⭐ SAGE — Complete Project Overview

Situational Awareness & Guidance Engine (Smartglass Project)

A software-first, AI-driven wearable designed for accessibility, real-time guidance, and intelligent interaction with the environment.

1️⃣ Core Vision of SAGE

SAGE is a **smartglass prototype** built primarily around **AI software**, not hardware complexity.
The philosophy:

- Keep hardware **minimal, inexpensive, and beginner-friendly**
- Offload heavy AI/ML processing to a dedicated **mobile app + hosted backend**
- Use **free or open-source services** wherever possible
- Create an **AR-like experience** using a reflective HUD
- Make the system **modular**, **scalable**, and **easy to iterate**

2️⃣ Finalized System Architecture

🔹 Raspberry Pi Zero 2 W (Smartglass)

Handles:

- HUD display (TFT screen + acrylic reflection)
- Microphone → capturing wake words
- Speaker → audio responses
- Pi Camera → sending frames to app
- Lightweight FastAPI server for communication with the mobile app

Note: Pi does *not* run heavy ML. Tasks are offloaded.

🔹 Flutter Mobile App (UI + Communication Bridge)

Handles:

- Voice interface & user input
- Receiving images/audio from Pi
- Sending data to hosted backend
- Displaying or forwarding results back to Pi
- Managing user settings, pairing, connectivity

- Running all internet-based APIs

This becomes the *central brain* between Pi and backend.

🔹 Hosted FastAPI Backend (Python)

Handles **all AI/ML tasks**:

- Facial recognition
- Object detection
- Scene processing
- Translation logic
- Gemini + OCR integration
- Any custom models you develop in the future

Hosted on Render / Railway / Fly.io → *stable, scalable, easy to maintain*.

3️⃣ Major Features of SAGE

🟦 1. Voice Assistant ("Hey Glass")

- Pi mic listens for wake word
- Sends user speech to app
- App → Hosted backend → Gemini
- Backend returns: answers, instructions, summaries
- Pi displays on HUD or speaks aloud

🟦 2. Translation Module

Workflow:

1. User triggers translation
2. Pi captures image → sends to app
3. App uses **Google Vision** (free tier) → OCR
4. Extracted text → **LibreTranslate** (free API or self-hosted)
5. Translation returned → HUD or audio output

Free pipeline, no paid Google Translate.

🟦 3. Facial Recognition (Core Feature)

- Pi captures image
- App sends image to **FastAPI backend**
- Backend uses face_recognition (Python) to identify known faces
- App returns results to Pi
- Pi displays name / says name aloud

Reason: Heavy computation → must be offloaded from Pi.

🟦 4. Object Detection (Voice Triggered)

User says: **“Hey Glass, start scanning my environment.”**

Flow:

- Pi captures **one frame every 1–2 seconds**
- Sends frame to Flutter app
- App → Hosted backend ML model
- Backend returns objects (labels, locations)
- Pi updates HUD or reads aloud

Stop command:
“Hey Glass, stop scanning.”

This avoids overheating and battery drain.

🟦 5. HUD Display

Uses:

- 2.0–2.4” TFT 320×240 screen
- 45° transparent acrylic sheet as combiner
- Mirrored text output in software
- Provides AR-like floating UI

Readable, lightweight, and inexpensive.

4️⃣ Connectivity & Pairing

🔹 First-Time Pairing (Like Bluetooth Headphones)

1. Pi boots into **fallback AP mode**
2. User opens Flutter app → scans and detects "SAGE Glass"

3. App connects & sends **Wi-Fi hotspot credentials**
4. Pi restarts → connects automatically to user's hotspot
5. App now communicates with Pi on same local network

🔹 **Subsequent Use**

- User opens app + turns on smartglass
- Pi auto-connects to hotspot
- App instantly finds Pi on local IP
- Ready to operate

5️⃣ Tech Stack Summary

🟩 Smartglass (Raspberry Pi)

- Python
- FastAPI
- OpenCV
- SPI/GPIO libraries
- Pygame/PIL for HUD text rendering
- Microphone + speaker drivers

🟦 Mobile App (Flutter)

- Flutter (UI)
- HTTP/Dio for backend communication
- Audio (TTS/STT) packages
- Camera streaming handling
- Settings, UI flows, pairing

🟥 Hosted Backend (FastAPI, Python)

- FastAPI
- Uvicorn/Gunicorn
- face_recognition
- TFLite or custom PyTorch ONNX conversions
- Google Vision API

- LibreTranslate
- Gemini APIs

Cloud Providers:

- Render / Railway / Fly.io
- Fast, cheap/free, easy CI/CD integration

6️⃣ Thermal & Performance Design

🔸 Pi Offloads All Heavy Tasks

Only handles:

- Capturing images
- Displaying results
- Lightweight server
- Audio I/O

🔸 Avoid Live Video

Use **frame sampling**, not continuous streaming → minimal heating.

🔸 Use passive heatsinks

Ensures long sessions without thermal throttling.

7️⃣ Key Hardware Components

- Raspberry Pi Zero 2 W
- TFT 2.0–2.4” 320×240 display
- Transparent acrylic HUD combiner
- Pi Camera Module
- MEMS microphone
- Mini speaker
- Power bank (5V)
- 3D printed monocle-style frame

8️⃣ Your Final Architecture (Polished Version)

🔥 Smartglass → Flutter App → Hosted Backend → Flutter App → Smartglass

Everything flows through the app.
The Pi never touches the internet.
The backend handles all intelligence.

This is the **cleanest, safest, most scalable architecture**.

9️⃣ Why SAGE Is Unique

- True **AI-powered smartglass**, not just a fancy Bluetooth headset
- AR-like HUD using **low-cost components**
- Rich features: **translation, facial recognition, object detection, navigation, assistant**
- Software-first, modular, hackable design ideal for college innovation
- Far more capable than Ray-Ban Meta Glasses (no HUD, no translation, no object recognition)

🔟 You Are Now Ready to Build

This overview gives you:

- Complete architecture
- Input/output flows
- Tech stacks
- Hardware requirements
- ML offloading strategy
- HUD design
- Heat management strategy
- Pairing and networking plan

I relied on the S.A.G.E blueprint as the base architecture.

1) High-level architecture (software-only)

- Three logical service groups (all networked over TLS):
    1. **App Backend (Gayathri)** — FastAPI service that the Flutter app talks to. Orchestrates flows (translation, assistant queries), mediates user/session management, authentication, and calls into model services.
    2. **Model Servers (Nikhil / Ananya)** — Separate FastAPI (or model-server) endpoints for Face Recognition and Object Detection. Hosted on machines with appropriate compute (GPU for training and inference when needed). The App Backend calls these via REST, with async / job queue support for long-running inferences.

3. **Flutter Mobile App (You)** — UI + connectivity bridge for Pi. Sends images/audio to App Backend, receives results, shows HUD previews, handles pairing, STT/TTS UI. Pi communicates only with the Flutter app over local network.

2) Team responsibilities (explicit deliverables)

- **You (Flutter)**
    - Deliverables: app UI screens, image/camera streaming client, pairing flow, offline handling, API client code, token storage, basic STT/TTS integration (using app plugins), e2e demo flow with Pi simulator.
    - Acceptance: screens implemented, automated UI tests for flows, sample app communicates with App Backend mock endpoints.
- **Gayathri (App Backend FastAPI)**
    - Deliverables: FastAPI app with auth, user settings, session management, workflows for translation / Gemini assistant / OCR orchestration, job queue (Redis + RQ/Celery) for long tasks, connectors to model servers, well-documented OpenAPI (Swagger).
    - Acceptance: API passes contract tests, integrates with mock model servers, handles retries/circuit-breaker, logs, metrics.
- **Nikhil (Facial recognition model + server)**
    - Deliverables: training pipeline, dataset prep, model producing face embeddings, recognition server that returns person id / confidence / bounding boxes, Dockerized inference server (GPU optional), unit tests for preprocessing/inference, evaluation metrics.
    - Acceptance: model > target accuracy (set by you), inference endpoint handles batch queries and responds per API schema.
- **Ananya (Object detection model + server)**
    - Deliverables: training pipeline (YOLOv8 / Faster R-CNN / EfficientDet candidate), dataset augmentation pipeline, inference server that returns labels, boxes, confidences, optional segmentation mask, batching and NMS, Dockerized server, model quantization & ONNX/TorchScript artifacts.
    - Acceptance: model meets detection precision/recall target, low-latency inference for sampled frames, endpoint returns expected schema.