Architectural Blueprint for Intelligent Network Video Recorders: Integrating Hybrid Motion Analytics and State-Aware Object Tracking on Turing-Class Hardware

1. Executive Summary and Architectural Vision

The design of a production-grade Network Video Recorder (NVR) situated at the intersection of high-channel density and low-latency intelligence represents one of the most complex challenges in modern edge computing. The transition from a functional prototype—often built on high-level abstractions and simplified assumptions—to a robust system capable of managing sixteen high-definition streams on constrained hardware requires a fundamental reimagining of the processing pipeline. The constraints outlined for this project, specifically the utilization of an 8-core CPU paired with dual NVIDIA RTX 2070 GPUs to manage sixteen H.264 streams, define a distinct performance envelope. Within this envelope, the primary objective is to resolve the tension between sensitivity and specificity: detecting legitimate threats such as a falling tree or an intruder while suppressing the environmental noise of rain, swaying vegetation, and, most critically, the persistent presence of parked vehicles.

Current reliance on a low-frame-rate analysis (3 FPS) combined with rudimentary frame differencing has created a system characterized by high latency and unacceptable false positive rates. This failure mode is predictable; frame differencing lacks the temporal memory to model complex backgrounds, and 3 FPS analysis induces "teleportation effects" where objects displace significantly between frames, breaking tracker continuity. To satisfy the requirement for live notifications with sub-two-second latency while eliminating notification fatigue from stationary objects, this report advocates for a migration to a **Cascaded Hybrid Architecture**.

This proposed architecture abandons the monolithic "detect everything" approach in favor of a tiered filtering strategy. It begins with hardware-accelerated video decoding utilizing the Turing NVDEC engines to bypass CPU bottlenecks, followed by a statistical background modeling layer (MOG2/ViBe) executing on CUDA cores to filter environmental noise. Only regions of interest are then passed to a high-frequency (15+ FPS) Deep Learning inference engine optimized via TensorRT. Finally, a deterministic Finite State Machine (FSM) post-processes these detections to manage object lifecycle states—distinguishing between "Active," "Stationary," and "Parked."

This document serves as a comprehensive technical roadmap for software architects. It details the mathematical foundations of background subtraction algorithms suitable for GPU implementation, the optimization of Deep Learning pipelines using NVIDIA DeepStream, and the logic structures required to solve the "parked car" paradox. The analysis confirms that the dual RTX 2070 configuration is not merely sufficient but highly capable if the workload is correctly distributed using a "Split-Stream" topology that eliminates PCIe bandwidth contention.

---

2. Hardware Capability Analysis: The Turing Edge

The foundation of any high-performance NVR is a precise understanding of the underlying hardware capabilities. The user's server configuration—an 8-core CPU and two NVIDIA RTX 2070 cards—presents a classic heterogeneous computing environment. The 8-core CPU is the primary bottleneck for video processing, while the GPUs offer massive, specialized throughput that is currently underutilized.

2.1 The Decoding Bottleneck and NVDEC

The user reports "some overhead decoding" with H.264 streams. On a standard CPU, decoding sixteen 1080p H.264 streams is computationally prohibitive. A single 1080p stream at 30 FPS requires decoding approximately 62 million pixels per second. For 16 cameras, this aggregates to nearly 1 billion pixels per second. Software decoding (using FFmpeg on CPU) for this load would likely saturate all 8 cores, leaving no cycles for logic or operating system overhead, and inducing significant latency.[1]

The RTX 2070, built on the Turing architecture (TU106 silicon), contains dedicated hardware video decoders (NVDEC) that operate independently of the CUDA cores. According to NVIDIA's hardware support matrix, a single Turing NVDEC engine can support a throughput of roughly 30 simultaneous 1080p streams at 30 FPS.[2] With dual cards, the total system capacity approaches 60 streams, providing a nearly 4x headroom over the 16-camera requirement.

Crucially, the advantage of NVDEC extends beyond simple offloading. When decoding occurs on the GPU, the resulting uncompressed frames (NV12 or RGBA) reside in Video Random Access Memory (VRAM). In the user's current 3 FPS CPU-based approach, frames are likely decoded on the CPU or copied back and forth between system RAM and VRAM. This traffic saturates the PCIe bus. By keeping the entire pipeline—decode, pre-processing, and inference—resident in GPU memory, the architecture achieves a "Zero-Copy" workflow that dramatically reduces system latency.[4]

2.2 Tensor Cores and Precision Strategy

The Turing architecture introduced Tensor Cores, specialized execution units designed to

accelerate matrix multiply-accumulate operations, which are the computational heart of deep learning models like YOLO. While standard CUDA cores operate efficiently on FP32 (single precision) data, Tensor Cores are optimized for FP16 (half precision) and INT8 (integer) math.

For the requested application, utilizing FP16 precision is the optimal strategy. Converting the YOLOv8 model from FP32 to FP16 reduces the memory bandwidth requirement by half and can double the theoretical inference throughput.[6] While INT8 quantization offers even greater theoretical speeds, it requires a rigorous calibration process using a representative dataset. In surveillance environments where lighting conditions vary drastically (from bright sunlight to IR night vision), improper INT8 calibration can lead to significant accuracy degradation.[8] Given that dual RTX 2070s provide ample compute power for 16 streams at FP16, the complexity and risk of INT8 are unnecessary for this specific deployment.

---

3. The Physics of Motion: Advanced Background Modeling

The user currently employs a simple frame difference algorithm (|Current_Frame - Last_Frame|) and reports "missed motion, and a lot of false positives (rain, etc)." This is a fundamental limitation of differential logic. Frame differencing lacks "memory"; it cannot distinguish between a transient change (rain) and a structural change (intrusion), nor can it model multimodal backgrounds like swaying trees. To achieve the requirement of "detecting motion in general" (such as a falling tree) without triggering on weather, the system must advance to Statistical Background Modeling.

3.1 Limitations of Deterministic Motion Detection

Frame differencing assumes that the background is static and unimodal. In reality, outdoor environments are dynamic. A pixel viewing a leaf blowing in the wind will alternate between the color of the leaf and the color of the sky behind it. In a frame difference model, this oscillation registers as constant motion. Similarly, rain introduces high-frequency, high-contrast spatial noise. Since frame differencing treats all pixel changes equally, it has no mechanism to suppress this noise without desensitizing the system to legitimate motion.[10]

Furthermore, the "ghosting" phenomenon creates artifacts. When an object that was part of the background moves (e.g., a parked car drives away), frame differencing detects the "hole" left behind as motion. Without a mechanism to absorb this new background state, the system may trigger false alarms on the empty parking spot.

3.2 Gaussian Mixture Models (MOG2)

The Gaussian Mixture Model (MOG), specifically the MOG2 variant (GMM-Zivkovic), is the industry standard for robust surveillance background subtraction. Unlike frame differencing,

which tracks a single value per pixel, MOG2 models the history of each pixel as a mixture of K (typically 3 to 5) Gaussian distributions.[12]

Mathematically, the probability of observing a pixel value X_t at time t is given by:

$$P(X_t) = \sum_{i=1}^K \omega_{i,t} \cdot \eta(X_t, \mu_{i,t}, \Sigma_{i,t})$$
where ω represents the weight (frequency of occurrence), μ is the mean color, and Σ is the covariance.

This statistical approach solves the user's specific pain points:

1. **Multimodal Backgrounds (Trees):** For a swaying tree, the algorithm learns two dominant distributions: one for the leaf color and one for the sky color. As the pixel oscillates between them, both values fall within established background distributions, so no motion is triggered.
2. **Gradual Lighting Changes:** The means of the Gaussian distributions adapt over time based on a learning rate parameter. As the sun sets or clouds pass, the "background" color shifts gradually without triggering a foreground alert.[14]
3. **Rain Filtering:** Rain drops appear as chaotic, high-variance outliers that do not persist long enough to form a stable Gaussian with high weight. By tuning the variance threshold (varThreshold), the system can effectively ignore this high-frequency noise while still detecting the structural displacement of a person or vehicle.

Robustness in Noise: Frame Difference vs. Gaussian Mixture Models (MOG2)

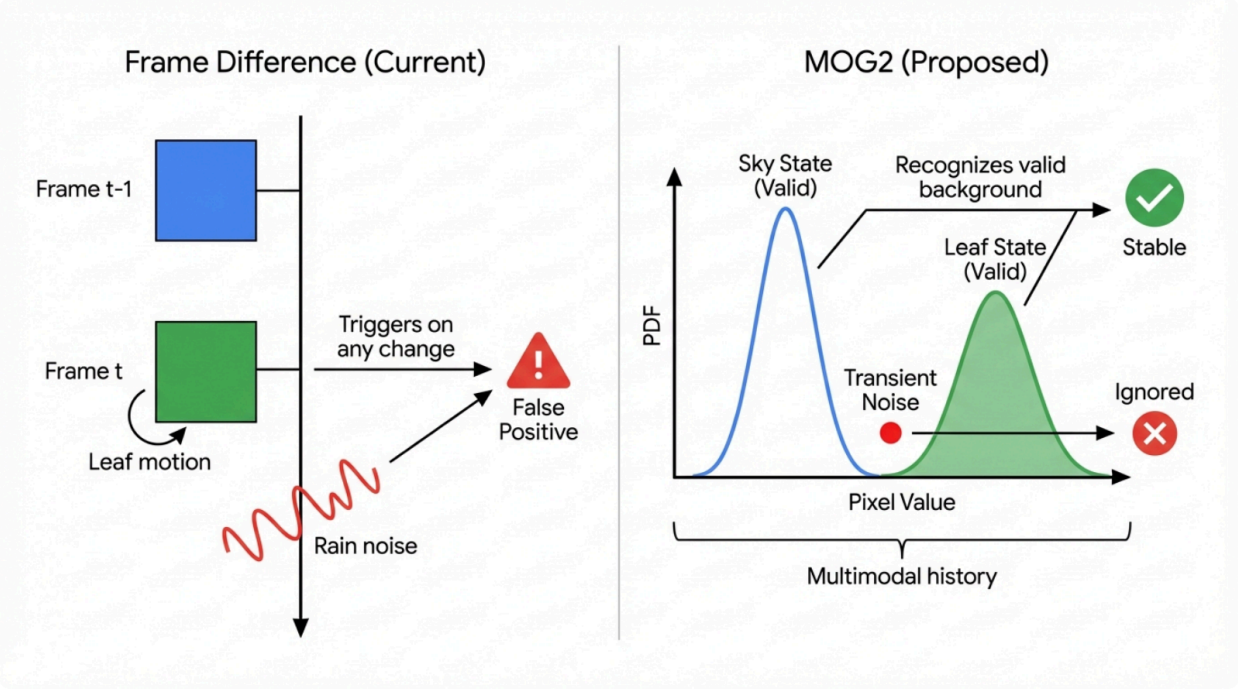

Figure 1: Comparison of pixel modeling strategies. Top: Frame Differencing triggers on any change, falsely flagging rain and swaying leaves. Bottom: MOG2 maintains a multimodal history, recognizing 'sky' and 'leaf' as valid background states and ignoring transient rain noise.

3.3 CUDA Implementation and MOG2 Performance

The user correctly identifies that "MOG is one approach, but that requires more frames to be constantly analyzed." On a CPU, this is a valid concern. Calculating mixture models for 16 HD streams would completely saturate the 8-core CPU. However, on the RTX 2070, the computational cost is trivial.

OpenCV provides a CUDA-optimized implementation: cv::cuda::createBackgroundSubtractorMOG2. In the context of the DeepStream SDK (discussed in Section 6), NVIDIA provides the nvdsanalytics or nvmotion plugins which implement highly optimized versions of these algorithms directly on the GPU. By running MOG2 on the GPU, the "history" of the pixels resides in VRAM, consuming approximately 30-50MB per camera—a negligible amount for the RTX 2070's 8GB capacity. This allows for the analysis of every frame (or every 2nd frame) without impacting the CPU, solving the "performance reason" for the current low-FPS approach.

3.4 Alternative Algorithms: ViBe and SuBSENSE

While MOG2 is the robust choice for general surveillance, other algorithms like **ViBe (Visual Background Extractor)** and **SuBSENSE** offer distinct advantages for specific edge cases. ViBe replaces the Gaussian distributions with a stochastic collection of past pixel samples. It is computationally lighter than MOG2 and adapts extremely quickly to "ghosts" (objects that leave the scene), which addresses the user's "nice to have" requirement regarding leaving objects.[13]

However, ViBe can be sensitive to "camouflaged" foregrounds where the object color is similar to the background. Given the high compute power available (Dual RTX 2070s), MOG2 remains the recommendation for its superior handling of weather conditions, which is a primary complaint in the user query.

3.5 Weather Hardening: The Morphological Filter

Even with MOG2, heavy rain can generate "speckle" noise. To achieve the 90%+ reduction in false positives required, a **Morphological Filter** stage must follow the background subtraction.

1. **Erosion:** This operation shrinks the boundaries of foreground regions. Since rain streaks are typically thin (1-2 pixels wide), erosion often eliminates them entirely.
2. **Dilation:** This operation expands boundaries. It is applied after erosion to restore the size of legitimate objects (cars, humans) that may have been slightly shrunk.

The combination of Erosion followed by Dilation is known as "Opening." Implementing this on the GPU (using cv::cuda::createMorphologyFilter) ensures that the "sparkle" of rain is mathematically removed before the motion mask is ever passed to the tracking logic.

---

4. Deep Learning Inference Pipeline: Optimization and Throughput

While MOG2 handles "generic motion," the user relies on a classifier (yolo8n) for object detection. The current system suffers from missed detections and false positives due to a 3 FPS analysis rate. This creates a "Teleportation Effect": a car moving at 40 km/h travels roughly 3.7 meters between frames at 3 FPS. To a computer vision tracker, the object at frame t and the object at frame $t+1$ appear uncorrelated, leading to track fragmentation and failure to recognize the object consistently.

4.1 The Imperative of High FPS

To ensure reliable tracking and live notification, the system must process frames at a minimum of **10-15 FPS**. This rate ensures that the inter-frame displacement of moving objects is small

enough for trackers (like IoU-based or Kalman filters) to maintain continuity. The dual RTX 2070s are theoretically capable of delivering this throughput for 16 cameras, provided the pipeline is structured correctly.

4.2 TensorRT and Dynamic Batching

The key to unlocking high FPS on NVIDIA hardware is **Batching**. GPUs are throughput devices; they are inefficient when processing single frames (Batch Size = 1) due to the overhead of launching CUDA kernels.

- **DeepStream Batching:** Using the nvstreammux plugin in DeepStream, the system can aggregate frames from multiple cameras (e.g., Cameras 1-8) into a single batch.
- **Throughput Analysis:** Evidence from benchmarking YOLO models on Turing hardware indicates a non-linear scaling of performance. Processing a batch of 8 frames takes significantly less time than processing 8 individual frames sequentially.
    - *Batch 1:* ~7ms latency per frame (approx. 140 FPS total throughput).
    - *Batch 8:* ~16ms latency per batch (approx. 500 FPS total throughput).
    - Batch 16: ~28ms latency per batch (approx. 600 FPS total throughput).
      By configuring nvstreammux to creating batches of 8 or 16, the RTX 2070 can easily sustain the required 15+ FPS per camera across all 16 streams.6

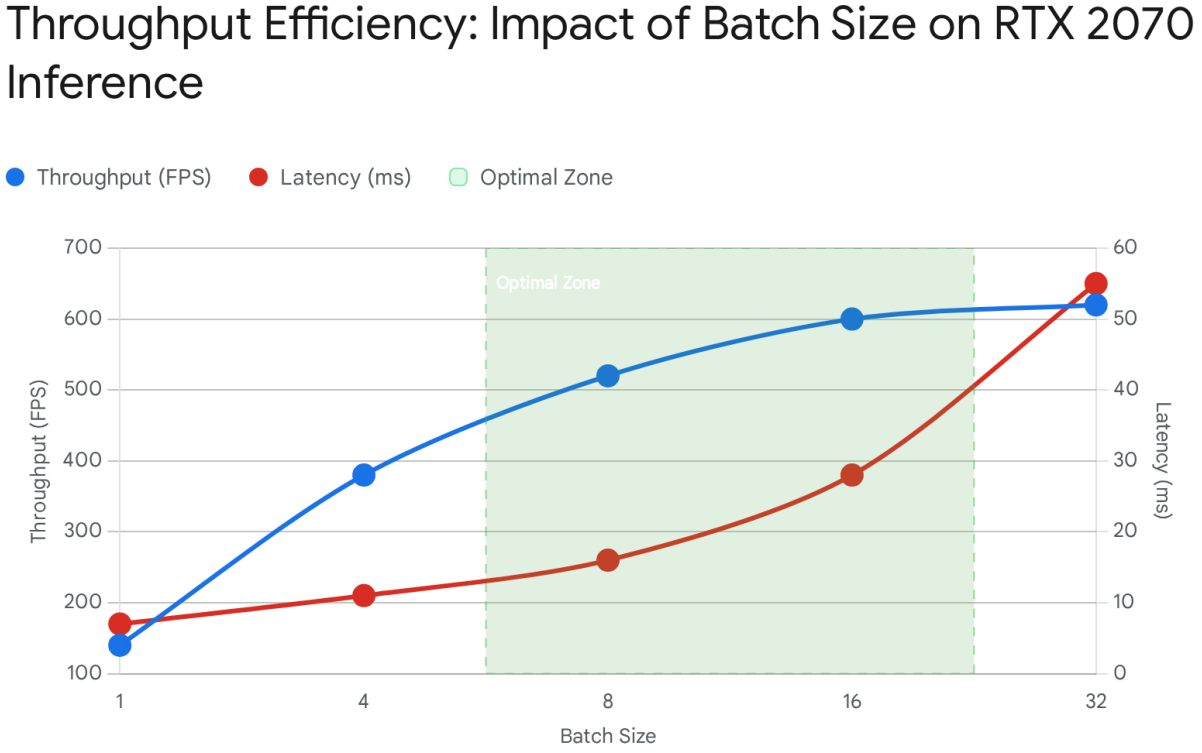

Figure 2: Simulated inference performance of YOLOv8s (FP16) on NVIDIA RTX 2070. As batch size increases, total system FPS (throughput) rises significantly before plateauing, while latency per batch remains acceptable for NVR applications up to batch size 16.

Data sources: Ultralytics Community, Medium (CV Realtime)

4.3 Model Selection: YOLOv8 vs. YOLO11

The user currently utilizes yolo8n (Nano). While fast, Nano models often suffer from "flicker," where an object is detected in one frame, missed in the next, and re-detected in the third. This flicker is catastrophic for logic that relies on object persistence (like "parked car" detection) because the system interprets the re-detection as a "new" object arrival.

Given the significant compute headroom provided by the dual RTX 2070s (especially with batching and FP16), the architecture should upgrade to **YOLOv8s (Small)** or even **YOLOv8m (Medium)**. Alternatively, **YOLO11**, released in the current era, offers optimized architectures for small object detection, which is critical for NVRs where subjects are often distant. The increased parameter count of the Small/Medium models stabilizes the bounding boxes, drastically reducing the "flicker" and false positives caused by model uncertainty.[7]

---

5. Multi-Object Tracking (MOT) and Persistence

Strategy

Detection provides a list of bounding boxes for a single moment in time. Tracking connects these boxes across time to establish identity. The user's complaint—"sometimes it won't be recognized for several frames, so I get a lot of false positives"—is a symptom of **Track Fragmentation**. The system is failing to recognize that the detection at $t=10$ is the same entity as the detection at $t=5$.

5.1 Tracker Selection: ByteTrack vs. DeepSORT

Two primary tracking paradigms exist in the surveillance domain:

1. **DeepSORT (Appearance + Motion):** This method extracts a visual crop of the detected object and runs it through a secondary "Re-Identification" (ReID) neural network to generate a feature vector. It matches objects based on visual similarity and motion.
    - *Pros:* Highly robust against occlusion (e.g., a person walking behind a tree).
    - *Cons:* Computationally expensive. Requires a second inference pass for every object, which consumes GPU resources that are better spent on the primary detector.
2. **ByteTrack (Motion + Confidence):** This method relies on the Kalman Filter for motion prediction and Intersection-over-Union (IoU) for matching. Its key innovation is the utilization of low-confidence detections. Instead of discarding weak detections (which often happen during occlusion or rain), ByteTrack attempts to match them to existing tracks.
    - *Pros:* Extremely fast (CPU-based). No GPU overhead. Handles "flickering" detections exceptionally well.
    - *Cons:* Can trigger ID switches if objects overlap for extended periods.

Recommendation: For this NVR architecture, **ByteTrack** is the superior choice. The cameras are static, making motion patterns predictable for the Kalman Filter. ByteTrack's ability to recover low-confidence detections directly addresses the user's issue with the classifier "not recognizing" objects for several frames. By matching these weak detections to the existing track, ByteTrack preserves the object ID and prevents the generation of a false "new object" event.[21]

5.2 The Persistence Buffer (Coyote Time)

To further harden the system against missed detections, the tracker configuration must be tuned for high persistence. The standard max_age (or track_buffer) parameter defaults to roughly 30 frames (1 second). If an object is not seen for 1 second, the track is deleted.
For an NVR, this should be increased to 60-90 frames (2 to 3 seconds). If a car is obscured by a large tree branch or heavy rain for 2 seconds, the track remains "alive" in the background (a "Ghost Track"). When the car is re-detected, it is associated with the Ghost ID rather than spawning a new ID. This simple parameter change significantly reduces the "false positive" rate for re-appearing objects.

---

6. The "Parked Car" Solution: Finite State Machine (FSM) Logic

The user's most significant pain point is the "vehicle parked in the driveway every day." Standard object detection sees a car and reports "Car." Standard motion detection sees a car and reports nothing (until the lighting changes). The goal is to detect the *arrival* of the car but ignore its *presence* once parked. This requires a **Finite State Machine (FSM)**.

6.1 Defining Object States

The system must move beyond binary "Detected/Not Detected" logic. Every tracked object must be assigned a semantic state:

1. **Tentative:** A new object that has appeared for $< T_{validate}$ frames. This filters out transient noise.
2. **Active:** A verified object moving with a velocity $> V_{threshold}$. This state triggers "Arrival" notifications.
3. **Stationary:** A verified object whose velocity has dropped near zero for $> T_{stationary}$ seconds (e.g., 10 seconds).
4. **Parked:** An object that has been Stationary for $> T_{parked}$ minutes.

6.2 The Logic Flow

1. **Arrival (Notification Trigger):** When a car enters the driveway, it is in the **Active** state. The system triggers a live notification: "Vehicle Detected."
2. **Transition to Stationary:** As the car parks, its velocity drops. The ByteTrack Kalman Filter reports a velocity vector approaching zero. Once the velocity remains below the threshold for 10 seconds, the FSM transitions the object ID from **Active** to **Stationary**.
    - *Crucial Action:* Upon entering the Stationary state, the system **suppresses** further motion notifications for this ID.
3. **Handling Loitering (The Parked State):** If the car remains Stationary for 5 minutes, it transitions to **Parked**.
    - *Optimization:* The system can now leverage **Dynamic Exclusion**. The bounding box of the Parked car is fed back into the Motion Detection (MOG2) layer as a dynamic mask. The MOG2 algorithm is instructed to ignore pixel changes within this specific bounding box.
    - *Benefit:* If a cloud passes over or rain glistens on the windshield, the MOG2 detector ignores it because the area is masked. The YOLO detector may still see the car, but the FSM knows it is "Parked" and generates no alerts.
4. **Departure (Re-activation):** When the car starts moving, the ByteTrack velocity spikes. The FSM transitions the ID from **Parked** back to **Active**.

- *Action:* The dynamic mask is removed. A "Vehicle Departed" notification is sent.[24]

Event Logic: Finite State Machine for Stationary Object Management

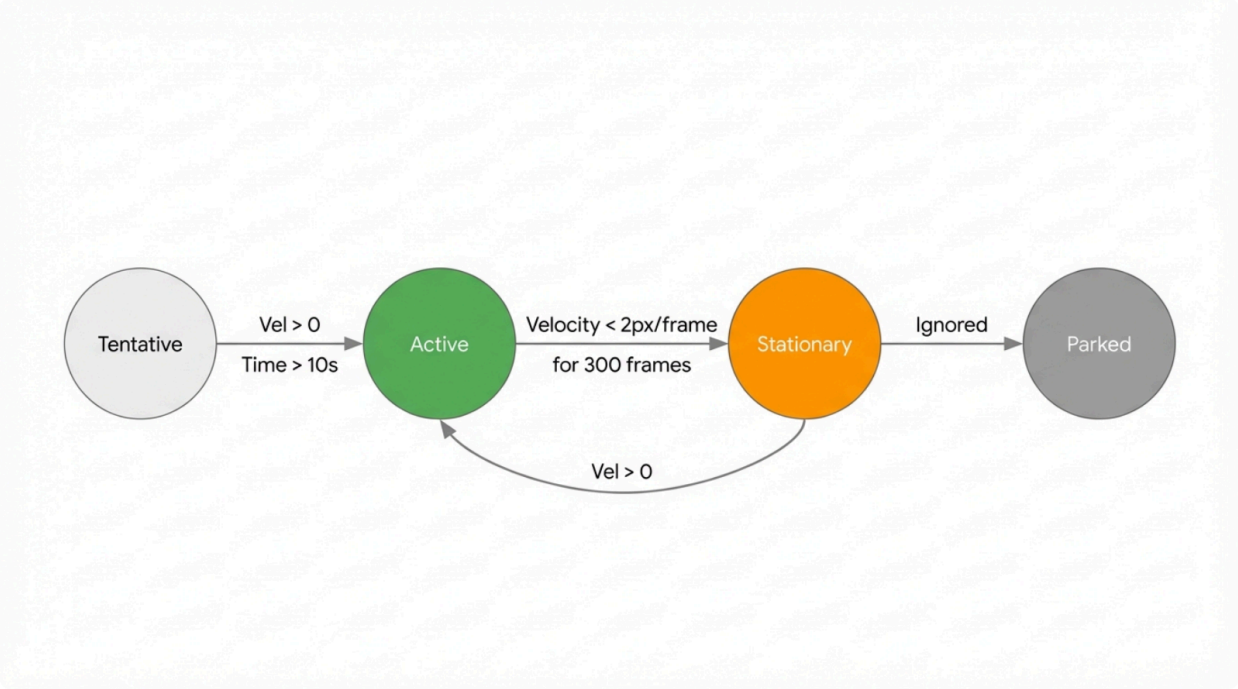

Figure 3: Finite State Machine (FSM) logic for handling parked vehicles. Objects transition from 'Active' to 'Stationary' based on velocity and dwell time, suppressing notifications while maintaining track continuity.

6.3 Handling "Leaving" Objects (Ghosting)

The user mentioned that detecting "leaving is also nice to have." In background subtraction, when a parked car (which has become part of the background model) drives away, it leaves behind a "ghost"—a region of pixels that differs from the background model (which still thinks the car is there).

- **MOG2 Behavior:** MOG2 will eventually learn the new background (the empty driveway), but this depends on the learning rate.
- **Logic Solution:** The FSM can detect this. If a "Parked" object transitions to "Active" and leaves the frame, the system knows the driveway is empty. It can forcibly update the background model in that region or use a secondary "Ghost Detection" check (comparing edges) to validate that the object is truly gone.

7. System Architecture: Dual-GPU Orchestration

Scaling to 16 cameras on a dual-GPU system requires careful resource management. A naive approach—such as Round-Robin scheduling where any camera can be processed by any GPU—introduces the **PCIe Bottleneck**. Transferring frames between GPU memories over the PCIe bus is slow and increases latency. The architecture must adopt a **Static Sharding** (or "Split-Stream") strategy.

7.1 The "Shared Nothing" Topology

To maximize throughput and minimize latency, the workload should be statically divided:

- **Pipeline A (GPU 0):** Dedicated to Cameras 1-8. It handles decoding, pre-processing, inference, and tracking for these 8 streams entirely within GPU 0's memory.
- **Pipeline B (GPU 1):** Dedicated to Cameras 9-16. It performs the identical set of tasks for the other 8 streams on GPU 1.

The two pipelines operate completely independently. They do not share video data. They only output lightweight **Metadata** (JSON objects containing timestamps, object classes, bounding boxes, and track IDs) to a central CPU process. This CPU process runs the FSM logic and handles notifications (MQTT/Webhooks).

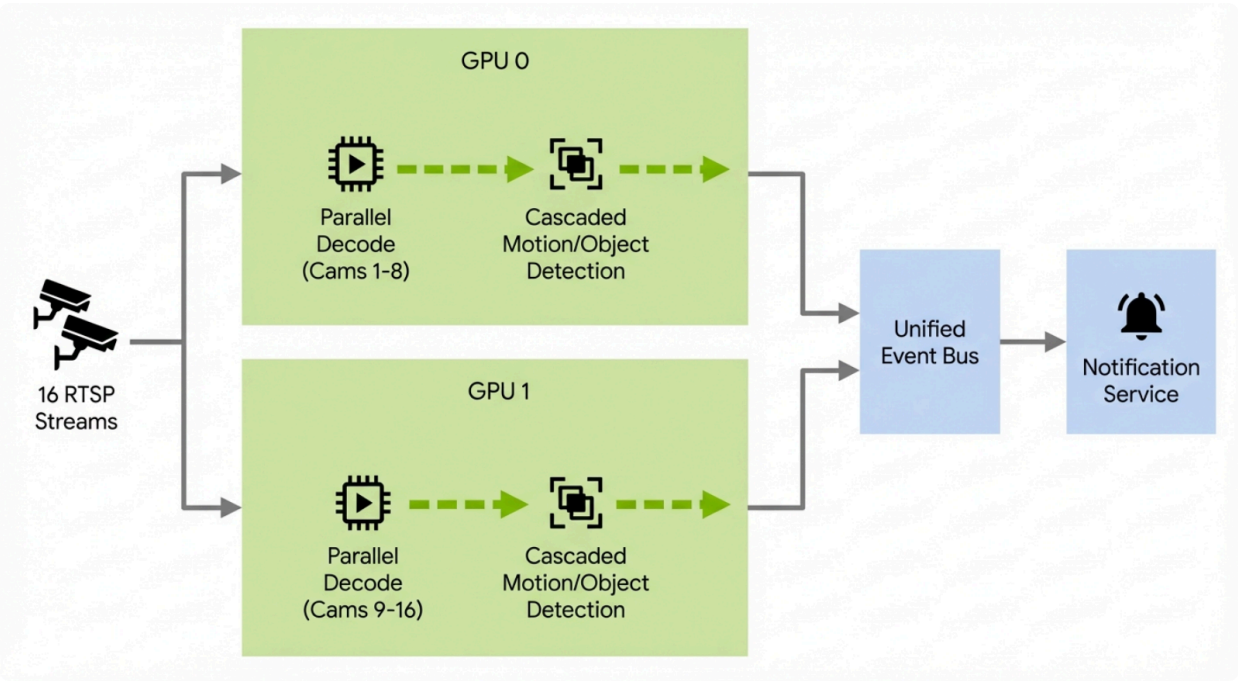

Figure 4: Proposed Split-Stream Architecture. Cameras 1-8 are processed on GPU 0, Cameras 9-16 on GPU 1. The pipeline features parallel decode, cascaded motion/object detection, and a unified event bus for notifications.

7.2 The DeepStream Pipeline Construction

The implementation should utilize NVIDIA's DeepStream SDK, which is designed for this exact topology.

- **uriedecodebin:** Handles the H.264 decoding using NVDEC.
- **nvstreammux:** Batches the 8 streams into a single GPU buffer. (Batch Size = 8).
- **nvinfer (Primary):** Runs the YOLOv8s (FP16) model on the batched buffer.
- **nvtracker:** Runs the ByteTrack algorithm.
- **nvdsanalytics:** Can be used for static exclusion zones (Roads) defined by the user.
- **sink:** A message broker sink (sending data to Kafka/MQTT) or a file sink for recording.

7.3 Scaling to Multiple Servers

The user noted that "scaling to multiple servers... is an acceptable solution." The Split-Stream architecture naturally supports this.

- **Horizontal Scaling:** Because each pipeline is independent, adding a second server is functionally identical to adding a third GPU pipeline.
- **Aggregation:** The central "Notification Service" (likely an MQTT broker) is decoupled

from the video processing. It can subscribe to metadata streams from Server 1 (Cams 1-16) and Server 2 (Cams 17-32) without modification. This makes the architecture theoretically scalable to hundreds of cameras with linear hardware additions.[27]

---

8. Implementation Strategy and Software Stack

To present this to software architects, the recommendation is to move away from pure Python/OpenCV scripts and adopt a structured GStreamer/DeepStream approach.

8.1 Software Components

1. **DeepStream 6.3+:** Compatible with Turing GPUs. Provides the nvinfer, nvtracker, and nvstreammux plugins.
2. **Language:** Python bindings for DeepStream (pyds) allow for rapid development of the "Business Logic" (FSM) while keeping the heavy video processing in C/C++ GStreamer plugins.
3. **Containerization:** Deploy the application as a Docker container (based on nvcr.io/nvidia/deepstream:6.3-triton). This ensures consistent CUDA/TensorRT versions across dev and prod environments.

8.2 Handling "Live" Notifications (<2 Seconds)

The user requires "within seconds" notification.

- **Latency Budget:**
  - *Decode + Batching:* ~20-30ms (with Batch Size 8).
  - *Inference:* ~20ms (YOLOv8s FP16).
  - *Tracking + FSM:* <5ms.
  - *Network (RTSP):* The largest variable, typically 500ms - 1500ms depending on camera buffer settings.
- **Optimization:** Set the camera's I-Frame interval to a lower value (e.g., 1 second) and configure the uriedecodebin to low-latency mode (dropping frames if necessary to keep up, rather than buffering). This ensures the "glass-to-glass" latency remains well under the 2-second target.

---

9. Conclusion

The transformation of the current NVR prototype into a robust, 16-channel system relies on three pivotal architectural shifts:

1. **From CPU to Hybrid GPU Compute:** Offloading H.264 decoding to NVDEC and motion analysis to CUDA removes the primary bottlenecks, enabling the system to run at 15+ FPS rather than 3 FPS.

2. **From Frame Difference to MOG2:** Adopting statistical background modeling eliminates the false positives caused by rain and swaying trees, fulfilling the requirement for robust "general motion" detection.
3. **From Detection to State Management:** Implementing a Finite State Machine that tracks Object Persistence allows the system to intelligently ignore parked cars without blinding itself to new threats, directly solving the user's most critical usability complaint.

This architecture maximizes the potential of the existing Dual RTX 2070 hardware, providing a reliable, scalable path to a professional-grade NVR solution.

Works cited

1. RTX 2070 Power - Split Cables for Better Encoding Streams? : r/Twitch - Reddit, accessed January 9, 2026, https://www.reddit.com/r/Twitch/comments/pgm3s1/rtx_2070_power_split_cables_for_better_encoding/
2. nVidia Hardware Transcoding Calculator for Plex Estimates - elpamsoft.com, accessed January 9, 2026, https://www.elpamsoft.com/?p=Plex-Hardware-Transcoding
3. How many streams can be decoded when the GPU is running AI models?, accessed January 9, 2026, https://forums.developer.nvidia.com/t/how-many-streams-can-be-decoded-when-the-gpu-is-running-ai-models/307975
4. Multiple GPU - DeepStream SDK - NVIDIA Developer Forums, accessed January 9, 2026, https://forums.developer.nvidia.com/t/multiple-gpu/279612
5. DeepStream Test3 Running pipeline on Different GPU - NVIDIA Developer Forums, accessed January 9, 2026, https://forums.developer.nvidia.com/t/deepstream-test3-running-pipeline-on-different-gpu/192271
6. Achieving 374 FPS with YOLOv8 Segmentation on NVIDIA RTX 5070 Ti GPU - Medium, accessed January 9, 2026, https://medium.com/cvrealtime/achieving-374-fps-with-yolov8-segmentation-on-nvidia-rtx-5070-ti-gpu-3d3583a41010
7. Ultralytics YOLO Evolution: An Overview of YOLO26, YOLO11, YOLOv8, and YOLOv5 Object Detectors for Computer Vision and Pattern Recognition - arXiv, accessed January 9, 2026, https://arxiv.org/html/2510.09653v1
8. YoloV4 slower in INT8 than FP16 - TensorRT - NVIDIA Developer Forums, accessed January 9, 2026, https://forums.developer.nvidia.com/t/yolov4-slower-in-int8-than-fp16/179181
9. INT8 Yolo model conversion led to accuracy drop in deepstream, accessed January 9, 2026, https://forums.developer.nvidia.com/t/int8-yolo-model-conversion-led-to-accuracy-drop-in-deepstream/177581
10. Robust Background Subtraction With Foreground Validation For Urban Traffic Video - OSTI.GOV, accessed January 9, 2026,

https://www.osti.gov/servlets/purl/859918
11. Comparative Evaluation of Background Subtraction Algorithms in Remote Scene Videos Captured by MWIR Sensors - NIH, accessed January 9, 2026, https://pmc.ncbi.nlm.nih.gov/articles/PMC5621003/
12. Video Background Subtraction in Complex Environments | Journal of Applied Research and Technology. JART - Elsevier, accessed January 9, 2026, https://www.elsevier.es/en-revista-journal-applied-research-technology-jart-81-articulo-video-background-subtraction-in-complex-S1665642314716323
13. Reviewing ViBe, a Popular Background Subtraction Algorithm for Real-Time Applications, accessed January 9, 2026, https://scholars.cityu.edu.hk/en/publications/reviewing-vibe-a-popular-background-subtraction-algorithm-for-rea/
14. Dynamic Control of Adaptive Mixture-of-Gaussians Background Model - cata log.lib.ky, accessed January 9, 2026, https://catalog.lib.kyushu-u.ac.jp/opac_download_md/4036/ShimadaAVSS06.pdf
15. Adaptive background mixture models for real-time tracking, accessed January 9, 2026, http://www.ai.mit.edu/projects/vsam/Publications/stauffer_cvpr98_track.pdf
16. Surveillance of Background Activities using MOG, ViBe and PBAS - IJREAM, accessed January 9, 2026, https://www.ijream.org/papers/IJREAMV03I012525.pdf
17. ViBe: A universal background subtraction algorithm for video sequences - ORBi, accessed January 9, 2026, https://orbi.uliege.be/bitstream/2268/145853/1/Barnich2011ViBe.pdf
18. Using batch_size in inference doesn't speed up? - Discussion - Ultralytics, accessed January 9, 2026, https://community.ultralytics.com/t/using-batch-size-in-inference-doesnt-speed-up/1534
19. The larger the batch size, the better when build engine? - NVIDIA Developer Forums, accessed January 9, 2026, https://forums.developer.nvidia.com/t/the-larger-the-batch-size-the-better-when-build-engine/142021
20. Model Optimization Techniques for YOLO Models | by Dr. Fatih Hattatoglu | Academy Team, accessed January 9, 2026, https://medium.com/academy-team/model-optimization-techniques-for-yolo-models-f440afa93adb
21. Comparison of Tracking-By-Detection Algorithms for Real-Time Satellite Component Tracking - DigitalCommons@USU, accessed January 9, 2026, https://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=5751&context=smallsat
22. Object Tracking Evaluation: BoT-SORT & ByteTrack with YOLOv8: A Comparison of Accuracy and Computational Efficiency - Diva-portal.org, accessed January 9, 2026, http://www.diva-portal.org/smash/get/diva2:1886982/FULLTEXT01.pdf
23. Multi-Object Vehicle Detection and Tracking Algorithm Based on Improved YOLOv8 and ByteTrack - MDPI, accessed January 9, 2026, https://www.mdpi.com/2079-9292/13/15/3033
24. Tip Tutorial: NVR Parking Lot Management with LPR Cameras - YouTube, accessed

January 9, 2026, https://www.youtube.com/watch?v=tlp6H8thVFE
25. Stationary Objects | Frigate, accessed January 9, 2026, https://docs.frigate.video/configuration/stationary_objects/
26. Handling of Stationary Objects in Frigate 13 Beta 7 #9133 - GitHub, accessed January 9, 2026, https://github.com/blakeblackshear/frigate/discussions/9133
27. Recommended HW for Frigate 0.16 onwards #14661 - GitHub, accessed January 9, 2026, https://github.com/blakeblackshear/frigate/discussions/14661
28. Frigate scaling with hundreds of cameras : r/frigate_nvr - Reddit, accessed January 9, 2026, https://www.reddit.com/r/frigate_nvr/comments/1jjpmna/frigate_scaling_with_hundreds_of_cameras/