Software Project

Smart AI Glass

A. Platform information (detailed)

1. Target hardware

- **Primary prototype:** Raspberry Pi 4 (4GB or 8GB) or similar ARM-based SBC.
- **Optional higher-power devices:** NVIDIA Jetson Nano / Xavier or Google Coral USB/PCIe TPU for faster inference.
- **Camera:** Raspberry Pi Camera Module v2 / 8–12 MP USB camera or wide-angle MIPI camera (supports 720p–1080p).
- **Microphone:** Single or small array microphone (16 kHz or 48 kHz sampling) with noise suppression capability.
- **Audio output:** On-temple mini-speaker or bone-conduction earphones; headphone jack or Bluetooth audio optional.
- **Power:** Rechargeable Li-ion battery pack; power budget planning (typical Pi + camera + audio ≈ 5–10W).
- **Storage:** microSD (32–128GB) or small SSD for models/datasets/logs.
- **Physical constraints:** small, lightweight enclosure; thermal considerations for continuous inference.

2. Operating system & runtime

- **OS:** Raspberry Pi OS (64-bit) or Ubuntu 22.04 (ARM) for embedded devices; Linux recommended for driver support.
- **Runtime:** Python 3.9+ (virtualenv / venv recommended). Use lightweight process supervisor (systemd) for production.
- **Libraries:** OpenCV, NumPy, TensorFlow / TensorFlow Lite, pytorch/ONNX runtime (if using PyTorch), Tesseract OCR, pyttsx3 or local TTS engine, speech/voice libraries.

- **Model deployment:** Prefer TensorFlow Lite / ONNX + hardware acceleration where available (Edge TPU, Jetson Tensors).

3. Connectivity & external interfaces

- **Optional:** Wi-Fi / Bluetooth for companion app, model updates, and log upload.
- **Companion app:** Mobile app (Android/iOS) or web UI for settings, log export and manual snapshot.
- **Local interfaces:** CLI tool for debugging; optional serial/USB debug port.

4. Development & CI

- **Version control:** Git (GitHub/GitLab).
- **Packaging:** requirements.txt, Dockerfile for reproducible dev environment (optional).
- **Monitoring:** lightweight health check endpoints or local status CLI.

B. Requirements specification (detailed)

1. Functional requirements (core features)

1. **Object detection (core):**
    - Detect and classify a set of common, safety-critical object categories (e.g., person, vehicle, stairs, door, chair, curb, traffic light, crosswalk sign).
    - For each detection return: label, confidence score (0–1), bounding box coordinates, timestamp.
2. **Text recognition (OCR):**
    - Detect text regions in the camera frame and extract textual content. Support printed text (signs, menus, labels).
    - Return text string, confidence, bounding box. Support multi-line reading.
3. **Scene summarization / description:**
    - Generate a short natural-language description of the scene (1–2 sentences) when requested. Use templated summarizer over detected objects.
4. **Voice-driven interface:**

   - Accept simple voice commands (examples): “Describe surroundings”, “Read text”, “Repeat”, “Language: English/Arabic”, “Volume up/down”.
   - Provide short audio confirmations (“Describing now”, “Reading text”, “I could not read that, try again”).
5. **Audio output:**
   - Convert summary or OCR result to speech using local TTS engine; support multiple languages (English + Arabic).
6. **Interaction logging & model metadata:**
   - Log every user interaction (command, timestamp, used model_id, confidence). Maintain model metadata (version, accuracy, date).
7. **Model management:**
   - Load/unload models, store model files locally; enable safe local updates (update only on consent via companion app).
8. **Settings storage:**
   - Persist user preferences (preferred language, volume, confidence thresholds) locally.

2. Non-functional requirements

- **Latency:** End-to-end latency targets: object detection response ≤ 500 ms (on accelerated hardware or optimized model), scene summary < 1.5 s. For SBC, use TensorFlow Lite/quantized models.
- **Accuracy:** Object detection average precision suitable for practical use (target: ≥ 75–85% on selected objects in typical conditions). OCR word accuracy ≥ 80% on printed standard fonts at reasonable distances.
- **Robustness:** Must handle variable lighting — degrade gracefully and inform user when confidence low.
- **Availability:** System should resume operation automatically after transient failures. Aim for 95% uptime during usage session.
- **Power efficiency:** Keep CPU/GPU utilization and power draw within battery limits for acceptable use time (target operational time e.g., > 3 hours under normal use).

- **Privacy & security:** Default offline processing; if network features enabled require explicit consent. Data encryption at rest for stored images/audio if retention enabled.
- **Usability (Accessibility):** Voice-first interface, brief and clear messages, confirmatory tones for actions.

3. Acceptance criteria (examples/test cases)

- **Object detection test:** On a test set of 20 representative images, system must correctly detect > 80% of target objects with confidence > threshold.
- **OCR test:** Given printed text at 20–50 cm, ≥ 85% word-level accuracy in normal lighting.
- **Voice command test:** System recognizes and executes > 90% of predefined commands in quiet environment.
- **Latency test:** 90th percentile end-to-end latency < 1.5 s on target hardware.

4. Error handling & edge cases

- **Low confidence:** If model confidence < threshold → TTS: “I’m not sure, please move closer or take a photo.”
- **Noisy audio:** Implement VAD + retry prompts; fall back to button-press activation if voice not recognized.
- **Low light:** Signal low-light condition and suggest moving to better lighting or switch to flash / snapshot mode.
- **Model load failure:** Notify user and fall back to a safe minimal mode (e.g., basic obstacle warning).

C. Data description (detailed)

1. Input data (live)

- **Camera frames**
  - Format: RGB image arrays (numpy) or encoded JPEG/PNG for snapshot storage.
  - Resolution: Prefer 1280×720 for OCR, 640×480 for real-time detection (configurable).
  - Frame rate: 10–15 FPS real-time processing target (may process every Nth frame to reduce load).

- **Audio**
    - Format: PCM 16-bit, mono. Sampling rate: 16 kHz (speech) or 48 kHz if supported.
    - Use VAD to detect start/stop of command.

2. Derived data (outputs from models)

- **Detection output structure (JSON example)**

```
{
  "timestamp": "2025-11-23T15:02:10Z",
  "text": "Welcome to Store",
  "confidence": 0.88,
  "bbox": [x, y, w, h]
}
```

- **OCR output structure**

```
{
  "timestamp": "2025-11-23T15:02:10Z",
  "text": "Welcome to Store",
  "confidence": 0.88,
  "bbox": [x, y, w, h]
}
```

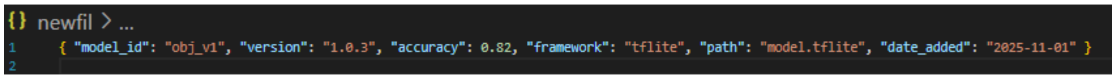
- **Scene summary**

```
{ "summary": "A person is standing two meters ahead. There is a chair on your right." }
```

3. Storage & file layout (on-device)

- /app/models/

    - object_detector/ (files: model.tflite, model.json)
    - ocr/ (tesseract data or model)
- /app/data/
    - /data/snapshots/ → saved images (PNG/JPEG) when user requests or for debug.
- /app/logs/
    - YYYY-MM-DD.jsonl → one JSON object per interaction (interaction logs).
- /app/config/
    - settings.json → user preferences, thresholds.
- Model metadata file example: /app/models/object_detector/model.json


newfil > ...

```json
{ "model_id": "obj_v1", "version": "1.0.3", "accuracy": 0.82, "framework": "tflite", "path": "model.tflite", "date_added": "2025-11-01" }
```

4. Data retention, privacy & encryption

- **Retention policy:** Default keep logs 30 days; allow user configuration via companion app.
- **Sensitive data:** Raw images/audio considered sensitive — only store when necessary and encrypted at rest (AES-256).
- **Anonymization:** Strip any PII before upload (default no upload). If any cloud processing enabled, require explicit opt-in and consent flow.

5. Annotation & training data

- **Dataset structure for training:**
    - /train/images/ → images with annotation files (COCO/YOLO format)
    - /train/annotations/ → COCO JSON or YOLO label files
- **Labels taxonomy:** provide a controlled set of object labels (define file labels.txt) matched across training and runtime.
- **Quality metrics:** dataset should include varied lighting, angles, occlusions; measure per-class map, per-OCR WER (word error rate).

6. Logging & analytics

- **Interaction log fields (per event)**

    - log_id, user_id (optional), timestamp, action (describe/read_text), result (success/fail), model_id, confidence, notes (optional), snapshot_path (optional)

- **Log format:** JSON Lines (.jsonl) to allow streaming and easy export.
- **Local analytics:** periodic aggregation (daily) to compute usage stats (commands per day, avg confidence).

7. Data validation & quality checks

- **Input validation:** verify frame size and audio sample rate; discard corrupted frames.
- **Postprocessing checks:** if detection bounding boxes are tiny or confidence extremely low, raise "low confidence" flag.
- **Monitoring metrics:** fps, inference time (ms), memory usage, CPU temp — log these for performance tuning.

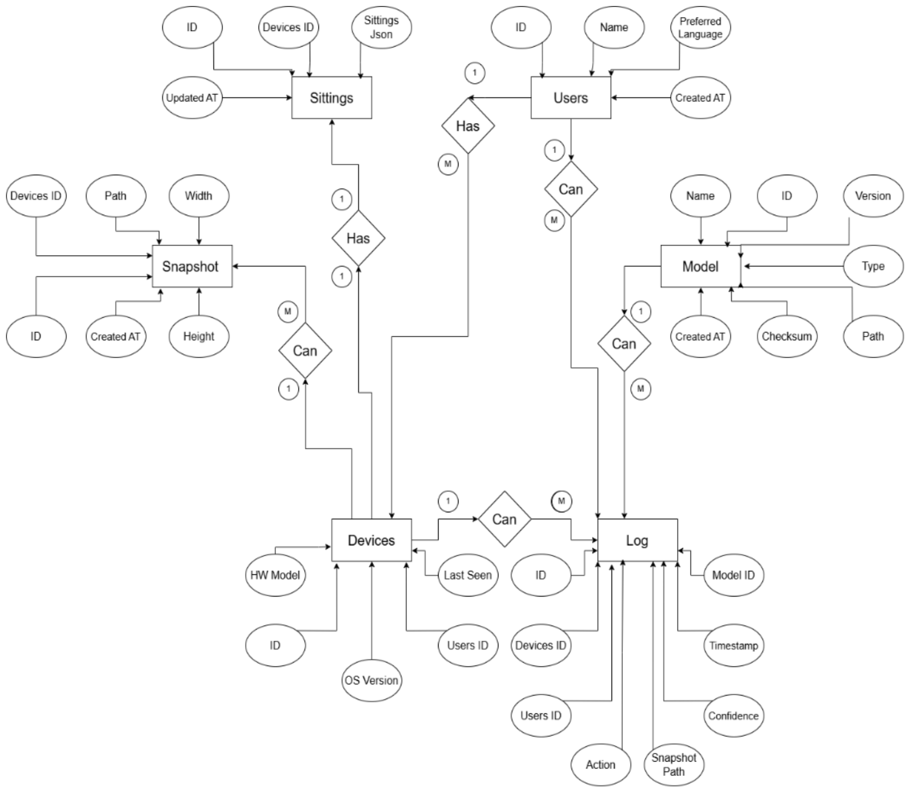
D. Configuration parameters (examples to include in settings.json)

```
{
  "model": {
    "object_detector": { "min_confidence": 0.5, "max_fps": 10 },
    "ocr": { "min_confidence": 0.6 }
  },
  "ui": { "language": "en", "volume": 0.8 },
  "logging": { "retain_days": 30, "log_level": "INFO" },
  "privacy": { "store_snapshots": false, "encrypt_at_rest": true }
}
```

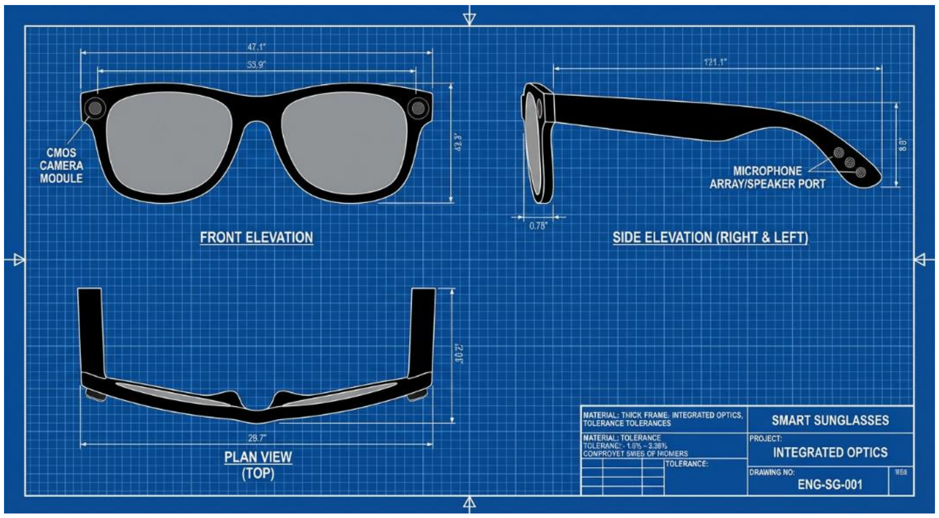
E. Short checklist to include with Design Inputs

- Specify exact SBC model and camera module for prototype.
- Choose model formats (TFLite / ONNX) and acceleration hardware (Edge TPU / Jetson).
- Define label taxonomy and OCR languages.
- Set retention and privacy defaults (no cloud by default).
- Prepare acceptance tests for detection/OCR/voice latency.

F. ER-Diagram

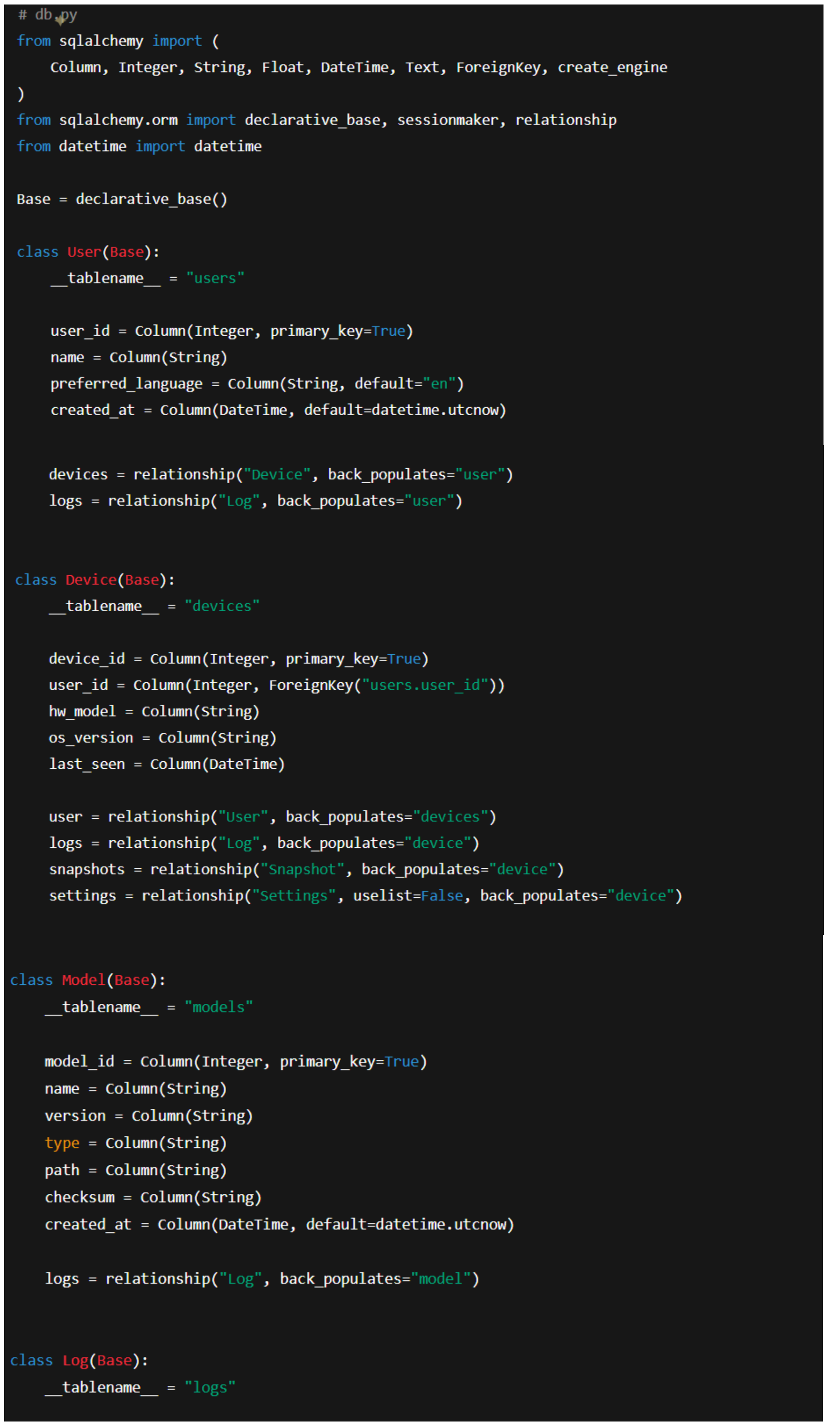


G. Engineering-Design



H. Python code

1. Database Models (Python + SQLAlchemy)



```python
# db.py
from sqlalchemy import (
    Column, Integer, String, Float, DateTime, Text, ForeignKey, create_engine
)
from sqlalchemy.orm import declarative_base, sessionmaker, relationship
from datetime import datetime

Base = declarative_base()

class User(Base):
    __tablename__ = "users"

    user_id = Column(Integer, primary_key=True)
    name = Column(String)
    preferred_language = Column(String, default="en")
    created_at = Column(DateTime, default=datetime.utcnow)

    devices = relationship("Device", back_populates="user")
    logs = relationship("Log", back_populates="user")


class Device(Base):
    __tablename__ = "devices"

    device_id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey("users.user_id"))
    hw_model = Column(String)
    os_version = Column(String)
    last_seen = Column(DateTime)

    user = relationship("User", back_populates="devices")
    logs = relationship("Log", back_populates="device")
    snapshots = relationship("Snapshot", back_populates="device")
    settings = relationship("Settings", uselist=False, back_populates="device")


class Model(Base):
    __tablename__ = "models"

    model_id = Column(Integer, primary_key=True)
    name = Column(String)
    version = Column(String)
    type = Column(String)
    path = Column(String)
    checksum = Column(String)
    created_at = Column(DateTime, default=datetime.utcnow)

    logs = relationship("Log", back_populates="model")


class Log(Base):
    __tablename__ = "logs"
    log_id = Column(Integer, primary_key=True)
    device_id = Column(Integer, ForeignKey("devices.device_id"))
    user_id = Column(Integer, ForeignKey("users.user_id"))
    action = Column(String)
    confidence = Column(Float)
    model_id = Column(Integer, ForeignKey("models.model_id"))
    snapshot_path = Column(String)
    timestamp = Column(DateTime, default=datetime.utcnow)

    user = relationship("User", back_populates="logs")
    device = relationship("Device", back_populates="logs")
    model = relationship("Model", back_populates="logs")


class Snapshot(Base):
    __tablename__ = "snapshots"

    snapshot_id = Column(Integer, primary_key=True)
    device_id = Column(Integer, ForeignKey("devices.device_id"))
    path = Column(String)
    width = Column(Integer)
    height = Column(Integer)
    created_at = Column(DateTime, default=datetime.utcnow)

    device = relationship("Device", back_populates="snapshots")


class Settings(Base):
    __tablename__ = "settings"

    settings_id = Column(Integer, primary_key=True)
    device_id = Column(Integer, ForeignKey("devices.device_id"))
    settings_json = Column(Text)
    updated_at = Column(DateTime, default=datetime.utcnow)

    device = relationship("Device", back_populates="settings")

engine = create_engine("sqlite:///smartglasses.db")
Base.metadata.create_all(engine)

Session = sessionmaker(bind=engine)
```

2. Core System Components

- Camera

```
# camera.py
import cv2

class FrameGrabber:
    def __init__(self, source=0, width=640, height=480):
        self.cap = cv2.VideoCapture(source)
        self.cap.set(3, width)
        self.cap.set(4, height)

    def read(self):
        ok, frame = self.cap.read()
        return ok, frame
```

- Preprocessing

```python
# preprocessing.py
import cv2

class Preprocessor:
    def process(self, frame):
        frame = cv2.resize(frame, (640, 480))
        return frame
```

- Object Detection (YOLO / TFLite stub)

```python
# object_detector.py
from dataclasses import dataclass
from typing import Tuple, List
import random

@dataclass
class Detection:
    label: str
    confidence: float
    bbox: Tuple[int, int, int, int]

class ObjectDetector:
    def __init__(self, model_path="models/yolo.tflite"):
        self.model_path = model_path

    def infer(self, frame) -> List[Detection]:
        return [
            Detection(
                label="person",
                confidence=round(random.uniform(0.85, 0.99), 2),
                bbox=(100, 80, 200, 300)
            )
        ]
```

- OCR Engine

```python
# ocr_engine.py
from dataclasses import dataclass
from typing import Tuple, List

@dataclass
class OCRResult:
    text: str
    confidence: float
    bbox: Tuple[int, int, int, int]

class OCREngine:
    def __init__(self, lang="eng"):
        self.lang = lang
    def infer(self, frame) -> List[OCRResult]:
        return [
            OCRResult(
                text="OPEN",
                confidence=0.93,
                bbox=(20, 20, 150, 60)
            )
        ]
```

- Scene Summarizer

```
# summarizer.py
class SceneSummarizer:
    def summarize(self, detections, ocr):
        parts = []

        if detections:
            d = detections[0]
            parts.append(f"There is a {d.label} ahead.")

        if ocr:
            parts.append(f"Detected text: {ocr[0].text}")

        return " ".join(parts) if parts else "No useful information detected."
```

- Text-to-Speech

```
# tts.py
import pyttsx3

class TTS:
    def __init__(self, language="en"):
        self.engine = pyttsx3.init()

    def speak(self, text):
        self.engine.say(text)
        self.engine.runAndWait()
```

- Logging System

```
# logger.py
import json, time, os

class Logger:
    def __init__(self, path="logs/"):
        os.makedirs(path, exist_ok=True)
        self.path = path

    def log(self, event):
        filename = os.path.join(self.path, time.strftime("%Y-%m-%d") + ".jsonl")
        with open(filename, "a") as f:
            f.write(json.dumps(event) + "\n")
```

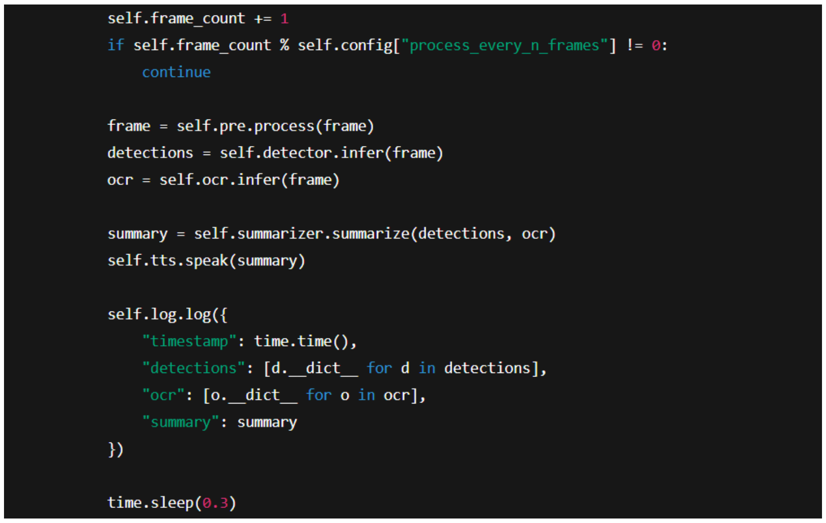
- Configuration Manager

```python
# config.py
import json, os

DEFAULT_CONFIG = {
    "language": "en",
    "volume": 0.7,
    "confidence_threshold": 0.5,
    "process_every_n_frames": 2
}

class ConfigManager:
    def __init__(self, path="config/settings.json"):
        self.path = path
        os.makedirs(os.path.dirname(path), exist_ok=True)
        if not os.path.exists(path):
            json.dump(DEFAULT_CONFIG, open(path, "w"), indent=2)

    def load(self):
        return json.load(open(self.path))
```

3. Main Runtime Controller

```python
# controller.py
import time
from camera import FrameGrabber
from preprocessing import Preprocessor
from object_detector import ObjectDetector
from ocr_engine import OCREngine
from summarizer import SceneSummarizer
from tts import TTS
from logger import Logger
from config import ConfigManager
```

```python
class MainController:
    def __init__(self):
        self.config = ConfigManager().load()
        self.camera = FrameGrabber()
        self.pre = Preprocessor()
        self.detector = ObjectDetector()
        self.ocr = OCREngine()
        self.summarizer = SceneSummarizer()
        self.tts = TTS(language=self.config["language"])
        self.log = Logger()
        self.frame_count = 0

    def run(self):
        while True:
            ok, frame = self.camera.read()
            if not ok:
                self.tts.speak("Camera error")
                continue
```



```
        self.frame_count += 1
        if self.frame_count % self.config["process_every_n_frames"] != 0:
            continue

        frame = self.pre.process(frame)
        detections = self.detector.infer(frame)
        ocr = self.ocr.infer(frame)

        summary = self.summarizer.summarize(detections, ocr)
        self.tts.speak(summary)

        self.log.log({
            "timestamp": time.time(),
            "detections": [d.__dict__ for d in detections],
            "ocr": [o.__dict__ for o in ocr],
            "summary": summary
        })

        time.sleep(0.3)
```

Team Members:

Ahmed Khaled	(ID: 2401483)
Yousef Ehab	(ID: 2401476)
Abdullah Ahmed	(ID: 2401519)
Mahmoud Ashraf	(ID: 2401495)
Zahraa Mahmoud	(ID: 2401493)
Samia Mahmoud	(ID:2401171)
Reem Mohamed	(ID: 2401280)
Fyrouz Mohamed	(ID: 2401100)
Mohamed Salah	(ID: 2401501)
Ahmed Samir	(ID: 2401386)