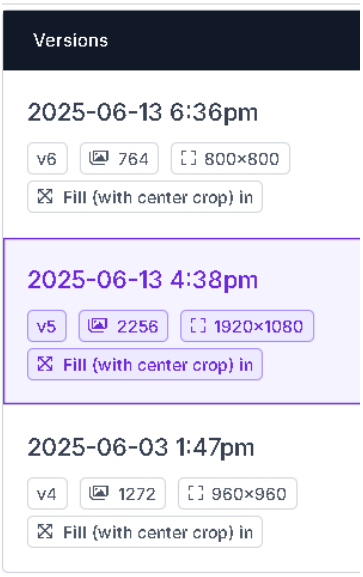
🍕 Pizza Store Hygiene Monitoring System

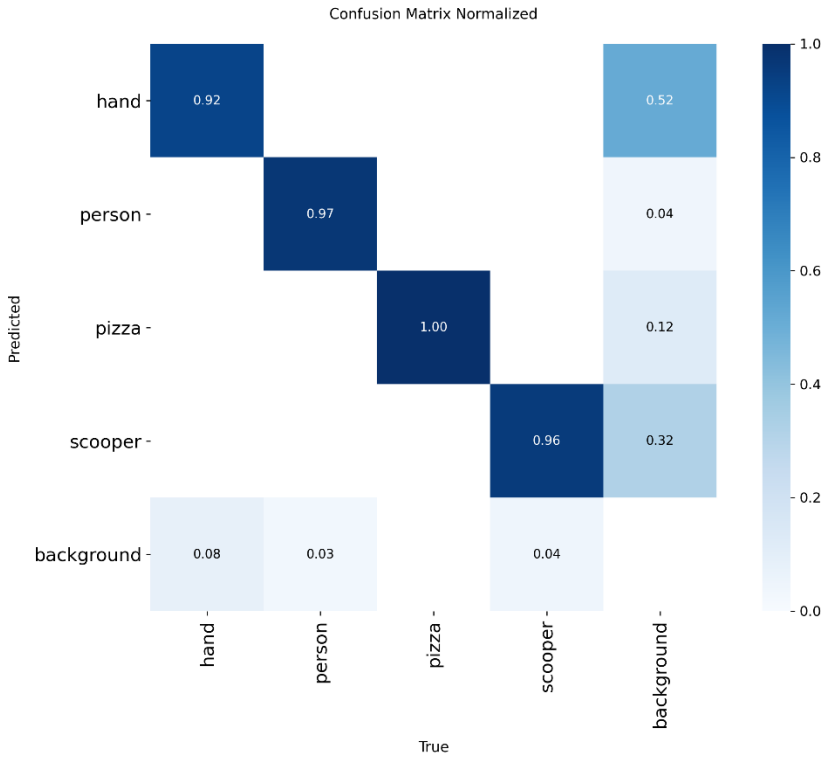
🎯 Project Objectives

- Monitor food safety compliance in real-time
- Detect violations when workers handle ingredients without proper tools
- Provide visual alerts and statistics on compliance
- Create a scalable system that can handle multiple workers simultaneously

1- Fine-Tuning

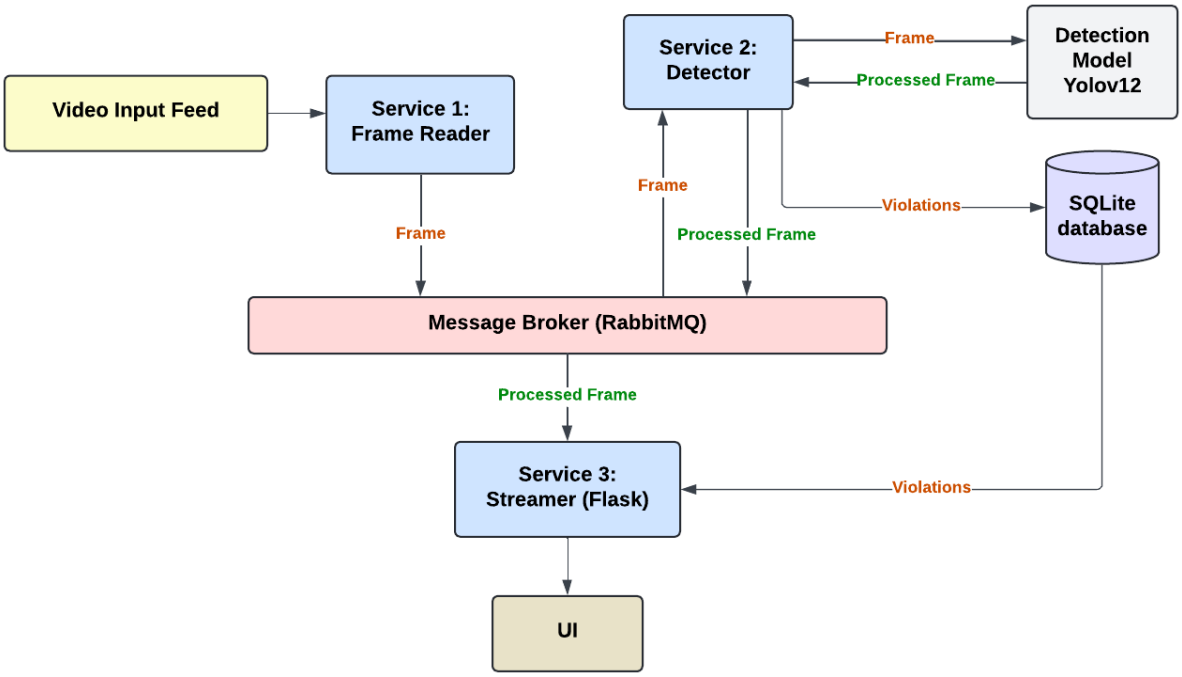
To enhance the model's performance in detecting hands, scoopers, pizzas, and workers more accurately within our specific pizza store environment, I fine-tuned the pretrained YOLOv12 weights on **another version of the provided dataset**.
This version of the dataset was derived from the original, augmented with various transformations to improve generalization and robustness.
This is the new model after fine-tuning: **best.pt**





2- Frame Reader Service

This project is architected using a **microservices** approach where each component is isolated by responsibility and communicates via **RabbitMQ** messaging queues.



The **Frame Reader Service** is responsible for reading video frames from a source (e.g., a file or camera) and sending them to the raw_frames RabbitMQ queue for further processing by the detection microservice.

🔍 **Main Responsibilities:**

- Reads video frames using OpenCV
- Encodes each frame into a base64 string
- Publishes frames to RabbitMQ queue at a controlled rate (~30 FPS)

1. **Frame Encoding for RabbitMQ Transfer**
   Each frame is encoded as a JPEG, then base64 encoded:

```
def encode_frame(frame):
  _, buffer = cv2.imencode('.jpg', frame)
  return base64.b64encode(buffer).decode('utf-8')
```

2. **Establishing Connection to RabbitMQ**

```
def connect_rabbitmq():
  connection =
pika.BlockingConnection(pika.ConnectionParameters(host=RABBITMQ_HOST))
```

```
    channel = connection.channel()
    channel.queue_declare(queue=RABBITMQ_QUEUE, durable=True)
    return channel
```

3. **Main Loop: Read, Encode, Publish**

```
while True:
    ret, frame = cap.read()
    if not ret:
        break

    frame_data = encode_frame(frame)
    channel.basic_publish(
        exchange='',
        routing_key=RABBITMQ_QUEUE,
        body=frame_data.encode('utf-8')
    )
    time.sleep(1 / 30)  # ~30 FPS
```

- **VIDEO_SOURCE**: Path to the video or camera source (from config.py)
- **RABBITMQ_QUEUE**: Queue name to which frames are published

3- Detection Service

This service is responsible for **processing incoming frames**, **detecting violations or safe behavior**, **annotating frames**, and **publishing results** to the next stage.
It uses **YOLOv12** for object detection and **ByteTrack** for person tracking, and it evaluates whether a pizza is picked up using a scooper or a bare hand, identifying **violations** and **safe pickups** accordingly.

1- Model Inference

Runs YOLOv12 on each frame using .track() for object detection and tracking.

```
results = model.track(frame, persist=True, tracker="bytetrack.yaml", conf=0.01, iou=0.3, verbose=False)[0]
```

2- Object Categorization

Filters out labels like hand, scooper, pizza, and person, and stores their bounding boxes.

3- Tracking Logic

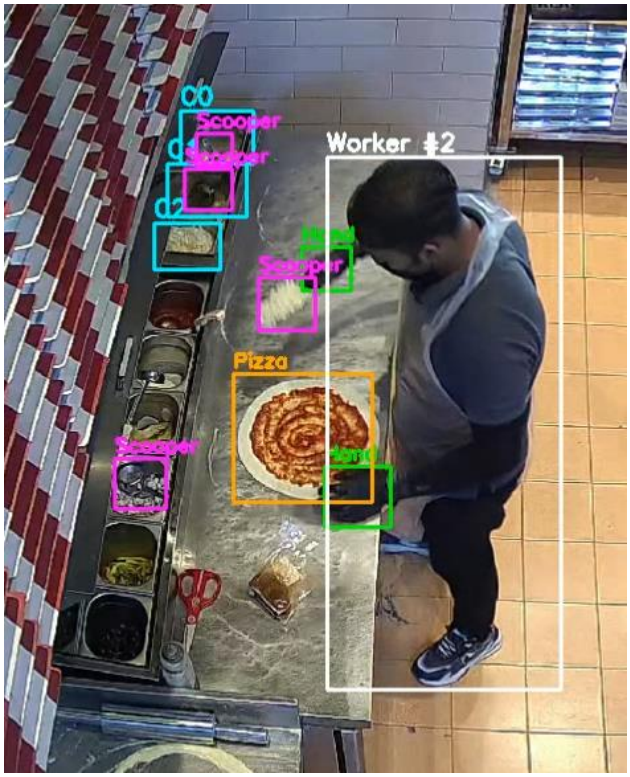
- Associates detected hands with people using distance matching.
- Maps each worker to a consistent ID using:

def get_consistent_worker_id(...)

4- Event Detection

- Detects if a worker hand enters the ROI (region of interest) without a scooper and touches a pizza.
- Validates events after a short buffer of 50 frames (for stability).
- Differentiates between:
  - **Violation:** Pizza touched without a scooper
  - **Safe Pickup:** Pizza touched with a scooper

➢ The ROIs here were the containers that had scoopers C0, C1 and C2, so they're static regions we adjust them in code on demand, like in this picture:



5- Frame Annotation

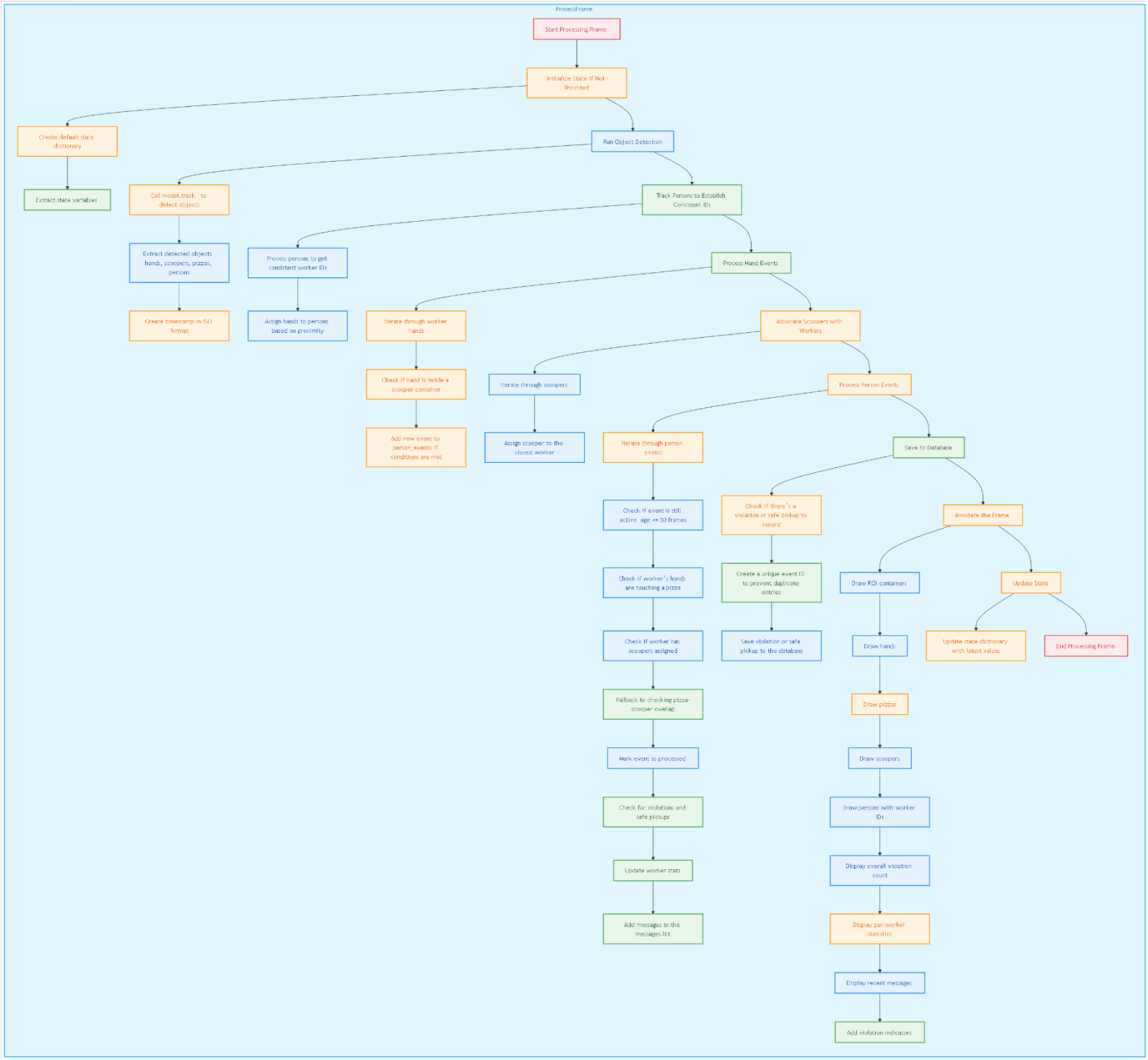
- Draws bounding boxes, ROIs, messages, and statistics for real-time feedback.

6- Logging & Storage

- Events are stored in an SQLite DB
- Subscribes to the RabbitMQ `raw_frames` queue.
- Applies the `process_frame()` function with shared persistent state.

- Publishes processed results (frames, metadata) to `processed_frames`

  save_violation(timestamp, "", labels, boxes, is_violation, is_safe_pickup, db_path)



4- Streaming Service

1- rabbit_consumer.py: RabbitMQ Stream Consumer

This service listens to the **processed frame queue** (containing inference results from the detector), decodes each base64-encoded image frame, and pushes the decoded frames to a **shared state buffer**.

Key logic:

- **Base64 decoding** of frames received from RabbitMQ:

```
def decode_base64_frame(b64_str):
    byte_data = base64.b64decode(b64_str)
    np_array = np.frombuffer(byte_data, dtype=np.uint8)
    return cv2.imdecode(np_array, cv2.IMREAD_COLOR | cv2.IMREAD_IGNORE_ORIENTATION)
```

- **Multithreaded decoding**: 3 worker threads decode frames from a queue:

```
def process_frame_worker(worker_id=0):
    while True:
        frame_data = frame_queue.get(timeout=5)
        frame = decode_base64_frame(frame_data)
        state.latest_frame = frame
```

- **Frame Queue** ensures backpressure when receiving frames faster than decoding:

```
frame_queue = queue.Queue(maxsize=60)
```

- **RabbitMQ Listener** (uses PROCESSED_QUEUE):

```
channel.basic_consume(queue=PROCESSED_QUEUE, on_message_callback=callback, auto_ack=False)
```

2- streaming_app.py: Live Video Dashboard

The Flask app renders a **dashboard** displaying the current video feed and violation statistics in real time.

Key features:

- **Live MJPEG Streaming** using generator:

```
def generate_frames():
  while True:
    frame = getattr(state, 'latest_frame', None)
    jpeg = cv2.imencode('.jpg', frame)[1].tobytes()
    yield (b"--frame\r\nContent-Type: image/jpeg\r\n\r\n" + jpeg + b"\r\n")
```

- **HTML Dashboard** (GET /) with:
  - Real-time stats using /summary endpoint
  - Embedded video via /video route
- **Summary API** (cached every second to avoid DB overload):

```
@app.route("/summary")
def get_summary():
  cursor.execute("SELECT COUNT(*) FROM violations WHERE is_violation = 1")
  cursor.execute("SELECT COUNT(*) FROM violations WHERE is_safe_pickup = 1")
```

- **Deployment with Waitress** (production-grade WSGI):

```
serve(app, host="0.0.0.0", port=8000)
```

5- Handled Cases & Edge Conditions

1- Multiple Workers in Frame

The system supports multi-worker detection and tracking using ByteTrack IDs in combination with a custom consistent worker ID assignment mechanism:

- Each tracked person is assigned a consistent worker_id using spatial continuity and object tracking.
- Hands detected in the frame are dynamically assigned to the closest worker using Euclidean distance from hand center to person center.
- This allows the system to accurately distinguish between workers, even if they're close to each other or temporarily occluded.

🔁 Hands and violations are tracked per worker independently using the worker_id for isolated logic and statistics.

2- False Positive Prevention: Worker Cleaning Without Pizza Touch

To prevent **false violations** when a worker is simply cleaning or organizing the station:

- The system ensures that a **violation is only triggered if the pizza was touched first** and no scooper was used **after** that.
- If a worker is seen using a scooper **without having touched a pizza**, **no violation or event is triggered**, this is treated as a **normal or cleaning behavior**.
- Only if the hand touches a pizza inside the ROI *before* scooper interaction is the event processed as a possible violation.

🛡️ This logic prevents flagging safe behaviors such as cleaning or preparing tools before serving.

6- Notes

- Currently, the major bottleneck in system speed is YOLO detection latency, especially on high-resolution frames. So, to significantly reduce detection time, you can convert the YOLOv12 model to **TensorRT** format. **TensorRT** optimizes inference on **NVIDIA GPUs** and is ideal for deployment. But ensure that your **GPU** supports TensorRT because unfortunately, mine doesn't.
- Using more diverse and annotated training data will significantly improve the detection model's accuracy and robustness in real-world scenarios.

7- Appendix

To see demo videos for the flow of the system, codes, and the setup steps to run the project:

Github Repo: **https://github.com/amira-medhat/Pizza-Store-Hygiene-Monitoring-System.git**