EchoSight Sentinel — I chose this name because "Echo" captures the system's audio intelligence, "Sight" reflects its computer vision, and "Sentinel" conveys active protection and vigilance. It's a real-time, multi-modal security assistant that watches and listens for intrusions and emergencies, fusing Detectron2-based visual detection with audio classification (e.g., glass breaks, gunshots, screams) to score severity and dispatch WhatsApp alerts with snapshots or audio snippets in seconds. If you need a one-line pitch: "It's an edge-powered security assistant that combines cameras and microphones to detect threats accurately and notify responders instantly, cutting false alarms and speeding up response."

What You'll Build

- A small, modular pipeline that continuously ingests RTSP video and microphone audio, runs Detectron2 and an audio classifier on rolling windows, fuses signals into severity-scored events, and dispatches WhatsApp alerts with evidence via a FastAPI backend and a minimal operator dashboard.

Architecture

- Ingestion: video_ingest reads RTSP frames; audio_ingest buffers PCM chunks.
- Models: vision_infer (Detectron2 COCO person/hazard subset), audio_infer (pretrained CNN on mel-spectrogram).

- Fusion: event_fuser applies rule-based severity, dedup, cooldown, and aggregation.
- Alerting: alert_service formats payloads and sends WhatsApp messages; optional SMS/email fallback.
- Backend & UX: FastAPI routes for alerts, health, acknowledgements; minimal dashboard.
- Storage & Metrics: local SQLite or JSONL logs, snapshots/audio clips; Prometheus metrics.

Week 1 — Environment, Ingestion, Vision Baseline

- Set up environment and repo scaffolding.

- Implement video ingest with ROI; implement audio capture ring buffer.

- Run Detectron2 person/hazard baseline; persist snapshots for detected events.

- Create Python env:

```
python -m venv venv
```

- Activate env:

```
venv\Scripts\activate
```

- Install core dependencies (run individually, verify GPU support first):

```
pip install fastapi uvicorn opencv-python numpy
torchaudio librosa prometheus-client python-dotenv
```

requests
```

- Install PyTorch with CUDA (match your GPU/CUDA; example uses CUDA 12.1):
```
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121
```
- Install Detectron2 wheel compatible with your Python/CUDA (adjust URL/version to your setup):
```
pip install detectron2
```
- Suggested repo layout:
```
src/
  ingest/
    video.py
    audio.py
  models/
    vision.py
    audio.py
  fusion/
    rules.py
  alerting/
    whatsapp.py
    payloads.py

```
  backend/
    app.py
    storage.py
  utils/
    config.py
    logging.py
data/
  snapshots/
  audio_clips/
  logs/
```

- Implement video ingestion with RTSP and ROI masking:

```
import cv2
import time
import numpy as np

class VideoIngest:
    # ... existing code ...
    def __init__(self, rtsp_url: str,
    sample_rate_hz: float = 5.0, roi_mask: np.
    ndarray | None = None):
        self.rtsp_url = rtsp_url
        self.sample_period = 1.0 / sample_rate_hz
        self.roi_mask = roi_mask  # binary mask
        matching frame size
        self.cap = cv2.VideoCapture(rtsp_url)
```

```
        if not self.cap.isOpened():
            raise RuntimeError(f"Failed to open
            RTSP: {rtsp_url}")
    # ... existing code ...
    def frames(self):
        last = 0.0
        while True:
            ok, frame = self.cap.read()
            if not ok:
                time.sleep(0.2); continue
            now = time.time()
            if now - last < self.sample_period:
                continue
            last = now
            if self.roi_mask is not None:
                frame = cv2.bitwise_and(frame,
                frame, mask=self.roi_mask)
            yield now, frame
    # ... existing code ...
```

- Implement audio ingestion with a ring buffer:

```
import queue
import threading
import time
import sounddevice as sd
import numpy as np
```

```
class AudioIngest:
    # ... existing code ...
    def __init__(self, sample_rate=16000,
    channels=1, window_seconds=1.0, max_queue=50):
        self.sample_rate = sample_rate
        self.channels = channels
        self.window_samples = int(sample_rate *
        window_seconds)
        self.q = queue.Queue(maxsize=max_queue)
        self.stream = None
    # ... existing code ...
    def _callback(self, indata, frames, time_info,
    status):
        if status:  # handle underruns/overruns
            pass
        audio = indata.copy().reshape(-1)
        self.q.put_nowait(audio)
    # ... existing code ...
    def start(self):
        self.stream = sd.InputStream(callback=self.
        _callback, samplerate=self.sample_rate,
        channels=self.channels)
        self.stream.start()
    # ... existing code ...
    def windows(self):
        buf = np.zeros(0, dtype=np.float32)
```

```
        while True:
            chunk = self.q.get()
            buf = np.concatenate([buf, chunk]).astype
            (np.float32)
            if buf.size >= self.window_samples:
                win = buf[:self.window_samples]
                buf = buf[self.window_samples:]
                yield time.time(), win
    # ... existing code ...
```

- Vision baseline inference with Detectron2:

```
from detectron2.config import get_cfg
from detectron2.engine import DefaultPredictor
from detectron2 import model_zoo

class VisionModel:
    # ... existing code ...
    def __init__(self, threshold=0.5,
    classes_of_interest=("person", "knife")):
        cfg = get_cfg()
        cfg.merge_from_file(model_zoo.get_config_file
        ("COCO-Detection/faster_rcnn_R_50_FPN_3x.
        yaml"))
        cfg.MODEL.WEIGHTS = model_zoo.
        get_checkpoint_url("COCO-Detection/
        faster_rcnn_R_50_FPN_3x.yaml")
```

```
        cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST =
        threshold
        self.predictor = DefaultPredictor(cfg)
        self.classes_of_interest = set
        (classes_of_interest)
        from detectron2.data import MetadataCatalog
        self.meta = MetadataCatalog.get(cfg.DATASETS.
        TRAIN[0])
    # ... existing code ...
    def infer(self, frame):
        out = self.predictor(frame)
        boxes = out["instances"].pred_boxes.tensor.
        cpu().numpy()
        scores = out["instances"].scores.cpu().numpy
        ()
        classes = out["instances"].pred_classes.cpu
        ().numpy()
        names = [self.meta.get("thing_classes")[c]
        for c in classes]
        keep = [(b, s, n) for b, s, n in zip(boxes,
        scores, names) if n in self.
        classes_of_interest]
        return keep
    # ... existing code ...
```

Week 2 — Audio Baseline, Thresholding, Fusion Rules

- Build audio classifier using mel-spectrograms; start with a pretrained CNN to speed up MVP.
- Calibrate thresholds per class using pilot recordings; compute simple PR metrics.
- Implement fusion: rule evaluations, severity scoring, cooldown, and dedup.
- Audio model with mel-spectrogram features:

```
import torch
import torchaudio
import librosa
import numpy as np

class AudioModel:
    # ... existing code ...
    def __init__(self, device="cuda"):
        self.device = device if torch.cuda.
        is_available() else "cpu"
        # Example: use a small CNN from torch.hub or
        your own trained weights.
        # Placeholder linear head on top of log-mel;
        replace with real model.
        self.sample_rate = 16000
        self.classes = ["glass_break", "gunshot",
        "scream", "forced_entry"]
        self.thresholds = {"glass_break": 0.6,
        "gunshot": 0.7, "scream": 0.6,
```

```
        "forced_entry": 0.65}
    # ... existing code ...
    def _mel(self, audio):
        mel = librosa.feature.melspectrogram
        (y=audio, sr=self.sample_rate, n_fft=1024,
        hop_length=256, n_mels=64)
        logmel = librosa.power_to_db(mel).astype(np.
        float32)
        return torch.tensor(logmel)[None,
        None, ...]  # [B, C, M, T]
    # ... existing code ...
    def infer(self, audio_window):
        x = self._mel(audio_window).to(self.device)
        # Dummy scores; plug in real model forward
        scores = {k: float(np.clip(np.random.rand(),
        0, 1)) for k in self.classes}
        events = {k: s for k, s in scores.items() if
        s >= self.thresholds[k]}
        return scores, events
    # ... existing code ...
```

- Fusion rules and severity scoring:

```
import time
from dataclasses import dataclass

@dataclass
```

```
class Event:
    ts: float
    site: str
    vision: list  # [(box, score, name)]
    audio: dict   # {class: score}
    severity: int
    reason: str

class Fuser:
    # ... existing code ...
    def __init__(self, cooldown_sec=10,
    dedup_window_sec=5):
        self.cooldown_sec = cooldown_sec
        self.dedup_window_sec = dedup_window_sec
        self._last_sent_ts = 0.0
        self._last_reason = ""
    # ... existing code ...
    def score(self, vision, audio):
        has_person = any(n == "person" for _, _, n
        in vision)
        has_glass = "glass_break" in audio
        has_gun = "gunshot" in audio
        has_scream = "scream" in audio
        severity = 0; reason = "no-event"
        if has_person and has_gun: severity, reason
        = 3, "person+gunshot"
        elif has_person and has_glass: severity,
```

```
        reason = 3, "person+glass_break"
        elif has_person and has_scream: severity,
        reason = 2, "person+scream"
        elif has_gun or has_glass: severity, reason
        = 2, "audio_threat"
        elif has_person: severity, reason = 1,
        "person_only"
        return severity, reason
    # ... existing code ...
    def should_send(self, ts, severity, reason):
        if severity == 0: return False
        if (ts - self._last_sent_ts) < self.
        cooldown_sec and reason == self._last_reason:
            return False
        self._last_sent_ts = ts; self._last_reason =
        reason
        return True
    # ... existing code ...
```

Week 3 — Alerting, Backend, Minimal Dashboard

- Integrate WhatsApp Cloud API; send text + link to snapshot/audio first; add media later.
- Build FastAPI to list alerts, acknowledge, serve snapshots/audio clips, and expose /metrics .
- Add Prometheus counters for latency, delivery success; persist alerts in SQLite or JSONL.

- WhatsApp alert service (text payload first):

```
import os
import requests

class WhatsAppClient:
    # ... existing code ...
    def __init__(self, phone_number_id: str,
    access_token: str):
        self.base = f"https://graph.facebook.com/v20.
        0/{phone_number_id}/messages"
        self.token = access_token
        self.headers = {"Authorization": f"Bearer
        {self.token}", "Content-Type": "application/
        json"}
    # ... existing code ...
    def send_text(self, to_number: str, body: str):
        payload = {
            "messaging_product": "whatsapp",
            "to": to_number,
            "type": "text",
            "text": {"preview_url": False, "body":
            body}
        }
        r = requests.post(self.base, json=payload,
        headers=self.headers, timeout=10)
        r.raise_for_status()
```

```
        return r.json()
    # ... existing code ...
```
- Backend wiring and endpoints:
```
from fastapi import FastAPI
from fastapi.responses import JSONResponse
from prometheus_client import Counter, Summary,
generate_latest, CONTENT_TYPE_LATEST
import time

app = FastAPI()
ALERTS = []
SEND_LATENCY = Summary("alert_send_latency_seconds",
"Latency for sending alerts")
DELIVERY_OK = Counter("alerts_delivered_ok_total",
"Successful alert deliveries")

@app.get("/alerts")
def list_alerts():
    # ... existing code ...
    return JSONResponse(ALERTS)
# ... existing code ...
@app.post("/ack/{idx}")
def ack(idx: int):
    # ... existing code ...
    if 0 <= idx < len(ALERTS): ALERTS[idx]["ack"] =
```

```
    True
    return {"ok": True}
# ... existing code ...
@app.get("/metrics")
def metrics():
    # ... existing code ...
    return JSONResponse({"metrics": generate_latest
    ().decode("utf-8")},
    media_type=CONTENT_TYPE_LATEST)
# ... existing code ...
def record_alert(event, send_func):
    start = time.time()
    try:
        res = send_func()
        DELIVERY_OK.inc()
    finally:
        SEND_LATENCY.observe(time.time() - start)
    ALERTS.append({"ts": event.ts, "severity": event.
    severity, "reason": event.reason, "ack": False})
    return True
# ... existing code ...
```

- Run backend during development:

```
uvicorn src.backend.app:app --reload
```

Week 4 — Pilot, Tuning, Hardening

- Deploy pipeline at pilot site; run continuously; collect logs and metrics.
- Tune thresholds and fusion logic; add ROI masks; enable cooldown/dedup tweaks to minimize alert storms.
- Add privacy options: masking configured ROIs, media retention limits, and simple auth for dashboard.
- Conduct acceptance checks (latency P95, precision/recall on labeled snippets, delivery success, uptime).
- Package as Docker images (Windows + WSL2 recommended) or run as a service; write a short runbook.

Detailed Developer Steps

- Config management: put site config in .env or config.yaml with RTSP_URL , WHATSAPP_PHONE_ID , WHATSAPP_ACCESS_TOKEN , TO_NUMBER , SITE_NAME , ROI_MASK_PATH , AUDIO_SR , FRAME_RATE .
- Ingestion loop: start VideoIngest.frames() and AudioIngest.windows() in separate threads or asyncio tasks; push outputs to a processing queue.
- Processing pipeline: for each video frame/audio window timestamp bucket, run VisionModel.infer(frame) and AudioModel.infer(audio) ; combine into Fuser.score() ; if should_send , persist snapshot/audio clip to data/ and enqueue alert.
- Alert dispatch: call WhatsAppClient.send_text with formatted message: "{site} {ts} severity={n} reason={r}

snapshot={url} audio={url}" ; add retries and exponential backoff on failure; record metrics and logs.
- Dashboard: render ALERTS list as a simple HTML page or JSON for a lightweight frontend; include filters by severity/site; add /health route to expose ingestion/model status.
- Metrics: expose counters/summaries; add your own: vision_infer_ms , audio_infer_ms , events_sent_total , events_dropped_total , false_positive_reports_total .
Requirements to Prepare

- Hardware placed and tested: verify RTSP connectivity (rtsp://user:pass@host:port/stream) and mic SNR (quiet vs loud).
- Data capture: collect short clips of target audio classes and typical ambient noise; save 5–10 snapshots with motion/lighting variations.
- BSP setup: WhatsApp Cloud account, phone_number_id , and access_token ; test with a sandbox number before production.
Technical Stack

- Python 3.10+, FastAPI , uvicorn for backend; opencv-python for RTSP frames; torchaudio / librosa for audio features; detectron2 for vision; requests for API calls; prometheus-client for metrics; SQLite/JSONL for MVP storage; Docker optional for packaging.

- Acceleration: use CUDA with PyTorch; consider ONNX + TensorRT export for Detectron2 later; keep v1 simple and reliable.

Why This Approach

- Rule-based fusion is transparent and quick to iterate for MVP; it gives control over false positives while you gather pilot data.
- WhatsApp Cloud API provides reliable, audited messaging where operators already work; text first, media second simplifies early integration.
- FastAPI is lightweight and fast, perfect for a single-site MVP with minimal dashboards and health endpoints.

What You'll Have After Version 1

- A running pilot that detects people and key audio threats, fuses signals, and sends WhatsApp alerts with context in ~1–2 seconds.
- A small operator dashboard and health metrics, plus snapshots/audio evidence stored locally with retention controls.
- Baseline KPIs and a tunable pipeline ready for more classes, better models, and multi-site scaling.

If you want, I can tailor the code skeletons to your exact hardware (GPU/CPU) and RTSP stream format, and add a ready-to-run main.py that wires ingestion → models → fusion → alerting.