Summary

This document outlines the significantly refactored software architecture for the perception pipeline. Responding to the need for enhanced performance analysis, modularity for ROS integration, and precise latency measurement, the application has been restructured into a clean, object-oriented, multi-threaded pipeline.

This new architecture successfully encapsulates all processing logic within a TrackingPipeline class, managed by a centralized config.hpp file. Key features have been implemented to empower performance engineering, including on/off toggles for CPU-intensive modules and a high-precision, per-stage profiling system. This document serves as a guide to the new code structure and details the procedures for conducting the requested performance and latency analysis.

2. Code Structure Overview

To meet the requirement for a modular and ROS-ready design, the previous monolithic application has been broken down into the following logical components within the object_detection/ directory:

- **config.hpp**: A centralized header file containing all user-configurable parameters and feature flags. This is the primary file for controlling the pipeline's behavior without recompiling the core logic.
- **tracking_pipeline.hpp**: The main interface file. It defines the TrackingPipeline class and the FrameData structure, providing a clean API for the entire system.
- **tracking_pipeline.cpp**: The core implementation file. It contains the detailed logic for the TrackingPipeline class, including the management of the three primary worker threads (Pre-process, Inference, Post-process).
- **main.cpp**: The application's simple entry point. Its sole responsibility is to create a configuration, instantiate the TrackingPipeline class, and execute its run() method.
- **utils/**: This directory continues to hold supporting code, including the Hailo inference wrapper (async_inference) and the new, self-contained global_stabilizer class.
- **tracking_lib/**: The external, pre-compiled ByteTrack library, which is treated as a modular dependency.

This structure ensures a clean separation of configuration, interface, and implementation, making the system robust, maintainable, and easy to integrate with larger frameworks like ROS.

3. Component Breakdown & Functionality

3.1. config.hpp (The Control Panel)

This file is the primary tool for controlling the pipeline's features for testing and analysis.

- **What it does:** It defines the PipelineConfig struct, which holds all operational parameters.

- **Key Flags:**
  - **bool enable_global_stabilization**: This is the main switch for the CPU-intensive video stabilization module.
    - true: (Default) The optical flow-based stabilization is active. This provides the most stable tracking when the camera is in motion but consumes significant CPU resources.
    - false: The stabilization module is completely bypassed. This dramatically reduces CPU load and allows for precise measurement of its performance impact.
  - **bool enable_visualization**: Controls all on-screen display features.
    - true: (Default) An OpenCV window will show the video feed with bounding boxes and tracking information.
    - false: Runs the pipeline in "headless" mode. No GUI is created.
  - **bool enable_profiling_log**: Toggles the detailed per-stage latency logging.
    - true: (Default) For every frame, a detailed timing breakdown will be printed to the console.
    - false: Disables the console logging for cleaner output during normal operation.

3.2. tracking_pipeline.hpp / .cpp (The Engine)

- **What it does:** This class encapsulates the entire application. The constructor initializes the Hailo model, video capture, and the tracker. The run() method launches three parallel worker threads:
  1. **preprocess_worker()**: This thread is responsible for capturing frames, resizing them, and (if enabled) performing global video stabilization. It places the prepared data into a queue for the next stage.
  2. **inference_worker()**: This thread waits for pre-processed data, sends it to the Hailo NPU for inference, and places the results into a second queue.
  3. **postprocess_worker()**: This thread waits for inference results, performs tracking using the BYTETracker library, handles all visualization, and generates the final PTZ guidance commands.
- **Asynchronous Operation:** The use of three threads and two thread-safe queues ensures that the pipeline stages operate asynchronously. A slow stage (like pre-processing with stabilization) will not block a fast stage (like post-processing), maximizing throughput.

3.3. global_stabilizer.hpp / .cpp (The Stabilization Module)

- **What it does:** This class contains all the logic for the global video stabilization. It uses OpenCV's optical flow to estimate camera motion and applies an affine warp to the image to counteract it. All internal state (like the previous frame's data) is managed within this class, keeping the main pipeline code clean.

4. How to Conduct Performance & Latency Analysis

The new structure is designed to make this process straightforward.

4.1. Measuring the Impact of Global Stabilization

To measure the precise CPU and latency cost of the stabilization module, perform the following two tests on the same input video:

1. **Test A (Stabilization ON):**
    - In config.hpp, ensure enable_global_stabilization = true;.
    - Recompile and run the application.
    - Observe the average CPU utilization using a system tool (e.g., top).
    - From the console output, record the average **Preproc:** time from the [PROFILE] logs.
2. **Test B (Stabilization OFF):**
    - In config.hpp, change the flag to enable_global_stabilization = false;.
    - Recompile and run the application.
    - Observe the new, lower average CPU utilization.
    - From the console output, record the new, lower average **Preproc:** time.

4.2. Measuring Pure Processing Latency (Headless Mode)

To measure the pipeline's raw performance without the overhead of drawing to the screen or writing a video file, configure the system as follows:

- In config.hpp:
    - Set enable_visualization = false;
    - Set save_output_video = false;
    - Ensure enable_profiling_log = true;
- **Analysis:** Run the application. The [PROFILE] logs will now show the true end-to-end software latency. The **Total E2E(End-to-End):** value represents the pure processing time from

frame capture to the final tracking result. This is the key metric for evaluating the core performance of the perception pipeline.

4.3. Understanding the Profiling Log

The console output provides a detailed breakdown for each frame:

```
[PROFILE] Frame 123: Preproc: 15.2ms | Infer: 24.1ms | Postproc: 2.8ms | Total E2E: 42.1ms
```

- **Preproc**: Time spent in the preprocess_worker (capture, resize, stabilization).
- **Infer**: Time from when the data is sent to HailoRT until the NPU result is ready. **This is the key metric for identifying the current ~460ms bottleneck for y11sperson.hef model.**
- **Postproc**: Time spent in the postprocess_worker (tracking, drawing preparation).
- **Total E2E**: The total software latency for the frame. The application's theoretical maximum FPS is 1000 / Total E2E.

5. Testing & Optimization Strategy

This section details the performance testing conducted on the pipeline, analyzes the results to identify the primary bottleneck, and proposes a clear, multi-pronged strategy to achieve the target performance with the yolov11s model architecture.

5.1. Performance Baseline Analysis

Initial testing was conducted using the integrated profiling tools to analyze the performance of various models and identify system bottlenecks.

A. Profiling Results with y11s_person.hef:
The pipeline was run with the 1-class y11s_person.hef model. The profiling output consistently revealed a critical performance issue:

```
[PROFILE] Frame X: Preproc: ~7ms | Infer: ~460ms | Postproc: ~3ms | Total E2E: ~470ms
```

- **Analysis:** The CPU-bound tasks (Preproc and Postproc) are extremely fast, executing in approximately 10ms combined. However, the **Infer stage exhibits an unexpectedly high latency of ~460ms**, resulting in an end-to-end application performance slow down.

B. Comparative Model Benchmarking:
To understand the hardware's potential, benchmarked several YOLO models using the hailortcli tool, which measures raw NPU throughput.

Model	Classes	Complexity (Compiler Output)	Benchmark FPS
yolov8n.hef	80	**Single Context**	**~45-50 FPS**
yolov8s.hef	1(Person only)	**Single Context**	**~80+ FPS (Est.)**
yolov11s.hef	80	Multi Context (2 Contexts)	~**30** FPS (Est.)
y11s_person.hef	1	**Multi Context (3 Contexts)**	**~23 FPS**

5.2. Root Cause Analysis: The Inference Bottleneck

The testing data points to two distinct issues contributing to the performance gap:

1. Critical Issue: CPU Fallback due to Data Format Mismatch
The ~460ms inference latency is caused by the HailoRT falling back to the CPU to perform a slow data format conversion. **This may be a software integration issue, not a hardware limitation.**

2. Architectural Issue: Model Complexity and Context Switching
The benchmark results highlight a significant performance difference based on model complexity.

- **(yolov8s)** once I trained with a kaggle IR person dataset with very few epochs, just testing the performance, it was quite good at fps. Can be have an option with training the same images what y11sperson.hef trained and tested the performance.
- **(y11s_person)** are inherently slower. The Hailo compiler had to split the model into **3 sequential parts (contexts)**. The NPU must execute Part 1, save the result, reconfigure itself, execute Part 2, and so on. This "context switching" introduces significant overhead, explaining why its theoretical maximum FPS (~23 FPS) is much lower than that of a single-context model. This is a **model architecture and compiler optimization issue. (Not sure about the compiler or conversion can handle this, it might be automatically setup while conversion, but this can decrese fps found while testing with larger context value)**

5.3. Proposed Optimization Plan

Step 1: Deeply analyzes about the Hardware-acceleration part, still I can’t find anything which can help to improve performance.

Step 2: Recommended - Model Re-Compilation with Pre-processing Fusion and Batching

To further reduce CPU load and maximize NPU efficiency, maybe re-compiling the y11s_person.hef from its original .onnx source file, do quantization or retrain some other version of yolo for doing testing and evaluating for better result according to expected outcomes.

Step 3: Batching

Running inference on batches will allow the NPU to operate at its maximum efficiency, pushing the NPU throughput . The end-to-end application FPS will improve accordingly.
(You already mention about batching issues, But can keep this an option for further testing).