ChatGPT

Real-Time Video Analytics Pipeline on NVIDIA Jetson Orin Nano: Research Roadmap

This roadmap outlines a comprehensive strategy for building a **highly optimized, real-time video analytics pipeline** on the Jetson Orin Nano and similar embedded AI platforms. It is organized into key categories reflecting the primary goals: camera capture, deep-learning inference, recording with annotation, system-level integration, advanced capabilities, and performance evaluation. Each section lists core research questions, state-of-the-art tools (including but not limited to NVIDIA's), implementation techniques, relevant search keywords, quantitative metrics, and suggested sources for further exploration. The focus is on a **modular, GPU-accelerated, multi-threaded design** that balances **low latency, power efficiency, and maintainability**.

1. Real-time Video Capture (Low-Latency Frame Acquisition)

- **Core Research Topics:** How to capture video frames with *minimal latency* and overhead, directly in a GPU-friendly format. Investigate USB webcam vs. CSI camera performance, driver capabilities (e.g. UVC for USB cameras), and methods to avoid unnecessary memory copies. Explore questions like *"How to achieve zero-copy frame transfer to GPU?"* and *"What capture pipeline yields the lowest end-to-end latency?"*.

- **State-of-the-Art Tools/Frameworks:** For Jetson and general Linux, **GStreamer** is a primary choice (with NVIDIA plugins like `nvvidconv` for zero-copy GPU buffers). **OpenCV** (with GStreamer backend) offers a simpler API for frame capture; it can leverage V4L2 under the hood for portability. Low-level **V4L2 (Video4Linux2)** APIs allow fine-grained control (e.g. using `ioctl` with memory-mapped (mmap) or DMA buffer modes). For CSI cameras on Jetson, NVIDIA's **LibArgus** can provide direct GPU buffers. *Non-NVIDIA:* on other platforms one might use OpenCV, GStreamer, or vendor SDKs (e.g. Intel Media SDK, OpenNI for depth cameras, etc.). Also consider specialized libraries (e.g. **libuvc** for USB cameras) or hardware-specific pipelines (like Raspberry Pi's MMAL or Arducam SDK) for broader applicability.

- **Implementation Techniques:** Use **zero-copy and pipelined capture** to minimize latency. On Jetson, leverage the unified memory architecture – capture frames into pinned CPU/GPU shared memory to avoid extra copies [1] [2]. For example, configure V4L2 to use `V4L2_MEMORY_DMABUF` so frames are captured directly into DMA buffers accessible by the GPU. Utilize hardware image processing: many USB cameras (e.g., Logitech C920) output compressed formats like MJPEG; you can let the Jetson's HW JPEG decoder or a fast CUDA decoder handle this in a separate thread. (Note: modern C920 models output YUYV or MJPEG but no longer H.264 [3] [4], so plan to decode MJPEG frames on the device.) If available, prefer uncompressed formats and GPU-accelerated color conversion (e.g. YUV to RGB via `nvvidconv` or a custom CUDA kernel) to avoid CPU bottlenecks. It's critical to tune camera settings: use the highest USB bandwidth mode (e.g. USB3 if possible) and disable any automatic controls that add latency. For lowest latency, a MIPI CSI-2 camera is ideal – it feeds frames through the ISP directly with lower overhead. In fact, NVIDIA notes that a CSI camera can achieve

significantly higher framerates and lower latency than an equivalent USB cam [5]. (One experiment on Orin Nano saw ~24 FPS with a CSI camera vs ~11 FPS with a USB webcam for the same model [6].) To maximize throughput on a USB camera, consider reducing resolution or frame rate to match processing capability, and use the camera's native format to avoid conversion (e.g. if the webcam provides MJPEG at 1080p30, capture that and decode on GPU). Enable driver *bulk mode* or *in-kernel frame buffering* if available to prevent frame drops.

- **Search Keywords & Technical Tags:** `V4L2 DMA buffer`, `GStreamer nvvidconv NVMM`, `UVC camera latency`, `Jetson zero-copy capture`, `OpenCV VideoCapture GStreamer`, `MIPI CSI vs USB performance`, `camera driver threaded capture`, `GPU image conversion CUDA`.

- **Benchmarks or Metrics to Evaluate:** Key metrics include **capture latency** (e.g. time from exposure to frame ready in GPU memory, target <~30ms for 30 FPS), **frame rate stability** (sustaining 30 FPS or desired rate with no drops), and CPU utilization during capture. Measure end-to-end *camera-to-GPU* memory transfer time; on Jetson, this should be a few milliseconds with zero-copy. Evaluate frame timing jitter (variance) – a real-time pipeline needs consistent inter-frame intervals. If using USB, monitor USB bandwidth usage (is the bus saturated?). For camera quality, track any frame desync or timestamps. For example, ensure that a 1080p @ 30 fps Logitech C920 (which supports 1920×1080 at 30 FPS [7]) indeed delivers ~33ms intervals. If using software decoding (MJPEG→RGB), measure the decode time per frame (Jetson's NVJPEG can decode JPEG in a few milliseconds). "Success" in this stage means **minimal capture latency (just a few ms overhead) and <5% CPU usage** for the capture thread, by offloading copying and color conversion to the GPU.

- **Sources to Explore:** NVIDIA Developer Forums and RidgeRun wiki for Jetson capture (e.g. discussions on **zero-copy V4L2** capture [8] [9]); GStreamer official docs for the `v4l2src` and NVIDIA codec plugins; OpenCV documentation on GStreamer pipelines. The blog *"Jetson Zero-Copy for Embedded Applications"* [1] [2] explains how unified memory can eliminate copies in Jetson's integrated GPU architecture. Also review the **Christian Mills tutorial** on Jetson Orin Nano, which shows setting up a CSI vs USB camera and a GStreamer pipeline in OpenCV [5]. Open-source projects like *ros-drivers/usb_cam* (Issue #320) illustrate using **DMA buffers and CUDA** for a Jetson camera driver [8]. These will provide implementation insights (e.g. sample code for allocating CUDA buffers and converting YUV to RGB on GPU).

2. Object Detection Inference (High-Performance Continuous Inference)

- **Core Research Topics:** Achieving **real-time object detection** on an embedded GPU. Key questions include: *Which neural network models provide the best speed/accuracy trade-off on Orin Nano?*; *What frameworks or inference engines maximize throughput?*; *How to optimize models (quantization, pruning) for edge inference?*; *How to pipeline inference to run in parallel with other tasks?* Since the Jetson Orin Nano has a modest GPU and no dedicated DLAs [10], we focus on GPU Tensor Cores for acceleration. Broader topics: one-stage vs two-stage detectors on edge devices, efficient backbone architectures (e.g. MobileNet, EfficientNet), and batch vs single-image inference in a streaming context. Also consider *continuous inference loops* – how to handle each frame arriving in succession without GPU idle time (overlap compute and data transfer using streams).

- **State-of-the-Art Tools/Frameworks: NVIDIA TensorRT** is the flagship – it compiles models to highly optimized engines (FP16 or INT8 precision) and is the go-to for maximum performance on Jetson [11]. NVIDIA's **DeepStream SDK** builds on TensorRT with a full pipeline (more on that later) and pre-optimized plugins (for decoding, inferencing, tracking). For a more framework-agnostic approach, **ONNX Runtime** is notable: it can run ONNX models using TensorRT as an execution provider, which Christian Mills demonstrated for YOLOX on Orin Nano [12]. This allows writing Python inference code while delegating heavy lifting to TensorRT. Other options include **PyTorch with TorchScript** (convenient but slower than TensorRT), **TensorFlow Lite (TFLite)** with GPU delegates (works on Jetson but not as optimized as TensorRT for CNNs), and even vendor-neutral engines like **TVM** or **OpenVINO** (OpenVINO is Intel-focused but supports NVIDIA GPUs via ONNX). Also consider model-specific libraries: e.g., **TensorRT YOLOv5** from NVIDIA/TAO toolkit or **TRT-optimized CV models** (NVIDIA has pretrained EfficientDet, SSD, etc.). *Non-NVIDIA alternatives:* On devices like Google Coral, one would use the Edge TPU runtime; on OpenCV AI Kit or others, use their NPU API. We mention these to ensure modularity – the pipeline should allow swapping the inference backend (e.g., using ONNX runtime on CPU vs TensorRT on GPU) with minimal changes.

- **Implementation Techniques: Model Optimization:** Use 16-bit or 8-bit precision. The Orin Nano's GPU has Tensor Cores (FP16/INT8); converting models to INT8 (with calibration or QAT) can boost speed ~2× at some cost to accuracy. For example, Mills quantized a YOLOX model to INT8 using ONNX-TensorRT for inference [13]. Leverage efficient architectures: e.g., YOLOv8-n or YOLOv5s, SSD-MobileNet, or NVIDIA's PeopleNet/DetectNet. These smaller models often run at 30+ FPS on Orin. Avoid very heavy models (e.g. Faster R-CNN or YOLOv8x) that exceed real-time budget. **Continuous Batch-1 Inference Loop:** design the code to fetch the next frame while the GPU is busy on the current frame. For instance, use separate threads or CUDA streams – one thread performs pre-processing on CPU for frame N+1 while the GPU processes frame N. Utilize TensorRT's asynchronous API: enqueue inference on a CUDA stream and do CPU work (post-processing of the previous frame, etc.) in parallel. **Zero-copy inference I/O:** Where possible, feed GPU data directly to the inference engine. For example, if using TensorRT C++ API, you can wrap a CV cudaGpuMat or DMA buffer as input without extra memcpy. If using DeepStream, the frames stay in GPU memory (as NVMM) through the pipeline. **Pipeline parallelism:** Consider a double-buffer scheme: maintain two sets of device buffers so you can copy new frame into one while the model computes on the other, thereby overlapping data transfer with computation. Additionally, explore the Jetson's capability to run multiple parallel streams if needed (the Orin can handle multiple networks if resources allow, e.g. a detection model and a secondary classification). **Alternative engines trade-offs:** TensorRT tends to give the best raw throughput on NVIDIA (often 2-3× faster than raw PyTorch [11]), but for development ease one might use ONNX Runtime or even run the model in PyTorch if performance is still acceptable. Always measure memory footprint too – TensorRT engines are optimized but can consume memory; ensure the 4–8GB RAM is sufficient for your model and frames. The inference step should ideally be under ~30ms for 30FPS; below we list some benchmark numbers.

- **Search Keywords & Technical Tags:** `TensorRT INT8 calibration`, `Jetson quantized model performance`, `DeepStream YOLO example`, `ONNX Runtime TensorRT EP Jetson`, `CUDA streams asynchronous inference`, `Edge TPU vs GPU benchmarks`, `Jetson Triton Inference Server`, `model pruning sparsity edge AI`.

- **Benchmarks or Metrics to Evaluate:** The primary metric is **inference latency per frame** (or throughput in FPS). On Jetson Orin Nano, aim for inference <33 ms to sustain 30 FPS; for many

models <20 ms is achievable. For example, a recent study found YOLOv8-nano runs in ~16 ms on Orin Nano (≈62 FPS) while a larger YOLOv8 model took ~50 ms (20 FPS) [14]. EfficientDet-Lite or SSD-mobilenet models were ~20 ms each (≈50 FPS) on TensorRT [14]. We summarize some model performance for context:

Model (TensorRT-optimized)	Inference Time (Jetson Orin Nano)
YOLOv8-nano (smallest YOLOv8)	~16 ms per frame [14] (~62.5 FPS)
YOLOv8-medium	~50 ms per frame [14] (~20 FPS)
EfficientDet-Lite0	~20 ms per frame [14] (≈50 FPS)
SSD-MobileNetv2	~20 ms per frame [14] (≈50 FPS)

Table: Example detection model speeds on Jetson Orin Nano (using TensorRT). These assume FP16 or INT8 optimization; FP32 (or using PyTorch without optimization) would be significantly slower (often 2-4×). **Other metrics:** measure **GPU utilization** (ideally high but leaving headroom for other tasks), and **power consumption** during inference. For power, Orin Nano should stay within its budget (e.g. 10-15W) even under load – check that running inference doesn't thermal throttle the device. Also evaluate **accuracy** of the detection on your target data (mAP, precision/recall) – ensure optimizations (INT8, etc.) did not degrade accuracy beyond acceptable levels. Success criteria might be, for instance, *"maintain mAP > 0.5 on test set while achieving 30 FPS at 720p resolution with <10W power"*. Additionally, consider **latency jitter**: the model should have stable inference times frame-to-frame to avoid buffering issues.

- **Sources to Explore:** NVIDIA's **Jetson Benchmark charts** (many blog posts or papers, e.g. an arXiv paper "Benchmarking Deep Learning Models on Edge Devices" provides insight into YOLOv8, SSD, EfficientDet on Orin Nano [15] [16]). The NVIDIA forums (search "Orin Nano TensorRT YOLO") often have users sharing performance figures and tips. The **Ultralytics YOLOv5 and YOLOv8 repositories** have Jetson deployment guides [17]. Look at **NVIDIA Tao Toolkit** – it provides pretrained detection models optimized for TensorRT (e.g. PeopleNet, TrafficCamNet) and metrics about their performance on Jetson. For implementation, examine the open-source **jetson-inference** project (by Dustin Franklin) which includes C++ and Python code for real-time detection using TensorRT on various models. Also, the blog post *"Deploying YOLOX on Jetson Orin Nano"* by Christian Mills is an excellent step-by-step resource [18] [13], illustrating the use of ONNX Runtime with TensorRT, INT8 quantization, and even tracking. For non-NVIDIA perspective, see Intel's OpenVINO benchmarks or Google Coral demos to understand alternative frameworks – this helps ensure our design remains modular and not Jetson-specific.

3. Recording & Labeling (Synchronized Video Storage with Annotations)

- **Core Research Topics:** How to **record the video feed with overlaid detection results (bounding boxes, labels)** in real-time, and log metadata (e.g. object classes, timestamps) for each frame or event. Key questions: *What is the most efficient way to encode video on an embedded device?* (Given limited hardware encoding on some platforms); *How to draw or overlay annotations without slowing down the pipeline?*; *How to synchronize metadata and video frames accurately?*; *What data formats and storage are suitable for later retrieval or training?* This involves exploring video codecs (H.264/H.265 vs

raw), container formats (MP4, MKV), and whether to record continuously or only on events. It also covers *labeling for training*: possibly storing frames and detection outputs for future model improvement.

- **State-of-the-Art Tools/Frameworks: GStreamer** again is a strong candidate for recording: it can fetch frames (as a GStreamer buffer) and pass them to encoders and file sinks. On Jetson platforms with hardware encoders (e.g. Xavier NX), elements like `nvv4l2h264enc` provide GPU-accelerated encoding, but **note:** Jetson Orin Nano **lacks a hardware video encoder** [19]. Therefore, recording on Orin Nano must use software encoding (CPU-based x264/x265 encoders) or an external hardware encoder. GStreamer's `x264enc` plugin or **FFmpeg** library (libx264) can encode H.264 on the CPU. NVIDIA's **DeepStream** SDK includes an `nvdsosd` (on-screen display) component to draw bounding boxes on frames in GPU memory, and sink plugins (like `nvvideoenc` for encoding and `filesink` or `splitmuxsink` for file output) – DeepStream essentially streamlines recording annotated video with minimal custom code. Another approach is using **OpenCV**: grab frames, draw rectangles with `cv::rectangle` or `cv2.rectangle` (if Python), and use `VideoWriter`. This is simpler but can be CPU-intensive for high resolution. For labeling data, one might output metadata in JSON or CSV form alongside the video – e.g., each frame's detections (class, bbox coordinates, confidence). There are also specialized **video annotation tools/formats**: for instance, writing to a **YOLO format text file** per frame, or using **ROS bag** if in a robotics context (ROS bag can record image topics and detection topics together). In modular design, you could consider a separate "logger" thread that receives detection results and writes to disk (to avoid slowing the main loop). If targeting a surveillance scenario, also consider **timestamp overlays** or saving segments of video when events occur. *Non-NVIDIA:* On other devices, similar principles apply – e.g., on a Raspberry Pi, use the hardware encoder via MMAL/OMX, or on Intel use QuickSync or software FFmpeg. The pipeline should allow swapping the recording backend (GStreamer vs OpenCV vs custom) as needed.

- **Implementation Techniques: Efficient Encoding:** since Orin Nano has no HW encoder, using **fast software encoding** is crucial. GStreamer `x264enc` with an *ultrafast* preset can encode 1080p at decent speeds by trading off compression efficiency for speed [20] [21]. (Experiments show using x264 *ultrafast* vs *veryslow* can raise encoding throughput from ~5 FPS to ~69 FPS at 1080p, at the cost of larger file size and lower quality [21].) Thus, choose a preset like *superfast/ultrafast* and a moderate bitrate to ensure real-time encoding. Also enable OpenGL/CUDA overlays if possible: one trick is to use **NVMM buffers and nvosd** (NVIDIA's on-screen display in DeepStream) to draw bounding boxes in the GPU, then feed into an encoder – this avoids bringing frames back to CPU for drawing. If coding manually, one can use **CUDA kernels or OpenGL** to draw on frame (e.g., using OpenGL FBO to overlay lines). If stuck with CPU drawing, restrict it to lower-res frames or only when needed. **Threaded I/O:** perform file I/O on a separate thread or use asynchronous writes. For example, a producer thread (after inference) could push annotated frames or metadata into a queue; a consumer thread handles encoding and disk writes. This decoupling prevents disk latency (or encoding spikes) from stalling the camera/inference loop. **Synchronization:** ensure each video frame's metadata carries a timestamp or frame index. You can maintain a log (e.g., CSV with `frame_number, timestamp, detections...`) to correlate with the video. If using GStreamer's built-in MP4 recording, you might embed metadata as subtitles or simply rely on parallel logs. **Storage considerations:** decide on recording format – continuous video vs short clips triggered by events (discussed more in Advanced section). Continuous recording might need a circular buffer or periodic file rollover (use GStreamer's `splitmuxsink` to split video into chunks by time or size). If labeling for training data, perhaps only save frames when certain objects appear, to save space. Also,

monitor storage write speed: an encoded 1080p30 stream might be ~5-10 Mbit/s; ensure the SD card or SSD can sustain this. Power-wise, software encoding will tax the CPU (as noted by NVIDIA, x264 encoding will use significant CPU and one should max out CPU frequency via `jetson_clocks` for consistent performance [22]). If this power use is problematic, consider lowering resolution (e.g. record 720p instead of 1080p to reduce CPU load ~50%). **Label rendering:** If high-quality recorded video with bounding boxes is needed, consider drawing boxes with semi-transparent edges or different colors per class – this might be done via a lightweight GPU shader. Otherwise, a simple rectangle is fine. The key is to *minimize overhead* of any annotation drawing.

- **Search Keywords & Technical Tags:** `Jetson Orin Nano x264enc performance`, `NVIDIA hardware encoder support`, `GStreamer nvosd example`, `OpenCV VideoWriter Jetson`, `synchronizing video with metadata`, `real-time annotation overlay`, `DeepStream record annotated video`, `ROI extraction for logging`.

- **Benchmarks or Metrics to Evaluate: Encoding/Recording FPS** – ensure the recording pipeline keeps up with the source (e.g., if camera/inference produce 30 FPS, the recorder must handle 30 FPS without dropping frames). Monitor **CPU usage** during recording; on Orin Nano, encoding might consume one or more CPU cores fully [23] [24]. The goal could be to keep total CPU < 70% to leave headroom for other tasks. Check **output video quality and size**: measure bitrate and frame quality (PSNR or just visual inspection) for chosen encoder settings. If using ultrafast preset, expect larger file sizes – verify that your storage can hold the data for the required duration (e.g., X hours of footage). Also measure the **time alignment** between recorded video and metadata timestamps – e.g., does frame #1000 in video correspond to the 1000th inference output in the log. This sync should be frame-accurate; one way to test is to see if known events (object appears) match the annotation timeline. For labeling, a metric of success is completeness and correctness of the annotation log (no frames missing, all detections recorded). Another metric: **latency added by recording** – ideally the act of recording/annotating should not add more than a few milliseconds to the pipeline latency (if it's well-threaded, it can be almost zero added latency, just some CPU load). If the system starts dropping frames during heavy scenes, that's a sign the recording might be too slow – adjust parameters accordingly.

- **Sources to Explore:** NVIDIA forum discussions on Orin Nano encoding (e.g., *"Jetson Orin Nano Encoding"* threads – confirming no HW encoder, and solutions using x264 [19]). The RidgeRun wiki article *"Software Encoders for Jetson Orin Nano"* provides detailed benchmarks of x264 encoding on Orin (frame rates at various presets, CPU usage) [20] [21]. Look at **GStreamer pipeline examples** for recording, such as RidgeRun's references or JetsonHacks' blog (e.g., "GStreamer Network Streaming and Save to File with C920" [25], which uses a C920's H.264 stream directly). The DeepStream SDK documentation and samples (like deepstream-test4) show how to use `nvdsosd` and `nvv4l2encoder` / `filesink` to record video with bounding boxes in a few lines of code. If using OpenCV, refer to OpenCV's official docs for `VideoWriter` and known issues on Jetson (some builds need FFMpeg support). Also, resources on **synchronization**: e-con Systems' application note on "timestamping frames on Jetson" [26] might be useful to ensure alignment. Finally, general surveillance system design papers can give insight (e.g., how long to retain video, using motion-only recording, etc.) which ties into the next section on advanced features.

4. System Integration (Robust Multi-Threaded Pipeline Design)

- **Core Research Topics:** Designing the pipeline as a **robust, autonomous system** that can run continuously and handle multiple tasks concurrently. Questions: *What is the best threading or process architecture to connect capture → inference → output without bottlenecks?*; *How to handle errors (camera disconnects, model failures) and recover?*; *Can the pipeline run autonomously at boot and be managed (start/stop or remote monitoring)?*; *How to keep components modular and swappable?* We also consider inter-thread communication (queues, ring buffers) and synchronization primitives. The goal is a blueprint for a **modular pipeline** where each component (capture, inference, etc.) could be replaced or upgraded independently (e.g., swap the model, or use a different camera) – facilitated by clean interfaces between them.

- **State-of-the-Art Tools/Frameworks: Threading libraries** (C++11 `<thread>`, Python `threading` or `multiprocessing`, etc.) are fundamental. In C/C++, frameworks like **Boost::LockFree queues** or **OpenMP** (for simple parallel loops) might help. In Python, because of the GIL, one may use multiprocessing or offload heavy work to C/C++ extensions – however, frameworks like **TensorRT** release the GIL during compute, so a multi-threaded Python pipeline can still work (capture and infer threads can run in parallel). Another approach is **message passing frameworks**: e.g., using **ROS2** in a robotics context – each component (camera, detector, logger) could be a node publishing/subscribing messages. ROS gives a modular, distributed design but adds overhead; however, it's useful if future robotics integration is planned (the camera driver, inference, etc., can be independent nodes). There's also NVIDIA's **Isaac SDK**/ROS GEMs for GStreamer and vision, though that might be overkill. **GStreamer** itself can be seen as an integration framework – a properly constructed GStreamer pipeline runs elements in separate threads (sources, sinks, etc., often have their own thread). So if using GStreamer deeply, a lot of the multi-threading is handled internally (and one can insert custom plugins for custom logic). Outside of these, consider *supervisory tools*: **systemd** (to auto-start the pipeline on boot and restart on crash), or Docker containers for each component (some edge systems containerize each part for modularity – e.g., one container for capture service, one for inference service like Triton, etc., communicating via ZMQ or REST). For debugging and monitoring, one can integrate with **Telemetry frameworks** (e.g., tegrastats for resource usage, or custom logs). The design should ensure the pipeline can run headlessly and be controlled or observed remotely if needed.

- **Implementation Techniques: Multi-threaded design:** A common pattern is the *producer-consumer* model with thread-safe queues. For example, Thread A (capture) grabs frames and pushes to a queue; Thread B (inference) pops frames and processes them; Thread C (recorder) receives processed frames or metadata for output. Use **lock-free queues or ring buffers** if possible for efficiency. Make sure to handle the case where the queue is full (e.g., drop frames or overwrite the oldest – better to drop input frames than to let latency grow unbounded). Tuning the queue length is important: a small queue keeps latency low at risk of dropping frames under load; a large queue buffers bursts but increases latency. Employ **asynchronous I/O** where possible: e.g., camera reads using non-blocking calls, file writes with async APIs or double-buffer file buffers. Protect shared resources with proper locks or use message passing to avoid shared state. **Error handling & watchdogs:** The pipeline should detect if the camera feed stops (e.g., USB unplugged) – one strategy is to monitor timestamps and if no new frame in X seconds, attempt to reinit the camera. Similarly, if the inference thread throws an exception (e.g., TensorRT error), catch it and possibly restart that component or the whole pipeline gracefully. Use try-catch around main loops and possibly a

supervisory thread or external watchdog that can restart threads that die. **Resource management:** Pin certain threads to specific CPU cores to ensure critical threads (camera capture) have enough CPU and don't contend with encoding thread, for instance. Or use nice values/priorities (on Linux, `pthread_setschedparam` or `chrt` for real-time scheduling if needed) for time-critical threads. For example, give the capture thread a real-time priority to minimize frame jitter [27]. Balance this with not starving inference (which mostly uses GPU but still needs some CPU). **Synchronization between threads:** use condition variables or semaphores if a stage should wait for signal (but in a streaming pipeline, usually you try to avoid hard waits and just use queues). **Modularity:** define clear interfaces for each module – e.g., the capture thread could be abstracted behind a class with a `read()` method that yields frames; the inference module could be swappable (today TensorRT, tomorrow maybe an ONNX CPU fallback). One can use an interface like `DetectorBase` with a method `detect(frame) -> results`. This way, testing different inference engines is easier. Similarly, use configuration files or parameters for things like camera resolution, model type, etc., rather than hardcoding – facilitating swapping components. **Pipeline as a service:** For autonomy, the whole pipeline could be wrapped in a service that starts at boot. Using Docker can isolate dependencies (e.g., one container with a specific OpenCV/GStreamer setup). If using multiple processes (for isolation or using multiple languages), consider lightweight protocols: e.g., using **ZeroMQ** or **gRPC** to pass frames or results between processes. This could enable an architecture where the camera process runs in C++ (fast capture), and it sends frames to an inference server (maybe running Triton or a Python script) – more complex, but very modular and scalable (even across machines). Given the single-device scope, threads in one process are simpler and avoid IPC overhead, but it's good to design with an eye towards possible distribution.

- **Search Keywords & Technical Tags:** `producer-consumer thread C++ queue`, `Jetson multi-thread real-time`, `lock-free ring buffer video`, `pipeline scheduling real-time`, `ROS2 vs GStreamer for video`, `Docker deployment Jetson video analytics`, `watchdog camera feed`, `thread priority scheduling Linux`.

- **Benchmarks or Metrics to Evaluate: End-to-end latency** is a top metric here – measure from the time a frame is captured to the time it's output (either shown on screen, saved, or an alert generated). This latency should ideally remain constant and low (e.g., <100 ms for real-time perception). Breaking it down: capture latency + inference time + post-processing + queue delays + encoding. If multi-threaded properly, these can overlap, so the pipeline latency might just be dominated by the slowest stage plus small overhead. We want to see minimal *queuing delay* (i.e., frames not piling up in queues). **Throughput** should match the camera FPS; no stage should be a severe bottleneck that lowers overall FPS (unless by design you deliberately skip frames). **CPU/GPU utilization:** The pipeline should utilize resources efficiently (e.g., GPU near 80-90% during inference, CPUs perhaps each 50-70% rather than one at 100% and others idle). If one core is maxed out (e.g., the encoding thread), that's a sign to optimize or distribute load. **Memory usage:** track that buffers aren't leaking – in a long-running process, memory should plateau. Using tools or just monitoring `/proc` over time can ensure we don't have a slow leak (which could eventually crash an autonomous system). **Robustness metrics:** Uptime (can it run for 24/7 without issues?), and if using a watchdog, count of auto-restarts (ideally zero in steady state). For modularity, a qualitative metric is how easily one can swap a component: e.g., replace the detection model with minimal code change – test this by integration testing a new model. Also, measure **frame drop rate**: ideally 0% under normal load; if overload occurs (e.g., too many objects causing slow drawing), the system might drop frames to recover – that's preferable to lag. So you might set a policy like "no more than 5% frames dropped

during peak activity". **Synchronization correctness:** ensure multi-threading isn't introducing race conditions – e.g., check that the bounding boxes drawn correspond to the correct frame (no off-by-one issues). One can test this by putting an identifiable marker in the scene and verifying the metadata aligns.

- **Sources to Explore:** General articles on real-time pipeline design, such as *"Real-Time Video Pipelines: Techniques & Best Practices"* [27] [28] which discusses multithreading, buffering, and modular design (including an emphasis on asynchronous, non-blocking processing). Look at the NVIDIA example **multicamera pipeline** project (GitHub NVIDIA-AI-IOT/jetson-multicamera-pipelines) which uses threading to handle multiple video streams – even if you have one camera, the techniques are similar. ROS2 resources: for instance, a ROS2 demo of image pipeline on Jetson to see how they structure the nodes (though ROS introduces overhead, it showcases separation of concerns). NVIDIA's *JetsonHacks blog or Jetson Zoo* might have tutorials on writing multi-threaded C++ capture programs. Also, consider the **NVIDIA Nsight Systems** profiling tool – it's a source in itself to find bottlenecks in a multi-thread system (Nsight can show CPU threads and GPU kernels timelines). For error recovery strategies, searching the forums for "Jetson camera timeout" or "Jetson long-run stability" can yield insight (some users share scripts to restart the pipeline if something goes wrong). In summary, leverage concurrency design patterns from producer-consumer examples and the wealth of open-source code (for instance, look at how OpenCV's `VideoCapture` is implemented with buffers, or how GStreamer handles its internal threading) to guide a solid integration.

Code Snippet: Multi-Threaded Frame Pipeline (pseudo-code in C++ style for illustration)

```
// Shared queue for frames (between capture and inference)
ConcurrentQueue<Frame> frameQueue;
std::atomic<bool> done(false);

// Thread A: Camera capture
std::thread tCapture([&](){
    Camera cam(...);
    while (!done) {
        Frame frm = cam.grabFrame();
        if (!frameQueue.try_push(frm)) {
            // Queue full: drop frame to keep pipeline realtime
        }
    }
});

// Thread B: Inference
std::thread tInfer([&](){
    Detector model(...); // e.g. TensorRT engine
    while (!done) {
        Frame frm;
        if (frameQueue.try_pop(frm)) {
            auto results = model.infer(frm);  // run object detection
            // (Optionally push results to another queue for recorder/display)
        } else {
```

```
            std::this_thread::yield(); // or small sleep, nothing to do
        }
    }
});

// ... (Thread C for recording or display could similarly pop results)
// Join threads at shutdown:
tCapture.join();
tInfer.join();
```

Explanation: We have a lock-free `ConcurrentQueue` (could be a ring buffer) to pass frames from the capture thread to the inference thread. The capture thread drops frames if the queue is full, ensuring no backlog (this keeps latency bounded at the cost of some frames). The inference thread constantly tries to pop frames; if none are available (maybe capture is slower), it yields CPU. In a real implementation, use appropriate synchronization (condition variables or semaphore) instead of busy-wait, and handle thread shutdown cleanly. This pattern decouples the timing of capture and processing, utilizing multi-core CPU and keeping the GPU busy. It's a simple starting point; more sophisticated pipelines might extend this with multiple queues (e.g., another for results) and more threads (for encoding, etc.), or use thread pools.

5. Advanced Extensions (Scalability and Intelligent Features)

- **Core Research Topics:** Building on the core pipeline, what *extended features* can we incorporate? This includes **event-triggered processing** (only react or record on certain conditions), **edge-cloud integration** (offloading tasks or sending alerts to cloud services), and **Re-Identification (ReID)** or multi-object tracking to maintain identities over time. Questions: *How can the system detect and respond to "events" of interest (e.g., a person entering frame) in real-time?*; *How to integrate with cloud or remote systems for additional processing or monitoring?*; *What algorithms can re-identify objects across frames or even across cameras?*; *How to scale to multiple cameras or devices in the future?* Even if starting with one camera, consider these as forward-looking research areas. Another aspect: **multi-object tracking (MOT)** – linking detection boxes frame-to-frame to output trajectories and counts, which ties into ReID when persisting identity over longer gaps or across different cameras.

- **State-of-the-Art Tools/Frameworks: Event triggering:** often done via simple algorithms like motion detection (e.g., background subtraction with OpenCV) or by analyzing detection results (e.g., if an object of class "person" with confidence >0.9 is present, trigger an event). Libraries like **OpenCV** provide methods for motion detection (absdiff frames, accumulate background). NVIDIA's **DeepStream** has a feature called **NvDsAnalytics** (in DeepStream 6+ extensions) which can do region-of-interest counting, line crossing triggers, etc., on top of detection – useful for smart surveillance (though this might be more than needed). **Edge-cloud integration:** Tools include **MQTT** or **Kafka** for messaging – the pipeline can publish events or frames to a cloud broker. DeepStream, for instance, directly supports MQTT/Kafka output for metadata [29]. There's also **RESTful APIs** or webhooks: e.g., the device could send an HTTP request to a server when an event happens (like "intruder detected"). For video streaming to cloud, protocols like **RTSP** or **WebRTC** are common (e.g., stream the video to a central server or cloud for remote viewing). NVIDIA's Metropolis ecosystem even has an SDK for connecting to Azure or AWS IoT services [30]. **Re-Identification and Tracking:** For MOT on Jetson, **ByteTrack** and **SORT/DeepSORT** are popular open-source trackers that can run in real-time (ByteTrack uses detection IOUs and is quite fast; DeepSORT uses a lightweight ReID

network to re-identify). Christian Mills's example uses ByteTrack on Orin Nano to track objects with minimal overhead [31]. NVIDIA's DeepStream includes a **tracker** plugin (NvDCF or IOU tracker) which can do object tracking on the device (NvDCF can incorporate ReID features as well). For person ReID specifically, there are models like **OSNet** or others that produce an embedding for each detection; one can run a small ReID model on detected person crops to maintain IDs even if the person leaves and re-enters. There are research papers on real-time edge ReID [32] and even GitHub projects for Jetson (e.g., *"Human-ReID-in-NVIDIA-Jetson"* uses a ResNet50-based ReID model on Jetson TX2 [33]). Also, **NVIDIA Triton Inference Server** can be considered if you want to host multiple models (say detection and a secondary ReID model) and potentially allow cloud queries – Triton can run on Jetson and manage multiple models with APIs for remote requests [34] [35]. In a multi-camera future, frameworks like **NVIDIA DeepStream** shine, as they can handle multiple input streams and even multi-camera analytics (like aggregating ReID across cameras, though that's complex). *Edge-cloud:* If scaling to many devices, one might use cloud management: e.g., use **Kubernetes at the edge** or simply IoT hubs (AWS IoT Greengrass, etc.) to deploy and update pipelines on devices. For now, focus on a single device sending data out in a lightweight manner (MQTT messages, occasional images).

- **Implementation Techniques: Event-based triggers:** Implement a module that monitors the inference results stream. This could be a callback or just part of the main loop that checks each frame's detections for certain conditions. For example, to do motion-based triggering, maintain a background model and compute difference; if motion magnitude exceeds a threshold, and maybe confirm via the detector that an object of interest is present, then "trigger" – e.g., start recording (if not already), or send an alert. Ensure hysteresis (don't trigger repeatedly for the same event). Many surveillance systems use a buffer: record video continuously but only keep it if an event occurs (pre-buffer 5 seconds before motion, etc.). This could be implemented by always recording to a ring buffer in RAM and writing to disk on trigger. **Edge messaging:** On event, publish a message. For instance, use an MQTT client (Paho MQTT in C++/Python) to send a JSON payload: `{ "device": "Cam1", "event": "person_detected", "time": "...", "details": {...} }`. Ensure this is done asynchronously (so network latency doesn't stall detection; use a separate thread or non-blocking call). If sending images, you might publish a URL or a thumbnail instead of full-res image to the cloud to save bandwidth. Alternatively, keep the heavy video on edge and only send metadata (and perhaps an ID or short clip if needed). **Cloud offloading:** If some analytics are too heavy for the edge (say face recognition on a detected face), the pipeline could capture the ROI and send it to a cloud service for processing. This requires robust networking code and fallback (if no connection, maybe queue the request or skip). **ReID & tracking:** To integrate a tracker like DeepSORT, you would run an additional step after detection. This usually involves maintaining a list of active tracks, predicting new positions (Kalman filter), matching new detections to tracks (Hungarian algorithm using IoU or ReID feature distances). Implementation on Jetson should be careful with performance – use a lightweight ReID model or use the CNN features from the detector if possible to save computation. Some modern trackers (ByteTrack) avoid a deep ReID model by using detection scores cleverly – those are very fast (essentially just some bounding-box association logic) and can easily run at 30+ FPS with minimal CPU. If using DeepSORT with a neural ReID, perhaps run the ReID model only on a subset of frames or when needed. You can also leverage the GPU for that model; if the model is small (e.g., a 50-layer ResNet) it might still be fine. Optimize tracking by using C++ for the heavy math (there are C++ implementations of ByteTrack, etc.). **Modular extensibility:** Design the pipeline such that these features can be toggled. For example, event-triggering could be configured via a JSON config (e.g., enable/disable motion detection or set which object classes trigger recording). ReID/tracking might be an optional module layered on top of

detection – if disabled, the pipeline simply won't do the tracking step. This keeps the base pipeline flexible for different deployments (surveillance vs robotics might have different needs). **Power and resource trade-offs:** Advanced features may consume more resources – e.g., constant cloud communication can use network and CPU, ReID uses extra GPU. Consider dynamic enabling: e.g., only run ReID on certain frames (sampled) or when multiple cameras are active hand off heavy tasks to a more powerful node.

- **Search Keywords & Technical Tags:** `Edge AI MQTT alert`, `DeepStream IoT integration`, `motion detection OpenCV MOG2`, `ByteTrack Jetson`, `DeepSORT TensorRT`, `multi-camera ReID Jetson`, `Triton server Jetson edge`, `cloud offload edge computing`, `event-based video recording`.

- **Benchmarks or Metrics to Evaluate: Event latency:** how quickly does the system react to an event? If a person appears, the goal might be to raise an alert within e.g. 500 ms. This includes detection latency plus network send. Test by scenario (introduce an object and measure time for an alert to reach a server or user). **False positive/negative rate for triggers:** If using motion or specific object triggers, measure the reliability (e.g., does it ever trigger on noise? Does it miss actual events?). This is more qualitative but important for a deployable system. **Network utilization:** if sending data to cloud, measure bytes per second or per event. Ensure it's within limits (especially for cellular-connected devices). Possibly use a metric like "<1 MB per hour when no events, <5 MB per event" or similar, depending on requirements. **ReID accuracy and performance:** Evaluate ReID with metrics like Rank-1 accuracy if possible (on a test set or scenario). Also measure the ID persistence in tracking – e.g., does the tracker keep consistent IDs for an object throughout its presence? If multi-camera, measure if ReID correctly matches the same person between cameras (that's quite challenging and beyond single device scope unless cameras are on one device). **Performance overhead:** tracking and ReID will add computation – verify the FPS doesn't drop below requirements when these are on. For instance, if ByteTrack adds only 2 ms per frame, that's fine; if a heavy ReID model adds 10 ms, ensure you still meet real-time or consider lowering frequency of that step. **Scalability:** if planning to extend to multiple streams, one could simulate or actually test with multiple video inputs (if hardware allows) to see how linear the performance scales or if any contention (e.g., memory bandwidth) arises. The modular design's success can be gauged by how easily new features plug in – e.g., integrating a cloud message in the pipeline should not require rewriting the core, just attaching an observer to events.

- **Sources to Explore:** NVIDIA DeepStream documentation on **message brokers and cloud** (e.g., how to use MQTT/Kafka in DeepStream) [29] – even if not using DeepStream directly, their design shows best practices (like using a separate thread to batch and send messages). Look into open-source projects like **Azure IoT Edge with DeepStream** or **AWS IoT Greengrass Jetson** – these show end-to-end examples of edge devices sending data to cloud [30]. For ReID and tracking, research papers such as *"Real-time person re-identification and tracking on edge devices"* [32] could give algorithms that are optimized for speed. The GitHub repository we found, *Human-ReID-in-NVIDIA-Jetson* [33], provides scripts and a pipeline using TensorRT for detection and TensorFlow for ReID on Jetson – reviewing its approach (they mention using a ResNet50 and the Argus camera interface) can spark ideas. OpenMMLab's **MMTracking** library has implementations of many trackers – though not Jetson-specific, one could see which algorithms are light enough. Also, the **PEEPS platform** Medium story (using Triton for ReID) [34] might be insightful for serving models in a scalable way. Lastly, consider reading about **edge video analytics systems** in academic literature (keywords: edge video

surveillance architecture) to foresee how a single-device solution can slot into a larger system (for instance, an architecture where edge devices do detection and send only “alerts” to a central server that does heavier analytics or long-term data storage).

6. Evaluation Metrics & Performance Targets

In developing this pipeline, it’s critical to define what “success” means quantitatively. Below we summarize key **evaluation metrics** for each major component and overall system, along with target values or thresholds:

- **Frame Latency (End-to-End):** This is the time from a frame entering the system (captured by the camera) to the time the results for that frame are output (e.g., bounding boxes drawn and recorded, or alert sent). Target end-to-end latency for real-time analytics is typically **<100 ms** (for live view use-cases, humans can notice >100–200ms delay). For many robotics scenarios, even ~200 ms might be acceptable, but lower is better especially if feedback or tracking is involved. Measure this by timestamping frames at capture and again when processing is done. Aim to keep this latency stable under load.

- **Throughput (FPS):** The system should sustain the input frame rate. For example, with a 30 FPS camera, the pipeline should output ~30 FPS (or whatever frame rate is desired) consistently. A drop in throughput indicates a bottleneck. We set a goal of **30 FPS at 1080p** with our chosen model (or higher if using 720p or a lighter model). Use benchmarks like those in Section 2 to choose an appropriate model such that the inference stage doesn’t become the limiter. If a trade-off is needed, one might run the camera at a slightly lower FPS (e.g., 20 FPS) to guarantee processing can keep up with worst-case scenarios.

- **Accuracy Metrics:** Since this is an analytics pipeline, ensure the object detection accuracy is adequate. Use mAP (mean Average Precision) on a validation dataset relevant to the use-case (e.g., persons and vehicles for surveillance) to quantify detection performance. If employing tracking/ReID, use metrics like IDF1 (ID score) or tracking precision/recall. Define a minimum acceptable accuracy (e.g., “person detection mAP >= 0.5 on COCO or a relevant test set”). Hitting performance targets should not come at the cost of unacceptable accuracy loss – if INT8 quantization drops accuracy too much, consider FP16 or retraining with quantization-aware training.

- **Resource Utilization:** Monitor **GPU utilization** (ideally high during inference but should drop when idle) and **CPU utilization** per core. A balanced pipeline might show, for example, GPU ~80% busy during inference, one CPU core ~80-90% for encoding (if software encoding), and remaining cores ~50% handling other tasks, leaving a bit of headroom. If any component consistently maxes out (100%), that’s a potential bottleneck or point of failure under stress. Also check **memory usage**: both RAM and GPU memory. The Orin Nano 4GB should be enough for a single model and video frames (which might use ~1GB or less), but if using Python or additional models, be wary of memory bloat. Memory usage should stay within limits with no significant leaks over time.

- **Power and Thermal:** For embedded devices, power is key. Use a tool like `tegrastats` on Jetson to measure power draw. The Orin Nano can operate in different power modes; ensure the pipeline runs within the intended power budget (e.g., if in a 10W mode, not exceeding it on average). Thermal

throttling can occur if the device overheats, causing performance to drop. Thus, measure the device temperature after prolonged operation; the design should either include adequate cooling or manage workload to avoid hitting thermal limits. A metric could be "Device does not throttle (stays below 80°C) during 24/7 operation at room temperature."

- **Latency per Component:** In addition to overall latency, track latency of each stage: capture (e.g., camera driver + any conversion), inference, post-processing (drawing, etc.), encoding, etc. This helps identify bottlenecks. For instance, if inference is 20 ms but encoding is taking 40 ms per frame, the encoding stage might govern the throughput and latency. We might set targets like "Inference latency <= 30 ms; capture+preproc <= 10 ms; annotation+encode <= 30 ms" as a breakdown for 30 FPS. Use profiling tools (Nsight Systems, as mentioned, or OpenCV's tick timings if using that) to log these.

- **Robustness and Uptime:** Define metrics for system reliability. For example, *Mean Time Between Failures (MTBF)* – though hard to measure in development, we want no crashes or memory exceptions in days of testing. Another metric: **Frame Drop Rate** – ideally 0% under normal conditions, and even under stress (lots of motion or large numbers of objects) should be very low. If drops occur, quantify them and ensure they happen in a controlled way (e.g., our queue dropping mechanism intentionally drops frames when overwhelmed – measure that scenario to verify it recovers and doesn't snowball into increasing latency).

- **Maintainability (Qualitative):** While harder to quantify, one can assess maintainability by how modular and clear the system is. For instance, measure **lines of code per component** (smaller, well-defined components are easier to maintain), or time taken to swap in a new model or camera. You can simulate that by a small test: try replacing the model with a different one (say a smaller YOLO) and see how much effort/code change – if minimal due to abstraction, that's a success of the modular design.

In summary, "success" for the project means **meeting real-time performance (30 FPS, sub-100ms latency) with acceptable accuracy, under the device's power/thermal constraints, sustained reliably over long durations.** Each metric above should be monitored during testing. Tools like **tegrastats**, **nsys (Nsight)**, and custom logging can be used to gather these numbers. Only when the pipeline hits these targets should it be considered production-ready for the intended surveillance/robotics use-case.

- **Sources to Explore:** Refer to the arXiv paper for **edge object detection benchmarking** [36] [37] – it enumerates metrics like energy per inference and accuracy trade-offs. The It-Jim blog on video pipelines also highlights importance of measuring frame loss and latency under multithreading [27] [28]. NVIDIA's official Jetson performance tuning guides (e.g., *"Maximizing Jetson Performance"*) provide tips on measuring and achieving consistent throughput. Additionally, tools like **MLPerf Inference Benchmark** results for edge devices can give an external reference for what performance to expect on Orin Nano for given models. Monitoring frameworks (some use **Prometheus/Grafana** on edge for metrics) might be overkill, but worth knowing. Overall, use a combination of NVIDIA's guidelines and your own scenario-specific requirements to finalize the performance thresholds.

By following this roadmap, an engineer or researcher can methodically explore and implement each aspect of a real-time video analytics pipeline on Jetson Orin Nano (and similar devices). The emphasis on modular design,

hardware acceleration, and careful evaluation will ensure the final system is not only fast, but also robust, scalable, and adaptable to future needs (additional cameras, new models, or cloud integration). The cited resources and tools provide further depth for each topic, enabling a deep research-driven development process. [5] [29]

---

[1] [2] Jetson zero-copy for embedded applications | fastcompression.com
https://www.fastcompression.com/blog/jetson-zero-copy.htm

[3] [4] PSA, Logitech has removed Hardware H.264 Encoder from some WebCams : r/linux
https://www.reddit.com/r/linux/comments/f2icry/psa_logitech_has_removed_hardware_h264_encoder/

[5] [6] [12] [13] [18] [31] Deploying YOLOX for Real-Time Object Tracking on Jetson Orin Nano – Christian Mills
https://christianjmills.com/posts/pytorch-train-object-detector-yolox-tutorial/jetson-object-tracking/

[7] Logitech C920 PRO HD Webcam, 1080p Video with Stereo Audio
https://www.logitech.com/en-eu/shop/p/c920-pro-hd-webcam

[8] [9] [hardware acceleration] usb_cam how to develop a high performance hardware accelerated camera driver in jetson orin · Issue #320 · ros-drivers/usb_cam · GitHub
https://github.com/ros-drivers/usb_cam/issues/320

[10] Jetson Orin Nano having difficulties to run models on DLA(s)
https://forums.developer.nvidia.com/t/jetson-orin-nano-having-difficulties-to-run-models-on-dla-s/256974

[11] Best way to deploy object detection model in jetson orin nano - Jetson Orin Nano - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/best-way-to-deploy-object-detection-model-in-jetson-orin-nano/284693

[14] [15] [16] [36] [37] Benchmarking Deep Learning Models for Object Detection on Edge Computing Devices
https://arxiv.org/html/2409.16808v1

[17] Quick Start Guide: NVIDIA Jetson with Ultralytics YOLO11
https://docs.ultralytics.com/guides/nvidia-jetson/

[19] [22] High-Speed Low-Latency Video Transmission Methods on Jetson Orin Nano - Jetson Orin Nano - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/high-speed-low-latency-video-transmission-methods-on-jetson-orin-nano/320352

[20] [21] [23] [24] NVIDIA Jetson Orin AGX - JetPack 5.0.2 - Performance Tuning - Software Encoders For Jetson Orin Nano
https://developer.ridgerun.com/wiki/index.php/NVIDIA_Jetson_Orin/JetPack_5.0.2/Performance_Tuning/Software_Encoders_For_Jetson_Orin_Nano

[25] GStreamer Network Video Stream and Save to File - JetsonHacks
https://jetsonhacks.com/2014/10/28/gstreamer-network-video-stream-save-file/

[26] NVIDIA Jetson TX2 Video Input Timing Concepts & Tutorials
https://developer.ridgerun.com/wiki/index.php/NVIDIA_Jetson_TX2_-_Video_Input_Timing_Concepts

[27] [28] Real-Time Video Pipelines: Techniques & Best Practices - it-jim
https://www.it-jim.com/blog/practical-aspects-of-real-time-video-pipelines/

29 NVIDIA DeepStream 7.0 Milestone Release for Next-gen Vision AI Development - Edge AI and Vision Alliance
https://www.edge-ai-vision.com/2024/05/nvidia-deepstream-7-0-milestone-release-for-next-gen-vision-ai-development/

30 How to integrate NVIDIA DeepStream on Jetson Modules with AWS ...
https://aws.amazon.com/blogs/iot/how-to-integrate-nvidia-deepstream-on-jetson-modules-with-aws-iot-core-and-aws-iot-greengrass/

32 Real-time person re-identification and tracking on edge devices with ...
https://dl.acm.org/doi/abs/10.1007/s10044-025-01492-z

33 GitHub - anonymous1computervision/Human-ReID-in-NVIDIA-Jetson: Person ReID(Re Identification) in NVIDIA Jetson TX2 for real time algorithm
https://github.com/anonymous1computervision/Human-ReID-in-NVIDIA-Jetson

34 35 Implementing a pipeline for Person Re-Identification (ReID) with NVIDIA's Triton Inference Server | by d*classified | d*classified | Medium
https://medium.com/d-classified/using-nvidias-triton-inference-server-for-person-re-identification-reid-f67013a9761a