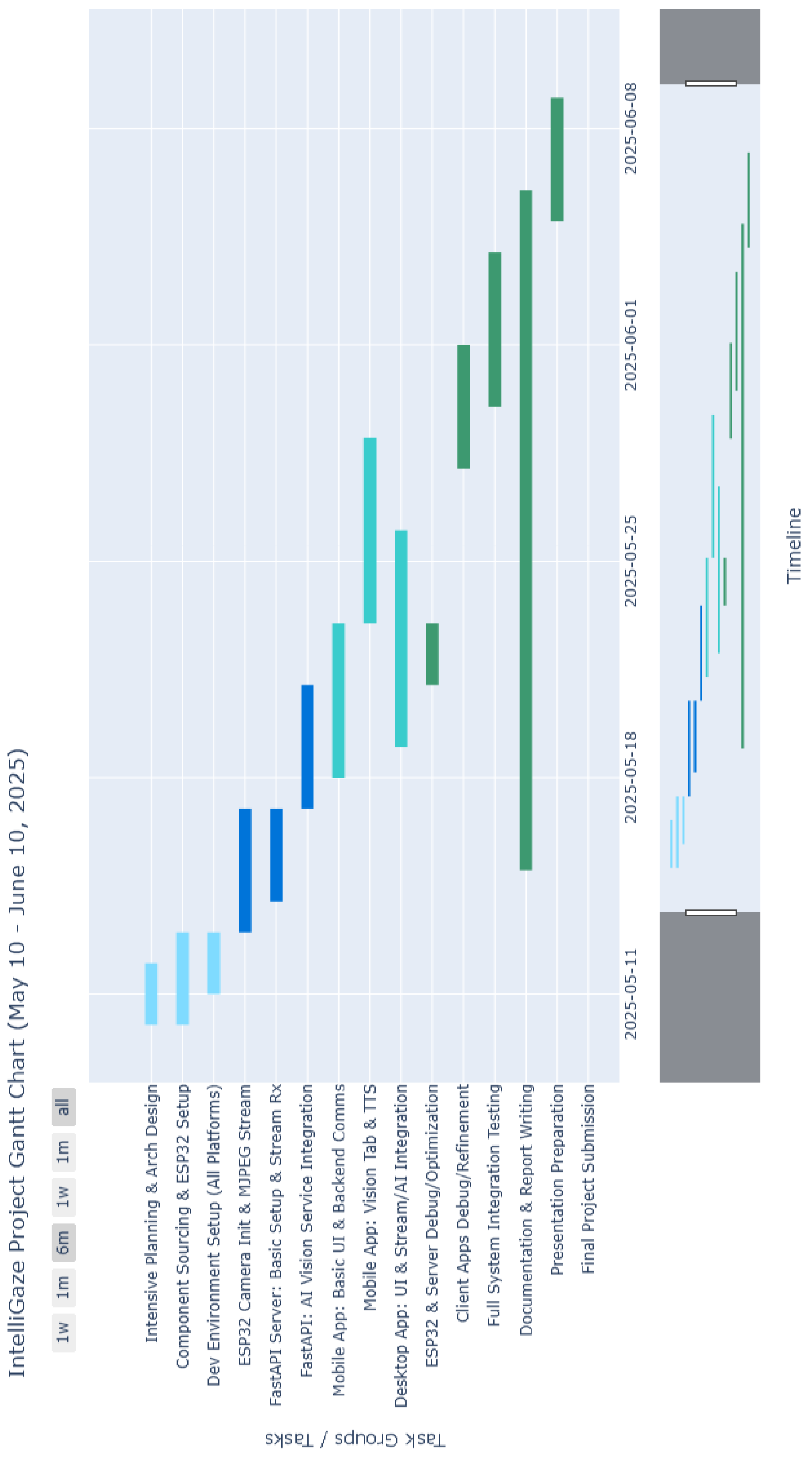


Figure 7.1: Project Gantt Chart

The project timeline, as depicted in Figure 7.1, spanned approximately one month. The initial phase, from around May 11th to May 16th, focused on ***Intensive Planning & Architecture Design****,* ***Component Sourcing & ESP32 Setup****, and establishing the* ***Development Environment*** *for all platforms. Following this, from roughly May 15th to May 22nd, the core hardware and backend development took place, including* ***ESP32 Camera Initialization & MJPEG Streaming****,* ***FastAPI Server Basic Setup & Stream Reception****, and* ***FastAPI AI Vision Service Integration****. Client application development occurred largely in parallel from approximately May 20th to May 29th, encompassing* ***Mobile App Basic UI & Backend Communications****, the* ***Mobile App Vision Tab & TTS*** *features, and the* ***Desktop App UI & Stream/AI Integration****. A dedicated period for* ***ESP32 & Server Debugging/Optimization*** *was scheduled around May 24th-26th, followed by* ***Client Apps Debugging/Refinement*** *from May 27th to June 1st, and* ***Full System Integration Testing*** *from May 29th to June 3rd.* ***Documentation & Report Writing*** *was an ongoing task from mid-May (around May 18th) through June 6th. The final stages involved* ***Presentation Preparation*** *(June 4th-8th) leading up to the* ***Final Project Submission*** *around June 10th, 2025.*

Bibliography

[1] Tiangolo. (n.d.). *FastAPI*. FastAPI Documentation. Retrieved May 20, 2025, from https://fastapi.tiangolo.com/

[2] Meta Platforms, Inc. (n.d.). *React Native – A framework for building native apps with React*. React Native Documentation. Retrieved May 20, 2025, from https://reactnative.dev/

[3] Riverbank Computing. (n.d.). *Introduction to PyQt6*. PyQt6 Documentation. Retrieved May 20, 2025, from https://www.riverbankcomputing.com/static/Docs/PyQt6/

[4] Espressif Systems. (n.d.). *Arduino ESP32 Core*. GitHub Repository. Retrieved May 20, 2025, from https://github.com/espressif/arduino-esp32

[5] OpenCV team. (n.d.). *OpenCV (Open Source Computer Vision Library)*. OpenCV Official Site. Retrieved May 20, 2025, from https://opencv.org/

[6] Axios Community. (n.d.). *Axios - Promise based HTTP client for the browser and node.js*. GitHub Repository. Retrieved May 20, 2025, from https://github.com/axios/axios

[7] Expo. (n.d.). *Expo - Make apps with React*. Expo Documentation. Retrieved May 20, 2025, from https://docs.expo.dev/

[8] Groq. (n.d.). *Groq API Documentation*. GroqCloud Console. Retrieved May 20, 2025, from https://console.groq.com/docs

[9] AI-Thinker. (n.d.). *ESP32-CAM Development Board Datasheet*. (Typically found via distributors or AI-Thinker's website, provide specific URL if available).

[10] Python Software Foundation. (n.d.). *Python Language Reference*. Python.org. Retrieved May 20, 2025, from https://www.python.org

[11] Google. (2024, May). *Gemma 3 outperforms other models in its class*. Google Blog. Retrieved May 20, 2025, from https://blog.google/technology/developers/gemma-3/

[12] Google Developers. (2024, May). *Introducing Gemma 3*. Google Developers Blog. Retrieved May 20, 2025, from https://developers.googleblog.com/en/introducing-gemma3/

Appendix A

Project Directory Structure

The IntelliGaze project is organized into several main directories, each containing specific components of the system. The high-level structure is as follows:

```
camera-feed/            # ESP32-CAM Firmware
  camera-feed.ino/
    camera-feed.ino
Desktop/                # PyQt6 Desktop Application
  app.py/
    app.py
Flask_server/           # FastAPI Backend Server
  server.py/
    server.py
inteligaze/             # React Native Mobile Application (Expo)
  .gitignore
  .vscode/
    settings.json/
  app/                  # Expo Router app directory
    _layout.tsx
    (tabs)/             # Tab-based navigation setup
      _layout.tsx
      explore.tsx
      index.tsx
      settings.tsx
      vision.tsx
    +not-found.tsx
  app.json              # Expo configuration file
  assets/               # Static assets (fonts, images) - (Inferred, not explicitly
  components/           # Reusable React Native components
    CaptureControls.tsx
    Collapsible.tsx
    ... (other UI components) ...
    ui/                 # More specific UI elements (Card, IconSymbol)
  constants/            # Global constants (e.g., Colors.ts)
  hooks/                # Custom React hooks
  scripts/              # Utility scripts (e.g., reset-project.js)
  utils/                # Utility functions (e.g., playGroqTTS.ts)
  package.json          # NPM package dependencies and scripts
  README.md
  tsconfig.json         # TypeScript configuration
Test/                   # Test scripts and files
  index.html            # HTML test page for vision endpoint
  test_fastapi.py       # Python script to test FastAPI server
  test.py               # Python script for image upload test
web/                    # Web Client Prototype
  index.html
  script.js
  style.css
```

Appendix B

Key Configuration Files

B.1 Expo Configuration (`inteligaze/app.json`)

This file configures the Expo build and runtime environment for the React Native mobile application.

```json
{
  "expo": {
    "name": "inteligaze",
    "slug": "inteligaze",
    "version": "1.0.0",
    "orientation": "portrait",
    "icon": "./assets/images/icon.png",
    "scheme": "inteligaze",
    "userInterfaceStyle": "automatic",
    "newArchEnabled": true,
    "ios": {
      "supportsTablet": true
    },
    "android": {
      "adaptiveIcon": {
        "foregroundImage": "./assets/images/adaptive-icon.png",
        "backgroundColor": "#ffffff"
      },
      "edgeToEdgeEnabled": true
    },
    "web": {
      "bundler": "metro",
      "output": "static",
      "favicon": "./assets/images/favicon.png"
    },
    "plugins": [
      "expo-router",
      [
        "expo-splash-screen",
        {
          "image": "./assets/images/splash-icon.png",
          "imageWidth": 200,
          "resizeMode": "contain",
          "backgroundColor": "#ffffff"
        }
      ],
      "expo-audio"
    ],
    "experiments": {
      "typedRoutes": true
    }
  }
}
```

Listing B.1: Snippet from inteligaze/app.json

B.2 React Native Dependencies (`inteligaze/package.json`)

The main dependencies for the mobile application are listed here.

```json
{
  "dependencies": {
    "@expo/vector-icons": "^14.1.0",
    "@react-native-async-storage/async-storage": "2.1.2",
    "@react-navigation/bottom-tabs": "^7.3.10",
    "@react-navigation/elements": "^2.3.8",
    "@react-navigation/native": "^7.1.6",
    "axios": "^1.9.0",
    "expo": "~53.0.9",
    "expo-audio": "~0.4.5",
    "expo-av": "~15.1.4",
    "expo-blur": "~14.1.4",
    "expo-constants": "~17.1.6",
    "expo-font": "~13.3.1",
    "expo-haptics": "~14.1.4",
    "expo-image": "~2.1.7",
    "expo-linking": "~7.1.5",
    "expo-router": "~5.0.6",
    "expo-splash-screen": "~0.30.8",
    "expo-status-bar": "~2.2.3",
    "expo-symbols": "~0.4.4",
    "expo-system-ui": "~5.0.7",
    "expo-web-browser": "~14.1.6",
    "react": "19.0.0",
    "react-dom": "19.0.0",
    "react-native": "0.79.2",
    "react-native-gesture-handler": "~2.24.0",
    "react-native-reanimated": "~3.17.4",
    "react-native-safe-area-context": "5.4.0",
    "react-native-screens": "~4.10.0",
    "react-native-web": "~0.20.0",
    "react-native-webview": "13.13.5"
  }
}
```

Listing B.2: Dependencies from inteligaze/package.json

Appendix C

Core Software Libraries and Frameworks

This section lists the primary software libraries and frameworks utilized in the development of each major component of the IntelliGaze system.

C.1 ESP32-CAM Firmware (`camera-feed.ino`)

- **Arduino Core for ESP32:** Base platform for ESP32 development.
- **`WiFi.h`:** For WiFi connectivity.
- **`esp_camera.h`:** For interfacing with the ESP32 camera module.
- **`esp_http_server.h`:** For creating the HTTP server to stream MJPEG.
- **`esp_timer.h`:** For timer functionalities (though not explicitly used for complex timing in the provided snippet, it's a core ESP-IDF component).

C.2 FastAPI Backend Server (`Flask_server/server.py`)

- **Python 3:** Core programming language.
- **FastAPI:** Modern, fast (high-performance) web framework for building APIs.
- **Uvicorn:** ASGI server to run FastAPI applications.
- **HTTPX:** Asynchronous HTTP client for making requests to the ESP32 stream and the external AI vision service.
- **Loguru:** Library for pleasant and powerful logging.
- **OpenCV-Python (`cv2`):** Used for decoding and potentially processing image frames from the ESP32 (though direct processing is minimal in the current server logic, it's used for validation).
- **Pydantic:** For data validation and settings management (used implicitly by FastAPI for request/response models).
- **Base64, Time, Threading:** Standard Python libraries.

C.3 React Native Mobile Application (`inteligaze/`)

- **React Native:** Framework for building native mobile applications using JavaScript/TypeScript and React.
- **Expo:** Platform and toolset for building universal React applications (iOS, Android, Web).
  - **Expo Router:** File-system based routing.
  - **Expo Audio/AV (`expo-av`):** For playing audio (TTS).
  - **Expo FileSystem (`expo-file-system`):** For saving temporary audio files.
  - **Expo Haptics, Expo Image, Expo Linking, Expo SplashScreen, Expo StatusBar, Expo Symbols, Expo WebBrowser, Expo Constants, Expo Font, Expo Blur, Expo SystemUI**
- **TypeScript:** Superset of JavaScript adding static typing.
- **Axios:** Promise-based HTTP client for making API requests to the backend and Groq TTS.
- **React Navigation (`@react-navigation/*`):** For navigation between screens and tabs.
- **AsyncStorage (`@react-native-async-storage/async-storage`):** For persistent local storage (settings).
- **React Native Reanimated, React Native Gesture Handler, React Native Safe Area Context, React Native Screens, React Native WebView**

C.4 PyQt6 Desktop Application (`Desktop/app.py`)

- **Python 3:** Core programming language.
- **PyQt6:** Set of Python bindings for Qt 6, used for creating the graphical user interface.
- **OpenCV-Python (`cv2`):** For fetching and processing (encoding/decoding) video frames from the ESP32 or webcam.
- **Requests:** Synchronous HTTP client for making requests to the external AI vision service.
- **NumPy:** For numerical operations, often used with OpenCV.
- **Base64, Sys, Threading (QThread):** Standard Python libraries and PyQt threading.