QUANTIC TECH ANALYSIS PVT LTD

A ROADMAP TO COMPUTER VISION

INTERNSHIP DOCUMENTATION

SUBMITTED BY:

ASHISH ARVIND M

DURATION

16 June 2025 to 9 July 2025

TABLE OF CONTENTS

S.NO	TITLE	PAGE
1	Ex 1. Overview on Object Detection	1
2	Ex 2. Overview on Object Tracking	3
3	Ex 3. Facial Recognition and Tracking	5
4	Ex 4. Person Detection (Entry Exit counting)	12
5	Ex 3. Hardhat detection (Safety Compliance)	15
6	Ex 6. Queue and Multithreading	17

EXERCISE 1: OVERVIEW ON OBJECT DETECTION

OBJECTIVES:

- To understand the core architecture of You Only Look Once (YOLOv8) model
- To learn the key network layers of YOLOv8
- Compare the model with other object detection models to understand shortcomings and improvements.

WHAT IS OBJECT DETECTION?

- Location and classification of objects in image or video is object detection
- Done through bounding boxes (location) and image labels (classification)

MODELS FOR OBJECT DETECTION:

- R-CNN
- Faster R-CNN
- Single Shot Detection (SSD)
- You Only Look Once (YOLO)
- R-CNN and Faster RCNN are two stage detection models while YOLO is based on single shot detection (SSD)

YOLOv8:

- 3 main layers
    - Backbone
    - Neck
    - Head

BACKBONE:

- Extracts feature from the input image using a series of convolutional layers.
- **Cross Stage Partial (CSP) Connection:** Neural network architecture design technique introduced to enhance learning efficiency and reduce computational cost while maintaining or improving accuracy.
- **SPPF (Spatial Pyramid Pooling Fast):** Captures multi-scale features and is an optimized version of the SPP module

NECK:

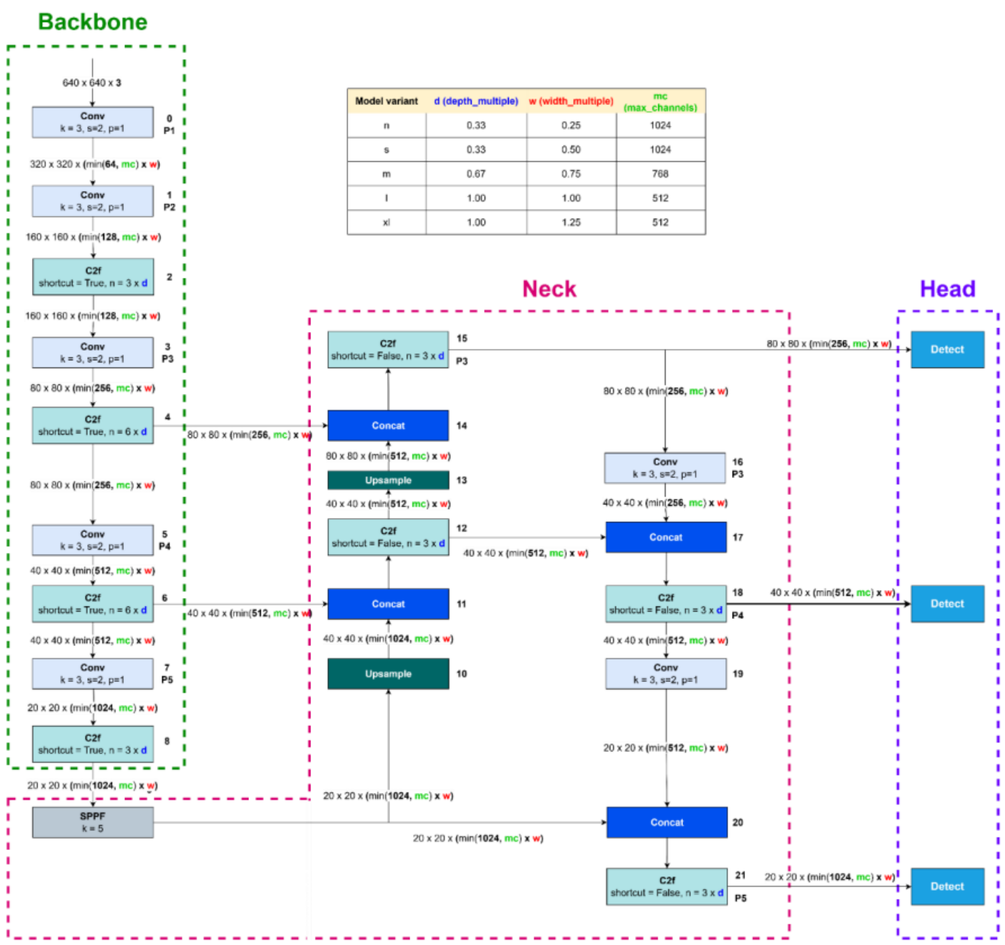
- Merges and processes features from different stages of the backbone to capture information at various scales.
- **C2f Module:** Combines high-level semantic features with low-level spatial information for improved detection of small objects
- **Feature Fusion Layers:** Integrate features from different backbone stages to enhance detection across object sizes

HEAD:

- Predicts bounding boxes, objectness scores, and class probabilities for objects in the image.
- **Detection Modules:** Multiple modules that predict large, medium and small objects separately.
- **Decoupled Head:** YOLOv8 uses a decoupled head, separating objectness and class prediction branches for improved accuracy

ADVANTAGES OVER OTHER MODELS:

- Single shot model, completes detection in one single pass through neural network.
- YOLO simultaneously predicts bounding boxes and class probabilities. Helps in observing global context instead of localized region detection. Reduces false positives significantly
- **Grid-based prediction:** Divides images into grids, each predicting multiple objects, improving small-object detection
- **Optimized backbones:** CSPDarknet and SPPF modules enhance feature extraction while reducing computational load
- As a result of all of this, YOLOv8 provides high, near real-time frame rates (45fps), useful for real-time applications



Backbone
640 x 640 x 3
Conv k = 3, s=2, p=1 0 P1
320 x 320 x (min(64, mc) x w)
Conv k = 3, s=2, p=1 1 P2
160 x 160 x (min(128, mc) x w)
C2f shortcut = True, n = 3 x d 2
160 x 160 x (min(128, mc) x w)
Conv k = 3, s=2, p=1 3 P3
80 x 80 x (min(256, mc) x w)
C2f shortcut = True, n = 6 x d 4
80 x 80 x (min(256, mc) x w)
Conv k = 3, s=2, p=1 5 P4
40 x 40 x (min(512, mc) x w)
C2f shortcut = True, n = 6 x d 6
40 x 40 x (min(512, mc) x w)
Conv k = 3, s=2, p=1 7 P5
20 x 20 x (min(1024, mc) x w)
C2f shortcut = True, n = 3 x d 8
20 x 20 x (min(1024, mc) x w)
SPPF k = 5
Neck
C2f shortcut = False, n = 3 x d 15 P3
Concat 14
80 x 80 x (min(512, mc) x w)
Upsample 13
40 x 40 x (min(512, mc) x w)
C2f shortcut = False, n = 3 x d 12
Concat 11
40 x 40 x (min(1024, mc) x w)
Upsample 10
20 x 20 x (min(1024, mc) x w)
80 x 80 x (min(256, mc) x w)
Conv k = 3, s=2, p=1 16 P3
40 x 40 x (min(256, mc) x w)
Concat 17
40 x 40 x (min(512, mc) x w)
C2f shortcut = False, n = 3 x d 18 P4
40 x 40 x (min(512, mc) x w)
Conv k = 3, s=2, p=1 19
20 x 20 x (min(512, mc) x w)
Concat 20
20 x 20 x (min(1024, mc) x w)
C2f shortcut = False, n = 3 x d 21 P5
Head
80 x 80 x (min(256, mc) x w)
Detect
40 x 40 x (min(512, mc) x w)
Detect
20 x 20 x (min(1024, mc) x w)
Detect

Model variant	d (depth_multiple)	w (width_multiple)	mc (max_channels)
n	0.33	0.25	1024
s	0.33	0.50	1024
m	0.67	0.75	768
l	1.00	1.00	512
xl	1.00	1.25	512

EXERCISE 2: OVERVIEW ON OBJECT TRACKING

OBJECTIVES:

- To get an overview of different tracker types
- To learn basics of Deep SORT
- To learn basics of BoT-SORT, ByteTrack

WHAT IS OBJECT TRACKING?

- Fundamental task in computer vision that involves detecting and following objects across video frames while maintaining consistent identities over time.
- Modern object tracking predominantly follows the tracking-by-detection concept.

KEY CONCEPTS:

- **Kalman filters** – Predict the future position of an object based on past positions.
  - Kalman filter tracks an object's position and speed
  - Predicts object position in next frame, even after brief occlusion
  - Updates the prediction with actual new detection when it reappears
- **Hungarian algorithm** - method to find the best match between objects detected in the current frame and tracked objects from previous frames, using cost matrices to find the least cost based on distance
- The Kalman Filter predicts where each person should be.
- New detections come in from the frame. The Hungarian Algorithm matches each new detection to the predicted tracks.
- Kalman Filters then update their predictions with the new matched detections

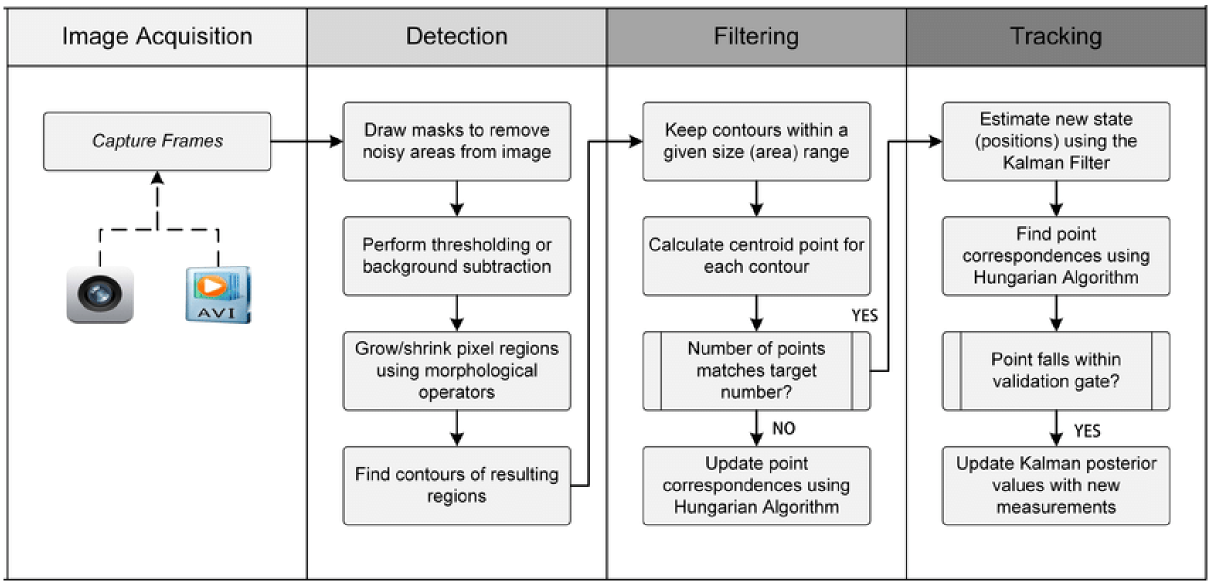


Image Acquisition
Detection
Filtering
Tracking
Capture Frames
AVI
Draw masks to remove noisy areas from image
Perform thresholding or background subtraction
Grow/shrink pixel regions using morphological operators
Find contours of resulting regions
Keep contours within a given size (area) range
Calculate centroid point for each contour
Number of points matches target number?
YES
NO
Update point correspondences using Hungarian Algorithm
Estimate new state (positions) using the Kalman Filter
Find point correspondences using Hungarian Algorithm
Point falls within validation gate?
YES
Update Kalman posterior values with new measurements

MODELS FOR OBJECT TRACKING:

- Centroid tracker
- Deep SORT
- BoT-SORT
- ByteTrack

Centroid tracker:

- Introduction of deep learning networks to extract appearance descriptors, for visual characteristics
- Useful for re-identification
- Combines this with motion similarity calculated from IoUs

Deep SORT:

- Introduction of deep learning networks (CNN) to extract **appearance descriptors**, for visual characteristics. Useful for re-identification
- Combines this with motion similarity calculated from IoUs and cosine distance metrics.
- Matched tracks are updated with new detections, while unmatched detections initialize new tracks.

ByteTrack:

- A tracking by detection method, which also includes **low confidence** detections along with the usual high confidence detections
- Firstly, the high confidence detections are matched with the tracks using IoU and Hungarian algorithm
- Unmatched tracks are once again tried to be matched with the low confidence detections. Therefore, it is a **two-pass** method

BoT-SORT:

- A souped-up ByteTrack. Includes appearance-based reID and motion modelling using Kalman tuning.
- This also gets the low confidence detection feature from ByteTrack.
- More robust in occluded cases, thanks to reID and Kalman motion modelling
- But speed is slower (reID feature extraction and Kalman model)

EXERCISE 3: FACIAL RECOGNITION AND TRACKING

Objectives:

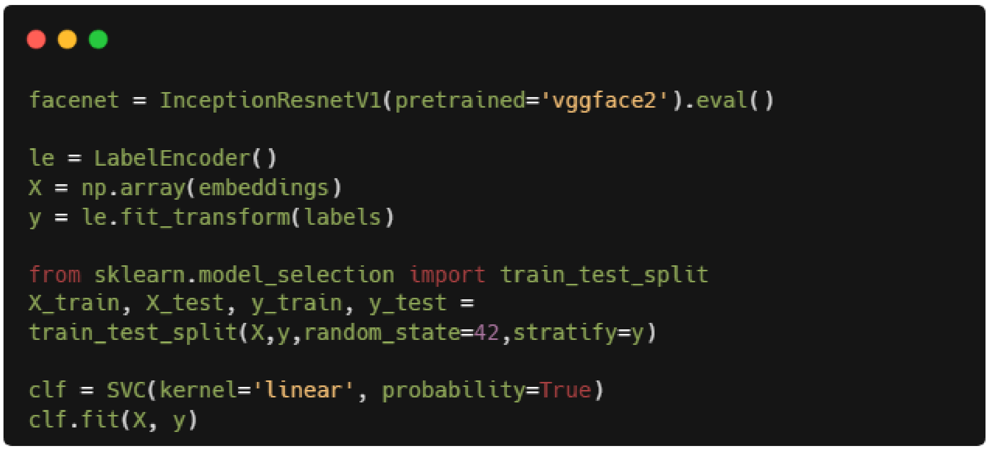
- Load webcam input and detect faces using YOLO
- Track faces using Deep SORT and Centroid Tracker
- Perform face recognition using FaceNet or PyTorch models

1. FACE DETECTION AND RECOGNITION

Libraries used:

- OpenCV
- Ultralytics YOLOv8
- Face-Net

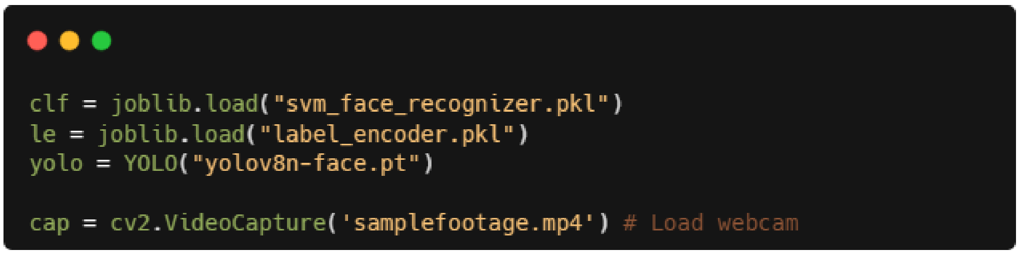
Step 1: Get a dataset of faces and get the **Face Net** embeddings, which can be used to train a model like **SVM** or **K-means classifier**. In this case, **SVM** has been used.



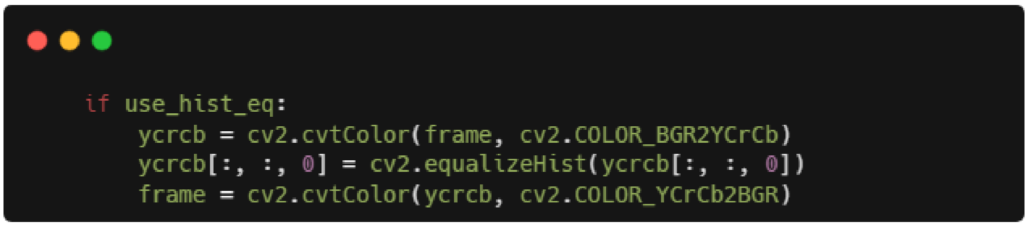
```
facenet = InceptionResnetV1(pretrained='vggface2').eval()

le = LabelEncoder()
X = np.array(embeddings)
y = le.fit_transform(labels)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X,y,random_state=42,stratify=y)

clf = SVC(kernel='linear', probability=True)
clf.fit(X, y)
```

Step 2: Load the saved models along with a YOLO detector for recognition.

Step 3: Apply **histogram equalization** for image enhancement. The code given below includes a toggle to switch between normal frame and histogram equalized frame

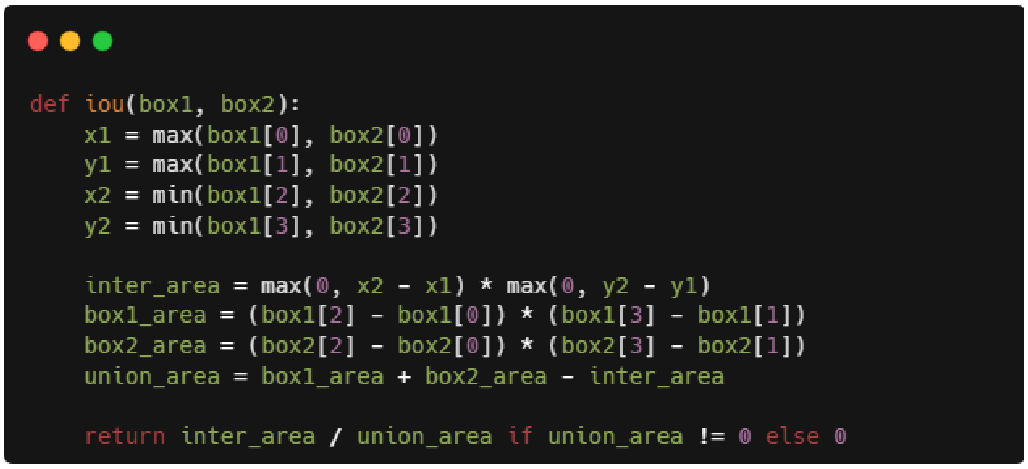


```
clf = joblib.load("svm_face_recognizer.pkl")
le = joblib.load("label_encoder.pkl")
yolo = YOLO("yolov8n-face.pt")

cap = cv2.VideoCapture('samplefootage.mp4') # Load webcam
```

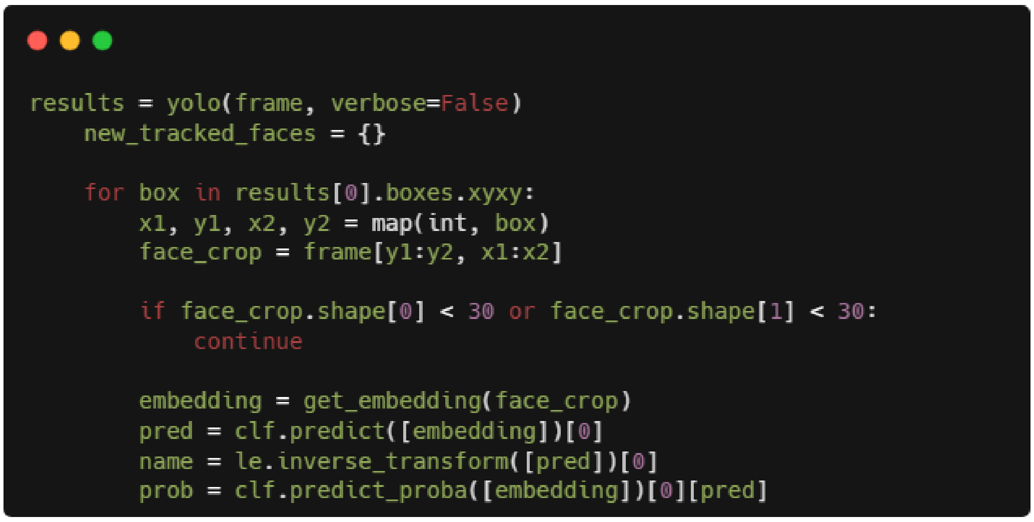


```
    if use_hist_eq:
        ycrcb = cv2.cvtColor(frame, cv2.COLOR_BGR2YCrCb)
        ycrcb[:, :, 0] = cv2.equalizeHist(ycrcb[:, :, 0])
        frame = cv2.cvtColor(ycrcb, cv2.COLOR_YCrCb2BGR)
```

Step 4: Use **Intersection over Union (IoU)** to implement a simple tracking logic to avoid flickering and disappearing bounding boxes.

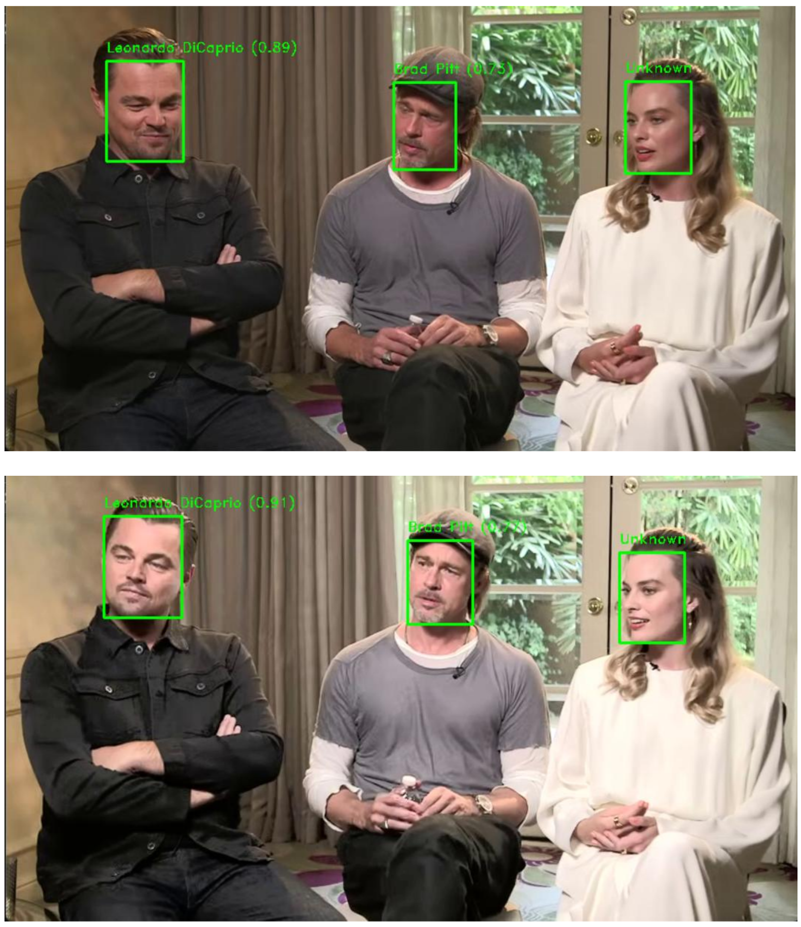


```
def iou(box1, box2):
    x1 = max(box1[0], box2[0])
    y1 = max(box1[1], box2[1])
    x2 = min(box1[2], box2[2])
    y2 = min(box1[3], box2[3])

    inter_area = max(0, x2 - x1) * max(0, y2 - y1)
    box1_area = (box1[2] - box1[0]) * (box1[3] - box1[1])
    box2_area = (box2[2] - box2[0]) * (box2[3] - box2[1])
    union_area = box1_area + box2_area - inter_area

    return inter_area / union_area if union_area != 0 else 0
```



```
results = yolo(frame, verbose=False)
    new_tracked_faces = {}

    for box in results[0].boxes.xyxy:
        x1, y1, x2, y2 = map(int, box)
        face_crop = frame[y1:y2, x1:x2]

        if face_crop.shape[0] < 30 or face_crop.shape[1] < 30:
            continue

        embedding = get_embedding(face_crop)
        pred = clf.predict([embedding])[0]
        name = le.inverse_transform([pred])[0]
        prob = clf.predict_proba([embedding])[0][pred]
```

- The output of this code, tested on a sample video, gives us recognition and tracking with flicker free bounding boxes.
- The class and confidence are also displayed on the bounding box.
- A toggle **'h'** has been set to switch to a histogram-equalized view in real time.



Histogram equalization OFF vs ON

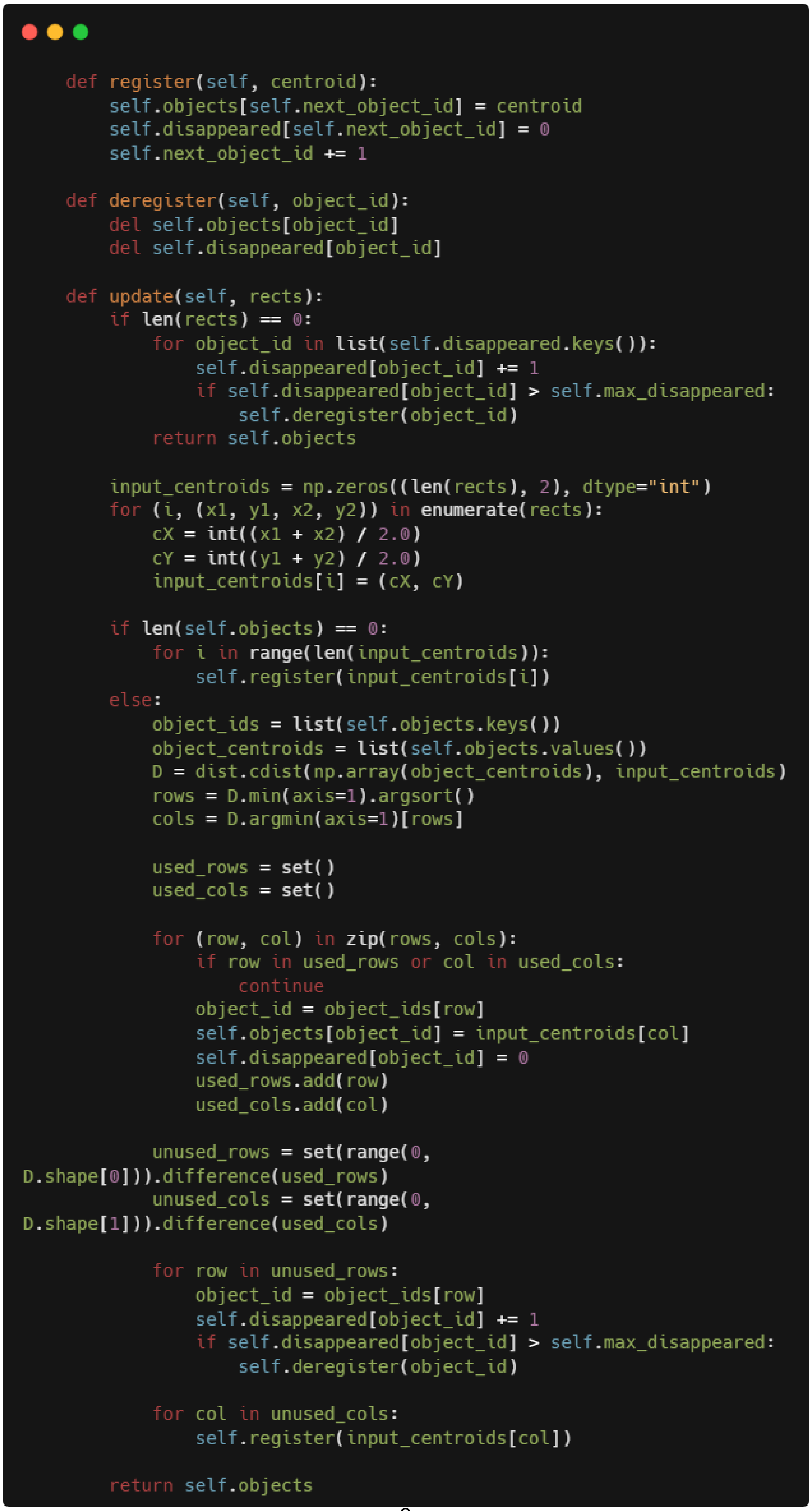
2. FACE TRACKING USING CENTROID TRACKER AND DEEP SORT

Libraries used:

- OpenCV
- Deep SORT Realtime

Step 1: Define the centroid tracker logic as a function first in **centroid_tracker.py**

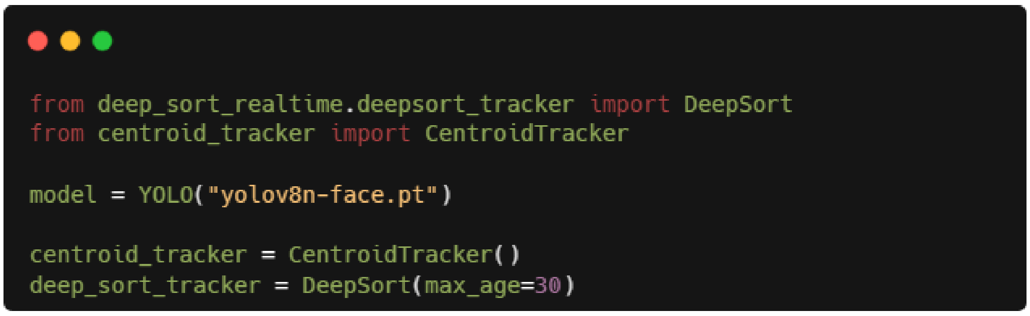
```
from scipy.spatial import distance as dist
from collections import OrderedDict
import numpy as np

class CentroidTracker:
    def __init__(self, max_disappeared=20):
        self.next_object_id = 0
        self.objects = OrderedDict()
        self.disappeared = OrderedDict()
        self.max_disappeared = max_disappeared
```



```
    def register(self, centroid):
        self.objects[self.next_object_id] = centroid
        self.disappeared[self.next_object_id] = 0
        self.next_object_id += 1

    def deregister(self, object_id):
        del self.objects[object_id]
        del self.disappeared[object_id]

    def update(self, rects):
        if len(rects) == 0:
            for object_id in list(self.disappeared.keys()):
                self.disappeared[object_id] += 1
                if self.disappeared[object_id] > self.max_disappeared:
                    self.deregister(object_id)
            return self.objects

        input_centroids = np.zeros((len(rects), 2), dtype="int")
        for (i, (x1, y1, x2, y2)) in enumerate(rects):
            cX = int((x1 + x2) / 2.0)
            cY = int((y1 + y2) / 2.0)
            input_centroids[i] = (cX, cY)

        if len(self.objects) == 0:
            for i in range(len(input_centroids)):
                self.register(input_centroids[i])
        else:
            object_ids = list(self.objects.keys())
            object_centroids = list(self.objects.values())
            D = dist.cdist(np.array(object_centroids), input_centroids)
            rows = D.min(axis=1).argsort()
            cols = D.argmin(axis=1)[rows]

            used_rows = set()
            used_cols = set()

            for (row, col) in zip(rows, cols):
                if row in used_rows or col in used_cols:
                    continue
                object_id = object_ids[row]
                self.objects[object_id] = input_centroids[col]
                self.disappeared[object_id] = 0
                used_rows.add(row)
                used_cols.add(col)

            unused_rows = set(range(0,
D.shape[0])).difference(used_rows)
            unused_cols = set(range(0,
D.shape[1])).difference(used_cols)

            for row in unused_rows:
                object_id = object_ids[row]
                self.disappeared[object_id] += 1
                if self.disappeared[object_id] > self.max_disappeared:
                    self.deregister(object_id)

            for col in unused_cols:
                self.register(input_centroids[col])

        return self.objects
```

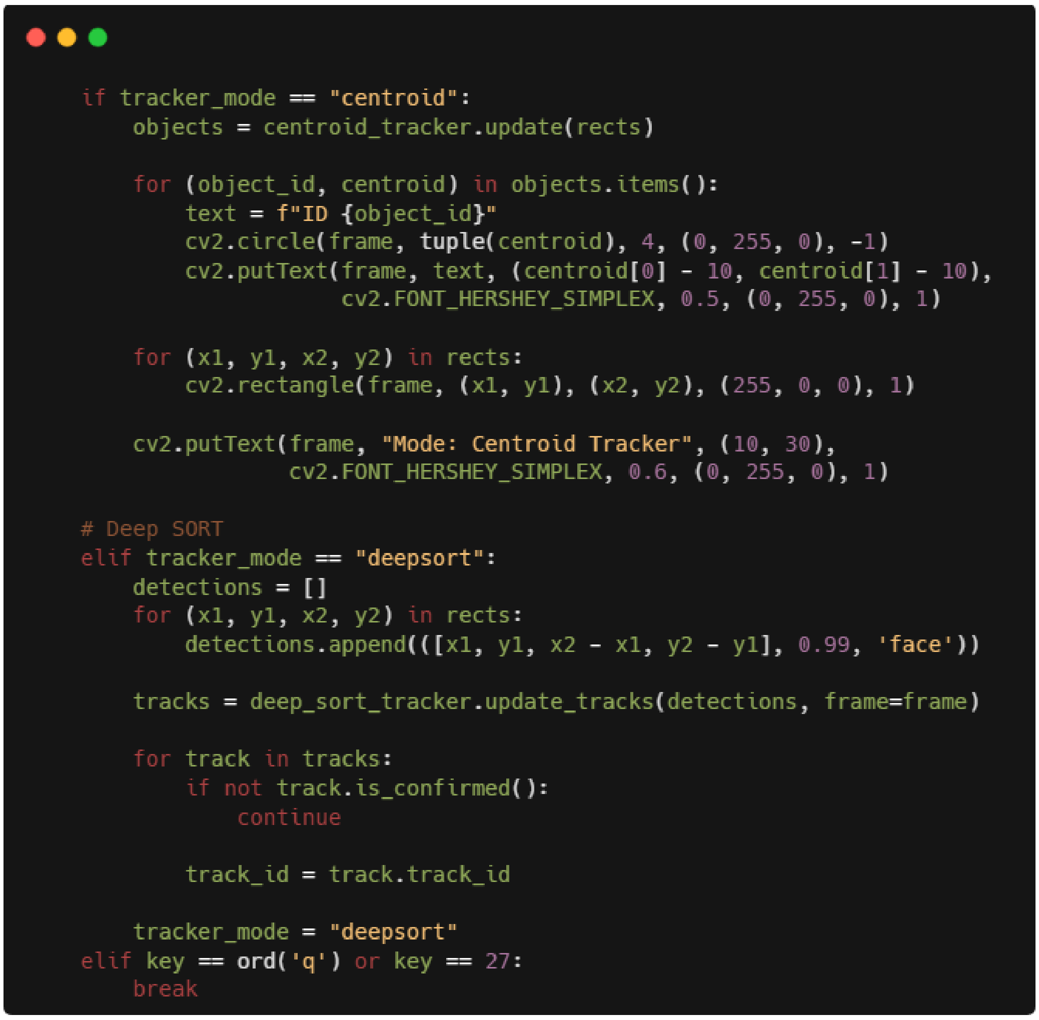
Step 2: Import the Deep SORT function from the ***deep_sort_realtime*** package.

Step 3: Import the Centroid Tracker function in the combined code file as follows.

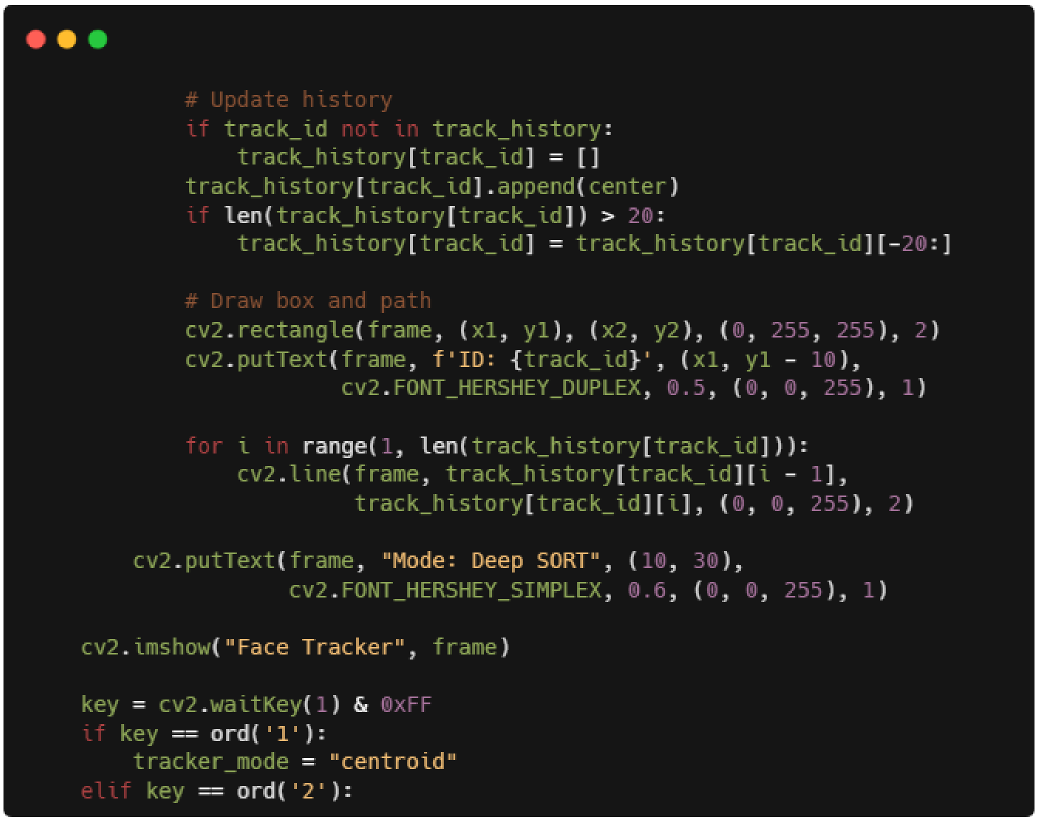


```
from deep_sort_realtime.deepsort_tracker import DeepSort
from centroid_tracker import CentroidTracker

model = YOLO("yolov8n-face.pt")

centroid_tracker = CentroidTracker()
deep_sort_tracker = DeepSort(max_age=30)
```

The below code has a key toggle to switch between the centroid tracker and Deep SORT in real time (1 for centroid tracker, 2 for Deep SORT)

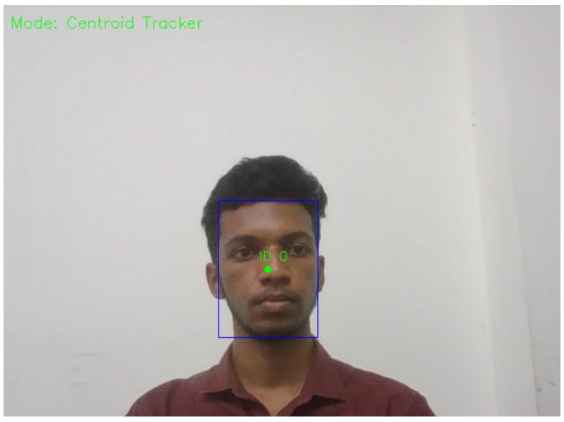


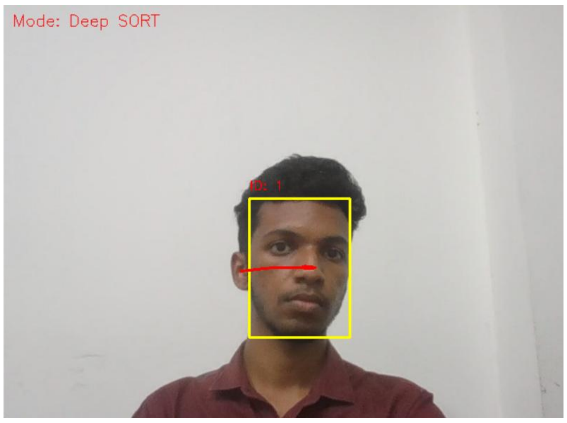
```
    if tracker_mode == "centroid":
        objects = centroid_tracker.update(rects)

        for (object_id, centroid) in objects.items():
            text = f"ID {object_id}"
            cv2.circle(frame, tuple(centroid), 4, (0, 255, 0), -1)
            cv2.putText(frame, text, (centroid[0] - 10, centroid[1] - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)

        for (x1, y1, x2, y2) in rects:
            cv2.rectangle(frame, (x1, y1), (x2, y2), (255, 0, 0), 1)

        cv2.putText(frame, "Mode: Centroid Tracker", (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0), 1)

    # Deep SORT
    elif tracker_mode == "deepsort":
        detections = []
        for (x1, y1, x2, y2) in rects:
            detections.append(([x1, y1, x2 - x1, y2 - y1], 0.99, 'face'))

        tracks = deep_sort_tracker.update_tracks(detections, frame=frame)

        for track in tracks:
            if not track.is_confirmed():
                continue

            track_id = track.track_id

        tracker_mode = "deepsort"
    elif key == ord('q') or key == 27:
        break
```



```
            # Update history
            if track_id not in track_history:
                track_history[track_id] = []
            track_history[track_id].append(center)
            if len(track_history[track_id]) > 20:
                track_history[track_id] = track_history[track_id][-20:]

            # Draw box and path
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 255), 2)
            cv2.putText(frame, f'ID: {track_id}', (x1, y1 - 10),
                        cv2.FONT_HERSHEY_DUPLEX, 0.5, (0, 0, 255), 1)

            for i in range(1, len(track_history[track_id])):
                cv2.line(frame, track_history[track_id][i - 1],
                         track_history[track_id][i], (0, 0, 255), 2)

        cv2.putText(frame, "Mode: Deep SORT", (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 1)

    cv2.imshow("Face Tracker", frame)

    key = cv2.waitKey(1) & 0xFF
    if key == ord('1'):
        tracker_mode = "centroid"
    elif key == ord('2'):
```

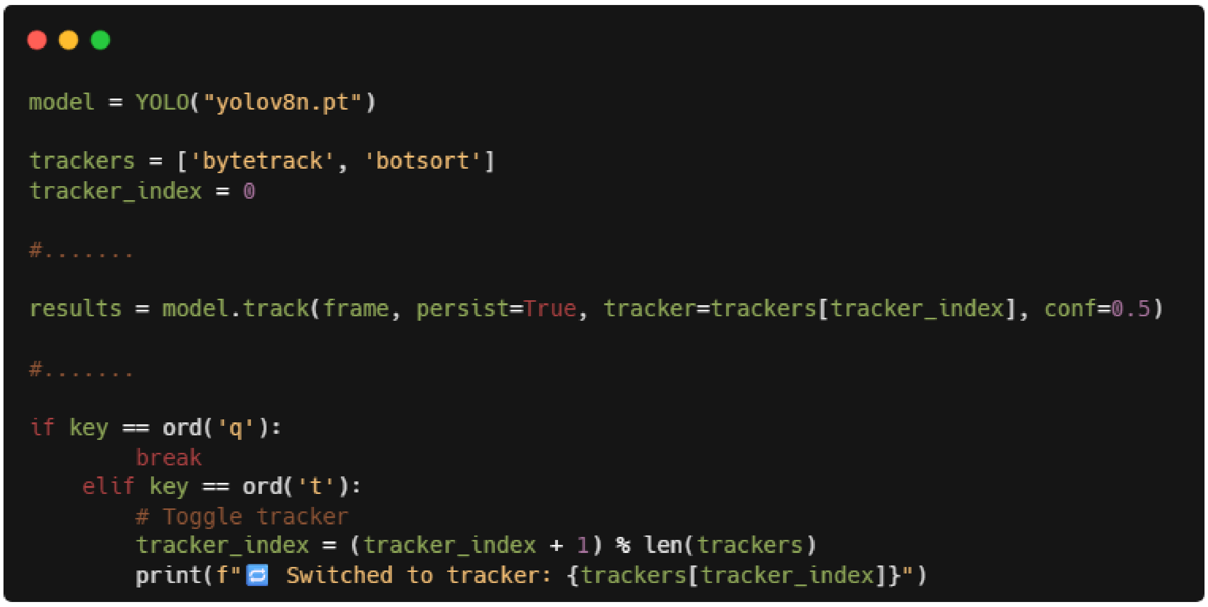
OUTPUT





3. FACE TRACKING USING BoT-SORT AND BYTETRACK

- Although these algorithms haven't been implemented in this duration due to challenges faced during setup, provided below is the simple code for implementation of both BoT-SORT and ByteTrack.
- The rest of the logic is same as the Deep SORT and Centroid tracker implementation



```python
model = YOLO("yolov8n.pt")

trackers = ['bytetrack', 'botsort']
tracker_index = 0

#.......

results = model.track(frame, persist=True, tracker=trackers[tracker_index], conf=0.5)

#.......

if key == ord('q'):
        break
    elif key == ord('t'):
        # Toggle tracker
        tracker_index = (tracker_index + 1) % len(trackers)
        print(f"🔄 Switched to tracker: {trackers[tracker_index]}")
```

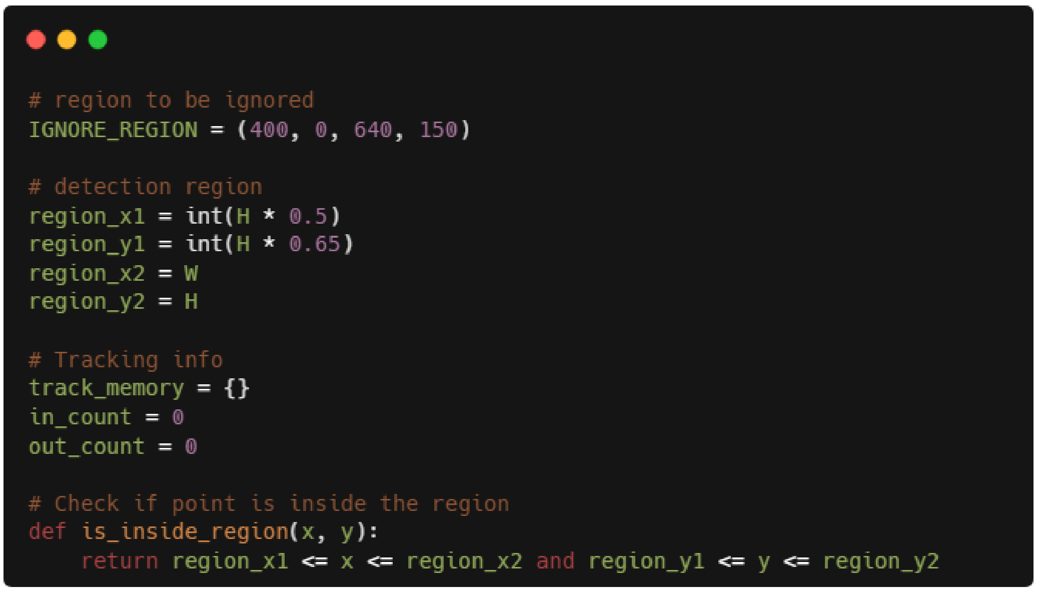
EXERCISE 4: PERSON DETECTION (ENTRY EXIT COUNTING)

OBJECTIVEs:

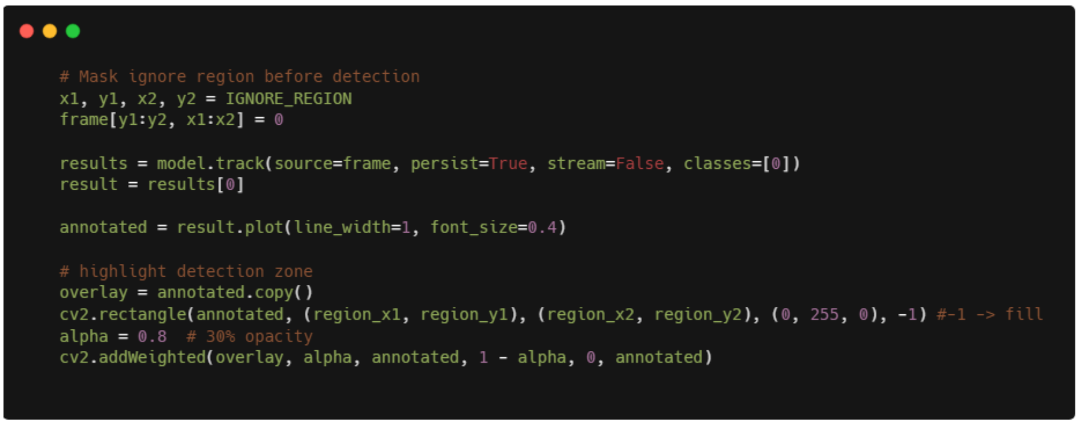
- Load video footage (e.g. Entrance camera)
- Detect and count people moving in and out

ENTRY EXIT DETECTION USING INBUILT yolo.track()

- We utilized the **yolov8l.pt** model for enhanced person detection accuracy.
- For tracking, we employed the **model.track() function** provided by the YOLO framework, which uses an integrated implementation of the **ByteTrack algorithm**. This delivered more consistent and reliable results compared to Deep SORT.
- **Test videos** depicting entry and exit scenarios were **recorded in-house** to evaluate the system's performance
- **Entry and exit counts** were tracked by defining a specific detection zone, and incrementing the respective count whenever a detected person's **centroid crossed** the predefined region.
- As the chosen environment had a lot of reflective surfaces, it resulted in reflections being assigned rogue IDs. As a solution, the problematic zone was blacked out and ignored during person detection

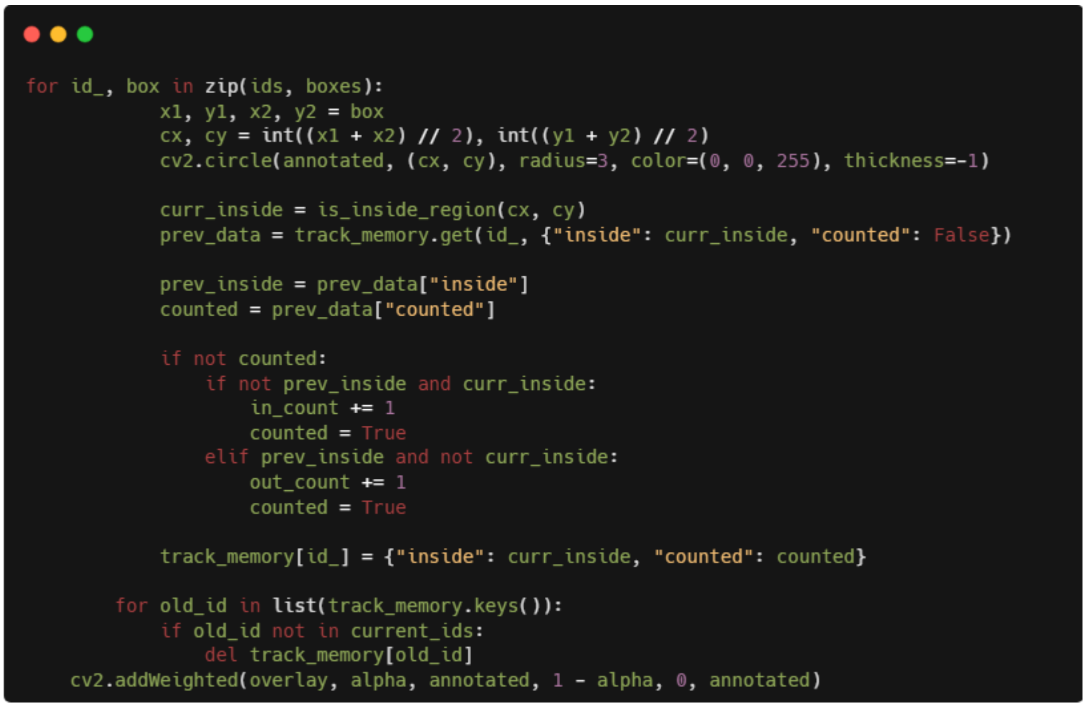


```python
# region to be ignored
IGNORE_REGION = (400, 0, 640, 150)

# detection region
region_x1 = int(H * 0.5)
region_y1 = int(H * 0.65)
region_x2 = W
region_y2 = H

# Tracking info
track_memory = {}
in_count = 0
out_count = 0

# Check if point is inside the region
def is_inside_region(x, y):
    return region_x1 <= x <= region_x2 and region_y1 <= y <= region_y2
```



```python
# Mask ignore region before detection
x1, y1, x2, y2 = IGNORE_REGION
frame[y1:y2, x1:x2] = 0

results = model.track(source=frame, persist=True, stream=False, classes=[0])
result = results[0]

annotated = result.plot(line_width=1, font_size=0.4)

# highlight detection zone
overlay = annotated.copy()
cv2.rectangle(annotated, (region_x1, region_y1), (region_x2, region_y2), (0, 255, 0), -1) #-1 -> fill
alpha = 0.8  # 30% opacity
cv2.addWeighted(overlay, alpha, annotated, 1 - alpha, 0, annotated)
```

<u>Model logic</u>

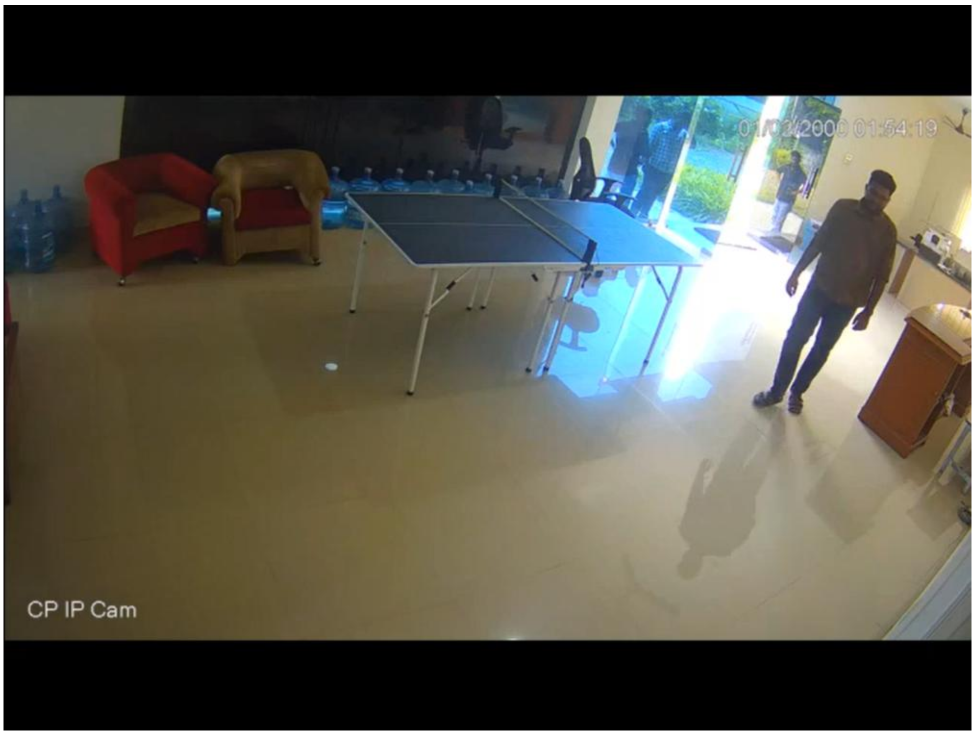


```python
for id_, box in zip(ids, boxes):
            x1, y1, x2, y2 = box
            cx, cy = int((x1 + x2) // 2), int((y1 + y2) // 2)
            cv2.circle(annotated, (cx, cy), radius=3, color=(0, 0, 255), thickness=-1)

            curr_inside = is_inside_region(cx, cy)
            prev_data = track_memory.get(id_, {"inside": curr_inside, "counted": False})

            prev_inside = prev_data["inside"]
            counted = prev_data["counted"]

            if not counted:
                if not prev_inside and curr_inside:
                    in_count += 1
                    counted = True
                elif prev_inside and not curr_inside:
                    out_count += 1
                    counted = True

            track_memory[id_] = {"inside": curr_inside, "counted": counted}

        for old_id in list(track_memory.keys()):
            if old_id not in current_ids:
                del track_memory[old_id]
    cv2.addWeighted(overlay, alpha, annotated, 1 - alpha, 0, annotated)
```

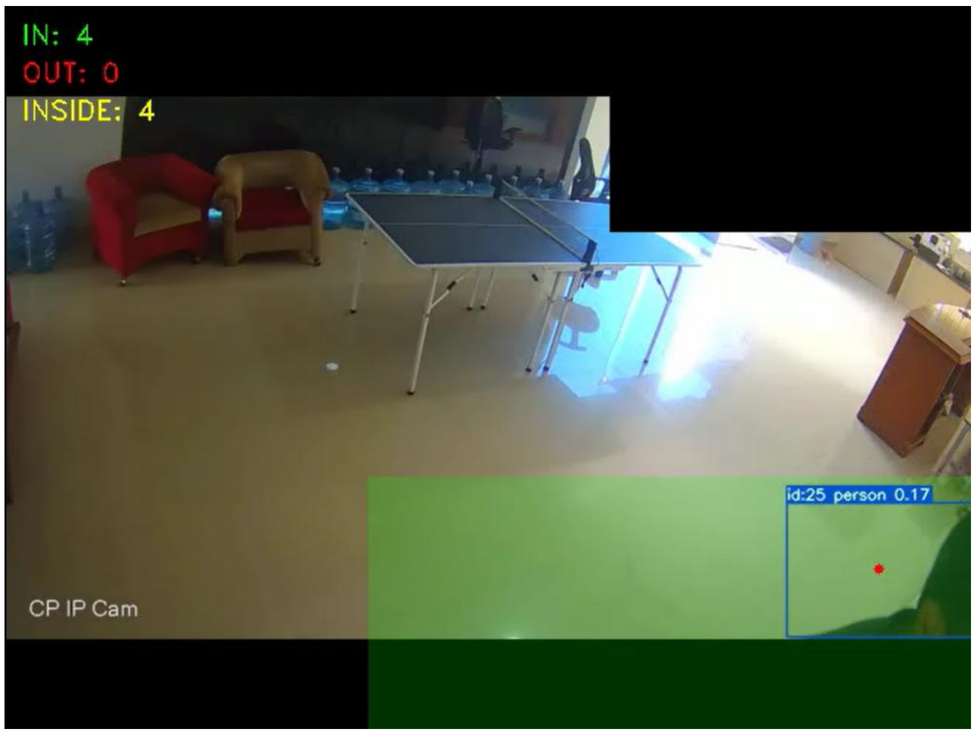
CHALLENGES FACED:

- Reflections assigned rogue IDs. Fixed by defining an IGNORE region (blacked out in output)
- Same IDs triggering multiple in and outs, leading to inaccurate counts. Fixed by defining a set of IDs, so that one ID can trigger only one in or out.

INPUT VIDEO



OUTPUT VIDEO

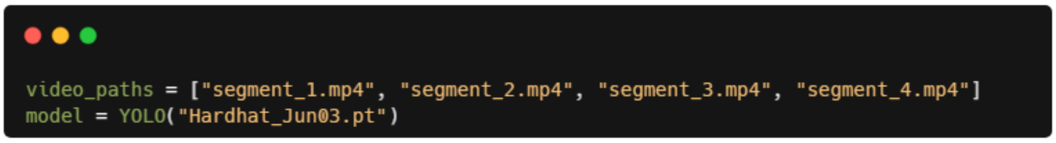


EXERCISE 5: HARDHAT DETECTION (SAFETY COMPLIANCE)

OBJECTIVES:

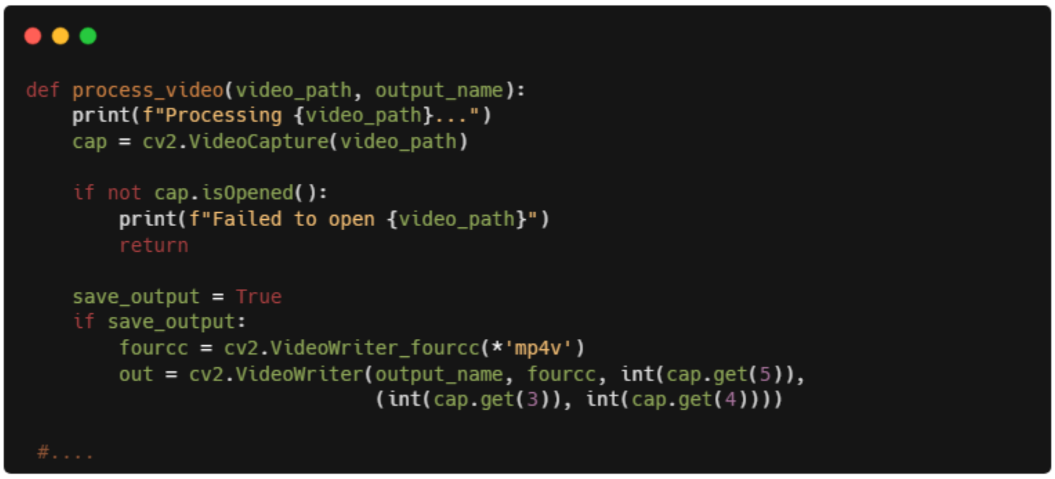
- Detect presence or absence of helmets

- A proprietary YOLO-based object detection model, **Hardhat_Jun03.pt**, which was trained in-house, was used in this task
- As part of performance optimization, multithreading was implemented to process four separate video streams concurrently for hardhat detection. This parallel processing approach aims to improve throughput and leverage available system resources more effectively.
- Results indicate that the multithreaded solution, while not outperforming a single instance of detection in terms of speed, is notably more efficient than sequentially processing each video independently.

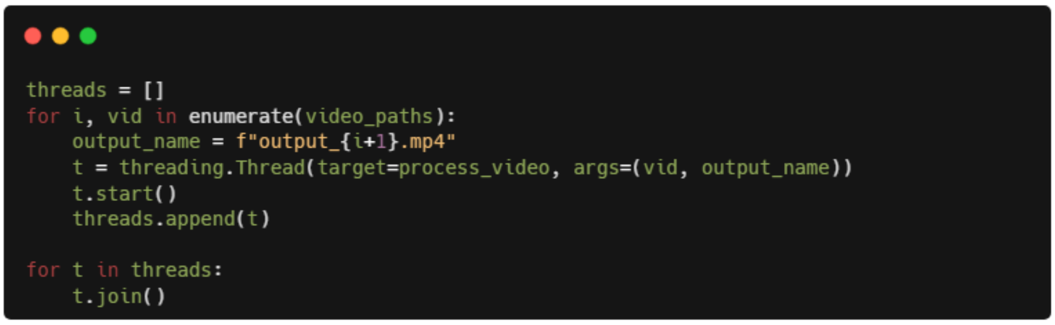


```
video_paths = ["segment_1.mp4", "segment_2.mp4", "segment_3.mp4", "segment_4.mp4"]
model = YOLO("Hardhat_Jun03.pt")
```

- The whole detection function is defined as ***process_video.*** The threading logic is shown as follows



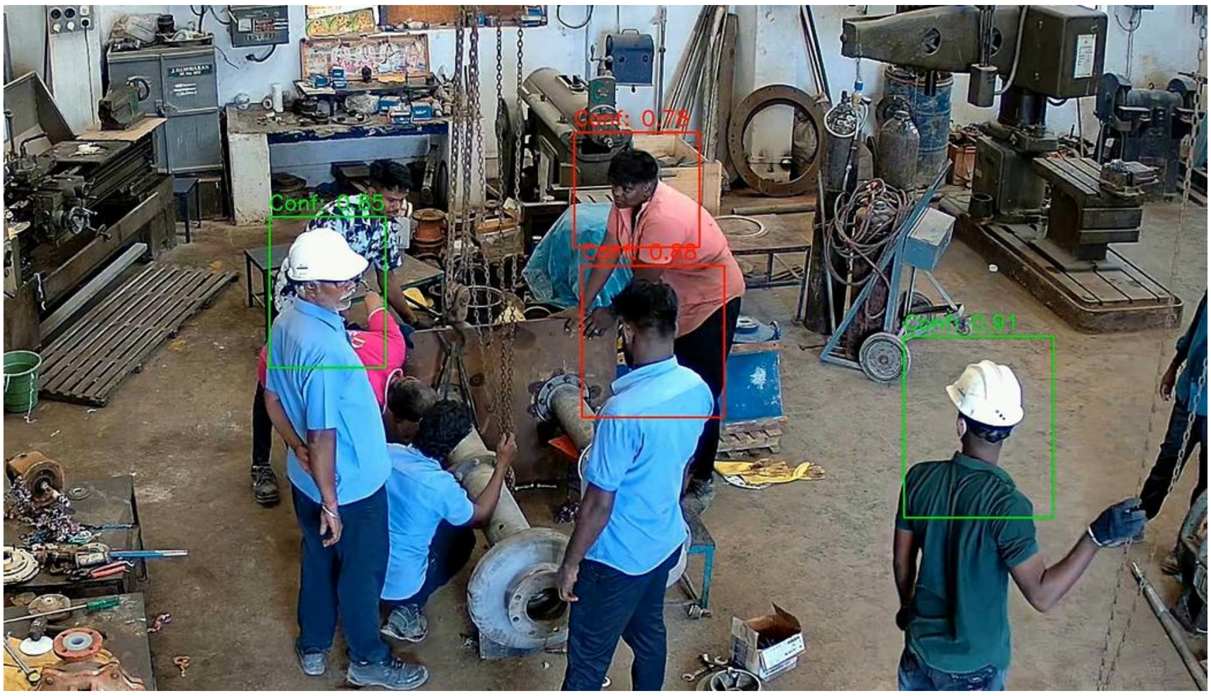
```
def process_video(video_path, output_name):
    print(f"Processing {video_path}...")
    cap = cv2.VideoCapture(video_path)

    if not cap.isOpened():
        print(f"Failed to open {video_path}")
        return

    save_output = True
    if save_output:
        fourcc = cv2.VideoWriter_fourcc(*'mp4v')
        out = cv2.VideoWriter(output_name, fourcc, int(cap.get(5)),
                              (int(cap.get(3)), int(cap.get(4))))

 #....
```



```
threads = []
for i, vid in enumerate(video_paths):
    output_name = f"output_{i+1}.mp4"
    t = threading.Thread(target=process_video, args=(vid, output_name))
    t.start()
    threads.append(t)

for t in threads:
    t.join()
```

INPUT VIDEO

OUTPUT VIDEO



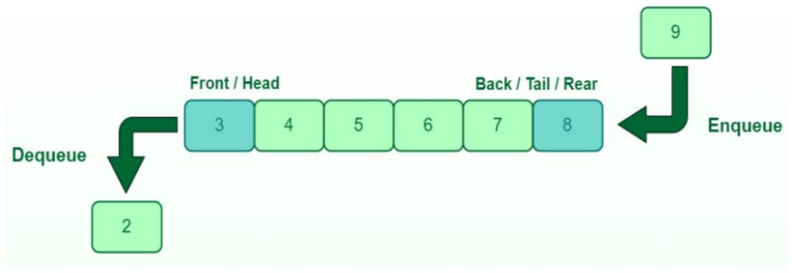
EXERCISE 6: QUEUE AND MULTITHREADING

UNDERSTANDING QUEUES:

- A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, where the first element added is the first one to be removed.
- Python offers three main queue types, each serving different use cases
  - **FIFO Queue**: The standard queue where elements are processed in the order they were added. This is ideal for most applications where fairness and order preservation are important.
  - **LIFO Queue**: A Last-In-First-Out queue that operates like a stack. The most recently added element is the first to be removed.
  - **Priority Queue**: Elements are kept sorted and the lowest-valued item is retrieved first. This is useful when certain tasks need higher priority processing.

BASIC QUEUE OPERATIONS:

- put(item)
- get()
- qsize()
- empty()
- full()

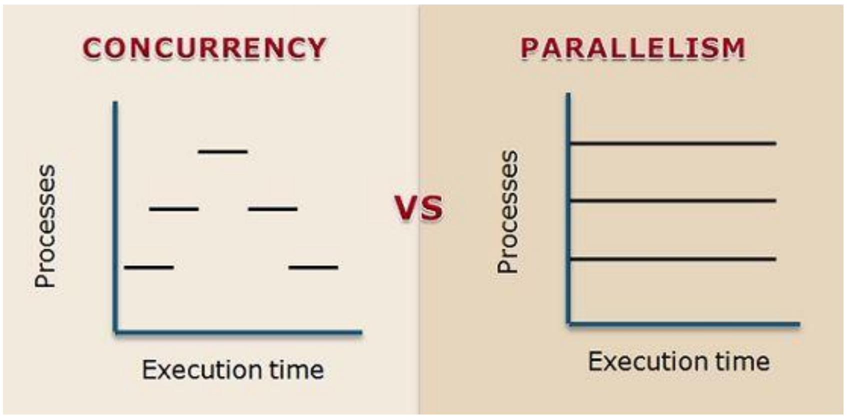


QUEUE IN COMPUTER VISION:

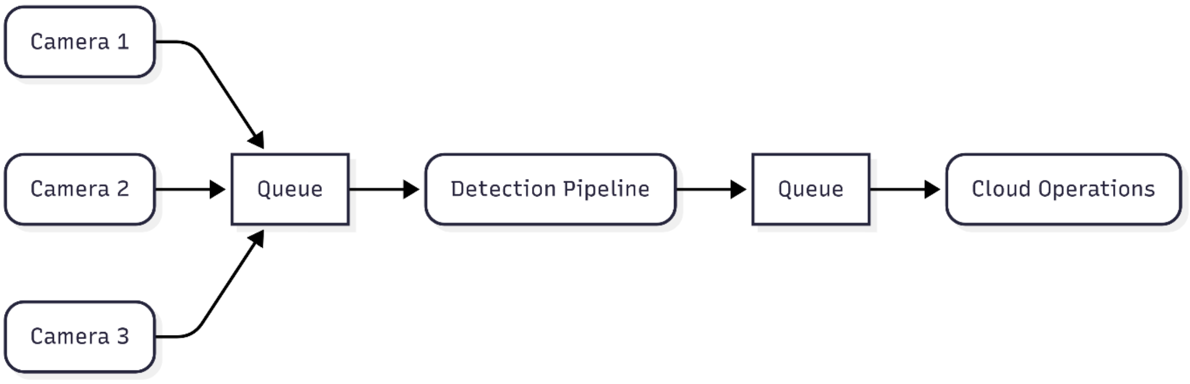
- **Input Queue Management:** Raw video frames from CCTV cameras are initially placed into input queues, where they await processing by computer vision algorithms. Each camera stream maintains its dedicated queue to prevent cross-contamination of data.
- **Processing Queue Distribution**: Video frames are distributed across multiple processing workers through message queues, enabling concurrent analysis of safety violations. This approach allows systems to scale the processing capacity based on demand.
- **Result Aggregation Queues**: Processed results, including detected violations and alerts, are collected through output queues before being forwarded to monitoring systems or databases

MULTITHREADING IN PYTHON:

- Multithreading allows multiple threads to execute concurrently within a single process, sharing the same memory space.
- The **Global Interpreter Lock (GIL)** is a mutex that prevents multiple threads from executing Python bytecode simultaneously in CPython
- While Python's Global Interpreter Lock (GIL) limits true parallelism for CPU-bound tasks, threading excels for I/O-bound operations
- Threading is ideal for
  - Tasks that spend time waiting for other resources
  - Lightweight concurrency
  - Shared memory requirement
- The difference between multithreading in Python and other languages is **concurrency** vs **parallelism**
- In Python, while multiple tasks are making progress at the same time, they aren't necessarily running at the same time
- But in other languages, multithreading allows multiple tasks to run at the same time i.e. **in parallel**.
-



VISUALISATION OF REAL TIME APPLICATION PIPELINE


Camera 1
Camera 2
Camera 3
Queue
Detection Pipeline
Queue
Cloud Operations