Project Documentation

Addressing Real-Time Inference Performance

Developed by *Farah Ahmed*

Email: *farahhahmed01@gmail.com*

1. The Challenge of Real-Time Performance

A primary objective of the "Pizza Store Scooper Violation Detection" project was to provide a real-time video stream with live violation analysis. During development, after the successful implementation of the multi-service pipeline, a significant performance bottleneck was identified: the video stream displayed on the frontend UI was not smooth, appearing choppy and "frame-by-frame" rather than as a fluid, real-time video.

This document details the investigation into this performance issue, the software-based optimization strategies that were implemented, and the final analysis of the underlying hardware limitations that define the boundaries of real-time performance on CPU-based systems.

2. Initial Diagnosis: Identifying the Bottleneck

The system is composed of three main services: a **Frame Reader Service** that sends frames, a **Detection Service** that analyzes them by doing inference and run the detection logic, and a **Streaming Service** that displays them. Analysis of the RabbitMQ message queues revealed that messages (raw video frames) were accumulating in the **video_frames** queue, waiting to be processed.

This confirmed that the **Detection Service** was the bottleneck. It was unable to process frames as quickly as the **Frame Reader** was sending them. The root cause was the computationally expensive nature of the AI inference step, specifically the **model.track()** function, which was being executed on every single frame received.

3. Optimization Strategy 1: Throttled Inference

The first and most significant software optimization implemented was **Throttled Inference**.

Concept:

The core idea is to decouple the heavy AI inference from the main processing loop. The AI model does not need to be run on every single frame of a high-FPS video, as the position of objects does not change dramatically between consecutive frames.

Implementation:

- An **INFERENCE_FPS** configuration variable was introduced in the **Detection Service** to define how many times per second the AI model should run (e.g., 10 times per second).
- A **skip_interval** was calculated based on the incoming frame rate. For example, to achieve 10 inference FPS from a 30 FPS video source, the script would run the AI model on one out of every three frames.

- A **last_results** global variable was created to store the output of the most recent AI inference.
- The main **process_frame** function was modified. The slow **model.track()** function was placed inside a conditional block (**if frame_idx % skip_interval == 0:)**. For all "skipped" frames, the service would instead use the data from **last_results** to run the much faster violation logic and draw annotations.

Result: This strategy yielded a very small improvement. The Detection Service was able to process frames a little bit quickly. But, choppiness remained significantly.

4. Optimization Strategy 2: Pipeline Optimization

To further reduce the load on the system, a second layer of optimizations was applied, focusing on the data itself.

Concept: The less data each service has to handle, the faster the entire pipeline will run.

Implementation:

- **JPEG Quality Reduction:** When encoding the frames to JPEG format, the quality was set to 80%. This reduces the byte size of each message sent through RabbitMQ.

Result: These changes further improved the fluidity of the video stream by reducing network traffic and the amount of data the **Detection Service** had to decode and process. But, it still did not achieve a perfectly seamless, real-time feel.

5. Final Analysis: The Core Issue of CPU vs. GPU Inference

After implementing all practical software optimizations, the remaining choppiness was correctly identified as a hardware limitation. The system was being developed and tested on a machine without a dedicated NVIDIA GPU, meaning all AI inference was running on the CPU (an AMD Ryzen).

Technical Explanation: Deep learning models like YOLO are composed of massive numbers of parallel mathematical operations (matrix multiplications).

- An **NVIDIA GPU** is a highly specialized processor with thousands of cores designed to execute these simple, repetitive tasks in a massively parallel fashion. Using NVIDIA's **CUDA** platform, it can solve these math problems almost instantly.
- A **CPU**, by contrast, has a few very powerful cores designed for complex, sequential tasks. When forced to run AI inference, it must perform those thousands of calculations one after another. Even a high-end CPU will be an order of magnitude slower than a compatible GPU for this specific type of workload.

Conclusion: The final analysis is that while the implemented software optimizations are the correct and standard industry approach to making real-time vision possible on CPUs, achieving perfectly fluid, high-frame-rate performance for this demanding task is ultimately constrained by the hardware. The remaining latency is not a bug in the logic, but the physical time required for the CPU to perform the complex calculations of AI inference. For a production deployment requiring a perfectly smooth stream, running the **Detection Service** on a machine with a compatible NVIDIA GPU would be the required next step.