Urban Safety: Implementing Video Segmentation and Object Detection in Python and Building a Live Website

Group 1: Data Science

Bett Kipkemoi

Dominic Ong'aro

Faith Ngina

Dennis Mmenya Mwange

Video analytics pipeline for public safety

Features:

- Simulates simple synthetic public-space video with moving "people" and "vehicles".

- Runs motion segmentation (background subtraction) and object detection (Faster R-CNN if available, falls back to OpenCV HOG for people detection).

- Simple centroid tracker to detect loitering, crowding, and stationary vehicles (possible illegal parking).

- Produces an annotated output video, per-frame CSV log, and a short JSON report with actionable strategies.

Usage:

- Simulate and process:

python sentiment-analysis.py --simulate --out annotated.mp4 --log events.csv –report report.json

- Process an existing video:

python sentiment-analysis.py --input input.mp4 --out annotated.mp4 --log events.csv --report report.json

Dependencies:

- python >= 3.8

- opencv-python

- numpy

- pandas

- torch, torchvision

Note: This is a prototype for research/teaching. For production use, use optimized detectors, robust trackers, privacy-preserving pipelines, and follow local legal/regulatory guidance.

Reporting to the Website: Operational Logic

1. Data Acquisition

High-definition video feeds are ingested from a network of 1,200+ CCTV nodes distributed across Nairobi's Central Business District. Feeds are encrypted and transmitted via secure fiber optic links to the central processing unit.

2. Computer Vision Processing

Real-time AI models process video frames at 30fps.

- **Object Detection:** Identifying pedestrians, vehicles, and unattended objects.
- **Density Mapping:** Calculating crowd volume per square meter.
- **Anomaly Detection:** Flagging rapid movements, loitering, or traffic disruptions.\

3. Analysis and Correlation

Detected events are cross-referenced with historical data and temporal patterns. The system identifies deviations from the norm (e.g., a crowd forming at an unusual hour) to assess potential safety risks.

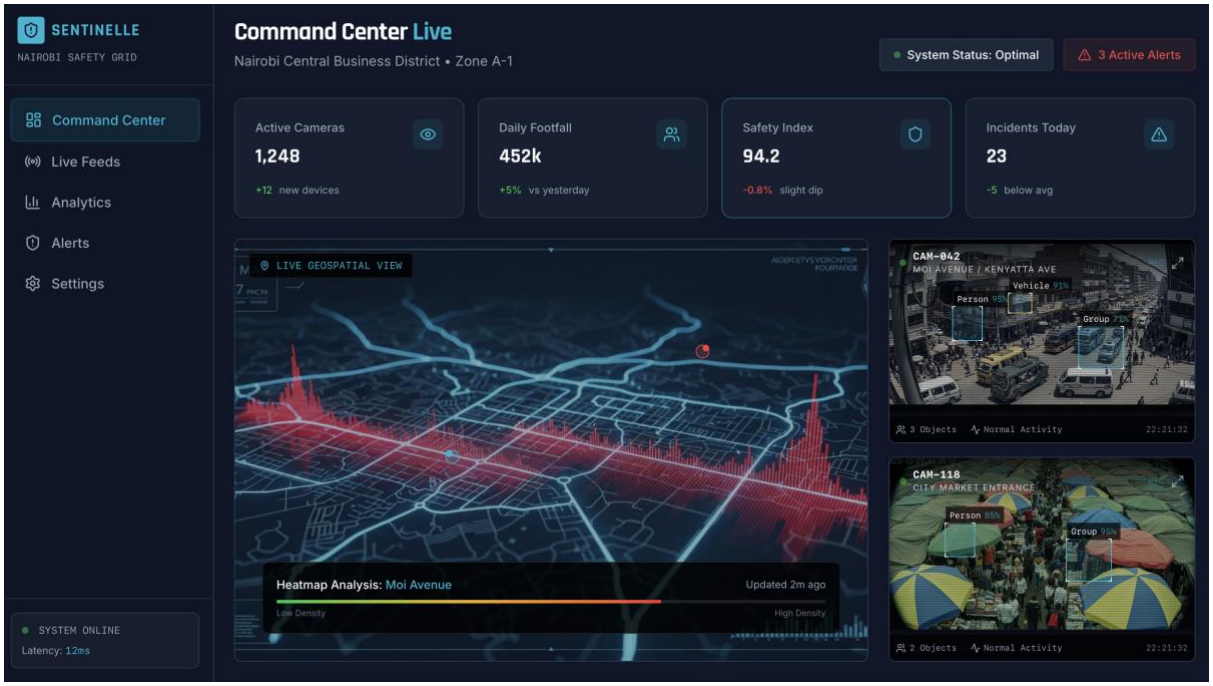
4. Response and Alerting

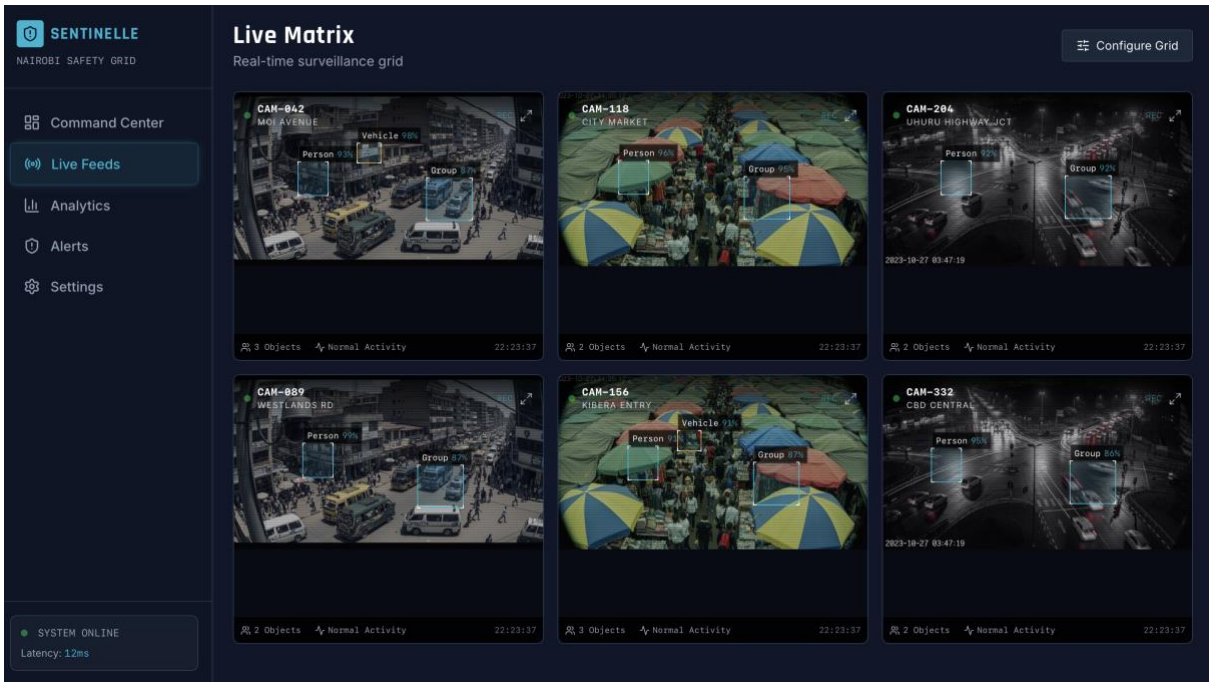
Verified incidents trigger automated alerts to the Command Center dashboard. High-priority threats (Tier-1) can be configured to automatically dispatch notifications to relevant field units or emergency services.

Daily Recommendations

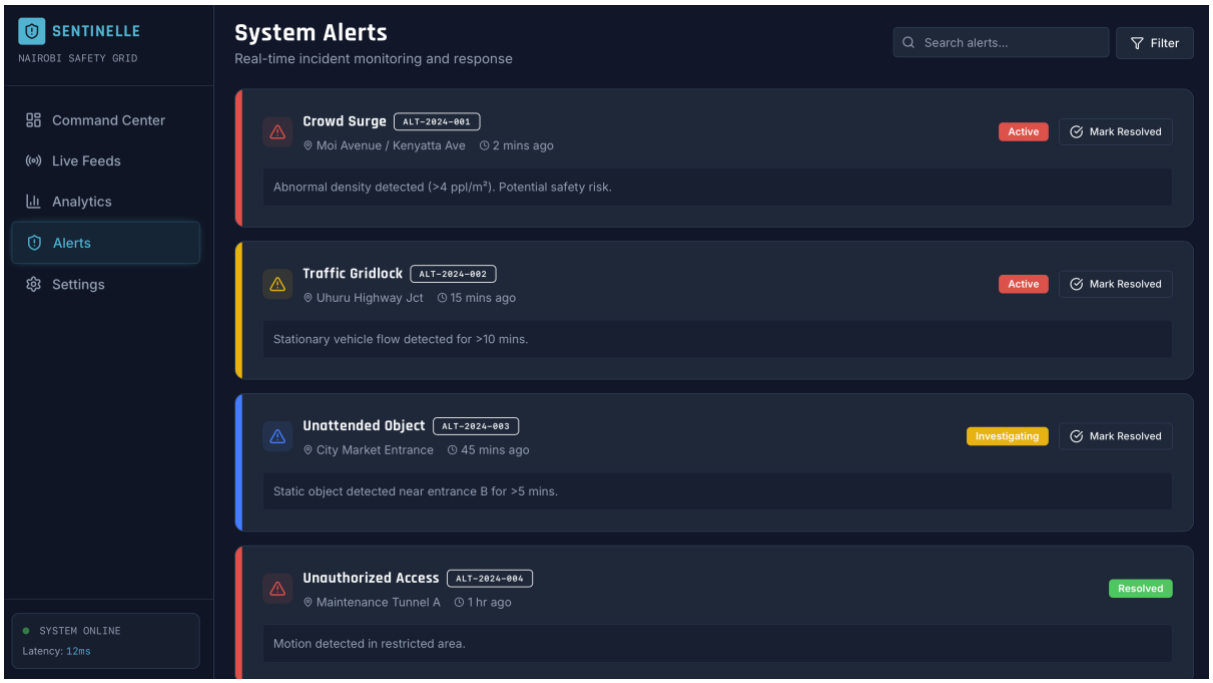
Zone	Observation	Recommended Action	Priority
Moi Avenue	Crowd density > 85% capacity	Deploy crowd control unit; divert pedestrian flow	**High**
City Market	Increased petty theft reports (14:00-16:00)	Increase visible patrol presence during peak hours	**Medium**
Uhuru Highway	Traffic signal latency causing gridlock	Recalibrate signal timing software	**Low**

Site Overview: https://urban-watch-ai--bettkipkemoi.replit.app





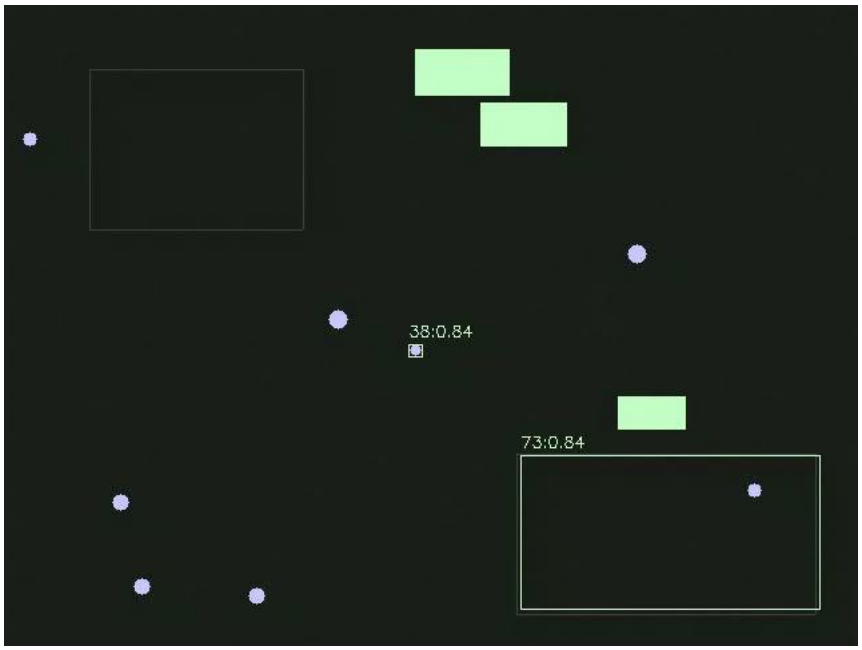

SENTINELLE
NAIROBI SAFETY GRID
Command Center
Live Feeds
Analytics
Alerts
Settings
SYSTEM ONLINE
Latency: 12ms
Safety Analytics
Deep dive analysis and predictive modeling
Last 7 Days
Export Report
Temporal Incident Distribution
Correlation between time of day and reported safety incidents
1000
750
500
250
0
04:00
visitors : 45
incidents : 0
00:00
04:00
08:00
12:00
16:00
20:00
23:59
Incident Categorization
Breakdown by type
Traffic
Loitering
Unattended Object
Crowd Surge
Actionable Recommendations
AI-generated strategies based on current data
HIGH
Increase Patrols in Zone B (Moi Ave)
Unusual spike in crowd density detected between 18:00 - 20:00. Recommend deploying 2 additional units.
MEDIUM
Optimize Traffic Light Timing at Junction 4
Traffic congestion correlating with increased petty theft incidents. Reducing stop time may improve flow and safety.
Predictive Risk Assessment
Forecast for next 24 hours
Crowd Surge Risk
Medium (45%)
Traffic Gridlock Risk
High (82%)
Public Disorder Risk
Low (12%)


SENTINELLE
NAIROBI SAFETY GRID
Command Center
Live Feeds
Analytics
Alerts
Settings
SYSTEM ONLINE
Latency: 12ms
System Alerts
Real-time incident monitoring and response
Search alerts...
Filter
Crowd Surge
ALT-2024-001
Moi Avenue / Kenyatta Ave
2 mins ago
Active
Mark Resolved
Abnormal density detected (>4 ppl/m²). Potential safety risk.
Traffic Gridlock
ALT-2024-002
Uhuru Highway Jct
15 mins ago
Active
Mark Resolved
Stationary vehicle flow detected for >10 mins.
Unattended Object
ALT-2024-003
City Market Entrance
45 mins ago
Investigating
Mark Resolved
Static object detected near entrance B for >5 mins.
Unauthorized Access
ALT-2024-004
Maintenance Tunnel A
1 hr ago
Resolved
Motion detected in restricted area.

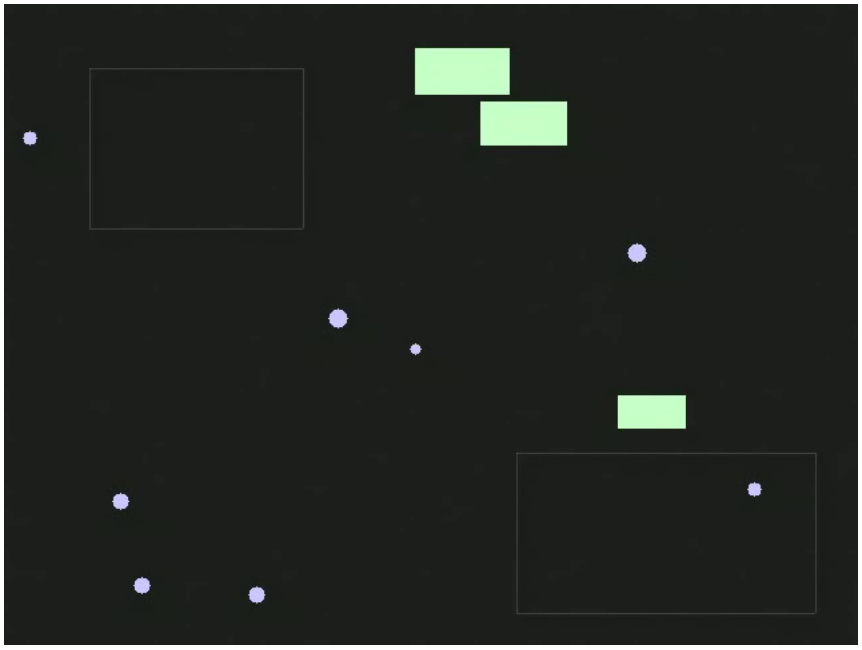
reports.json


```
{
  "video": "simulated_input.mp4",
  "processed_frames": 50,
  "total_people_observations": 0,
  "total_vehicle_observations": 0,
  "hotspots": [],
  "recommendations": [
    "Increase patrol presence and lighting in identified hotspots (grid cells with high people counts).",
    "Install time-limited parking restrictions and signage at bus_stop / market areas; enforce to reduce illegal parking.",
    "Deploy crowd management protocols (additional personnel, temporary barriers) during peak times identified by spikes in crowd alerts.",
    "Use real-time alerts to dispatch response teams for loitering and stationary vehicles, and integrate with local traffic enforcement.",
    "Complement camera analytics with anonymization (blurring faces) and data governance policies to protect privacy."
  ]
}
```


annotated.mp4



simulated.mp4



Code:

```
import argparse
import cv2
import numpy as np
import time
import os
```

```
import csv
import json
from collections import deque, defaultdict
import torch
import torchvision
from torchvision import transforms
import pandas as pd

"""
Video analytics pipeline for public safety

"""



try:
    TORCH_AVAILABLE = True
except Exception:
    TORCH_AVAILABLE = False


# -----------------------
# Simulation
# -----------------------
def simulate_video(path="simulated.mp4", n_frames=600, size=(640, 480), fps=20):
    """Generate a synthetic public-space video with moving rectangles."""
    w, h = size
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    out = cv2.VideoWriter(path, fourcc, fps, (w, h))

    rng = np.random.RandomState(42)
    people = []
    vehicles = []

    # spawn some initial actors
    for _ in range(8):
        people.append({
            "pos": np.array([rng.randint(0, w), rng.randint(0, h)]).astype(float),
            "vel": (rng.rand(2) - 0.5) * 4,
            "size": rng.randint(8, 16),
            "color": (255, 200, 200)
```

```
        })
    for _ in range(3):
        vehicles.append({
            "pos": np.array([rng.randint(0, w), rng.randint(0, h)]).astype(float),
            "vel": (rng.rand(2) - 0.5) * 6,
            "size": rng.randint(24, 40),
            "color": (200, 255, 200)
        })

    # define some zones (x1,y1,x2,y2) for "no-parking" or bus stops
    zones = {
        "market": (int(0.1*w), int(0.1*h), int(0.35*w), int(0.35*h)),
        "bus_stop": (int(0.6*w), int(0.7*h), int(0.95*w), int(0.95*h)),
    }

    for f in range(n_frames):
        frame = np.full((h, w, 3), 30, dtype=np.uint8)

        # draw zones lightly
        for name, (x1,y1,x2,y2) in zones.items():
            cv2.rectangle(frame, (x1,y1), (x2,y2), (60,60,80), 1)

        # spawn occasional new people
        if rng.rand() < 0.03:
            people.append({
                "pos": np.array([rng.randint(0, w), rng.randint(0, h)]).astype(float),
                "vel": (rng.rand(2) - 0.5) * 4,
                "size": rng.randint(8, 16),
                "color": (255, 200, 200)
            })

        # update and draw people
        for p in people:
            p["pos"] += p["vel"]
            # bounce at borders
            for i in (0,1):
                if p["pos"][i] < 0 or p["pos"][i] >= (w if i==0 else h):
                    p["vel"][i] *= -1
                    p["pos"][i] = np.clip(p["pos"][i], 0, (w-1) if i==0 else (h-1))
            cx, cy = int(p["pos"][0]), int(p["pos"][1])
```

```
                r = int(p["size"]/2)
                cv2.circle(frame, (cx, cy), r, p["color"], -1)

        # update and draw vehicles
        for v in vehicles:
            # occasional stop in bus stop zone to simulate parking
            if np.random.rand() < 0.002:
                v["vel"] = np.array([0.0, 0.0])
            # random resume
            if np.random.rand() < 0.01 and np.linalg.norm(v["vel"]) < 0.5:
                v["vel"] = (rng.rand(2) - 0.5) * 6
            v["pos"] += v["vel"]
            for i in (0,1):
                limit = w if i==0 else h
                if v["pos"][i] < 0 or v["pos"][i] >= limit:
                    v["vel"][i] *= -1
                    v["pos"][i] = np.clip(v["pos"][i], 0, limit-1)
            cx, cy = int(v["pos"][0]), int(v["pos"][1])
            s = int(v["size"])
            cv2.rectangle(frame, (cx-s, cy-s//2), (cx+s, cy+s//2), v["color"], -1)

        # add synthetic background motion (e.g., trees shimmer)
        noise = (rng.rand(h, w) * 2).astype(np.uint8)
        frame = cv2.add(frame, np.stack([noise]*3, axis=2))

        out.write(frame)

    out.release()
    return path

# ------------------------
# Detector helpers
# ------------------------
COCO_PERSON_ID = 1
COCO_VEHICLE_IDS = {3, 6, 8}  # car, bus, truck (approx)
COCO_LABELS = None

def load_coco_labels():
    # short COCO label list (index starts at 1)
    labels = {1: "person", 2:"bicycle",3:"car",4:"motorcycle",5:"airplane",6:"bus",
```

```
                    7:"train",8:"truck",9:"boat",10:"traffic light"}
    return labels

def load_torch_detector(device="cpu"):
    global COCO_LABELS
    COCO_LABELS = load_coco_labels()
    if not TORCH_AVAILABLE:
        return None, device
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
    model.to(device)
    model.eval()
    transform = transforms.Compose([
        transforms.ToTensor()
    ])
    return (model, transform), device

# -------------------------
# Simple centroid tracker
# -------------------------
class CentroidTracker:
    def __init__(self, max_distance=50, max_lost=30):
        self.next_id = 1
        self.objects = dict()  # id -> (centroid, last_seen_frame, trace deque)
        self.max_distance = max_distance
        self.max_lost = max_lost

    def update(self, detections, frame_idx):
        # detections: list of (x, y) centroids
        assigned = set()
        new_objects = dict()

        # simple greedy assignment
        for oid, (centroid, last_seen, trace) in list(self.objects.items()):
            best = None
            best_d = None
            for i, c in enumerate(detections):
                if i in assigned:
                    continue
                d = np.linalg.norm(np.array(centroid) - np.array(c))
                if d <= self.max_distance and (best is None or d < best_d):
```

```
                        best = i
                        best_d = d
                if best is not None:
                        c = detections[best]
                        trace.append(c)
                        if len(trace) > 64:
                                trace.popleft()
                        new_objects[oid] = (c, frame_idx, trace)
                        assigned.add(best)
                else:
                        # not matched: keep but mark lost
                        if frame_idx - last_seen <= self.max_lost:
                                new_objects[oid] = (centroid, last_seen, trace)
                        # else drop

        # add unmatched detections
        for i, c in enumerate(detections):
                if i in assigned:
                        continue
                oid = self.next_id
                self.next_id += 1
                dq = deque([c], maxlen=64)
                new_objects[oid] = (c, frame_idx, dq)

        self.objects = new_objects
        return self.objects

# ----------------------
# Processing pipeline
# ----------------------
def process_video(input_path, output_path="annotated.mp4", log_path="events.csv", report_path="report.json",
                          device="cpu", run_detector=True, detect_every=1):
    # try to load detector
    detector, device = None, device
    if run_detector:
        det_res = load_torch_detector(device=device)
        if det_res[0] is None:
                detector = None
                print("Torch unavailable: falling back to HOG people detector.")
        else:
```

```
            detector = det_res[0]
    else:
        detector = None

    cap = cv2.VideoCapture(input_path)
    if not cap.isOpened():
        raise FileNotFoundError(f"Cannot open video {input_path}")

    fps = cap.get(cv2.CAP_PROP_FPS) or 20.0
    w = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    h = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    out = cv2.VideoWriter(output_path, fourcc, fps, (w,h))

    # background subtractor
    backSub = cv2.createBackgroundSubtractorMOG2(history=500, detectShadows=True)

    # tracker for people and vehicles (separate)
    person_tracker = CentroidTracker(max_distance=60, max_lost=int(fps*3))
    vehicle_tracker = CentroidTracker(max_distance=80, max_lost=int(fps*10))

    # prepare HOG fallback
    hog = None
    if detector is None:
        hog = cv2.HOGDescriptor()
        hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

    frame_idx = 0
    logs = []
    hotspots = defaultdict(int)

    while True:
        ret, frame = cap.read()
        if not ret:
            break
        frame_idx += 1
        t0 = time.time()

        fgmask = backSub.apply(frame)
        # morphological cleanup
```

```
        fg = cv2.medianBlur(fgmask, 5)
        _, fg = cv2.threshold(fg, 200, 255, cv2.THRESH_BINARY)
        fg = cv2.morphologyEx(fg, cv2.MORPH_CLOSE, np.ones((5,5), np.uint8))

        people_centroids = []
        vehicle_centroids = []
        detections_out = []

        if detector and frame_idx % detect_every == 0:
            model, transform = detector
            img_t = transform(frame).to(device)
            with torch.no_grad():
                preds = model([img_t])[0]
            boxes = preds["boxes"].cpu().numpy()
            labels = preds["labels"].cpu().numpy()
            scores = preds["scores"].cpu().numpy()

            for box, label, score in zip(boxes, labels, scores):
                if score < 0.6:
                    continue
                x1,y1,x2,y2 = box.astype(int)
                cx = int((x1+x2)/2); cy = int((y1+y2)/2)
                lbl = COCO_LABELS.get(int(label), str(int(label)))
                detections_out.append((lbl, score, (x1,y1,x2,y2)))
                if int(label) == COCO_PERSON_ID:
                    people_centroids.append((cx, cy))
                if int(label) in COCO_VEHICLE_IDS:
                    vehicle_centroids.append((cx, cy))
        else:
            # fallback: use background segments and simple contours; detect people via HOG
            contours, _ = cv2.findContours(fg, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
            for cnt in contours:
                x,y,ww,hh = cv2.boundingRect(cnt)
                area = ww*hh
                if area < 200: continue
                cx = x+ww//2; cy = y+hh//2
                # classify by size heuristics
                if hh < 30:
                    people_centroids.append((cx, cy))
                    detections_out.append(("person", 0.5, (x,y,x+ww,y+hh)))
```

```
                else:
                    vehicle_centroids.append((cx, cy))
                    detections_out.append(("vehicle", 0.5, (x,y,x+ww,y+hh)))

        # HOG refine people (optional)
        if hog is not None:
            rects, weights = hog.detectMultiScale(frame, winStride=(8,8), padding=(8,8), scale=1.05)
            for (x,y,ww,hh), wgt in zip(rects, weights):
                if wgt < 0.6: continue
                cx = int(x+ww/2); cy = int(y+hh/2)
                people_centroids.append((cx, cy))
                detections_out.append(("person_hog", float(wgt), (x,y,x+ww,y+hh)))

    # update trackers
    persons = person_tracker.update(people_centroids, frame_idx)
    vehicles = vehicle_tracker.update(vehicle_centroids, frame_idx)

    alerts = []

    # detect loitering: person whose trace is within small radius for long time
    for pid, (centroid, last_seen, trace) in persons.items():
        if len(trace) >= int(fps*10):  # been tracked for >10s
            pts = np.array(trace)
            dmax = np.max(np.linalg.norm(pts - pts[0], axis=1))
            if dmax < 20:
                alerts.append({"type":"loitering","id":pid,"pos":centroid})
                # annotate
                cv2.putText(frame, f"LOITER-{pid}", (centroid[0]+5, centroid[1]-5),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,0,255), 2)

    # stationary vehicle detection -> possible illegal parking
    for vid, (centroid, last_seen, trace) in vehicles.items():
        if len(trace) >= int(fps*8):
            pts = np.array(trace)
            dmax = np.max(np.linalg.norm(pts - pts[0], axis=1))
            if dmax < 10:
                alerts.append({"type":"illegal_parking","id":vid,"pos":centroid})
                cv2.putText(frame, f"PARK-{vid}", (centroid[0]+5, centroid[1]-5),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,165,255), 2)
```

```
            # crowd detection: many people in small area
            if len(people_centroids) >= 6:
                # compute pairwise density
                pts = np.array(people_centroids)
                if pts.shape[0] >= 6:
                    # compute bounding box area
                    x_min, y_min = pts.min(axis=0)
                    x_max, y_max = pts.max(axis=0)
                    area = (x_max - x_min + 1) * (y_max - y_min + 1)
                    density = len(people_centroids) / max(area, 1)
                    if density > 0.002:  # heuristic density threshold
                        alerts.append({"type":"crowd", "count": len(people_centroids), "bbox":
[int(x_min),int(y_min),int(x_max),int(y_max)]})
                        cv2.rectangle(frame, (int(x_min),int(y_min)), (int(x_max),int(y_max)), (0,0,255), 2)
                        cv2.putText(frame, f"CROWD {len(people_centroids)}", (int(x_min), int(y_min)-8),
                                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,0,255), 2)

            # annotate detections
            for lbl, score, box in detections_out:
                x1,y1,x2,y2 = box
                color = (200,200,255) if "person" in lbl else (200,255,200)
                cv2.rectangle(frame, (x1,y1), (x2,y2), color, 1)
                cv2.putText(frame, f"{lbl}:{score:.2f}", (x1, max(0,y1-6)),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.4, color, 1)

            # annotate trackers
            for pid, (centroid, last_seen, trace) in persons.items():
                cv2.circle(frame, (int(centroid[0]), int(centroid[1])), 3, (255,0,0), -1)
                cv2.putText(frame, f"P{pid}", (int(centroid[0])+4, int(centroid[1])+4),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.4, (255,0,0), 1)
            for vid, (centroid, last_seen, trace) in vehicles.items():
                cv2.circle(frame, (int(centroid[0]), int(centroid[1])), 3, (0,255,0), -1)
                cv2.putText(frame, f"V{vid}", (int(centroid[0])+4, int(centroid[1])+4),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.4, (0,255,0), 1)

            # write frame
            out.write(frame)

            # logging
            ts = frame_idx / fps
```

```
            entry = {
                "frame": frame_idx,
                "time_s": round(ts,2),
                "people_count": len(people_centroids),
                "vehicle_count": len(vehicle_centroids),
                "alerts": alerts
            }
            logs.append(entry)

            # hotspot accumulation by simple grid
            for c in people_centroids:
                gx = int(c[0] / (w/10))
                gy = int(c[1] / (h/10))
                hotspots[(gx,gy)] += 1

            if frame_idx % int(fps*10) == 0:
                print(f"[{frame_idx}] people:{len(people_centroids)} vehicles:{len(vehicle_centroids)}
alerts:{len(alerts)} dt:{time.time()-t0:.2f}s")

    cap.release()
    out.release()

    # save logs
    df = pd.DataFrame([
        {"frame": e["frame"], "time_s": e["time_s"], "people_count": e["people_count"],
         "vehicle_count": e["vehicle_count"], "alerts": json.dumps(e["alerts"])}
        for e in logs
    ])
    df.to_csv(log_path, index=False)

    # summarize hotspots top-k
    hot = sorted(hotspots.items(), key=lambda x: -x[1])[:8]
    hot_zones = []
    for (gx,gy), cnt in hot:
        hot_zones.append({"grid": [int(gx),int(gy)], "count": int(cnt)})

    # generate simple report with actionable strategies
    report = {
        "video": os.path.basename(input_path),
        "processed_frames": frame_idx,
```

```
            "total_people_observations": int(sum([e["people_count"] for e in logs])),
            "total_vehicle_observations": int(sum([e["vehicle_count"] for e in logs])),
            "hotspots": hot_zones,
            "recommendations": [
                "Increase patrol presence and lighting in identified hotspots (grid cells with high people counts).",
                "Install time-limited parking restrictions and signage at bus_stop / market areas; enforce to reduce illegal parking.",
                "Deploy crowd management protocols (additional personnel, temporary barriers) during peak times identified by spikes in crowd alerts.",
                "Use real-time alerts to dispatch response teams for loitering and stationary vehicles, and integrate with local traffic enforcement.",
                "Complement camera analytics with anonymization (blurring faces) and data governance policies to protect privacy."
            ]
        }

        with open(report_path, "w") as f:
            json.dump(report, f, indent=2)

        print(f"Processing complete. Annotated video: {output_path}, logs: {log_path}, report: {report_path}")
        return output_path, log_path, report_path

# ------------------------
# CLI
# ------------------------
def main():
        p = argparse.ArgumentParser(description="Video analytics prototype for urban public safety.")
        p.add_argument("--simulate", action="store_true", help="Create a simulated video and process it.")
        p.add_argument("--input", type=str, help="Input video path.")
        p.add_argument("--out", type=str, default="annotated.mp4", help="Annotated output video path.")
        p.add_argument("--log", type=str, default="events.csv", help="CSV log output.")
        p.add_argument("--report", type=str, default="report.json", help="JSON report output.")
        p.add_argument("--device", type=str, default="cpu", help="torch device (cpu or cuda).")
        p.add_argument("--frames", type=int, default=600, help="Frames to simulate when --simulate.")
        args = p.parse_args()

        if args.simulate:
            sim_path = "simulated_input.mp4"
            print("Generating simulated video:", sim_path)
            simulate_video(path=sim_path, n_frames=args.frames)
```

```
            input_path = sim_path
        else:
            if not args.input:
                p.print_help()
                return
            input_path = args.input

        # try to use GPU if requested
        device = args.device
        if device != "cpu" and TORCH_AVAILABLE and torch.cuda.is_available():
            device = "cuda"
        else:
            device = "cpu"

        process_video(input_path, output_path=args.out, log_path=args.log, report_path=args.report,
device=device)

if __name__ == "__main__":
        main()
```