App name - Drishti

#1 — Executive / User-facing explanation (plain language)

Project Drishti is a real-time safety platform for very large gatherings (Mahakumbh 2028). Imagine a command center that watches thousands of people across ghats, temples, and transit hubs — detects dangerous crowding/panic early, helps reunite lost people, and sends calm, multilingual instructions to nearby speakers and pilgrims' phones.

Main things a user (admin / volunteer / pilgrim) experiences:

Admin / Command Center

- Live map with colorful heatmap overlays showing crowd density per zone (green → red).
- Real-time incident feed: each incident card shows a summary, urgency (low/medium/high/critical), photos or snapshots, estimated people count, suggested action, and assign/acknowledge buttons.
- Open any camera feed (live or near-live snapshot) and see the analysis results side-by-side.
- Trigger multilingual voice/text alerts to zones (auto-generated wording with voice audio).
- Search "lost-person" database and see candidate matches with confidence score.

Volunteer / Responder App

- Lightweight app mode to receive assignments, see location of assigned incident, mark resolved, and upload photos/voice updates.
- Two-way messaging with command center for instructions.

Pilgrim App (public)

- Panic button (sends geo + optional image/audio).
- Report forms (congestion, medical, lost child).
- Receive targeted alerts: "Move slowly toward Exit C", in Hindi / English / Marathi or other configured languages.
- Lost & Found: you can upload a photo; the system attempts to match.

Outcome users get

- Faster response, earlier interventions to avoid stampedes, fewer missing persons, and post-event analytics to improve future planning.

#2 — High-level architecture (one-sentence summary)

Drishti = **Edge + Ingest → Cloud AI (OpenAI + vector DB) → Orchestration / Events → Realtime UI & Mobile**. Each piece is modular and can be scaled independently.

#3 — Feature-by-feature: what it does, how it behaves, and acceptance criteria

I'll list each feature, the UX, the backend work, and the acceptance test.

1) Live Crowd Heatmap (Map)

What user sees

- A Leaflet map with color heatmap overlay. Zoom & pan. Click a zone to see density history and current incidents.

How it's built

- Backend receives frame-level density estimates (every camera snapshot → analysis).
- For each zone we compute people_estimate or density_score (0–1) and store time series.
- Frontend subscribes to WebSocket events to update heatmap tiles or draws canvas-based heatmap from points.

Acceptance

- Heatmap updates within 2–4s of a new analysis.
- Color legend consistent: <0.2 green, 0.2–0.5 amber, 0.5–0.8 orange, >0.8 red.

2) Real-time Incident Feed & Incident Cards

User

- Feed shows newest incidents at top, with severity, small image, location, summary and quick actions (assign/ack).

Backend

- When analysis returns risk_level >= medium, create event record and publish via Socket.IO.
- Event payload contains structured JSON (see schema below).

Acceptance

- Incident appears on dashboard within 2s of event creation.
- Clicking “Acknowledge” sets event status to acknowledged and notifies the server.

3) Live Camera / Snapshot Viewer

User

- Click a camera → open live panel. For MVP we use frequent snapshots (1–2s) instead of full WebRTC; later we can add WebRTC/HLS.

Backend

- Edge nodes or server produce periodic snapshots into S3 (MinIO). Presigned URLs served to frontend.

Acceptance

- Snapshots refresh every 1–2s, not causing UI freeze. Latency p95 < 3s.

4) Crowd Behavior & Panic Detection (AI)

What it does

- Detects sudden surges, directional panic movements, blocking at exits, sudden drop in local velocity (people stopping), and small fights that could escalate.

How

- System sends snapshot(s) to OpenAI Vision (prompt requests structured JSON about density and behaviors). Also can run lightweight optical-flow/anomaly detection at edge to prefilter.

Acceptance

- In 80%+ of simulated panic scenarios, system flags risk_level high or critical.
- False positive rate < 20% under normal crowd movement.

5) Lost & Found (Image Matching)

User

- Upload a person photo → system returns candidate matches (blurry faces redacted by default) for admins to verify.

How

- Option A (privacy-first): compute face embeddings on-edge (FaceNet, local lib), store embedding in vector DB (Milvus/Pinecone). Use nearest-neighbor to find matches.
- Option B (if using OpenAI): derive visual embeddings via OpenAI Vision + store vectors in vector DB.

Acceptance

- System returns top-3 candidates within 3s; correctness (admin-verified) in test set ≥ 70%.

6) Crowd-sourced reports (Pilgrim submissions)

User

- Text, photo, or voice. Backend normalizes into structured report and attaches severity suggestions.

How

- If voice: Whisper → text → GPT normalization to JSON. If text or photo: GPT cleans and classifies.
- Attach location based on phone GPS.

Acceptance

- Reports appear to admins within 2s of submission. Transcription accuracy acceptable for short utterances.

7) Multilingual Voice & Push Alerts

User

- Admin can send “Auto-generate calm instructions” — chooses languages — TTS audio is generated & sent to speakers and push messages.

How

- GPT crafts concise instruction. TTS API returns audio file (e.g., mp3). Backend routes audio to speakers or mobile (via FCM or direct playback).

Acceptance

- TTS audio plays correctly on tested speaker endpoints and mobile push includes both text & audio.

8) Volunteer Tasking & Assignment

User

- Admin assigns incident to volunteer(s). Volunteer receives push + instructions and marks resolved.

How

- Events include assignee_id and notifies volunteer via Socket.IO and push.

Acceptance

- Assignment delivered within 2s; volunteer can mark resolved and this updates event status.

9) Post Event Analytics / Reports

User

- Export: heatmap timelines, incident log, response times, average densities per zone.

How

- System aggregates stored event & frame time-series and produces CSV or PDF reports.

Acceptance

- Report generation for 1 week of data completes within acceptable time (e.g., < 60s for aggregated summary).

#4 — Data model examples (concrete schemas)

frames (one snapshot)

frame_id TEXT PRIMARY KEY

source_id TEXT

s3_url TEXT

ts_utc TIMESTAMP

width INT

height INT

fps_estimate FLOAT

analyses

```
{
  "analysis_id": "uuid",
  "frame_id": "uuid",
  "crowd_density": "low|medium|high|critical",
  "estimated_people": 450,
  "risk_level": "none|low|medium|high|critical",
  "detected_behaviors": ["panic_movement","blocking_exit"],
  "confidence": 0.87,
  "raw_response": { ... } // full JSON from OpenAI for auditing
}
```

reports

```
{
  "report_id": "uuid",
  "user_id": "nullable-hash",
  "type": "panic|congestion|medical|lost_person|hazard",
  "lat": 23.1765,
  "lon": 75.7885,
  "text": "string",
```

"media_url": "s3://bucket/key",

"transcript": "if voice",

"status": "new|triaged|assigned|resolved",

"created_at": "timestamp"

}

events

{

"event_id":"uuid",

"kind":"analysis|report|manual",

"source_frame_id":"uuid",

"related_report_id":"uuid",

"severity":"low|medium|high|critical",

"zone_id":"ramghat-A",

"summary":"short human readable text",

"assigned_to":"user-id",

"created_at":"timestamp",

"resolution":{"state":"open|closed","closed_by":"user","closed_at":"timestamp"}

}

#5 — Data flow & sequence steps (detailed)

I'll give the flow for the two primary inputs: camera frames and pilgrim reports.

A) Camera snapshot → incident pipeline (step-by-step)

1. **Edge snapshot**
    - Camera → edge node (Jetson / local gateway) → generate JPEG snapshot every 1–2 seconds.
    - Edge optionally runs cheap prefilter (people counter or motion detection). If below threshold (empty scene), it skips sending to reduce cost.
2. **Upload**
    - Edge posts snapshot to backend POST /api/frames/upload with source_id.
    - Backend saves to S3 / MinIO (private bucket) and stores frame metadata.
3. **Background analysis**
    - Backend enqueues job (Celery/RQ/BackgroundTasks).
    - Job downloads image (or points OpenAI to presigned URL), calls **OpenAI Vision/Responses** with a prompt that requests JSON describing density & behaviors (see prompt examples below).
    - Parse response → create analyses record.

4. **Decision**
    - If risk_level >= medium, create event and compute summary text via GPT (concise instruction + action).
    - Publish event to Socket.IO channel incident and to push-notification channel if needed.
5. **UI**
    - Dashboard receives socket event → shows incident card + update heatmap.
6. **Responder**
    - Admin assigns → notify volunteer → volunteer acts → marks resolved → event closed; record response time.

B) Pilgrim report → incident pipeline

1. **User action**
    - User presses panic button or uploads photo/audio.
2. **Upload**
    - App uploads media to backend POST /api/reports/submit including geolocation.
3. **Transcription/Normalization**
    - If audio: backend calls Whisper → transcript.
    - Use GPT to normalize text → classify type and estimate urgency (e.g., “crowd pushing, many elderly trapped” → risk medium/high).
4. **Vectorization (lost-person / matching)**
    - If photo: create embedding (local face embedding or OpenAI visual embedding) and search vector DB for matches.
5. **Event creation**
    - Create report and possibly event if severity ≥ threshold → publish via Socket.IO.

#6 — Concrete example prompts (how we ask OpenAI)

Important: never send raw PII (unredacted faces, personal phone numbers) to third-party services without consent. Where possible, pre-filter or anonymize.

A) Image analysis prompt template (for OpenAI Vision/Responses)

You will be given a single image URL. Analyze the scene and return EXACTLY a JSON object (no extra text) with fields:

```
{
  "crowd_density": "none|low|medium|high|critical",
  "estimated_people": integer or null,
  "risk_level": "none|low|medium|high|critical",
  "detected_behaviors": ["panic_movement","blocking_exit","stampede_wave","scattered_fighting","obstruction"],
  "confidence": float (0-1)
```

}

Details: estimate people and note behaviors such as sudden directional movement, piling up near an exit, or visible panic gestures. Only return JSON.

Image URL: {IMAGE_URL}

B) Report normalization prompt template

Input: a free text report (possibly from speech transcription) and optional media URL and lat/lon.

Task: Extract JSON:

{

"type": "panic|congestion|medical|lost_person|hazard",

"summary": "one-sentence summary",

"severity": "low|medium|high",

"recommended_action": "string (short instruction for volunteer/PA)",

"lang": "hi|en|mr|..."

}

Return JSON only.

Text: {REPORT_TEXT}

Media: {MEDIA_URL}

Location: {LAT},{LON}

C) TTS prompt (text generation)

Given event summary and zone name, generate a short calm instruction for the public, suitable for loudspeaker, under 20 seconds. Provide versions for English and Hindi. Output JSON:

{

"en": "...",

"hi": "..."

}

#7 — Tech stack & why (short)

- **FastAPI** — fast to develop, async-friendly for background tasks and WebSockets.
- **React + Tailwind + Leaflet** — rapid frontend with a polished map UX.
- **Flutter** — single mobile codebase for Android/iOS.
- **MinIO (S3), Postgres, Redis, RabbitMQ, Milvus** — solid dev stack: object storage, relational data, cache/pubsub, message bus, vector DB.
- **OpenAI** — Vision, Responses, Embeddings, Whisper, and TTS to avoid building and maintaining complicated deep learning models from scratch.

- **Docker / Kubernetes** — runs anywhere and scales.

#8 — Security & privacy (practical rules)

1. **Rotate the key now.** If you pasted it anywhere, treat it as compromised.
2. Put OPENAI_API_KEY and all credentials in **secret manager** (Replit Secrets, Vault, AWS Secrets Manager).
3. **Device authentication**: mTLS / per-device certs or unique tokens for edge devices.
4. **Limit PII going to third parties**: blur faces or only send anonymized vectors or cropped non-identifiable images when possible.
5. **RBAC** & audit logs: every admin action logged; roles: admin/commander/medic/volunteer/read-only.
6. **Encryption** at rest and in transit (TLS, S3 server-side or encrypted objects).
7. **Retention**: raw footage lifecycle 7–30 days; snapshots 30 days; event logs 1–3 years for audits.

#9 — Cost, performance & scaling considerations (real-world)

- OpenAI per-image / per-response cost: expect non-trivial costs if you analyze every frame. Use edge prefiltering to sample only important frames.
- Use caching and deduping to avoid sending repeated images.
- Use vector DB for efficient nearest-neighbor lookups (Milvus/Pinecone).
- Use autoscaling in k8s for backend and message consumers.
- Monitor OpenAI rate limits & add circuit-breakers and fallback (e.g., degrade to local heuristics).

#10 — Testing & validation plan (concrete tests)

Unit tests

- /api/frames/upload stores file and enqueues job (mock OpenAI).
- /api/reports/submit handles audio → transcribes (mock Whisper).

Integration

- Start local infra (docker-compose) and run a simulated camera sender (ffmpeg loop) to upload frames and verify event creation.

Load

- Simulate 50 cameras at 1 snapshot / 2s and 200 concurrent dashboards; measure CPU, memory, and queue backlog.

Security

- Attempt unauthenticated device ingest (should fail).
- Attempt to access presigned URL after expiry (should fail).

Acceptance

- Admin dashboard shows incidents created from simulated panic images and volunteer receives assignment.

#11 — Developer build plan — step-by-step (smallest details)

I'll list ordered implementation steps you (or your team) can follow.

Step 0 — Prep

- Create repo with backend/, frontend/, mobile/, infra/.
- Add .env.example and README.
- Add secret manager instructions.

Step 1 — Local infra

- Create docker-compose.yml for Postgres, Redis, MinIO, RabbitMQ, Milvus.
- Start docker compose up -d.

Step 2 — Backend skeleton

- FastAPI project layout: app/main.py, api/frames.py, api/reports.py, services/storage.py, services/openai_client.py, models/.
- Implement S3 upload with MinIO using boto3.
- Add BackgroundTasks for analysis.

Step 3 — OpenAI client plumbing

- Implement async HTTP client for OpenAI endpoints.
- Implement analyze_image_frame() using prompt template above and parse JSON safely (robust JSON parsing).

Step 4 — Event system

- When analysis returns risk_level >= threshold create events and publish via Socket.IO.
- Implement sio.emit("incident", payload).

Step 5 — Frontend MVP

- Map (Leaflet) + incident list + socket client.
- Hook to /api/frames/upload test endpoint.

Step 6 — Mobile MVP

- Flutter app with Panic button calls /api/reports/submit, attach location & optionally media.

Step 7 — Vector DB & Lost & Found

- Hook embeddings endpoint to create vectors → store in Milvus.
- Search workflow and admin UI to confirm matches.

Step 8 — TTS & Multilingual alerts

- Use GPT to generate text; use TTS to create audio files; store audio in S3 and provide playback endpoints.

Step 9 — Tests & CI/CD

- Add pytest tests and basic GitHub Actions for lint/test/build. Add Docker images.

Step 10 — Harden & scale

- Add RBAC, mTLS for devices, encryption, monitoring (Prometheus + Grafana), and scaling policies.

#12 — Example developer commands & quick test

From repo root:

```
# start infra
cd infra
docker compose up -d

# start backend dev
cd backend
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
export OPENAI_API_KEY="(use secret manager in production)"
uvicorn app.main:app --reload --port 8000

# test frame upload (example)
curl -X POST "http://localhost:8000/api/frames/upload" \
  -F "source_id=demo-01" \
  -F "file=@/path/to/test.jpg"
```

#13 — What I will build (if you want me to implement next)

I can produce any of the following immediately, ready to paste into your repo:

- Full FastAPI backend code with working OpenAI request templates (no key embedded), S3 upload, background job skeleton, Socket.IO emission, and unit tests.
- React frontend that connects to the socket and shows map/incident list and snapshot viewer.
- Flutter mobile skeleton with panic button and upload flows.
- Docker Compose infra for local dev and a minimal Kubernetes manifest for production.

You earlier asked me to produce all code for backend/frontend/mobile and I already provided scaffolds. I can now expand any of them into production-ready modules (e.g., real Alembic migrations, complete vector DB integration, or detailed Socket.IO auth).

TL;DR — What you'll build (MVP)

1. **Backend (FastAPI)** — receives frames & reports, stores media, calls OpenAI for analysis, stores events, emits real-time updates.
2. **Realtime Engine (Socket.IO / WebSockets + Redis/RabbitMQ)** — pushes events to dashboard/mobile.
3. **OpenAI integrations** — Vision for frame analysis, Whisper for voice, Embeddings for semantic search, TTS for alerts. OpenAI Platform+3OpenAI Platform+3OpenAI Platform+3
4. **Frontend (React + Tailwind)** — admin console: live feeds, heatmaps, incidents, controls.
5. **Mobile (Flutter)** — pilgrim/volunteer app: reports (photo/voice/text), panic button, push notifications.
6. **Infra (Docker Compose → Kubernetes)** — Postgres, MinIO (S3), Redis, RabbitMQ / Kafka, Vector DB (Milvus/Pinecone).
7. **Security & privacy** — mTLS for devices, server-side key usage, PII minimization, retention policies.

Workspace & prerequisites (do *this* first)

Commands assume Linux/macOS. Use WSL on Windows.

1. Install:
    - Python 3.11+ (pyenv recommended)
    - Node.js 18+ (nvm recommended)
    - Docker & Docker Compose
    - Flutter SDK (for mobile)
    - Git
2. Create directories:

mkdir drishti && cd drishti

mkdir backend frontend mobile infra docs

3. **Secrets** — locally use a .env file for dev but in prod use a secrets manager (Vault / AWS Secrets Manager / GCP Secret Manager / GitHub Secrets).
   Create .env.dev (never commit):

OPENAI_API_KEY=sk-...

S3_ENDPOINT=http://minio:9000

S3_ACCESS_KEY=minioadmin

S3_SECRET_KEY=minioadmin

DATABASE_URL=postgresql://postgres:postgres@postgres:5432/drishti

REDIS_URL=redis://redis:6379/0

RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672/

VECTORDB_URL=http://milvus:19530

Again: rotate your pasted key now. Do not push .env to git.

Repo layout (recommended)

```
drishti/
├– backend/
|  ├– app/
|  |  ├– main.py
|  |  ├– api/
|  |  |  ├– frames.py
|  |  |  ├– reports.py
|  |  ├– services/
|  |  |  ├– openai_client.py
|  |  |  ├– storage.py
|  |  |  ├– vectordb.py
|  |  ├– models/
|  |  |  ├– db.py
|  |  |  ├– schema.py
|  ├– Dockerfile
|  └– requirements.txt
├– frontend/
|  ├– src/
|  ├– package.json
├– mobile/
|  ├– pubspec.yaml
├– infra/
|  ├– docker-compose.yml
|  ├– k8s/
└– docs/
```

Step-by-step build (smallest detail — Phase 0 → MVP)

Phase 0 — Initialize repo & local infra

1. git init and create a root .gitignore:

```
# .gitignore
.env*
```

```
/node_modules
/.venv
*.pyc
/dist
```

2. **Docker Compose** (infra/docker-compose.yml) — minimal dev stack: Postgres, Redis, MinIO, RabbitMQ, Milvus (vector DB), backend service:

```
version: '3.8'
services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: drishti
    ports: ["5432:5432"]
    volumes: ["pgdata:/var/lib/postgresql/data"]
  redis:
    image: redis:7
    ports: ["6379:6379"]
  minio:
    image: minio/minio
    command: server /data
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: minioadmin
    ports: ["9000:9000"]
  rabbitmq:
    image: rabbitmq:3-management
    ports: ["5672:5672","15672:15672"]
  milvus:
    image: milvusdb/milvus:v2.2.9-20230912-d4b3b78
    environment:
      # use default dev settings
    ports: ["19530:19530"]
```

```
  backend:
    build: ./backend
    ports: ["8000:8000"]
    environment:
      - DATABASE_URL=postgresql://postgres:postgres@postgres:5432/drishti
      - REDIS_URL=redis://redis:6379/0
      - S3_ENDPOINT=http://minio:9000
    depends_on: ["postgres","redis","minio","rabbitmq","milvus"]
volumes:
  pgdata:
```

3. Start stack:

```
docker compose -f infra/docker-compose.yml up -d --build
```

Phase 1 — Backend skeleton (FastAPI) — wire it up

1. Backend requirements.txt

```
fastapi
uvicorn[standard]
sqlalchemy[aio]
asyncpg
pydantic
boto3
python-dotenv
aiohttp
redis
pika        # RabbitMQ client
httpx
openai      # OpenAI Python SDK (or use official client if available)
python-multipart
psycopg2-binary
python-jose[cryptography]  # for JWT
```

2. Backend Dockerfile

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
```

```
RUN pip install -r requirements.txt
COPY . .
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]
```

3. Minimal FastAPI app — backend/app/main.py

```
from fastapi import FastAPI
from app.api import frames, reports

app = FastAPI(title="Drishti Backend")

app.include_router(frames.router, prefix="/api/frames", tags=["frames"])
app.include_router(reports.router, prefix="/api/reports", tags=["reports"])
```

4. Frame ingest endpoint — backend/app/api/frames.py

```
from fastapi import APIRouter, UploadFile, File, Form, Depends
from app.services.storage import store_file_public
from app.services.openai_client import analyze_image_frame
from app.models.db import create_frame_record

router = APIRouter()

@router.post("/upload")
async def upload_frame(source_id: str = Form(...), file: UploadFile = File(...)):
    # 1) save file to object storage (MinIO / S3)
    url = await store_file_public(file, source_id)
    # 2) create DB record (frame)
    frame_id = await create_frame_record(source_id, url)
    # 3) call OpenAI analyze (async background ideally)
    analysis = await analyze_image_frame(url, frame_id)
    return {"frame_id": frame_id, "analysis": analysis}
```

5. Storage helper — backend/app/services/storage.py

```
import boto3
import os
from aiofiles import open as aioopen

s3 = boto3.client(
```

```
    "s3",
    endpoint_url=os.getenv("S3_ENDPOINT"),
    aws_access_key_id=os.getenv("S3_ACCESS_KEY"),
    aws_secret_access_key=os.getenv("S3_SECRET_KEY"),
)

async def store_file_public(upload_file, source_id):
    # read file bytes
    contents = await upload_file.read()
    key = f"{source_id}/{upload_file.filename}"
    s3.put_object(Bucket="drishti-media", Key=key, Body=contents, ACL="private")
    # presigned url
    url = s3.generate_presigned_url('get_object', Params={'Bucket':'drishti-media','Key':key}, ExpiresIn=3600)
    return url
```

(**Note:** use private S3 and presigned URL. Don't make raw video publicly accessible.)

Phase 2 — OpenAI integration (Vision + Text + Embeddings + Speech)

Guiding principle: All OpenAI calls must be server-side; never call OpenAI directly from client/mobile. Keep the key on the server and read it from secrets.

1) Configure OpenAI client (server-side)

backend/app/services/openai_client.py:

```
import os
import httpx
OPENAI_KEY = os.getenv("OPENAI_API_KEY")

HEADERS = {"Authorization": f"Bearer {OPENAI_KEY}"}

async def analyze_image_frame(image_url: str, frame_id: str):
    """
    Send the image URL to OpenAI Vision/Responses API to get structured JSON:
    e.g., crowd_density, estimated_people, risk_level, behaviors
    """
    # Example: send a prompt asking for a strict JSON output. Use the official responses endpoint.
    payload = {
```

```
    "model": "gpt-4o-mini",  # choose appropriate vision-enabled model
    "input": [
      {"role": "user", "content": f"Analyze the image at {image_url}. Output JSON: {{crowd_density, estimated_people, risk_level (none|low|medium|high|critical), detected_behaviors:list}} and nothing else."}
    ]
  }
  async with httpx.AsyncClient(timeout=30.0) as client:
    r = await client.post("https://api.openai.com/v1/responses", headers=HEADERS, json=payload)
    r.raise_for_status()
    return r.json()
```

Read the official Vision & Responses docs for exact payload format and to pick the suitable model names and fields. (Docs: Vision guide, Responses API). OpenAI Platform+1

Why server-side? You must keep your API key secret, control rate limits, and enforce privacy (throttle/obfuscate PII) before sending to OpenAI.

---

2) Example: Embeddings for semantic search / lost-and-found

Use embeddings to index textual descriptions of images or OCR results. Official Embeddings docs: OpenAI Platform+1

backend/app/services/embeddings.py:

```
import os
import httpx
OPENAI_KEY = os.getenv("OPENAI_API_KEY")
EMBED_URL = "https://api.openai.com/v1/embeddings"

async def create_embedding(text: str, model="text-embedding-3-large"):
  async with httpx.AsyncClient() as client:
    r = await client.post(EMBED_URL, headers={"Authorization":f"Bearer {OPENAI_KEY}"}, json={"input": text, "model": model})
    r.raise_for_status()
    return r.json()["data"][0]["embedding"]
```

Store resulting vectors in your vector DB (Milvus or Pinecone) with metadata (image_url, frame_id, timestamp). Use vector DB nearest-neighbor search for matches.

---

3) Speech → text (use Whisper / Speech-to-text) for voice reports

- Use the official Speech-to-Text docs for endpoint details. OpenAI Platform+1

Flow:

- Pilgrim uploads short voice clip (mobile app).
- backend uploads clip to S3 (or directly stream).
- Call OpenAI Speech-to-Text endpoint, get transcript → store as a report and run the same GPT normalization flow.

---

4) Text → speech (TTS) for loudspeaker announcements

- Use official TTS docs to create audio files for multilingual announcements. OpenAI Platform+1

Flow:

- Admin triggers an alert (or system auto-generates).
- Use OpenAI to generate calm instruction text in target languages (GPT prompt).
- Pass text to TTS → get audio → send to edge speaker (or play on mobile).

Phase 3 — Data model and DB schemas (minimum)

Use Postgres (SQLAlchemy / Alembic). Minimal tables:

- sources (source_id, name, type, location, protocol, status)
- frames (frame_id, source_id, s3_url, timestamp)
- analyses (id, frame_id, json_payload, risk_level, created_at)
- reports (report_id, user_id/anon, type, text, media_url, geo, status)
- events (event_id, kind, severity, zone_id, related_frame_id, created_at)

Add indices on created_at, zone_id, frame_id.

Phase 4 — Real-time notifications & frontend wiring

Real-time channel:

- Use **Socket.IO** (backend) + Redis pub/sub adapter for horizontal scale.
- When analyze_image_frame() returns event with risk_level high/critical → push via Socket.IO to connected dashboard clients and to push-notification service for mobile.

Example (FastAPI + Socket.IO via python-socketio / uvicorn):

```
# create a socket server and emit

import socketio

sio = socketio.AsyncServer(async_mode='asgi')

app = FastAPI()

app.mount("/ws", socketio.ASGIApp(sio))


# when an event occurs

await sio.emit("incident", {"event_id":event_id, "severity":"critical"})
```

Frontend (React) essentials

- Map using **Leaflet.js** and heatmap plugin (leaflet-heat).
- WebRTC viewer in a panel for live feed via backend signaling (Janus/Pion) - but MVP: show snapshot frames that update every 1–2s.
- Event list with priority ordering and manual ack/assign controls.

Key components:

- MapView.jsx — renders zones and heatmap overlay from analyses API.
- StreamPanel.jsx — pull presigned HLS or WebRTC.
- IncidentCard.jsx — shows summary (auto-generated by GPT), images, actions.

Phase 5 — Mobile (Flutter) core flows (very small details)

1. Permissions: location + camera + microphone.
2. Panic button: POST /api/reports with:
    - type=panic
    - current geo (lat, lon)
    - optional image or audio (upload to server)
3. Upload code sample (Dart):

```
var request = http.MultipartRequest('POST', Uri.parse('$BACKEND/report'));
request.fields['type'] = 'panic';
request.fields['lat'] = lat.toString();
// attach file
request.files.add(await http.MultipartFile.fromPath('media', filePath));
var resp = await request.send();
```

Server will store media, transcribe audio (Whisper), call OpenAI to normalize and assess severity, then push notifications.

Phase 6 — Vector DB & Lost & Found workflow (small details)

1. When a *lost person photo* is submitted:
    - Option A (privacy-preserving): run face-encoding locally at edge (FaceNet / Face-Recognition lib) to get embedding → store embedding in your vector DB (encrypted) with metadata and retention policy. Use local face matching for quick results. (This avoids sending raw faces to external APIs.)
    - Option B (if you must use OpenAI): send visual info + prompt to extract descriptors (not ideal for face matching). **Prefer hybrid approach** — face embeddings on-edge + text embeddings via OpenAI for fuzzy matching.
2. For cross-matching:
    - Query vector DB for nearest neighbors (Milvus / Pinecone).

- If confidence > threshold, flag candidate and notify admins for human review.

Privacy: only show matches to verified admin roles; blur faces in any public UI.

Phase 7 — Ops, monitoring, and testing

1. **Logging & Observability**
   - Centralized logs (ELK / Loki).
   - Metrics: stream up/down, per-source fps, OpenAI latency & errors, request counts (rate limit hits).
2. **Alerting**
   - PagerDuty / Slack channels for critical system health.
3. **Chaos tests**
   - Simulate network loss, delayed responses from OpenAI, disk full on MinIO.
4. **Load tests**
   - Use k6 or locust to test /api/frames/upload at anticipated rate.

Phase 8 — Security & Compliance (detailed steps)

1. **Key & secrets**
   - Put OPENAI_API_KEY in secret manager.
   - On CI, use GitHub Actions secrets to set environment variables for deploy.
2. **Least privilege**
   - If OpenAI supports scissors/scoped keys or separate billing keys for dev vs. prod, use them.
3. **PII Handling**
   - Encrypt raw media at rest.
   - Auto-expiry lifecycle: raw video = 7–30 days (configurable), snapshots = 30 days, analysis logs = 1–3 years for audit.
   - Redact or blur faces for public exports.
4. **Legal & ethics**
   - Get local govt approvals to process CCTV / face matching.
   - Post signage informing attendees about camera usage.
5. **Access control**
   - Implement RBAC: roles → superadmin, command_officer, medic, volunteer, read_only. Enforce on every endpoint.

Phase 9 — CI/CD & Deployment (practical)

1. **GitHub Actions** to:

- Lint & run unit tests (backend).
- Build Docker images & push to registry (GHCR / ECR).
- Deploy to Kubernetes (ArgoCD or kubectl apply).

2. **Kubernetes**
   - Use HPA for backend pods.
   - Use node pools with GPU for inference (if you later do any local model inference).
   - Use Ingress controller (NGINX) with TLS certs (Let's Encrypt or managed certs).

Phase 10 — Tests & acceptance criteria (tangible)

Write tests that cover:

- POST /api/frames/upload stores media and returns an analysis JSON within 3s in normal conditions.
- Transcribe a sample 10s audio file → text accuracy threshold (human reviewed).
- Embedding & vector DB insertion + query returns expected nearest neighbor for test data.

MVP Acceptance:

- Admin opens dashboard, sees heatmap updated from sampled frames.
- System auto-detects a simulated panic frame and generates an incident card (severity & suggested action).
- Pilgrim app can submit a panic report that appears in the admin UI within 2s–4s.

Example: Full small-path demo you can run locally (quickstart)

1. Start infra:

```
docker compose -f infra/docker-compose.yml up -d --build
```

2. Enter backend, create .env with keys (remember to rotate real keys later):

```
cp backend/.env.example backend/.env
# edit backend/.env and set OPENAI_API_KEY to your secret (dev only)
```

3. Start backend:

```
docker compose up -d backend
# or run locally
cd backend
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
uvicorn app.main:app --reload
```

4. Use curl to test frame upload (simulate with local image):

```
curl -X POST "http://localhost:8000/api/frames/upload" \
```

-F "source_id=demo-01" \

-F "file=@tests/data/panic_frame.jpg"

You should receive a JSON response with frame_id and an analysis object (from OpenAI).

OpenAI-specific notes & citations (you must read)

- **Vision & multimodal analysis** — use the Vision / Responses API for sending images and getting structured responses. The docs describe how to send images and prompts and what models to choose. OpenAI Platform+1
- **Embeddings** — create embeddings via the Embeddings API and store vectors for semantic search. Model choices and best practices are in the Embeddings guide. OpenAI Platform+1
- **Speech-to-text (Whisper)** — use Whisper endpoints for transcription of voice reports. See the speech-to-text guide. OpenAI Platform+1
- **Text-to-speech (TTS)** — OpenAI supports TTS for multilingual alerts; see the TTS docs for streaming/producing audio files. OpenAI Platform+1
- **General API reference** — for exact endpoint shapes, rate limits, and request/response schemata check the OpenAI API reference. OpenAI Platform

---

Ethics & privacy final reminder

This system handles face data and public safety. Include explicit approvals from local authorities and consult legal counsel. Prefer **on-edge** face embedding & matching to avoid uploading identifiable faces to third-party APIs where possible.

ADDITIONAL PROMPT

You are Replit AI. Build me a **full-stack web + mobile ready application** called **Project Drishti** for crowd safety and surveillance at Mahakumbh 2028.

⚡ Requirements:

1. **Tech Stack**

- **Backend**: FastAPI (Python) with uvicorn

- **Frontend**: React + Tailwind + Leaflet.js + Socket.IO

- **Mobile**: Flutter (for pilgrims/volunteers) [provide repo instructions & starter code]

- **Infra**: Docker Compose for Postgres, Redis, MinIO (S3), RabbitMQ, Milvus (vector DB)

2. **Backend Features**

- API endpoints:

- `/api/frames/upload` → accepts CCTV/drone image frame, stores in MinIO, calls **OpenAI Vision API** to analyze crowd density, panic detection, and behaviors. Return structured JSON.

- `/api/reports/submit` → accepts pilgrim/volunteer reports (panic button, lost person, congestion) via text, photo, or audio.

- If audio: upload → OpenAI Whisper Speech-to-Text → text → GPT normalization into JSON.

- WebSocket/Socket.IO channel `/ws` → streams real-time incidents to frontend dashboard.

- Services:

- **Storage** (MinIO/S3 signed URLs).

- **OpenAI Client**:

- Vision (GPT-4o) → crowd analysis.

- Text (GPT-4.1) → summarization & multilingual alerts.

- Embeddings (text-embedding-3-large) → lost & found / vector DB search.

- Speech-to-Text (Whisper) → pilgrim audio → text.

- Text-to-Speech → generate multilingual safety announcements.

- Database Models (SQLAlchemy + Alembic):

- `sources` (cameras, drones, sensors)

- `frames` (id, source_id, url, timestamp)

- `analyses` (frame_id, crowd_density, estimated_people, risk_level, behaviors)

- `reports` (report_id, type, lat, lon, text, media_url, status)

- `events` (event_id, severity, zone_id, linked_frame/report)

- BackgroundTasks → offload OpenAI API calls so API is non-blocking.

- RBAC authentication (JWT with roles: admin, police, medic, volunteer).

- Tests (`pytest`) for endpoints and AI integrations.

3. **Frontend (React) Features**

- **Dashboard UI** with Tailwind:

- **Map View** (Leaflet.js heatmap) overlay showing live crowd density from analyses.

- **Live Incident Feed** (Socket.IO updates).

- **Video/Frame Panel** → fetch presigned HLS/WebRTC or snapshots.

- **Controls**:

- Acknowledge incidents

- Assign to volunteers

- Trigger multilingual announcements (backend calls TTS → plays on client).

- **API Integration** with backend using Axios.

- Real-time updates with Socket.IO.

- Minimal but clean design (grid layout, cards, alerts with severity colors).

4. **Mobile App (Flutter) Features**
   - Screen 1: Location + Panic Button → sends report to `/api/reports/submit`.
   - Screen 2: Upload photo (lost person) → backend stores, generates embedding, matches against vector DB.
   - Screen 3: Voice report (mic) → upload audio → Whisper transcription → backend normalizes.
   - Use `flutter_dotenv` to configure BACKEND_URL.
   - Show success/failure via Snackbars.

5. **Infra Setup**
   - `docker-compose.yml` with services:
     - Postgres (DB)
     - Redis (cache/pubsub)
     - MinIO (S3 object store)
     - RabbitMQ (messaging)
     - Milvus (vector DB)
     - Backend service (build from backend/Dockerfile)
   - Mount volumes for persistence.
   - Pre-create MinIO bucket `drishti-media`.

6. **Security**
   - Use **Replit Secrets Manager** to store `OPENAI_API_KEY`, DB creds, and MinIO keys.
   - Backend must **read API key from env var** (`os.getenv("OPENAI_API_KEY")`).
   - No hardcoding keys in repo.
   - HTTPS enabled in production.
   - Role-based JWT auth.

7. **Testing & Validation**
   - Write `pytest` unit tests for `/api/frames/upload` and `/api/reports/submit`.
   - Simulate image uploads and mock OpenAI responses.
   - Add Cypress/E2E tests for frontend (optional stretch).

8. **Deployment**
   - Provide `Procfile` for Replit (backend via uvicorn).
   - Provide `replit.nix` config to install dependencies (Python 3.11, Node.js 18, PostgreSQL client).

- Provide instructions for mobile app to connect locally & in prod.

9. **Developer UX**
   - Pre-configure hot-reload for backend (uvicorn `--reload`) and frontend (`npm run dev`).
   - Clear README with:
     - How to run `docker-compose up`
     - How to set Replit secrets (`OPENAI_API_KEY`)
     - How to start frontend (`npm start`)
     - How to run mobile app (`flutter run`)

💡 **Replit, please generate:**

- Full `backend/app/` code (FastAPI) with all services, API routes, DB models, OpenAI integrations.
- Full `frontend/src/` (React + Tailwind + Leaflet + Socket.IO).
- `mobile/` folder with Flutter skeleton and Panic button flow.
- `infra/docker-compose.yml` with Postgres, Redis, MinIO, RabbitMQ, Milvus.
- `README.md` with full setup + run instructions.
- `replit.nix` with Python, Node.js, and Flutter dependencies.
- Tests (`pytest` for backend).

⚠️ After project generation:

- Go to Replit **Secrets Manager** → set:
  - `OPENAI_API_KEY` = your rotated OpenAI key.
  - `S3_ACCESS_KEY`, `S3_SECRET_KEY`, `DATABASE_URL`, etc.
- Don’t commit your keys.
- Use `os.getenv("OPENAI_API_KEY")` inside backend.

Question - i have no cameras right now then how can my app curreltly get the data from and also how can i test the app

Simulate a Camera Feed

You don't need a real CCTV — you can **script ffmpeg or Python OpenCV** to feed frames at intervals:

ffmpeg -re -stream_loop -1 -i test_crowd_video.mp4 -vf fps=1 frames/frame_%04d.jpg

- This generates a new frame every second from a test video.
- Then you POST those frames into your /api/frames/upload endpoint.

So your backend thinks they're live CCTV snapshots.

---

3. Crowd Reports Simulation

Since pilgrims/volunteers also contribute data:

- You (or testers) can use the mobile/web app to submit **fake reports**:
  - "There's congestion near Gate 2" (text only).
  - Record a short audio clip saying "Someone fainted here" → backend runs Whisper → GPT normalizes.
  - Upload a photo of a random person → backend treats as "lost person report."

📌 This lets you test **report ingestion, Whisper transcription, JSON structuring, and event creation.**

---

4. Lost Person Testing Without Real People

- Use **public domain sample portraits** (like Unsplash, Generated Photos, or AI-generated faces).
- Upload one as a "lost person report."
- Upload a different image of the same person (cropped, blurred, or with a different background) → test if your vector DB search finds them.

📌 Even without real pilgrims, you can test **facial matching workflow.**

---

5. Replay Mode (Synthetic Data Generator)

You can build a **data generator script** that periodically:

- Picks a random test image → uploads to /api/frames/upload.
- Generates a random "pilgrim report" JSON (text or fake lost-person image).
- Occasionally generates "panic event" with severity = high.

This way, your dashboard always looks like something is happening.
✅ Great for **demo day** even if you don't have real hardware.

✅ **How to Test the Whole App Right Now**

Step 1: Backend Test (No Frontend Yet)

- Run your FastAPI backend with uvicorn.
- Use curl or Postman to:
    - Upload test.jpg → see JSON from OpenAI.
    - Submit fake report → see normalized output.

Step 2: Frontend Dashboard Test

- Start your React dashboard.
- Start the backend with simulated data feeder (Python script).
- Watch live map + incident feed update with fake crowd densities.

Step 3: Mobile App Test

- Run Flutter app in emulator.
- Press Panic button → see event created on dashboard.
- Upload sample lost person photo → check if "match" comes back.

Step 4: End-to-End Demo (Simulation)

- Start data generator (uploads test crowd images every 2s).
- Start backend + frontend.
- Invite a friend to open the pilgrim/volunteer app.
- Within a few minutes, your dashboard shows live "incidents," reports, and even lost person test cases.

✅ **What You'll Achieve Before Real Cameras**

- You can fully validate your **pipelines, database models, AI calls, event system, and UI**.
- When you finally plug in real cameras (via RTSP snapshots), nothing changes in your backend — it's just another input stream.
- You'll already know the system **works at scale** with synthetic data.