

Edge Inference for Drones: Architectures, Hardware, and Sensor Fusion Across the Full Spectrum

Made by [Laurenz Bougan](#)

Chapter 1: Introduction: Why Edge Inference on Drones Is a Distinct Problem

The constraints of edge inference on autonomous aerial platforms are not merely engineering inconveniences. They are the defining force behind every architectural choice that follows. A drone does not have the luxury of offloading computation to a remote server when latency budgets are measured in milliseconds and radio links are unreliable or absent. What runs on the vehicle must run fast, run correctly, and run within a power envelope that competes directly with the motors keeping the platform airborne.

This is not the same problem as deploying a model on a mobile phone, or compressing a network to run on an embedded controller in an automotive system. Those problems are hard. The drone problem is harder in several specific ways, and understanding exactly why requires sitting with the constraints before reaching for solutions.

Consider power first. A 250-gram racing quadrotor draws roughly 150 to 200 watts through its propulsion system during aggressive flight. Its total battery capacity might be 20 watt-hours. Every watt consumed by an onboard compute module is a watt unavailable to the motors, and the relationship is not linear: reducing available motor power shortens flight time, which constrains mission radius, which changes what the vehicle can be used for. A 5-watt inference board on a nano-class platform is not a footnote. It is a meaningful fraction of the vehicle's energy budget, and the engineers who build these systems know it. At the other end of the taxonomy, a 30-kilogram industrial platform can tolerate a 65-watt GPU module because the propulsion system is drawing kilowatts, but that tolerance introduces its own thermal problems at altitude where convective cooling degrades. Power is never free, and on airborne platforms it is more expensive than almost anywhere else a compute module might be deployed.

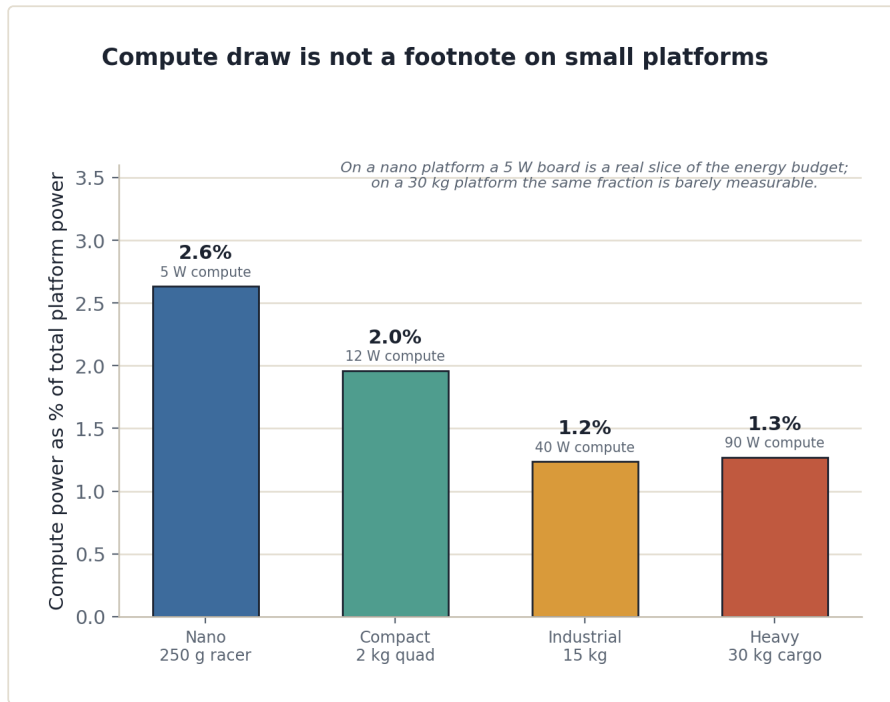


Figure 1.2 — Compute power as a share of the total platform energy budget across drone classes.

Weight compounds the problem. Adding 100 grams to a nano-class drone can reduce hover endurance by 15 percent or more, depending on the motor and battery configuration. Compute hardware has mass, and so do the cooling solutions that make it thermally viable. A heatsink adequate for a Jetson Orin NX in a ground-vehicle application may be completely impractical on a 2-kilogram inspection drone. This forces designers toward fanless or minimal-cooling configurations that tolerate sustained thermal throttling, or toward hardware choices that prioritize power efficiency over raw throughput even when throughput is what the mission demands.

Size closes the triad. Aerodynamic drag, center-of-gravity constraints, and mechanical integration tolerances all impose geometric limits on what can physically be carried. A PCB that would be unremarkable in a server room becomes an engineering problem when it needs to fit inside a 120-millimeter frame between battery cells and the flight controller stack.

Latency is a separate constraint that interacts with all three. Perception for autonomous flight is not a batch processing problem. Obstacle avoidance at 10 meters per second leaves roughly 150 milliseconds before a detected object becomes a collision, and that budget must cover sensor acquisition, preprocessing, inference, and enough time for the flight controller to execute an avoidance maneuver. In practice, inference latency budgets for safety-critical perception tasks on fast platforms are 20 to 50 milliseconds, and in some configurations tighter than that. A model that achieves excellent accuracy at 300 milliseconds per frame is not a model that can be used in flight. This is a hard cut, not a soft tradeoff, and it eliminates most architectures that perform well on standard desktop benchmarks before they are even considered.

Connectivity, or more precisely its absence, is the third force shaping the design space. A drone operating in a GPS-denied indoor environment, a radio-frequency-congested industrial site, or a remote agricultural corridor may have no uplink at all. Even when a link exists, its bandwidth is constrained and its latency is variable. Offloading heavy inference to a ground station introduces round-trip latency that is incompatible with real-time control, and it creates a single point of failure: if the link drops, the vehicle must continue operating on whatever intelligence is available locally. The practical implication is that any computation the mission depends on must be able to run fully on the vehicle. This is not a design preference. It is an operational requirement.

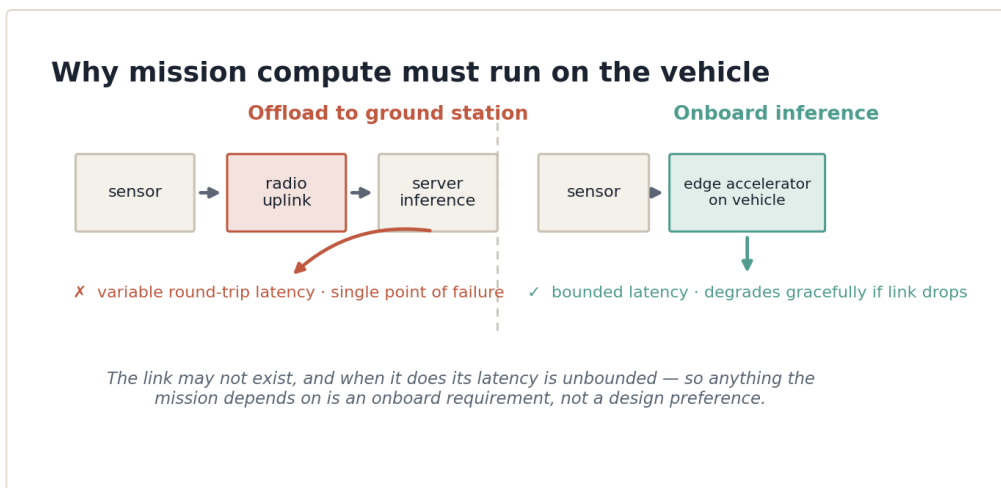


Figure 1.3 — Offloading inference to a ground station introduces unbounded latency and a single point of failure; mission compute must run onboard.

These three forces, SWaP, latency, and connectivity, constitute the constraint envelope within which all of drone edge inference operates. They are not independent. A choice that reduces power consumption may increase latency. A choice that reduces weight may reduce the sensor suite available for perception. Navigating this space requires not just knowing which models are accurate, but understanding which models survive contact with real hardware under real operating conditions.

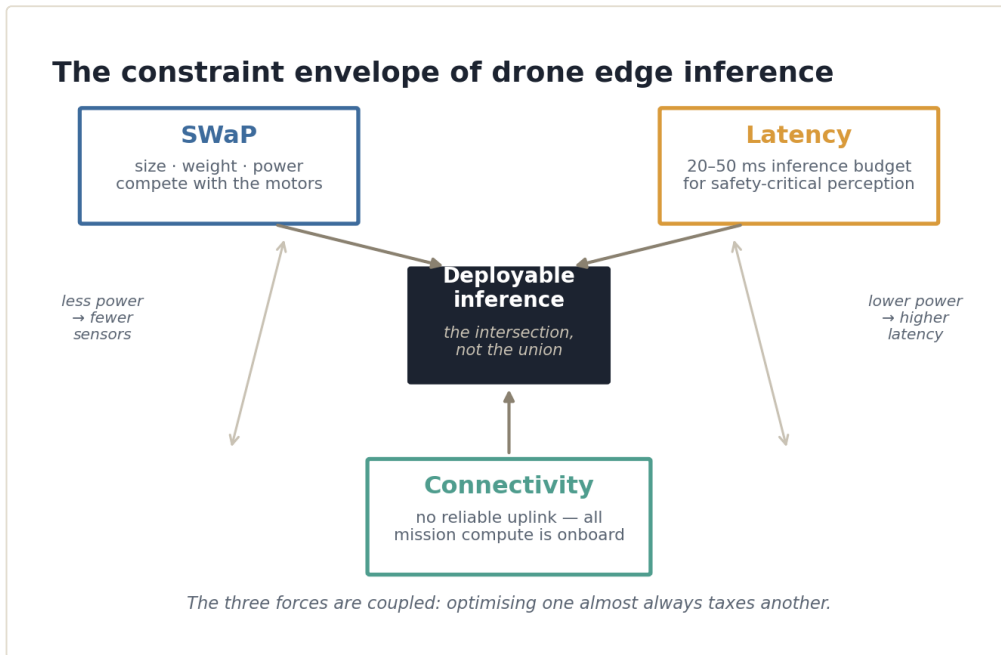


Figure 1.1 — SWaP, latency, and connectivity form a single coupled constraint envelope; only their intersection is deployable.

This paper treats inference architecture as a first-class citizen of drone system design. That means examining not just which models perform well on benchmark datasets, but which models survive contact with real hardware: the quantized INT8 network that fits inside a 4 MB weight budget, the pruned backbone that sustains 60 FPS on a mobile NPU at 2 watts, the early-exit transformer that bails out of its forward pass when confidence is already sufficient. These are not compromises. They are the craft.

One of the structuring decisions of this paper is to treat lidar, radar, and vision as co-equal modalities throughout. That choice requires justification, because the field has not always treated them that way. Camera-based perception dominates the published literature on drone autonomy, in part because camera hardware is cheap and small, in part because computer vision as a discipline has decades of accumulated momentum, and in part because the benchmark datasets that define research progress have historically been image-centric. None of those reasons is an argument about sensor physics.

Radar sees through fog, smoke, and precipitation that defeats both cameras and most lidar systems. It provides direct velocity measurements via Doppler shift that cameras can only infer through optical flow, with all the limitations that implies. Millimeter-wave radar modules small enough for nano-class platforms now exist and are appearing in consumer electronics. The inference architectures for radar are less mature than those for vision, and that gap is narrowing rapidly.

Lidar provides accurate depth and reflectivity information across wide fields of view without the scale ambiguity inherent in monocular vision. Solid-state lidar units are now available at form factors and power levels that make them viable on compact

industrial platforms. The point cloud representations they produce require inference architectures that are architecturally different from image-based networks, and understanding those architectures is necessary for anyone designing perception systems for the next generation of drone hardware.

Treating vision as the primary modality and the others as supplements is a bias that made sense in 2015. It makes less sense now, and it will make less sense still in five years. This paper is organized around the assumption that a practitioner building a perception system for an industrial drone in 2025 or later needs to understand all three modalities and the architectures suited to each, not just the camera pipeline with lidar treated as an optional depth channel.

The organizational framework used throughout is a taxonomy of four drone classes. These classes are defined not by product names or regulatory categories but by compute budget, battery capacity, and payload tolerance, because those are the quantities that actually determine what inference architectures are viable.

The first class is nano and micro platforms, generally below 250 grams total takeoff weight. These are the most constrained systems in the taxonomy. Available compute is typically a microcontroller-class processor or a small NPU integrated into a system-on-chip, with a total inference power budget of 1 to 2 watts. Sensor options are limited to lightweight cameras, minimal IMU suites, and small radar units. Lidar at this weight class is rare and will be treated as an emerging rather than established capability.

The second class is compact quadrotors in the 250-gram to 2-kilogram range. This is the most commercially common drone class and spans a wide range of capability. At the lower end of this class, inference hardware is still tightly constrained, typically a mobile SoC drawing 3 to 5 watts. At the upper end, a compact Jetson-class module becomes viable. Sensor suites can include stereo cameras, downward-facing lidar, and millimeter-wave radar.

The third class is mid-range industrial platforms in the 2- to 25-kilogram range. This class includes inspection drones, mapping platforms, and agricultural systems. Compute budgets open up to modules drawing 10 to 65 watts, making full-precision inference with capable backbones feasible. Sensor suites can include spinning or solid-state lidar, multiple camera rigs, and dedicated radar units.

The fourth class is large autonomous systems above 25 kilograms. This includes autonomous cargo vehicles, larger agricultural platforms, and experimental long-endurance systems. Compute budgets can accommodate server-class processors in some configurations. The constraint structure is still defined by SWaP, but the tradeoffs are different in character, with thermal management and system integration complexity becoming dominant concerns.

The paper works upward through the stack from sensors to architectures to fusion strategies, using this taxonomy as the reference frame for every claim about feasibility.

A claim that a given architecture is viable on an edge drone is not meaningful without specifying which edge drone. A claim that is valid for a 15-kilogram industrial platform may be completely false for a 150-gram nano quadrotor. The taxonomy makes those distinctions explicit and keeps them explicit throughout.

Chapter 2 develops the taxonomy in full, characterizing realistic sensor suites, SoC options, and inference time budgets for each class. Chapter 3 covers sensor fundamentals, explaining how the native output format of each modality shapes model input requirements and preprocessing cost. Chapter 4 surveys the hardware landscape, from mobile-class SoCs and NPUs to FPGA-based accelerators and the OS and runtime environments that run on them. Chapter 5 goes algorithm-deep on model compression, covering quantization, pruning, and knowledge distillation as they are actually applied to drone inference workloads. Chapters 6, 7, and 8 cover the inference architectures for vision, lidar, and radar respectively, with attention to the algorithmic mechanics that produce the compute savings that make edge deployment possible. Chapter 9 examines sensor fusion, covering both late fusion approaches and the joint multimodal architectures that have recently become practical at the edge. Chapter 10 synthesizes all of this into concrete end-to-end inference stack descriptions for each drone class in the taxonomy. Chapter 11 surveys the current state of the art and the open problems that define where the field is heading.

The goal throughout is to give a reader with a serious engineering background a complete picture of the design space: what the constraints are, what the hardware options are, what the algorithmic tools are, and how to assemble them into a system that actually flies.

Chapter 2: Drone Taxonomy: Compute Classes and Their Constraints

Before any model architecture can be evaluated for edge drone deployment, the platform running that model must be characterized precisely. "Drone" spans nearly four orders of magnitude in gross takeoff weight and an equally wide range of onboard power budgets, and the engineering decisions that are correct for one point in that space are often flatly impossible at another. This chapter builds the four-tier taxonomy that the rest of the paper uses as its reference frame, working through each tier's physical constraints in enough detail to make the downstream claims about hardware, model compression, and sensor fusion grounded rather than generic.

The four tiers are defined by total platform mass and the corresponding payload and power budgets that mass implies:

- Tier 1: Micro platforms, sub-250 g total takeoff mass
- Tier 2: Light tactical platforms, 250 g to 2 kg
- Tier 3: Industrial mid-range platforms, 2 kg to 10 kg
- Tier 4: Heavy autonomous platforms, 10 kg to 30 kg

Mass is the primary axis because it drives propulsive power, and propulsive power sets the total energy budget from which everything else is allocated. An increase in compute payload does not just add that payload's own mass; it increases rotor loading, which increases motor current draw, which reduces flight endurance unless battery mass also increases, which loops back into total mass. The interaction is nonlinear. A compute board that adds 80 g and draws 8 W additional power can reduce the flight time of a 300 g platform by 30 to 40 percent. The same board added to a 5 kg platform is barely measurable in endurance terms.

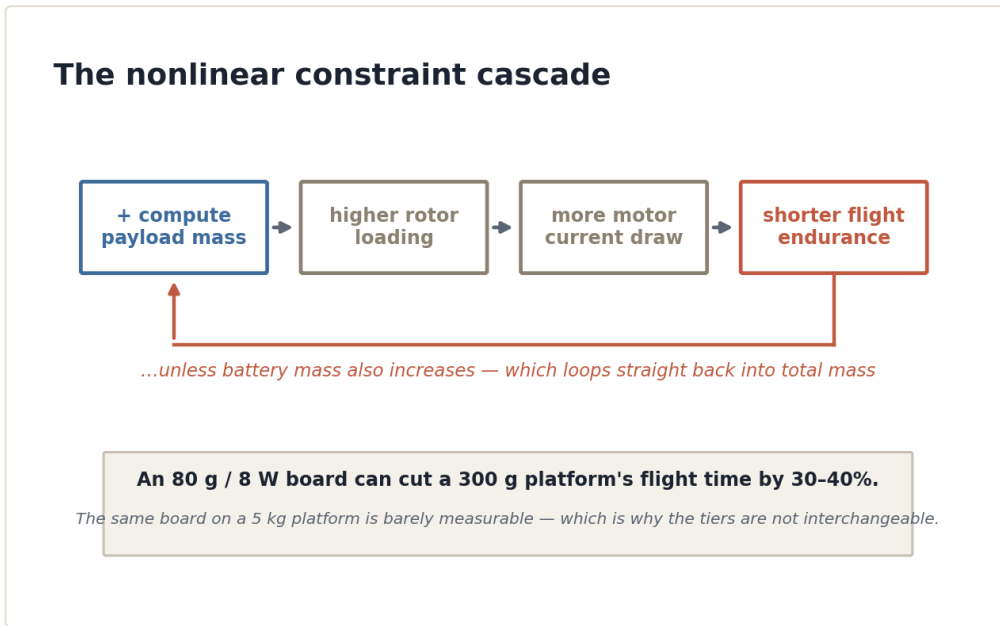


Figure 2.2 — The nonlinear constraint cascade: added compute mass loops back into total platform mass through the battery.

That asymmetry is why the taxonomy tiers are not interchangeable and why characterizing each one separately is necessary.

The four-tier taxonomy: mass drives every other budget

	Tier 1 Micro	Tier 2 Light tactical	Tier 3 Industrial	Tier 4 Heavy autonomous
takeoff mass	< 250 g	0.25-2 kg	2-10 kg	10-30 kg
compute power	1-3 W	5-15 W	10-30 W	50-100 W
payload budget	20-60 g	100-500 g	1-2 kg	≈ 3 kg
inference silicon	MCU / tiny NPU	Mobile NPU SoC	GPU-class SoC	Multi-SoC

Mass sets propulsive power, propulsive power sets the energy budget — and the compute payload is allocated from whatever is left. The interaction is nonlinear.

Figure 2.1 — The four-tier taxonomy: mass sets propulsive power, which sets every downstream budget.

Tier 1: Micro Platforms, Sub-250 g

The 250 g figure is not arbitrary. In most regulatory jurisdictions it represents the lightest category requiring registration, which has encouraged substantial engineering activity targeting exactly that ceiling. More importantly, it represents a hard physical constraint: a sub-250 g platform must allocate most of its mass budget to battery, frame, motors, and propellers, leaving very little for a compute payload.

Typical payload budgets in this class run from 20 g to 60 g total for sensors and compute combined. That is not a budget for a Jetson module. It is a budget for a microcontroller-class flight controller, a single lightweight sensor, and a minimal inference accelerator if one fits at all. Power available for the compute subsystem is typically 1 W to 3 W, imposed by the combination of battery capacity (usually 300 to 800 mAh at 1S to 2S LiPo, meaning 3.7 to 7.4 V) and the thermal constraints of a chassis with no active cooling and almost no thermal mass.

At 1 to 3 W, inference options are narrow. The realistic compute hardware at this tier is dominated by microcontroller-class processors running at sub-1 GHz, Cortex-M7 or Cortex-A53 class chips, or purpose-built ultra-low-power inference devices such as the Syntiant NDP series or the Himax WE-I Plus. Lattice Semiconductor's crossover FPGAs in the iCE40 and ECP5 families have found use here, consuming milliwatts for fixed-function inference pipelines. The STM32H7 family occupies the high end of the microcontroller space and can execute small CNN inference workloads, but with severe model size constraints given its on-chip SRAM measured in kilobytes and flash measured in megabytes.

Model design for this tier is not a matter of selecting an efficient backbone; it is a matter of asking whether any learned model is deployable at all given the available memory and compute. A MobileNetV1 model at INT8 quantization requires roughly 4 MB of weight storage and produces 218 million multiply-accumulate operations per inference pass on a 224x224 input. That is not feasible on a Cortex-M7 at 480 MHz in a 33 ms inference window. What is feasible is either: aggressively width-reduced variants of MobileNet (width multiplier of 0.25 or smaller), TinyML-class networks explicitly designed for sub-100 KB parameter budgets such as MCUNet or TinyBERT-like distilled classifiers, or fixed-function signal processing pipelines with small learned decision stages at the output.

Latency requirements at this tier depend on the use case, but obstacle avoidance imposes the hardest constraint. A platform traveling at 2 m/s needs to detect and begin responding to an obstacle within roughly 50 to 100 ms to execute any meaningful avoidance maneuver at a physically plausible deceleration. That 50 to 100 ms total budget includes sensor acquisition, preprocessing, inference, and flight controller response. The inference slice of that budget is realistically 10 to 30 ms. On a 1 W processor, that limits inference to models that execute in well under one million operations per millisecond, which pushes strongly toward ultra-compact architectures or non-neural detection heuristics augmented by learned classifiers.

Thermal management at Tier 1 is passive and minimal. There is no heat sink, no fan, no thermal interface material worth the mass. The compute chip must be selected to stay within junction temperature limits purely through power dissipation limits and the small convective advantage of rotor-induced airflow. In practice this means any chip

running at Tier 1 must be specified for sustained inference at its rated TDP, not peak, because the thermal environment offers no burst capacity.

The dominant sensor at this tier is a monocular camera, often a global-shutter VGA or 720p unit in the 1 g to 3 g range, sometimes augmented by a single-chip mmWave radar such as the Texas Instruments IWR1843 in its lightest form factor. LiDAR is essentially absent at Tier 1 in meaningful scanning form; single-beam time-of-flight sensors such as the STMicroelectronics VL53 series appear for altitude hold but provide no lateral scene information useful for inference.

Tier 2: Light Tactical Platforms, 250 g to 2 kg

This tier is the most commercially diverse. It includes consumer racing drones, inspection-class platforms, military scout vehicles, and the majority of research quadrotors. The mass ceiling of 2 kg, combined with modern LiPo battery energy density, makes flight times of 20 to 40 minutes realistic with moderate sensor and compute payloads.

Payload budgets at this tier run from 100 g to 500 g, with compute power envelopes of 5 W to 15 W. That range opens the door to mobile-class SoCs: the Qualcomm Snapdragon 820 or Flight Pro, lower-power Jetson variants such as the Jetson Nano at its 5 W mode, and dedicated neural accelerators such as the Intel Myriad X (now discontinued but widely fielded in this tier) and the Hailo-8M. The Raspberry Pi CM4 appears frequently as a low-cost flight computer with Vision Processing Units added as co-processors for inference.

Memory bandwidth is the most important secondary constraint at Tier 2 after raw power. The Jetson Nano in 5 W mode provides 25.6 GB/s of shared LPDDR4 memory bandwidth across CPU, GPU, and any camera ISP processing. A monocular depth network running at 30 Hz on 640x480 inputs can consume 8 to 12 GB/s of that bandwidth in feature map activations alone, leaving insufficient headroom for concurrent sensor preprocessing unless the pipeline is carefully staged. This bandwidth pressure is why, at Tier 2, the choice of whether to run inference on the GPU or offload it to a dedicated NPU has significant downstream effects on pipeline structure.

Latency budgets at Tier 2 are similar in shape to Tier 1 but somewhat more relaxed at the lower end of platform speeds. A 1 kg inspection drone hovering at a structure can tolerate 100 to 200 ms total perception latency for anomaly detection tasks. The same platform in autonomous navigation mode at 5 m/s needs 30 to 50 ms end-to-end. The 33 ms figure corresponding to a 30 Hz control loop is a practical minimum for any platform performing active obstacle avoidance, and achieving it at Tier 2 requires INT8 inference on a hardware-accelerated backend rather than floating-point inference on a CPU.

Thermal behavior at this tier introduces a new design constraint: intermittent active cooling via rotor downwash is available during flight but absent on the ground. Platforms that throttle compute during preflight or when motors are not spinning must handle the thermal transient at motor start. Some Tier 2 systems solve this by running inference at reduced frequency until rotor airflow is established. The Jetson Nano's thermal envelope is sufficient for sustained 5 W compute with the standard heatsink under rotor downwash, but can throttle under sustained full-speed inference on a warm day without airflow.

The sensor suite at Tier 2 expands meaningfully. Stereo cameras become practical: the Intel RealSense D4xx series, for instance, fits within Tier 2 payload budgets and provides depth maps usable directly as inference inputs. Solid-state LiDAR units such as the Livox Mid-360 at roughly 265 g and 5 W are on the edge of feasibility for the upper end of this tier. Single-chip mmWave radar is comfortably within range. The practical sensor suite for an aggressive Tier 2 platform is a stereo camera plus radar, or a monocular camera plus low-density solid-state LiDAR, but rarely all three simultaneously given payload constraints.

Model architecture implications at Tier 2 are significant. This is the tier where EfficientDet-D0 or D1, YOLOv5n or YOLOv5s, and MobileNetV3-Small become the natural choices for object detection. These models, quantized to INT8 and deployed through TensorRT on a Jetson Nano or through the Hailo SDK on Hailo-8M, can achieve 30 Hz inference on VGA inputs within the power budget. Depth estimation networks from the MiDaS family can run at reduced resolution. Semantic segmentation is possible at low resolution. Point cloud inference from sparse LiDAR data using PointNet++-derived architectures is feasible if the point cloud is small (under 4096 points per frame), but sparse convolutional networks are not practical on hardware without sparse convolution support, which the Jetson Nano GPU provides only partially.

Tier 3: Industrial Mid-Range Platforms, 2 kg to 10 kg

Tier 3 is where autonomous aerial systems begin to look like serious compute platforms rather than constrained embedded devices. A 5 kg industrial drone can carry 1 to 2 kg of payload and supply 25 to 50 W to compute and sensor subsystems from a 6S or higher battery pack. This is sufficient for a Jetson Xavier NX (10 W to 20 W), a Jetson Orin NX (10 W to 25 W), or a mid-range FPGA such as the Xilinx Zynq UltraScale+ in its automotive-grade MPSoC variants. Qualcomm's RB5 platform, though originally targeting robotics, appears in this tier when packaged appropriately.

The Jetson Orin NX at 15 W offers 70 TOPS at INT8, 16 GB of LPDDR5 at 102 GB/s bandwidth, and runs a full embedded Linux stack with CUDA, TensorRT, and ONNX Runtime support. For Tier 3, this is a capable and well-supported inference platform. The bandwidth figure is important: 102 GB/s allows concurrent inference across multiple sensor streams without the bandwidth starvation that constrains Tier 2 pipelines. A Tier 3 pipeline can realistically run simultaneous LiDAR inference, stereo

depth estimation, and a detection head on a fused feature representation, all within 33 ms, if the pipeline is correctly parallelized.

Thermal management at Tier 3 becomes a systems engineering problem rather than a physics constraint. A 25 W compute payload generates heat that must be conducted to a heatsink and convected by airflow. Rotor downwash provides variable airflow depending on throttle state, which varies significantly with maneuver load. The Jetson Orin NX thermal design guide specifies a heatsink-to-ambient thermal resistance sufficient for sustained full-power operation at 25 degrees C ambient; at 40 degrees C ambient, throttling may begin unless the thermal interface is carefully designed. Industrial Tier 3 platforms typically mount the compute module to the airframe with a thermal pad to a machined aluminum mounting plate that acts as a distributed heatsink, relying on the underside exposure to prop wash.

Latency at Tier 3 tightens for faster platforms. Agricultural inspection drones moving at 10 to 15 m/s need perception latency under 20 ms to provide meaningful stopping distance. That requires not just fast inference but pipelined sensor capture: the camera frame triggering must be synchronized with LiDAR scan timing such that the data used in one inference pass was acquired with a known, small temporal offset. At Tier 3, implementing this synchronization properly through hardware trigger lines and timestamped sensor drivers becomes important and is not optional on fast platforms.

The realistic sensor suite for Tier 3 includes solid-state LiDAR (Livox Mid-360, Ouster OS0-32), mmWave radar (TI AWR2944 or Ainstein US-D1), and stereo or multi-camera vision, all simultaneously. This is the first tier where all three modalities can operate concurrently and contribute to a fused inference pipeline. The LiDAR at this class generates 40,000 to 200,000 points per second depending on configuration, which produces point clouds in the range of 10,000 to 60,000 points per 33 ms frame, requiring voxelization or pillar-based encoding before inference.

Model architecture at Tier 3 expands to include: YOLOv8m or YOLOv8l for vision detection, PointPillars for LiDAR with a voxel resolution of 0.2 m or finer, EfficientViT-M4 or similar efficient transformer backbones with moderate input resolution, and radar processing networks operating on range-Doppler maps at the resolution the AWR2944 provides (typically 256x128 or similar). CenterPoint-style detection heads become feasible, and BEV fusion of LiDAR and camera features through a learned projection layer is within reach if the fusion module is kept compact.

Tier 4: Heavy Autonomous Platforms, 10 kg to 30 kg

Tier 4 platforms are autonomous delivery vehicles, large agricultural systems, and research platforms where sensor suite and compute are constrained by regulation and cost more than by physics. A 15 kg platform with 3 kg available for compute and sensors can supply 50 to 100 W to that subsystem. That is a Jetson AGX Orin (up to 60 W, 275 TOPS at INT8) or a multi-board configuration pairing a Jetson Orin NX with a

Hailo-8 or FPGA co-processor. Some platforms in this class use industrial embedded computing modules from ADLINK or Kontron that combine x86 processing with PCIe-attached inference cards.

At 275 TOPS, the AGX Orin can run inference workloads that would be completely impractical one or two tiers down: large transformer backbones, multi-scale BEV fusion networks, CenterFusion or TransFusion architectures, dense semantic segmentation at near-full resolution, and concurrent multi-object tracking. The 64 GB/s memory bandwidth (in the AGX Orin's 32 GB variant) remains a constraint for very large models, but the absolute bandwidth figure is sufficient for most multi-modal inference pipelines at 30 Hz.

Thermal management at Tier 4 becomes the primary design concern rather than power budget. A 60 W compute module in an enclosed nacelle can reach dangerous junction temperatures quickly without forced convection. Many Tier 4 platforms use sealed, thermally isolated compute bays with embedded heat pipes transferring heat to an external fin array exposed to airflow. The AGX Orin's active cooling kit adds mass and volume but is frequently necessary. Thermal throttling under sustained inference load is a real operational concern and must be validated at maximum ambient temperature before deployment.

Latency at Tier 4 allows more pipeline depth. A 20 kg delivery drone cruising at 8 m/s has less need for the sub-20 ms obstacle avoidance latency of a faster platform, and its trajectory planning runs at a longer horizon. However, the consequence of a false negative at Tier 4 is more severe given the platform's kinetic energy, which means reliability requirements are higher even if raw latency is more relaxed. Redundant inference paths and sensor fallback logic become engineering requirements, not optional features.

The sensor suite at Tier 4 is the full complement: spinning or solid-state LiDAR with 64 or more beams (Ouster OS1-64, Hesai AT128), multiple mmWave radar units for 360-degree coverage, and a multi-camera vision array providing panoramic coverage. LiDAR at this class generates 1.3 million or more points per second, producing frames of 40,000 to 100,000 points per 33 ms window that exceed what PointNet++-based architectures can process in latency budget on even the AGX Orin without voxelization. Sparse convolutional networks with GPU-resident sparse tensor formats become the practical requirement.

Constraint Cascades and Cross-Tier Comparisons

The tiers are not just quantitative gradations; they produce qualitatively different design decisions. To make that concrete, it is useful to trace how a single requirement, obstacle avoidance with 10 m detection range, propagates differently through each tier.

At Tier 1, 10 m detection range from a monocular camera on a moving platform requires either a learned depth estimator or a stereo pair. A learned monocular depth

estimator cannot run within the power and latency constraints, so the practical choice is a single-beam ToF sensor for close-range detection augmented by optical flow for motion estimation, accepting reduced detection range and reliability.

At Tier 2, the same requirement is addressable with a stereo camera running a lightweight disparity network at reduced resolution, or a solid-state LiDAR on the upper end of the tier. The stereo approach requires enough GPU or NPU performance to run disparity estimation at 30 Hz, which on a Jetson Nano in 5 W mode requires a network significantly smaller than anything in the standard depth estimation literature without modification.

At Tier 3, the requirement is straightforwardly met: solid-state LiDAR provides direct 10 m range measurement, and the inference pipeline enriches that with semantic labels from the concurrent camera stream. The engineering challenge shifts from feasibility to latency optimization and sensor synchronization.

At Tier 4, the challenge is not detecting obstacles at 10 m but managing the data rates and reliability requirements of a sensor suite that sees obstacles at 10 m from multiple perspectives simultaneously, and fusing that information into a consistent representation that the planner can act on.

The same pattern repeats for every inference task considered in later chapters. The tier determines not just which models run but which problems are even correctly framed at that level of compute.

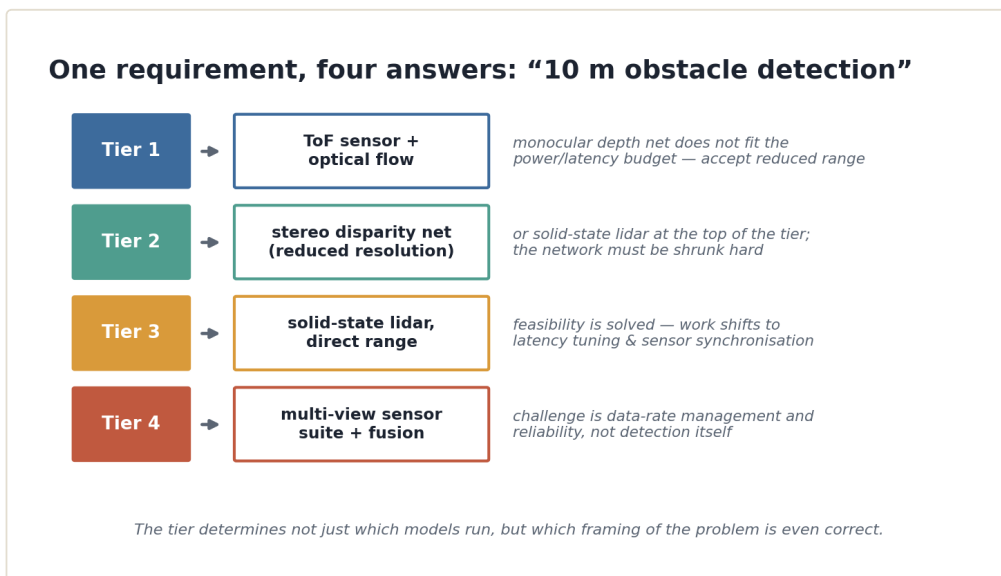


Figure 2.3 — One requirement, four answers: how “10 m obstacle detection” resolves differently per tier.

One cross-cutting constraint deserves explicit mention: memory. Across all tiers, the ratio of on-chip SRAM to total model size shapes whether inference can be executed without external memory transactions, and external memory transactions drive both latency and power. A Cortex-M7 with 1 MB of SRAM cannot hold even a minimal

MobileNetV1 INT8 model in cache; it must page weights from flash, which costs energy and time. A Jetson Orin NX with 16 GB of LPDDR5 holds the entire multi-modal inference pipeline in memory simultaneously, but its bandwidth still limits how quickly activations can flow between layers on large models. The bandwidth-to-compute ratio, measured as bytes per operation, is the figure that determines whether a model is memory-bound or compute-bound on a given chip, and nearly all edge inference on drones is memory-bound at some point in the pipeline. This matters because memory-bound inference does not benefit from higher TOPS counts; it benefits from higher memory bandwidth or from reducing the activation size that must be transferred.

The taxonomy established in this chapter will be invoked by name throughout the remaining chapters. When Chapter 7 states that PointPillars is a Tier 3 architecture, that claim is precise: it means that PointPillars, at the voxel resolutions and point cloud densities produced by the solid-state LiDAR sensors appropriate for that platform class, can be executed within the latency budget of a Jetson Orin NX operating at 15 W with TensorRT INT8 optimization. When the same chapter states that sparse convolutional networks require Tier 4 hardware for practical deployment, that claim is equally precise. The taxonomy converts vague statements about feasibility into specific claims that can be tested, contradicted, or refined as hardware and algorithms continue to develop.

Chapter 3: Sensor Modalities: LiDAR, Radar, and Vision on the Edge

Every inference pipeline starts with a sensor, and the sensor determines almost everything that follows. Not abstractly, but concretely: the data structure emitted by a camera is fundamentally different from the data structure emitted by a LiDAR, which is fundamentally different from what an FMCW radar returns. These differences are not cosmetic. They propagate through preprocessing, through model architecture, through memory layout, and through the fusion logic that combines them. Understanding why each modality produces the data it does requires understanding the physics that generates it, and understanding the physics makes the architectural choices downstream feel inevitable rather than arbitrary.

This chapter covers each of the three primary modalities in turn, treating them with equal seriousness. Vision has the largest body of embedded deployment literature, but that does not make it the most tractable modality on drones. LiDAR provides geometric precision that cameras cannot match but at the cost of sparsity patterns that break standard convolutional assumptions. Radar is frequently treated as a supplementary modality, but its ability to measure radial velocity directly and operate through precipitation and dust makes it architecturally irreplaceable in certain deployment contexts. The goal here is a precise mental model of what each sensor hands to the inference engine and why that matters.

The camera is the oldest and most thoroughly studied sensor in autonomous perception, and its operating principle needs little elaboration. A lens focuses reflected ambient light or active illumination onto a photodetector array. Each pixel integrates photons over the exposure interval and produces an intensity value. In color cameras, a Bayer filter mosaic assigns each pixel a red, green, or blue spectral sensitivity, and demosaicing reconstructs a three-channel RGB image. The result is a dense 2D array of values sampled at a uniform spatial grid.

The word "dense" is doing significant work in that description. A 1280x720 RGB frame contains 921,600 pixels, each represented as three unsigned 8-bit integers, totaling approximately 2.76 MB before any compression. At 30 frames per second that is 82.8 MB/s of raw data flowing into the processor. The data is spatially coherent, meaning neighboring pixels represent geometrically adjacent scene locations, which is exactly the structure that convolutional neural networks exploit. A 3x3 convolutional kernel applied to an RGB frame performs a local spatial integration that is directly meaningful in image coordinates.

What cameras do not natively provide is depth. The projection from 3D world to 2D image plane discards the distance dimension. Monocular depth estimation, which attempts to recover depth from a single frame using learned scene statistics, is possible but produces relative depth estimates with scale ambiguity and significant

error at distance. Stereo cameras recover metric depth through disparity estimation, but stereo matching on embedded hardware is expensive, and the baseline between lenses limits reliable depth range. For a drone operating in cluttered environments at 5 to 30 meters, monocular depth estimation is often too uncertain for obstacle avoidance, and stereo matching at full resolution frequently exceeds latency budgets on Tier 1 and Tier 2 hardware. These limitations are not failures of implementation; they are geometric facts about the image formation model.

The embedded camera hardware landscape for drones spans several categories. Rolling shutter CMOS sensors, found in most consumer cameras, read pixel rows sequentially rather than simultaneously, introducing a temporal distortion called rolling shutter artifact when the camera or scene is in motion. A drone banking at angular velocity while capturing a fast-moving object will see the image shear in proportion to the readout time. Global shutter sensors eliminate this by capturing all pixels at once, but they are more expensive and generally less sensitive. For high-speed flight or fast object tracking, global shutter sensors are architecturally necessary. The Tier 1 and Tier 2 platforms that must operate on tight cost budgets frequently accept rolling shutter and compensate in software, either by modeling the distortion explicitly or by running inference at a rate slow enough that the artifact is below threshold.

The interface between camera and processor matters considerably for edge inference. CSI-2 (Camera Serial Interface 2) is the dominant protocol for embedded cameras, providing a direct lane from sensor to SoC with minimal latency. USB cameras introduce additional latency through the USB stack and host controller, which is problematic for hard real-time pipelines. Cameras with onboard ISP (image signal processor) handle demosaicing, noise reduction, and exposure control in hardware before the data reaches the main processor, reducing the CPU/GPU preprocessing burden. On Jetson platforms, the Tegra ISP accepts CSI-2 streams and delivers processed YUV or RGB frames directly to memory, where the GPU can fetch them without involving the CPU. This pipeline matters because the latency contribution from sensor to processed frame in memory can be 5 to 10 ms, a significant fraction of a 33 ms frame budget.

The predominant preprocessing path for vision inference begins with normalization: subtracting per-channel means and dividing by standard deviations, typically those of the ImageNet distribution for pretrained backbones. This converts uint8 values in [0, 255] to float32 or INT8 values centered near zero, which is what the model expects. Resizing to the network's input resolution happens before normalization, and the choice of resize algorithm affects both quality and latency. Bilinear interpolation is standard. On hardware with dedicated ISP or NPU preprocessing pipelines, this entire path executes in hardware before the first inference kernel runs.

For edge inference on drones, the key architectural implication of camera data is that convolutional and attention-based models designed for dense 2D grids map directly

onto this data structure. There is no impedance mismatch between the sensor output and the model input. This is a genuine advantage. The disadvantage is that everything the model knows about the 3D world must be inferred from 2D projections, and that inference is inherently uncertain in ways that geometry alone cannot resolve.

LiDAR, which stands for Light Detection and Ranging, operates on a different physical principle. A pulsed or frequency-modulated laser emits light, typically in the near-infrared at 905 nm or 1550 nm, and the sensor measures either the time of flight of reflected pulses or the frequency shift of reflected continuous waves to compute range. A single laser beam at a known azimuth and elevation angle produces a single range measurement, which combined with the beam geometry gives a 3D point in polar coordinates, convertible to Cartesian (x, y, z) . A complete scan sweeps many beams across the scene in a pattern determined by the sensor's mechanical or solid-state architecture.

The mechanical rotating LiDAR, exemplified by the Velodyne HDL-64E and its descendants, arranges multiple laser/detector pairs vertically and spins the entire assembly horizontally. A 32-beam rotating LiDAR produces 32 concentric rings of points per revolution. At 10 Hz rotation rate, this yields approximately 320,000 points per second, each annotated with (x, y, z) coordinates and a reflectance intensity value. The data is sparse: at 30 meters range, the angular resolution of even a 32-beam unit leaves gaps of roughly 1 meter between rings at the horizontal extent of the scene. At 10 meters, the same gaps are 30 cm. This sparsity is not a defect; it is a fundamental property of the sampling geometry.

Solid-state LiDAR eliminates the rotating mechanism by using fixed arrays of laser emitters and detectors, often with MEMS mirrors or optical phased arrays to steer beams electronically. The advantages for drones are substantial: lower mass, higher shock tolerance, lower power, and more predictable failure modes. The Livox Avia, which draws around 9 W and weighs 490 g including housing, uses a non-repetitive scanning pattern that accumulates point density over time rather than completing uniform rings. The Hesai AT128, designed for automotive but appearing in some Tier 3 and Tier 4 drone integrations, provides 128 lines in a forward-facing wedge. Single-beam LiDAR altimeters appear even on Tier 1 platforms; the Benewake TFmini weighs 5 g and draws 40 mW, providing a single vertical range measurement at up to 12 meters.

The data structure produced by LiDAR is a point cloud: an unordered set of 3D coordinates with optional per-point attributes. This structure has no natural grid. It is not an image. You cannot apply a 3x3 convolutional kernel to a point cloud and expect meaningful spatial aggregation, because the points are not arranged on a regular grid in any of the three spatial dimensions. This is the fundamental architectural challenge that LiDAR inference must solve, and the various approaches to solving it define the entire landscape of LiDAR model architectures.

The first approach, exemplified by PointNet, processes each point independently through shared MLP layers and then aggregates globally with a symmetric function like max-pooling. This respects the unordered nature of the point cloud but loses local geometric structure. PointNet++ adds hierarchical local grouping, but the neighbor-finding operations are expensive on hardware that lacks efficient k-nearest-neighbor primitives.

The second approach, which has become dominant for embedded deployment, converts the point cloud to a structured representation before inference. Voxelization divides 3D space into a regular grid of cubes; points falling within each cube are aggregated into a fixed-length feature vector. The result is a sparse 3D grid that conventional 3D convolutions can process. Pillar encoding is a special case that collapses the vertical dimension, treating each vertical column of voxels as a single feature vector, producing a 2D bird's-eye-view representation that standard 2D CNNs can process efficiently. PointPillars, which Chapter 7 covers in depth, uses this approach and achieves inference latencies on Tier 3 hardware that would be impossible with point-based architectures.

The third approach projects the point cloud into 2D range images, where each pixel represents one beam at one azimuth angle. Rotating LiDAR data naturally occupies this representation; solid-state LiDAR data does not, because the scanning pattern is not a regular azimuth sweep. Range image methods are computationally attractive because standard 2D convolutions apply directly, but they discard the 3D geometry and handle occlusion poorly.

For drone applications specifically, the preprocessing pipeline from raw LiDAR data to model-ready representation involves several steps that are non-trivial on edge hardware. The raw point cloud arrives as a stream of timestamped polar coordinates from the sensor's UART, Ethernet, or proprietary interface. Motion distortion compensation is necessary because the drone moves during the scan, causing points from different times to be misregistered. This requires integrating the IMU or odometry estimate over the scan period and applying a per-point transformation. On Tier 1 hardware this step alone may be expensive; on Tier 3 hardware with a dedicated IMU coprocessor it can be done in real time.

Coordinate transformation from sensor frame to body frame to world frame adds additional matrix operations. Filtering to remove ground points or points below a minimum intensity threshold reduces the cloud density and the subsequent computation. Only after these steps is the point cloud ready for voxelization or pillar encoding. The full preprocessing pipeline for a solid-state LiDAR on a Tier 3 platform typically takes 3 to 8 ms, a cost that must be included in the latency budget alongside the model forward pass.

The embedded LiDAR hardware landscape for drones reflects a clear tier dependence. Tier 1 platforms carry single-beam altimeters at most. Tier 2 platforms can

accommodate short-range solid-state units like the Benewake CE30, which provides a 320x240 depth image out to 8 meters at 30 Hz and draws 1.5 W, effectively pre-converting the point cloud to a depth image and eliminating the unordered-set problem. Tier 3 and Tier 4 platforms can carry multi-beam rotating or solid-state units with sufficient density for full 3D obstacle mapping and object detection. The mass and power constraints established in Chapter 2 are the binding constraints on this choice.

FMCW radar, which stands for Frequency Modulated Continuous Wave radar, operates on principles that differ from both camera and LiDAR in ways that produce qualitatively different data and qualitatively different inference challenges. Understanding why requires a brief excursion into the signal processing.

An FMCW radar transmits a continuous sinusoidal signal whose frequency is swept linearly upward over a chirp interval, typically a few tens to hundreds of microseconds. The transmitted signal reflects off objects in the environment and returns to the receiver with a time delay proportional to range. Because the transmitted frequency has changed during the propagation time, mixing the received signal with the current transmitted signal produces a beat frequency proportional to that time delay, and thus proportional to range. A Fourier transform over the fast-time samples of one chirp produces a range profile: a 1D complex spectrum whose peaks correspond to reflectors at different distances.

When the radar transmits multiple chirps in sequence, the relative phase of the return from a moving object shifts between chirps because the object has moved during the inter-chirp interval. A second Fourier transform over the slow-time dimension, stacking consecutive chirp returns, produces a Doppler spectrum that reveals the radial velocity of each reflector. The 2D output of these two transforms is the range-Doppler map, a 2D complex tensor indexed by range bin and Doppler bin, where the magnitude at each cell indicates the power returned by a reflector at that range moving at that radial velocity. This is the fundamental data representation of FMCW radar.

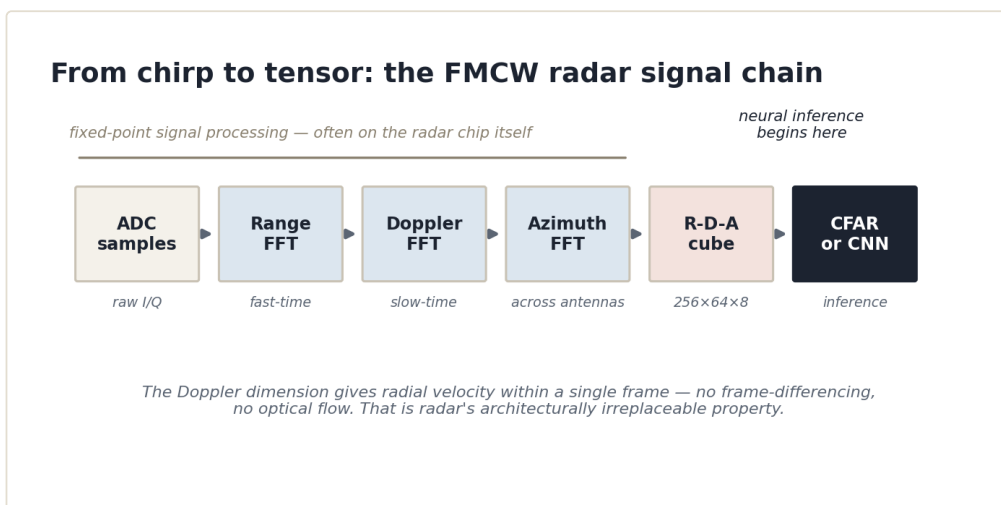


Figure 3.3 — The FMCW radar signal chain: from ADC samples through three FFTs to a range-Doppler-azimuth cube.

Multiple receive antennas add a third dimension. By comparing the phase of returns across spatially separated receive antennas, the radar infers the azimuth angle of each reflector through a third Fourier transform across the antenna aperture. This produces a range-azimuth-Doppler cube. For mmWave radar chips commonly found on drones, a typical configuration might be 256 range bins, 64 Doppler bins, and 8 virtual antennas, producing a tensor of approximately 256x64x8 complex values per frame. At 10 Hz frame rate, this is a modest data volume compared to camera or LiDAR, but the interpretation of individual tensor cells is nontrivial.

The key capability that distinguishes radar from both camera and LiDAR is direct velocity measurement. The Doppler dimension does not require differencing two frames to estimate velocity; it provides radial velocity for each detected reflector within a single frame. An object approaching at 3 m/s is distinguishable from a stationary clutter return within one range-Doppler frame. This is architecturally significant for drone inference pipelines because it means a radar-based detector can separate dynamic obstacles from static structure using single-frame information, which cameras cannot do without optical flow and LiDAR cannot do without frame-differencing.

The other significant property of radar is its behavior in adverse conditions. Radar signals at 77 GHz propagate through rain, fog, smoke, and dust with negligible attenuation compared to optical systems. A camera loses effective range in fog; LiDAR at 905 nm is affected by heavy rain and dust. An FMCW radar operating at 77 GHz loses almost none of its range capability in these conditions. For agricultural drones operating in dusty fields, inspection drones operating near industrial emissions, or search-and-rescue platforms operating in smoke, this property is not incidental.

The weaknesses of radar are angular resolution and target detectability for low-RCS objects. Angular resolution is limited by the physical antenna aperture, which on a drone-mounted radar chip is small. A TI AWR1843 evaluation module provides azimuth resolution of approximately 15 degrees with a small antenna array, compared to less than 0.2 degrees for a 32-beam LiDAR. This means radar can detect that something is present at a given range and velocity but cannot resolve its precise lateral position or shape. Thin cables, tree branches, and other geometrically complex structures may return weak or inconsistent radar echoes. These limitations are the reason radar is most powerful as a fusion component rather than a sole modality for precise 3D mapping.

The data tensor produced by a radar returns, before any detection processing, is noisy. Clutter, multipath reflections, and hardware imperfections produce false returns throughout the range-Doppler map. Traditional radar processing uses CFAR, which stands for Constant False Alarm Rate, detection to identify peaks in the range-Doppler map that exceed a locally adaptive threshold. CFAR produces a sparse set of detected targets, each represented as (range, velocity, azimuth, amplitude), analogous in structure to a point cloud. This sparse representation can be processed by point-based

networks similar to those used for LiDAR. Alternatively, the raw range-Doppler tensor can be passed directly to a CNN, treating the radar tensor as an image-like structure and letting the network learn feature extraction from raw complex-valued or magnitude data. Both paradigms appear in the literature and on deployed hardware, and they have different computational profiles that matter on edge hardware.

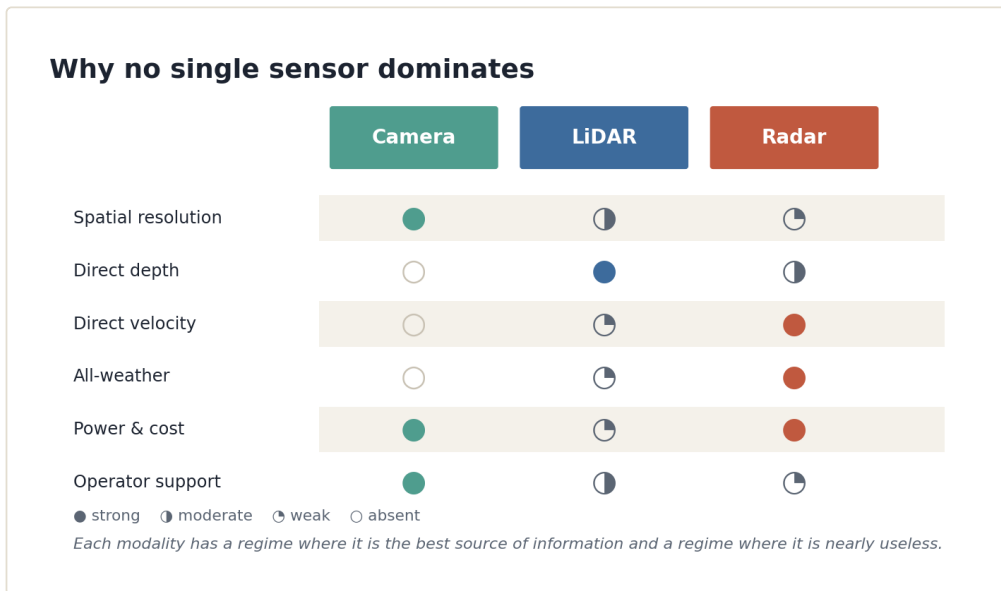


Figure 3.1 — Strength and weakness of camera, LiDAR, and radar across six attributes: no single sensor dominates.

The mmWave radar hardware available for drone integration spans a useful range. Texas Instruments offers the AWR1843 and AWR6843 as highly integrated single-chip solutions, drawing approximately 1.5 to 2.5 W, weighing a few grams, and requiring only a modest host processor for the inference step. The IMEC SAR radar platform offers higher angular resolution through synthetic aperture techniques at the cost of requiring controlled flight paths. Xaver and Ainstein produce application-specific drone radar modules with embedded CFAR processing, handing the host only detected targets rather than raw tensors. The trend in the hardware is toward more onboard preprocessing on the radar chip itself, reducing the bandwidth and compute burden on the main inference SoC.

Preprocessing on the host side, starting from a raw radar cube, involves the range FFT, Doppler FFT, azimuth FFT sequence described above, followed by magnitude computation and optionally CFAR detection. The FFT operations are highly regular and are excellent candidates for hardware acceleration; most radar-capable SoCs include FFT accelerators, and FPGAs implement the radar signal chain efficiently. When the radar chip delivers processed targets rather than raw I/Q samples, the host preprocessing cost drops dramatically, but control over the detection parameters is reduced.

Having examined each modality independently, the cross-modality comparison clarifies why no single sensor dominates. Cameras provide high spatial resolution and rich texture information at low cost and low power, but lack depth and suffer in low light and adverse weather. LiDAR provides precise 3D geometry with moderate spatial resolution, but produces sparse unstructured data that requires specialized architectures, at moderate to high cost and power. Radar provides direct velocity measurement and all-weather operation with coarse angular resolution and noisy returns, at low cost and low power.

The data structure each modality produces maps directly onto the compute primitives available on edge hardware. Camera frames map onto 2D convolutions, which all drone-deployable SoCs accelerate efficiently. LiDAR point clouds require either conversion to grid representations to recover convolution efficiency, or point-based operations that rely on irregular memory access patterns and are harder to accelerate. Radar tensors, when treated as 2D or 3D images, map onto convolutions, but the small spatial extent of the tensor compared to camera frames means the bottleneck is often preprocessing rather than inference. These are not obstacles to be overcome by trying harder; they are the physical realities that architecture choices must accommodate.

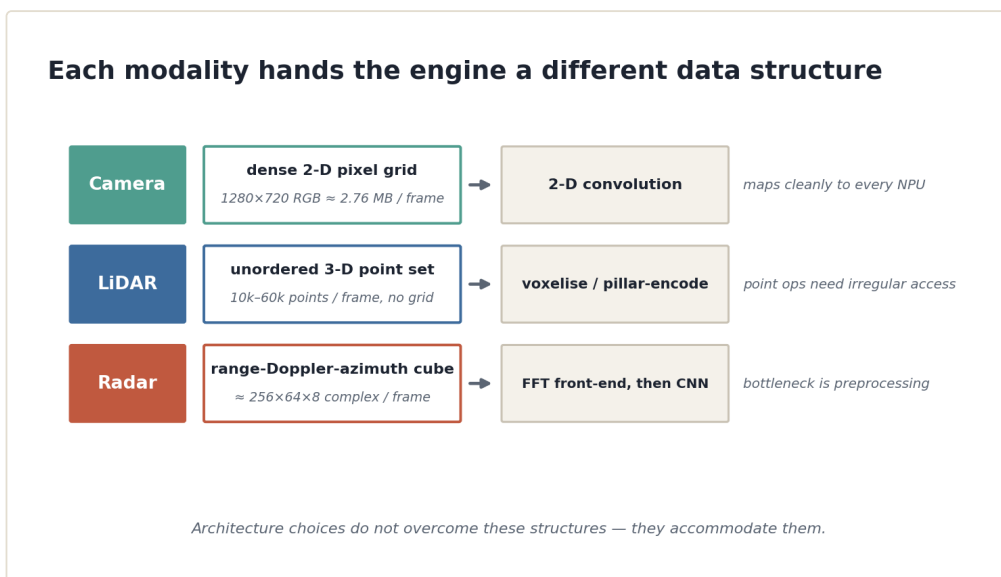


Figure 3.2 — Each modality hands the inference engine a different native data structure and compute primitive.

One further property deserves mention: temporal behavior. A camera captures the scene at a fixed frame rate determined by exposure time and readout speed. A LiDAR produces a complete scan at a rotation rate or accumulation interval. A radar produces a frame at a chirp sequence repetition rate. These rates are not the same, and they are not synchronized by default. A camera might run at 30 Hz, a solid-state LiDAR at 10 Hz, and a radar at 15 Hz. Fusing their outputs requires temporal alignment, and temporal alignment on edge hardware without a hardware synchronization trigger requires careful timestamping and interpolation logic. The latency implications of this

misalignment are real: if the fusion architecture waits for all three modalities to produce a frame before running inference, the effective output rate is limited to the slowest modality, and the waiting time costs latency budget. Asynchronous fusion architectures that update with whichever modality most recently produced data are more complex but more latency-efficient. Chapter 9 addresses these tradeoffs directly.

The sensor configurations associated with each tier from Chapter 2 now have precise meaning in terms of data volume, data structure, and compute requirements. A Tier 1 platform carrying only a camera and a single-beam altimeter is receiving a 640x480 dense image and one range scalar. A Tier 3 platform with a solid-state LiDAR, a 77 GHz radar, and a stereo camera is receiving a non-uniform point cloud, a range-Doppler cube, and two synchronized dense image streams simultaneously, and must fuse all three in time for a safe waypoint update. The gap between those two scenarios is not a matter of degree. It is a difference in the fundamental nature of the inference problem, rooted in the physics of what each sensor measures and the mathematics of what each sensor returns.

Chapter 4: Edge Hardware Landscape: SoCs, FPGAs, and Microcontrollers for Drone Inference

Edge Hardware Landscape: SoCs, FPGAs, and Microcontrollers for Drone Inference

The modality physics from the previous chapter explains what data arrives at the processor. What happens next is dictated entirely by the processor itself: its memory hierarchy, its available compute primitives, and the gap between peak TOPS on a datasheet and sustained throughput under the power constraints that a drone's battery actually enforces. These two chapters form a single argument. The sensor suite determines the data. The hardware determines what inference is possible on that data. The architecture must bridge them.

This chapter maps hardware to drone class with enough precision to make that bridge concrete. The mapping is not aspirational. It reflects what is flying today in commercial platforms, what is specified in reference designs from silicon vendors, and what the weight-power envelope of each class physically permits. A Class I nano-UAV cannot carry an Orin NX. A Class IV autonomous platform does not need to run on a Cortex-M7. The constraints are physical before they are economic, and the hardware families that result are genuinely different in the compute primitives they expose.

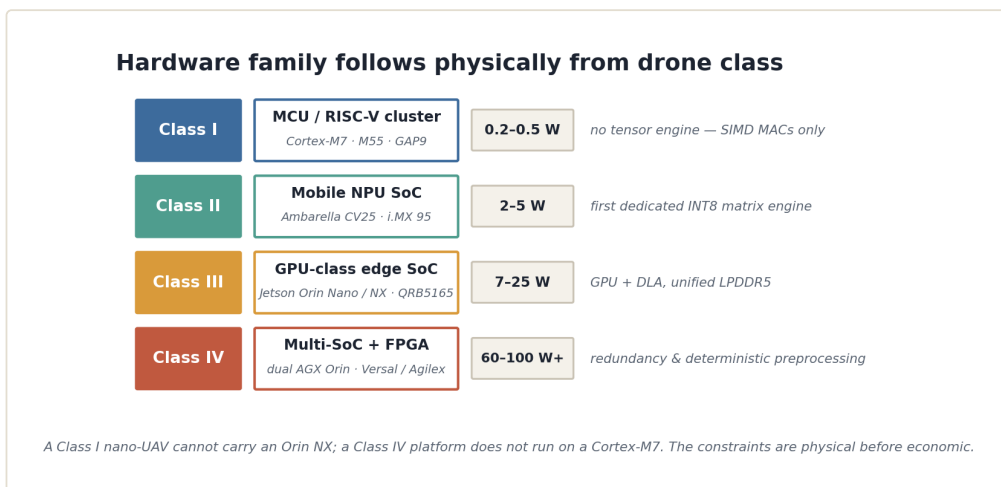


Figure 4.3 — Hardware family by drone class: the constraints are physical before they are economic.

The unit of comparison throughout is not raw TOPS. A tensor operation on a dedicated matrix engine is not the same as a tensor operation emulated across SIMD lanes. TOPS/W, sustained over a realistic inference workload, under thermal conditions representative of outdoor operation, is the number that matters. Every comparison here attempts to use that number rather than peak figures.

The Class I Platform: Microcontrollers and Their Ceilings

Class I drones carry payloads under 100 grams and operate on total system power budgets that leave perhaps 200 to 500 mW for inference. The silicon that fits into that envelope is dominated by two families: ARM Cortex-M series microcontrollers and, increasingly, RISC-V cores from vendors targeting embedded ML.

The Cortex-M7 is the practical ceiling of what most Class I platforms use today. Running at 480 MHz, with a six-stage superscalar pipeline and a hardware floating-point unit capable of single-cycle FP32 multiply-accumulate, the M7 can execute a shallow INT8 convolutional network at several frames per second on a VGA-resolution input if the network is designed carefully. The key qualifier is designed carefully. The M7 has no dedicated tensor acceleration. Its SIMD extension, Helium (MVE) on Cortex-M55 and later, expands that ceiling somewhat, providing 128-bit vector operations over INT8 and INT16 data that allow eight or sixteen multiply-accumulate operations per cycle. TensorFlow Lite Micro targets this architecture and provides kernels that use Helium intrinsics when available.

What the M7 and its successors cannot do is hide memory latency. Tightly coupled memory (TCM) on a Cortex-M7 is typically 512 KB to 1 MB. External flash or PSRAM adds latency that is not hidden by out-of-order execution, because these cores do not have out-of-order execution. The entire inference graph must be scheduled to keep activations in TCM during compute-intensive layers, or throughput collapses to memory bandwidth. This constraint shapes permissible model topology: layers must be narrow, activations must be small, and the programmer or compiler must manage tile sizes explicitly. CMSIS-NN, the ARM-maintained kernel library for Cortex-M inference, exposes this tiling explicitly so that operators working at this level understand what they are trading.

The RISC-V alternatives in this tier include the GreenWaves GAP9, a multi-core RISC-V cluster from GreenWaves Technologies that achieves substantially better performance per watt than a single M7 by executing a fine-grained parallel workload across eight compute cores plus a fabric controller. The GAP9 is notable because it was explicitly designed for sensory inference on battery-powered platforms and shipped with reference designs targeting ear buds and nano-drones. The Bitcraze Crazyflie research platform has used GAP8, its predecessor, to run convolutional networks for obstacle avoidance at sub-watt power budgets. The programming model is more demanding than Cortex-M because the application must manage DMA transfers between L2 shared memory and L1 per-core caches manually, but the throughput on INT8 convolution workloads is several times higher than a single M7 at comparable power.

The critical architectural fact for Class I inference is that there is no free convolution. Every multiply-accumulate comes from a general-purpose compute pipeline that was not designed specifically for neural network workloads. Network design for this tier is therefore less about choosing among existing model families and more about constructing the smallest network that accomplishes the detection task, subject to the

constraint that it fits in on-chip memory with no external memory access during inference. Models in the tens of kilobytes of parameter storage, with activations that peak at a few hundred kilobytes, are not exotic distillations of large networks. They are purpose-designed from scratch.

The Class II Platform: Mobile NPU SoCs

Class II drones carry payloads from 100 grams to roughly two kilograms and operate on system power budgets in the range of two to ten watts for the compute subsystem. This is the tier where dedicated neural processing units first appear as silicon features rather than vendor aspirations.

The Ambarella CV series represents one of the best-matched silicon families for Class II drone inference. The CV25 and CV28 SoCs combine ARM Cortex-A cores for general processing with a dedicated vector processing engine (the Cavalry engine in Ambarella's terminology) designed specifically for CNN inference on image and video data. Crucially, Ambarella targets the drone and automotive camera market directly, meaning the companion image signal processor is production-grade hardware, not a soft IP block. The CV25 consumes approximately two to three watts under active inference workloads, fits in a small BGA package appropriate for compact PCB designs, and provides enough CNN throughput to run a YOLOv8-nano class detector on 640x480 input in real time. The memory architecture includes an on-chip DRAM interface, removing the TCM constraint that dominates Class I design.

The NXP i.MX 9 series takes a different approach. The i.MX 93 and i.MX 95 include an eIQ Neutron NPU, an ARM-architecture neural accelerator with configurable data paths targeting INT8 workloads. The i.MX 95 also includes Cortex-A55 and Cortex-M7 cores in a heterogeneous arrangement, allowing safety-critical flight control to run in hard real-time on the M7 while perception inference runs on the A55 cluster and NPU simultaneously. This heterogeneous arrangement is increasingly the design pattern for Class II: one hard-real-time domain for flight control, one application-processor domain for inference, connected by a shared memory fabric with access controlled by the OS.

The ST Microelectronics STM32MP2 series occupies a similar niche with a neural processing unit called the NPU 1.8 TOPS core, targeting embedded vision applications. ST's positioning is explicitly toward the industrial IoT and edge AI markets, and the associated toolchain (STM32Cube.AI) generates deployment code from ONNX or Keras models with quantization applied automatically.

For Class II, the critical architectural shift from Class I is the existence of a hardware unit that can execute dense INT8 matrix multiply-accumulate operations in a single-cycle pipeline rather than across general-purpose SIMD lanes. This changes what model families are feasible. Depthwise separable convolutions, which are efficient on Class I because they reduce total multiply-accumulate count, become relatively less efficient on Class II NPUs because the depthwise step is a channel-wise operation that

may underutilize a matrix engine designed for dense matmul. The implication is that model topology choices that are optimal on a Cortex-M7 may not be optimal on a CV25, and vice versa. Profiling on target hardware is not optional; it is the only method that produces reliable latency estimates.

The memory hierarchy on Class II SoCs is more forgiving than Class I but still bounded. Typical on-chip SRAM is in the range of one to four megabytes, with LPDDR4 or LPDDR4X providing external bandwidth at fifteen to thirty gigabytes per second. The latency gap between on-chip SRAM and external DRAM remains substantial, which means large activations that spill from on-chip memory still carry a throughput penalty. Quantization from FP32 to INT8 is not just a model compression strategy at this tier; it is often the factor that determines whether a given model's peak activation tensor fits in on-chip memory, avoiding DRAM spill entirely.

The Class III Platform: GPU-Class Edge SoCs

Class III drones are industrial-grade platforms carrying two to twenty-five kilograms of total mass, with compute subsystems that can absorb ten to thirty watts continuously. This is the tier where small GPU silicon appears, and where the inference architecture landscape expands dramatically.

The NVIDIA Jetson Orin series is the dominant platform in this space for research and commercial applications requiring versatile deep learning support. The Orin Nano, at its seven-watt power mode, provides 20 TOPS of INT8 throughput from a combined GPU and deep learning accelerator (DLA) architecture. The Orin NX at ten watts provides 70 TOPS. The DLA is a fixed-function accelerator that executes a subset of operations (convolution, pooling, element-wise operations) with better energy efficiency than the GPU, while the GPU handles operations outside the DLA's supported set and provides the general programmability. The programming model exposed through TensorRT allows a network to be partitioned at compile time, with DLA-compatible subgraphs assigned to the DLA and GPU-fallback subgraphs running on CUDA cores. For inference-heavy workloads where the network is entirely DLA-compatible, the energy efficiency gain over GPU-only execution is substantial.

The Orin's memory architecture is unified: the CPU, GPU, and DLA share the same physical LPDDR5 pool through a central interconnect. This is architecturally significant because it eliminates the explicit copy operations that would otherwise be required to move activations from CPU memory to a discrete GPU's video memory. In a fused camera-lidar pipeline, for example, a preprocessing step on the CPU can write directly into a memory region that the DLA then reads for the first convolutional layer, without a DMA transfer across a PCIe or SPI boundary. The sustained bandwidth to that shared pool is approximately 68 GB/s on Orin Nano, which is sufficient to support concurrent sensor ingestion and inference without bandwidth starvation.

The Qualcomm RB5 (Robotics RB5) platform targets the same tier from a different direction. Built around the Qualcomm QRB5165 SoC, it combines Kryo 585 CPU cores with an Adreno 650 GPU, a Hexagon DSP with Hexagon Vector eXtensions (HVX), and a dedicated neural processing unit under the AI Engine umbrella. The combined INT8 throughput is rated at 15 TOPS. The Hexagon DSP is particularly relevant for radar signal processing workloads: its fixed-point vector operations are efficient for the FFT and CFAR operations that precede neural inference on radar data, allowing the signal processing front-end and the inference back-end to run on different compute units simultaneously without competing for GPU resources.

The Rockchip RK3588, popular in lower-cost Class III configurations and some Class II upper-range platforms, includes a 6 TOPS NPU alongside ARM Cortex-A76 and A55 cores. At roughly five to eight watts for the full SoC under load, it represents a cost-effective alternative for applications that do not require Orin-level throughput. Its NPU supports INT8 and INT4 quantization, and the RKNN toolchain provides deployment paths from PyTorch and ONNX models.

The architectural constraint that Class III hardware introduces, somewhat counterintuitively, is scheduling complexity. A Class I Cortex-M7 runs one thing at a time in a straightforward control flow. A Class III platform with a CPU cluster, a GPU, a DLA, and a DSP runs multiple concurrent workloads, and the competition for shared memory bandwidth among them can produce latency variability that is not visible in single-workload benchmarks. A perception pipeline that benchmarks at eight milliseconds in isolation may run at twelve milliseconds when the IMU processing, GPS filtering, and flight control tasks are also consuming CPU cycles and cache lines. Thermal throttling compounds this: the Orin Nano at seven watts in its power capped mode will reduce GPU and DLA frequencies if the SoC junction temperature exceeds threshold. In an enclosure on a drone without active cooling, that threshold can be reached within minutes of sustained flight in direct sunlight.

The Class IV Platform: Multi-Chip and Multi-SoC Configurations

Class IV drones are large autonomous platforms above twenty-five kilograms, capable of carrying compute payloads that would be impractical on smaller vehicles. Agricultural mapping aircraft, long-endurance surveillance platforms, and large cargo drones fall into this class. The compute budget can reach one hundred watts or more for the inference subsystem, and the architecture frequently involves multiple SoCs operating in a coordinated configuration.

The canonical multi-SoC arrangement for Class IV is two or more Jetson Orin NX or AGX Orin modules connected via high-speed interconnect, with sensor data distributed across them by a network-on-chip or a dedicated switch fabric. The NVIDIA Jetson AGX Orin at its full 60-watt mode delivers 275 TOPS INT8, combining a 2048-core Ampere GPU, a 12-core Cortex-A78AE CPU cluster, and two DLA engines. A dual-AGX Orin configuration provides over 500 TOPS sustained throughput, which is sufficient to run

separate model instances for each sensor modality in parallel and a separate fusion model in a third inference process, all at their respective sensor frame rates.

The engineering challenge at Class IV is not raw throughput but reliability and fault tolerance. A single SoC failure cannot be allowed to produce a loss of all perception capability on a twenty-five-kilogram platform operating beyond visual line of sight. Redundancy architectures therefore run independent inference pipelines on separate silicon, with a voting or arbitration layer that determines which pipeline's outputs to trust when they disagree. The Cortex-A78AE in the AGX Orin is specifically the automotive-enhanced variant, which includes lockstep execution capability for safety-critical tasks, reflecting the shared ancestry between autonomous driving and autonomous aviation at this tier.

FPGAs appear in Class IV configurations as pre-processing and co-processing accelerators rather than primary inference engines. A Xilinx Versal or Intel Agilex FPGA sitting between the sensor interfaces and the inference SoCs can handle sensor depacketing, time synchronization, point cloud preprocessing, and range-Doppler FFT computation at deterministic latency with zero jitter. The determinism is the value proposition: an FPGA running a fixed pipeline produces outputs at exactly the interval programmed into its logic, regardless of interrupt load or OS scheduling state. For a fusion architecture that must timestamp sensor data to within one millisecond for reliable alignment, that determinism is worth the added board space and design complexity.

Custom ASIC development enters Class IV at sufficient production volumes. Dedicated accelerators for 3D sparse convolution, designed to accelerate the type of lidar inference described in Chapter 8, have appeared in automotive silicon (Mobileye's EyeQ series, for example) and are being examined in defense and commercial aviation programs. At drone scale, the volumes that justify ASIC development are reached only in the largest production programs, which is why this chapter treats FPGAs and multi-SoC as the practical Class IV configuration rather than bespoke silicon.

Memory Hierarchy as Architecture Constraint

Across all tiers, the memory hierarchy is not a background detail. It is an active constraint on architecture design, as real as the MAC count limit.

On a Cortex-M7, the on-chip TCM is the only memory that sustains full compute throughput. On a Class II NPU SoC, the on-chip SRAM of two to four megabytes is the working set that sustains NPU utilization; spilling to LPDDR reduces effective throughput by a factor that depends on the bandwidth gap. On a Class III Orin, the shared LPDDR5 pool is large enough to hold multiple model weights and intermediate activations simultaneously, but the 68 GB/s bandwidth is shared among CPU, GPU, and DLA, meaning concurrent workloads create real bandwidth contention.

The practical implication is that INT8 quantization, discussed in depth in Chapter 6, is not purely a model quality tradeoff. It is a memory architecture alignment strategy. A model that fits its peak activation tensor in on-chip SRAM at INT8 but not at FP32 gains throughput from quantization that has nothing to do with reduced arithmetic cost. The reduction in bytes transferred from external memory is itself the speedup. This is why quantization is mandatory at Class I and strongly motivated at Class II, and why the argument for quantization at Class III is not primarily about fitting the model but about sustaining utilization without bandwidth starvation.

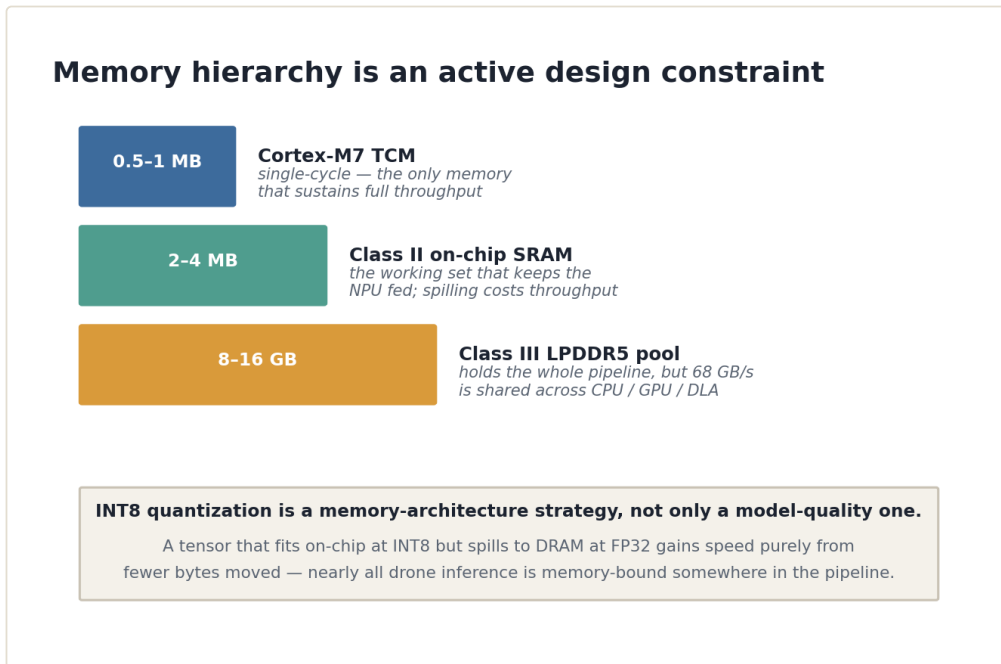


Figure 4.2 — The memory hierarchy as an active design constraint; quantization is a memory-architecture strategy.

Comparative Performance Summary

The following table summarizes sustained INT8 inference performance and efficiency across representative platforms for each drone class. Peak TOPS figures from vendor datasheets are noted separately from practical sustained figures derived from published benchmarks on representative workloads.

1. Class I, GAP9 (GreenWaves): Peak INT8 TOPS approximately 0.05, practical sustained TOPS approximately 0.03, power draw 50 to 100 mW, efficiency approximately 300 to 500 GOPS/W.
2. Class I, Cortex-M55 with Helium MVE: Peak INT8 TOPS approximately 0.02, practical sustained TOPS approximately 0.01, power draw 30 to 80 mW, efficiency approximately 200 to 300 GOPS/W.
3. Class II, Ambarella CV25: Peak INT8 TOPS approximately 2, practical sustained TOPS approximately 1.2, power draw 1.5 to 3 W, efficiency approximately 400 to 600 GOPS/W.

4. Class II, NXP i.MX 95: Peak INT8 TOPS approximately 4, practical sustained TOPS approximately 2.5, power draw 3 to 5 W, efficiency approximately 500 to 700 GOPS/W.
5. Class III, Jetson Orin Nano (7 W mode): Peak INT8 TOPS 20, practical sustained TOPS approximately 12, power draw 7 W, efficiency approximately 1.5 to 2 TOPS/W.
6. Class III, Qualcomm RB5 (QRB5165): Peak INT8 TOPS 15, practical sustained TOPS approximately 9, power draw 5 to 10 W, efficiency approximately 1 to 2 TOPS/W.
7. Class IV, Jetson AGX Orin (60 W mode): Peak INT8 TOPS 275, practical sustained TOPS approximately 180, power draw 60 W, efficiency approximately 3 to 4 TOPS/W.

Efficiency peaks in the Class II tier, not at the top of the hierarchy. This is not surprising: dedicated NPU silicon designed for a narrow workload at two to five watts can optimize its datapath far more aggressively than a large GPU, which carries die area for programmability and throughput diversity that Class II applications do not need. The Class IV AGX Orin recovers efficiency relative to Class III Orin Nano partly because larger die sizes amortize memory interface and control logic overhead across more compute units.

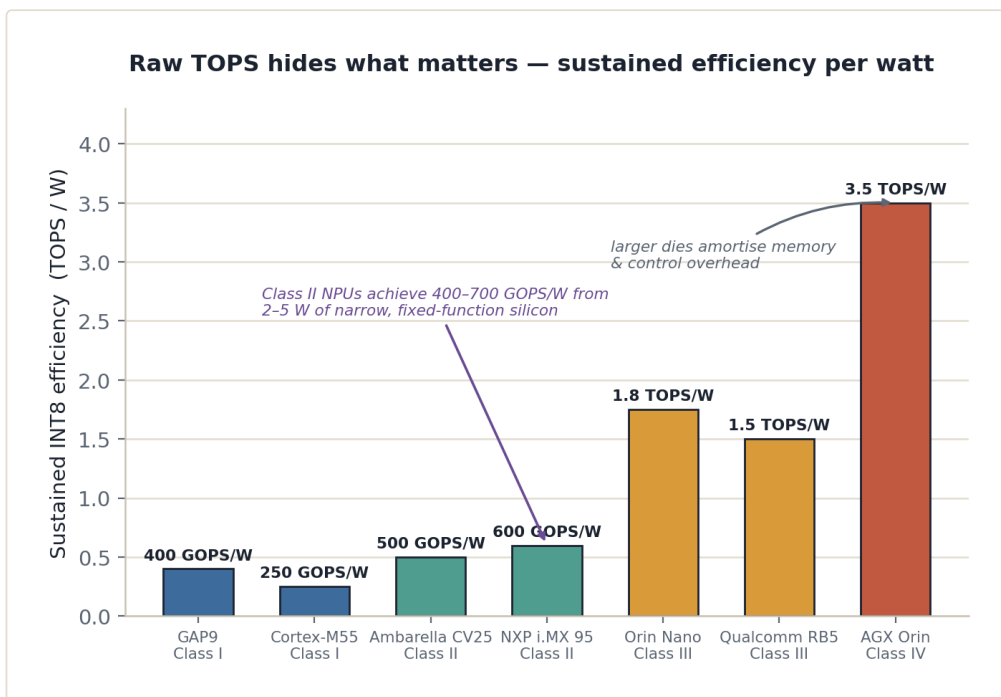


Figure 4.1 — Sustained INT8 efficiency per watt across representative platforms for each drone class.

Hardware Determines the Model, Not the Other Way Around

The conclusion this chapter forces is uncomfortable for machine learning practitioners accustomed to selecting architectures from benchmark leaderboards: the hardware platform comes first, and the architecture must be derived from it, not fitted to it after training. A model achieving 45.2 mAP on COCO is meaningless on a Class I platform if

it requires 8 MB of activation memory that cannot be staged through 512 KB of TCM without recomputing activations. A transformer architecture that achieves excellent detection quality on a Class III benchmark may not map efficiently onto a Class II NPU's fixed-function matrix engines, producing throughput one-quarter of what a structurally simpler CNN achieves on the same silicon.

This is the engineering contract that the remaining chapters operationalize. The sensor data structures from Chapter 3 arrive at the hardware described here. What can be done with that data, at what latency, at what power cost, and with what confidence is determined by the intersection of the two. Chapters 6 through 9 derive the architectures that fit within the boxes drawn here. Chapter 11 provides the framework for working through that derivation systematically for a given platform, mission, and sensor suite.

Chapter 5: Firmware and OS Layer: From RTOS to Embedded Linux

Firmware and OS Layer: From RTOS to Embedded Linux

The hardware described in Chapter 4 does not execute model weights directly. Between the silicon and the inference runtime sits a firmware and operating system layer that can add hundreds of microseconds of latency, misalign DMA transfers, delay interrupt service, and introduce non-determinism that makes an otherwise well-fitted architecture miss its timing budget. Understanding this layer is not optional for drone inference engineers. It is where the gap between benchmark numbers and deployed behavior lives.

The firmware stack is not uniform across drone classes. Class I platforms run bare-metal code or a lightweight RTOS on a single Cortex-M7 or RISC-V core with no virtual memory and no general-purpose scheduler. Class II platforms introduce mobile SoCs where a minimal embedded Linux instance may coexist with a dedicated NPU managed through a vendor firmware blob. Class III platforms run full embedded Linux with the real-time preemption patch or a dual-OS configuration pairing a Linux host with an RTOS co-processor. Class IV platforms may run multiple Linux instances across SoCs, connected by a high-bandwidth fabric. Each configuration imposes different constraints and exposes different knobs, and the failure modes at each level are distinct enough to treat separately.

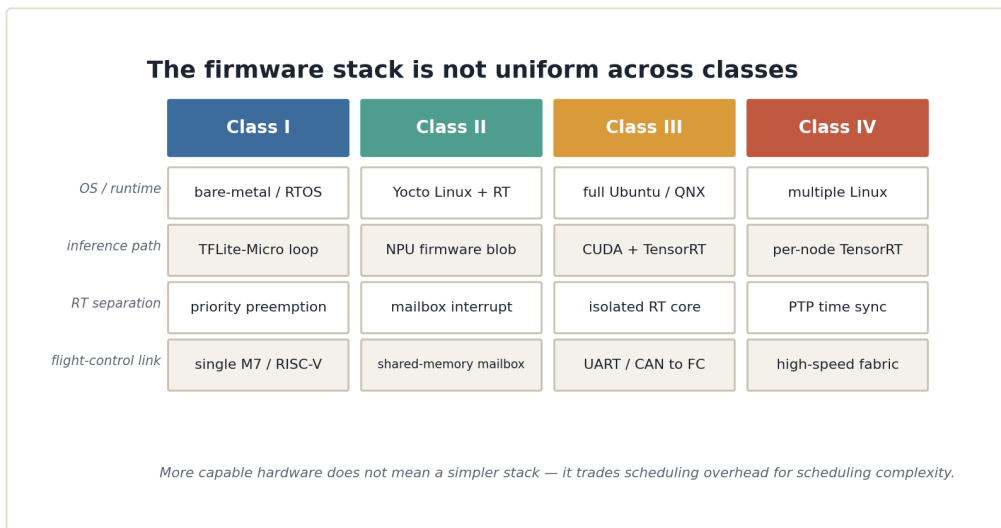


Figure 5.1 — The firmware and OS stack is not uniform across drone classes.

Bare-Metal and RTOS Environments for Class I

On a Class I platform such as an STM32H7 running at 480 MHz with 512 KB of tightly coupled memory, there is no operating system in any meaningful sense. The firmware is a superloop or an RTOS task set that runs entirely in physical address space with no

memory protection unit configuration separating inference code from flight control code. The advantage is zero scheduling overhead and zero kernel-to-user boundary cost. The inference forward pass is a function call. It completes when it completes.

The problem is that the superloop approach serializes everything. Sensor acquisition, IMU integration, flight controller updates, and inference must time-share a single execution thread. A 30 ms inference window on an 8-INT8-MAC-per-cycle microcontroller is not 30 ms of wall clock time available for inference. It is 30 ms total budget shared with all ISR handlers, DMA completion callbacks, and control loop iterations, some of which have hard real-time deadlines that inference does not have the authority to violate.

FreeRTOS resolves the serialization problem by introducing priority-based preemption. The flight control task runs at the highest priority with a 1 ms tick deadline. The inference task runs at a lower priority and can be preempted mid-forward-pass. This is safe only if the inference runtime is designed to tolerate preemption, meaning it does not hold any shared resource lock during the longest uninterruptible segment of execution. Most lightweight TFLite Micro deployments satisfy this property because they operate entirely within a statically allocated arena without dynamic allocation or global state mutation during the forward pass.

Zephyr OS offers a more structured alternative, with a device model, thread abstraction, and a kernel that enforces stricter resource separation. For Class I drone inference, Zephyr's advantage over FreeRTOS is primarily in driver quality and portability across Cortex-M and RISC-V targets, not in runtime scheduling behavior. The scheduling model is similar: a thread assigned to inference runs cooperatively or preemptively at a priority level below flight-critical threads, and the inference window is whatever CPU time remains after higher-priority work is satisfied.

The critical firmware-level knob in bare-metal and RTOS environments is interrupt coalescing. A camera frame arriving via SPI at 30 fps generates 30 interrupts per second. A lidar unit scanning at 10 Hz generates 10. Each interrupt, if handled immediately, inserts a context switch overhead of 200 to 400 CPU cycles on a Cortex-M7. For a platform executing inference at perhaps 5 to 10 million cycles per forward pass, a burst of sensor interrupts mid-inference can add 5 to 10 percent latency variance. The firmware pattern that suppresses this variance is to configure DMA to accumulate a full sensor frame into a ring buffer and fire a single completion interrupt per frame rather than per DMA burst. The inference task then reads from the ring buffer at the start of each inference cycle with no interrupt interaction during the forward pass.

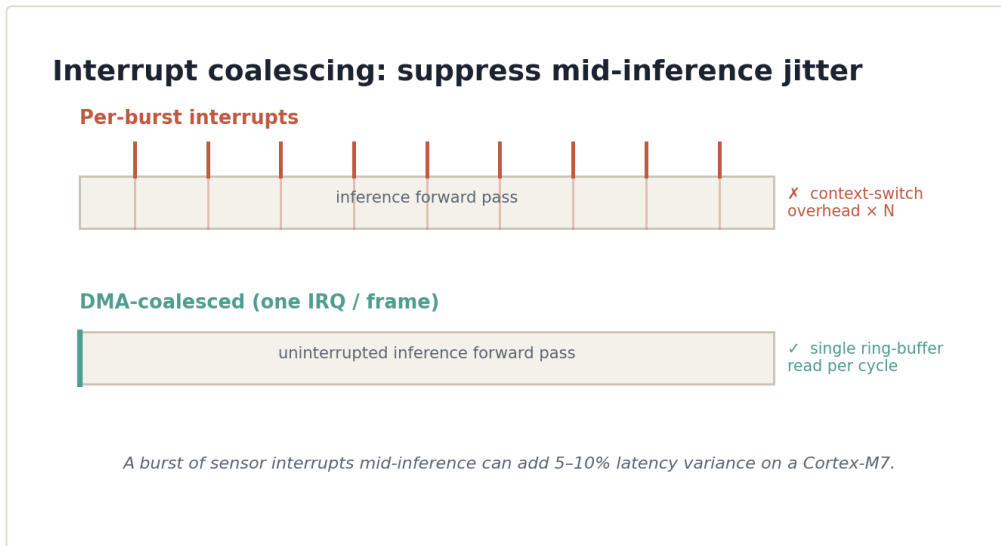


Figure 5.2 — Interrupt coalescing: a DMA ring buffer fires one interrupt per frame instead of one per burst.

Memory locking in RTOS environments means something different than in Linux. There is no virtual memory pager to lock against. The equivalent concern is ensuring that model weights and the inference activation arena are placed in the correct memory region at link time. On the STM32H7, DTCM (data tightly coupled memory) at 64 KB provides single-cycle latency for the core but is not accessible by DMA. SRAM accessible by DMA has one additional wait state. The activation buffer must be in DTCM for performance; the input tensor staging buffer must be in DMA-accessible SRAM to receive camera data without CPU involvement. Getting this mapping wrong by even one segment costs 10 to 15 percent throughput on compute-bound inference workloads.

Class II: The Mixed-Mode Problem

Class II platforms introduce a new category of complexity. A platform like the Ambarella CV25 or NXP i.MX 95 runs a multi-core SoC where one or more Arm Cortex-A application cores host an embedded Linux instance while a separate NPU or DSP subsystem runs under its own firmware blob, sometimes supplied as a black box by the silicon vendor. The two domains communicate through shared memory regions and a mailbox interrupt mechanism. This is architecturally clean but practically treacherous.

The embedded Linux instance on a Class II platform is typically a minimal Yocto-based build running a 5.x or 6.x kernel with the PREEMPT_RT patch applied. PREEMPT_RT converts most kernel spinlocks to sleeping mutexes, reduces interrupt handler duration by moving work to kernel threads, and allows user-space processes to achieve scheduling latencies in the 50 to 200 microsecond range rather than the millisecond range of a standard Linux build. For an inference task with a 20 ms latency budget and a sensor input that arrives every 33 ms, the scheduling jitter introduced by a non-RT kernel may consume 5 to 10 ms of that budget through random interference from kernel housekeeping tasks.

The SCHED_FIFO and SCHED_RR real-time scheduling policies in Linux are the primary user-space lever for managing this. An inference process pinned to a specific CPU core with SCHED_FIFO at priority 90 and all other processes at SCHED_OTHER will receive the CPU within one kernel tick of becoming runnable. CPU affinity assignment via taskset or the cpuset cgroup prevents the scheduler from migrating the inference thread to a core that is currently handling network or USB interrupts, which is a common source of latency spikes when camera data arrives over USB.

The NPU firmware blob on Class II platforms deserves explicit treatment because it is often the least visible and least controllable part of the stack. The NPU on a device like the CV25 has its own instruction set, its own on-chip memory, and its own scheduler. The Linux-side inference SDK submits a compiled network binary to the NPU via an ioctl call into a vendor kernel driver. The driver DMA-transfers the input tensor into NPU-accessible memory, signals the NPU firmware, and waits on a completion interrupt. The user-space inference call appears synchronous but blocks on a semaphore until the NPU interrupt fires.

The latency components in this path are: the Linux system call overhead for ioctl (typically 1 to 5 microseconds), the DMA transfer time for the input tensor (a 1 MB feature map at 6.4 GB/s costs roughly 150 microseconds), the NPU execution time (deterministic once started), and the interrupt service and semaphore wake latency (50 to 150 microseconds on a patched RT kernel). The sum of the non-inference components can reach 300 to 400 microseconds per forward pass, which is negligible for a 20 ms budget but significant for a 5 ms obstacle avoidance budget. The firmware knob that addresses this is the pinned DMA buffer: allocating the input tensor in a CMA (contiguous memory allocator) region that is permanently mapped for DMA eliminates one DMA setup operation per inference call, saving 50 to 100 microseconds.

Class III: Full Linux and Real-Time Co-processors

Class III platforms such as the Jetson Orin NX or Qualcomm RB5 run full Ubuntu or QNX-based Linux with the full kernel feature set, including GPU drivers, CUDA, and vendor inference SDK libraries like TensorRT. The inference runtime is no longer a lightweight operator loop but a compiled CUDA graph or TensorRT execution context that occupies its own CUDA stream and shares the GPU with any visualization or telemetry workloads.

The OS scheduling challenge on Class III shifts from latency minimization to isolation. A Jetson Orin NX has a 6-core Arm Cortex-A78AE CPU cluster and an Ampere GPU. The CPU cores are not interchangeable for inference purposes. The Cortex-A78AE supports memory tagging and ECC, and the Jetson firmware partitions cores into a real-time cluster and a general-purpose cluster through the CPUfreq and CPU isolation governors. Assigning the inference pipeline dispatch thread to an isolated core via isolcpus prevents the scheduler from placing any other user-space work on that core, which eliminates scheduling jitter from competing processes.

TensorRT on Jetson executes inference on the GPU, but the CPU is still responsible for enqueueing work into the CUDA stream, managing tensor lifetimes, and responding to the CUDA event that signals completion. This CPU involvement is small but non-zero. The established pattern for minimizing it is to use CUDA graphs, which pre-record the sequence of CUDA kernel launches for an inference pass and replay them with a single API call. Graph replay bypasses the CUDA driver overhead for individual kernel submissions, reducing CPU involvement per inference call from 50 to 100 microseconds to under 10 microseconds. TensorRT 8.x and later versions support CUDA graph capture natively for static-shape networks.

DMA on Class III platforms is handled differently than on Class II. The Jetson memory model exposes a unified memory space where CPU and GPU share physical DRAM, but access patterns still matter. Input tensors prepared by the camera driver in CPU-accessible memory must be made coherent before the GPU reads them. The two strategies are: using pinned (page-locked) host memory allocated with `cudaMallocHost`, which enables direct DMA from camera driver to pinned memory without a separate copy, and using `cudaMemcpyAsync` with a CUDA stream that overlaps the memory transfer with a preceding forward pass on the GPU. The choice between them depends on whether the camera ISP pipeline can target a CUDA-pinned buffer directly. The Jetson camera framework (`libargus` and `nvmm`) does expose zero-copy paths where ISP output is written directly into a `NvBuffer` that can be imported into CUDA with minimal overhead, and using this path correctly can save 2 to 4 ms per frame relative to a naive `memcpy` approach.

A common Class III configuration pairs the main SoC with a separate microcontroller handling flight control, connected via UART or CAN. In this architecture, the Jetson or RB5 runs inference and publishes detection or obstacle results to the flight controller over the low-latency link. The firmware concern here is the link scheduling: if the flight controller polls the inference SoC and the inference SoC is mid-forward-pass when the poll arrives, the response latency is determined by the remaining inference time plus the communication overhead. The design pattern that avoids this is a shared memory mailbox where the inference SoC writes results asynchronously and the flight controller reads the latest result at its own cadence, accepting that the result may be one frame stale. For obstacle avoidance at 10 m/s flight speed and 100 ms inference cadence, one frame of staleness corresponds to 1 m of position uncertainty, which must be accounted for in the safety margin of the avoidance algorithm.

Class IV: Multi-SoC Configurations

Class IV platforms assemble multiple compute nodes, each running its own Linux instance, connected by PCIe, Ethernet, or a proprietary high-bandwidth fabric. The firmware concerns at this level are primarily about synchronization across nodes and about ensuring that inference pipelines on different nodes can share a common time reference for sensor fusion.

The sensor fusion problem at Class IV scale requires that camera frames from node A and lidar point clouds from node B carry timestamps that are coherent to within the synchronization window of the fusion algorithm. Achieving this over Ethernet requires PTP (Precision Time Protocol) with hardware timestamping support in the NIC, which synchronizes clocks across nodes to within 1 microsecond when the network is not congested. Under congestion, PTP synchronization error grows to tens of microseconds. The firmware knob is traffic shaping: reserving a VLAN or a dedicated network interface for PTP traffic prevents telemetry or logging traffic from introducing queuing delay into the synchronization path.

CPU affinity and interrupt coalescing at Class IV are applied in the same manner as Class III but must be replicated consistently across all compute nodes. A common failure mode is that the operator tunes the primary inference node carefully but leaves the secondary nodes at default scheduling settings, so the secondary node's inference results arrive with 5 ms additional jitter that degrades fusion quality. The operational discipline required is that firmware configuration is version-controlled as part of the build system, not applied manually at deployment time.

The Latency Budget as a First-Class Firmware Artifact

Across all drone classes, the underlying principle is the same: the latency budget for one inference cycle is a resource that must be explicitly allocated across all firmware components that consume it. A budget of 33 ms for a 30 fps camera-based inference pipeline must account for:

1. Camera ISP pipeline delay from exposure completion to frame ready: typically 2 to 5 ms depending on ISP configuration.
2. DMA transfer from camera buffer to inference input tensor: 0.1 to 2 ms depending on tensor size and bus width.
3. RTOS or OS scheduling latency from inference task wake to CPU availability: 0.05 to 2 ms depending on platform and RT configuration.
4. Inference forward pass execution: 5 to 30 ms depending on model and platform.
5. Output tensor readback and post-processing (NMS, decode): 0.5 to 3 ms.
6. Result publication to flight controller or fusion module: 0.1 to 1 ms depending on communication mechanism.

The total of the non-inference components in this list can reach 10 ms on a poorly configured Class II system. That 10 ms is not free. It comes directly from the margin available for the model to be larger or for the system to process a more complex scene. Firmware optimization that reduces non-inference overhead from 10 ms to 3 ms is equivalent, in terms of inference budget, to deploying a model that is more than twice as fast per forward pass. This equivalence is why firmware engineering is not separable from model architecture engineering on constrained platforms.

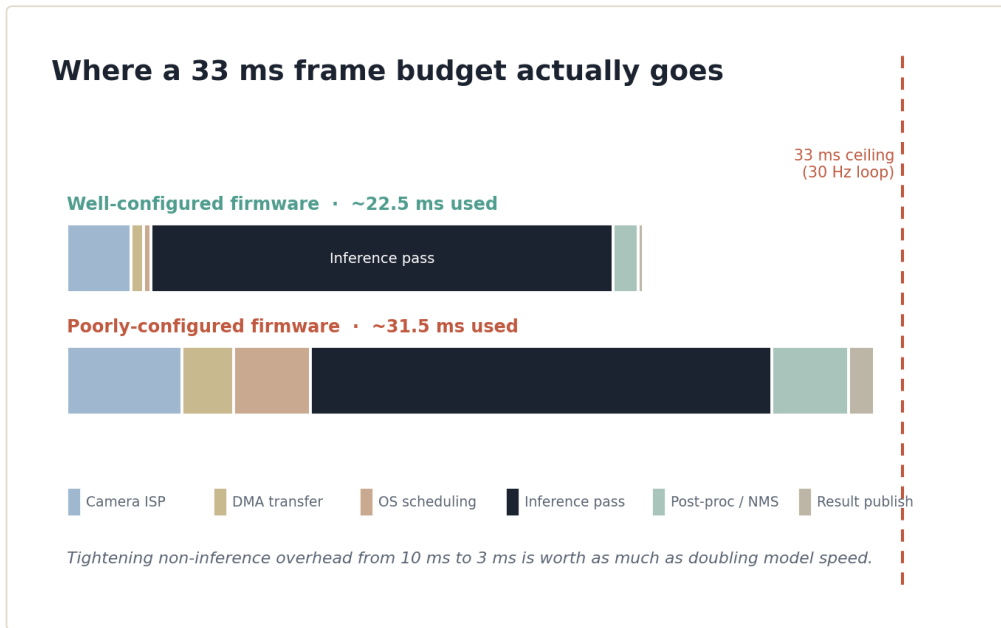


Figure 5.3 — Where a 33 ms frame budget goes: well- versus poorly-configured firmware.

The chapters that follow derive model architectures against timing budgets that assume a well-configured firmware layer. The interrupt coalescing patterns, DMA buffer strategies, CPU affinity assignments, and RT scheduling policies described here are preconditions for those budgets being achievable in practice. A model that fits its latency target in simulation but not on the deployed airframe has, almost invariably, encountered the firmware layer in a form its developers did not anticipate.

Chapter 6: Inference Architecture Fundamentals for Constrained Platforms

Inference Architecture Fundamentals for Constrained Platforms

The firmware layer establishes what time remains for inference. What happens inside that window is determined by a different set of decisions: which operations the model performs, in what numerical precision, at what structural density, and whether adjacent operations have been collapsed into forms the hardware can execute without stalling. These decisions are not independent of one another. Quantization changes the memory layout that pruning operates on. Pruning changes the activation statistics that quantization calibration observes. Operator fusion changes which quantization granularities are even expressible. The techniques compose, and understanding each one in isolation is necessary but not sufficient.

This chapter works through the four foundational compression and optimization techniques in order of their typical application in a deployment pipeline: quantization, pruning, knowledge distillation, and operator fusion. The treatment is algorithm-level. The goal is to make the mechanics precise enough that their interactions become visible and their failure modes predictable.

Quantization

A 32-bit floating-point weight consumes four bytes. An INT8 weight consumes one. The arithmetic implication is that an inference accelerator operating on INT8 data can issue four times as many multiply-accumulate operations per memory transaction compared to FP32, which on bandwidth-limited edge hardware is often the primary performance constraint rather than raw compute throughput. The memory saving compounds through the activation tensors as well, which for a convolutional network with large spatial feature maps are frequently the larger memory consumer during inference.

The mapping from a floating-point value x to an integer value x_q is:

$$x_q = \text{clamp}(\text{round}(x / s) + z, q_{\min}, q_{\max})$$

where s is a scale factor, z is a zero-point (an integer that represents the floating-point zero in the quantized domain), and the clamp bounds correspond to the representable range of the target integer type. For symmetric INT8, z is zero and q_{\min} and q_{\max} are -127 and 127. For asymmetric INT8, z is nonzero and the full range -128 to 127 is used. The inverse mapping for dequantization is:

$$x_{\text{approx}} = s * (x_q - z)$$

The quantization error introduced by this mapping is bounded by $s/2$ for values inside the clamp range. Values outside the clamp range suffer from saturation error, which is

unbounded and qualitatively different from the rounding error inside the range. Calibrating the scale factor s therefore requires choosing a clipping range that keeps the distribution of activations or weights within bounds while not wasting range on empty quantization bins.

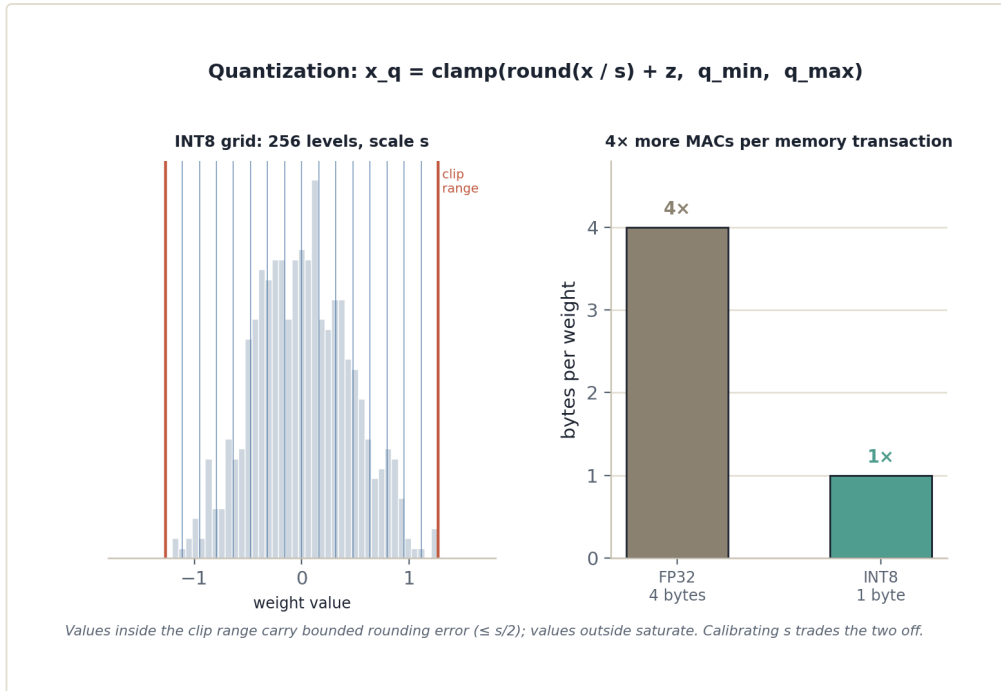


Figure 6.1 — The FP32-to-INT8 quantization mapping and the resulting fourfold reduction in bytes per weight.

Post-training quantization calibrates s using a representative dataset without retraining. The simplest calibration uses the observed min and max of the tensor across the calibration set. A better approach uses percentile clipping, discarding the top and bottom 0.1 to 1 percent of values, which prevents a single extreme activation from expanding the quantization range and wasting bins. The best approach for activations with long-tailed distributions minimizes the KL divergence between the original FP32 distribution and the quantized approximation over candidate clipping thresholds, which is the approach used by TensorRT's calibration engine. For weights, which have more predictable distributions, symmetric per-channel quantization with min-max calibration typically suffices.

Post-training quantization is fast but incurs accuracy losses that are model-dependent and sometimes unacceptable for precision-sensitive tasks like depth estimation or small-object detection. Quantization-aware training (QAT) recovers this accuracy by simulating quantization error during the forward pass of training so that the learned weights adapt to operate correctly under quantization. The simulation inserts fake quantization nodes that apply the round-then-clamp operation in the forward pass but pass gradients through the clamp as if it were the identity function, the straight-through estimator:

```

class FakeQuantize(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, scale, zero_point, q_min, q_max):
        x_q = torch.clamp(torch.round(x / scale) + zero_point,
                           q_min, q_max)
        x_dq = scale * (x_q - zero_point)
        ctx.save_for_backward(x, torch.tensor(q_min),
                              torch.tensor(q_max))

        return x_dq

    @staticmethod
    def backward(ctx, grad_output):
        x, q_min, q_max = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[x < q_min.item()] = 0.0
        grad_input[x > q_max.item()] = 0.0
        return grad_input, None, None, None, None

```

The QAT training loop wraps the model's existing training procedure with fake-quantize nodes inserted after every weight tensor and after every activation that will be consumed by a subsequent quantized operator. Scale and zero-point parameters are updated at each step using exponential moving averages of the observed min and max values, or are learned as parameters themselves in learned-scale variants. A complete QAT loop fragment for a single epoch looks like:

```

def qat_train_epoch(model, loader, optimizer, fake_quant_cfg):
    model.train()
    model.apply(torch.quantization.enable_fake_quant)
    model.apply(torch.quantization.enable_observer)

    for step, (images, labels) in enumerate(loader):
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    if step % fake_quant_cfg.observer_update_freq == 0:
        model.apply(torch.quantization.disable_observer)

```

The observer is disabled partway through training because allowing it to track activations indefinitely can cause the scale parameters to drift as the weight distribution continues evolving. A common schedule disables observers after 30 to 50 percent of QAT fine-tuning steps and then freezes batch normalization statistics after another 10 percent of steps.

Mixed-precision quantization applies different bit widths to different layers. The motivation is that not all layers are equally sensitive to quantization. The first convolutional layer operates directly on raw pixel values and is typically sensitive; its weight distribution spans a wide dynamic range, and INT8 rounding errors propagate into every subsequent layer. The last few layers before the detection head are similarly sensitive because their outputs are directly decoded into bounding box coordinates or class logits. Intermediate layers with abundant redundancy tolerate INT4 without

measurable accuracy loss. Assigning bit widths manually is impractical for large networks; automated mixed-precision search uses proxy metrics such as the Hessian trace of each layer's loss contribution or the sensitivity of task metrics to per-layer perturbations to rank layers by their quantization sensitivity and assign wider formats to the most sensitive ones.

Pruning

Pruning removes parameters from a trained network on the premise that a large fraction of the weight mass contributes negligibly to the network's output. The structural question is whether to remove individual weights (unstructured pruning) or entire groups such as filters, channels, or attention heads (structured pruning). The answer has direct hardware consequences.

Unstructured pruning produces sparse weight tensors where a fraction p of the values are forced to zero. The potential compute saving is real: a 50 percent sparse matrix-vector product requires only half the multiply-accumulate operations. But realizing that saving requires hardware or software that can exploit irregular sparsity, which most NPU accelerators and microcontroller SIMD units cannot. On a Cortex-M7 executing dense INT8 matrix multiplications via CMSIS-NN, an unstructured sparse weight tensor provides no latency reduction compared to the dense original because the hardware still executes the full dense multiply kernel. Unstructured pruning is therefore primarily useful on platforms with explicit sparse tensor support such as NVIDIA's Ampere and later GPU architectures with 2:4 structured sparsity, which occupies an intermediate position between fully unstructured and channel-level structured.

Structured pruning removes entire filters or channels, yielding a smaller dense network that executes without any sparse arithmetic. Removing the k -th output filter from a convolutional layer requires also removing the k -th input channel from the subsequent layer. The resulting network has a different topology but is architecturally identical to a smaller dense network and runs efficiently on any hardware.

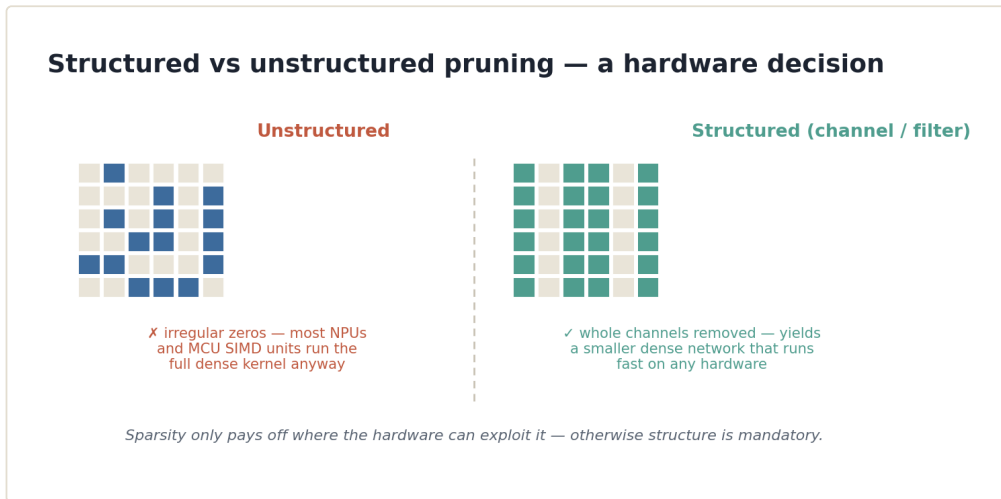


Figure 6.2 — Structured versus unstructured pruning: a hardware decision, not a stylistic one.

Magnitude-based structured pruning scores each filter by the L1 norm of its weight tensor and removes the lowest-scoring filters up to a target sparsity ratio. The scoring and removal for a single layer:

```
def prune_filters_l1(weight, prune_ratio):
    # weight shape: (out_channels, in_channels, kH, kW)
    scores = weight.abs().sum(dim=(1, 2, 3))
    num_prune = int(prune_ratio * weight.shape[0])
    threshold = scores.topk(num_prune, largest=False).values.max()
    keep_mask = scores > threshold
    return keep_mask

def apply_filter_mask(model, layer_name, keep_mask):
    layer = dict(model.named_modules())[layer_name]
    layer.weight.data = layer.weight.data[keep_mask]
    layer.out_channels = keep_mask.sum().item()
    if layer.bias is not None:
        layer.bias.data = layer.bias.data[keep_mask]
```

The `keep_mask` is then propagated to the subsequent layer to trim its input channels correspondingly. This propagation must traverse the full dependency graph of the network; skip connections in ResNet-style architectures and concatenation points in EfficientDet-style feature pyramids require careful bookkeeping to maintain architectural consistency.

Gradient-based pruning criteria replace magnitude with a first-order or second-order approximation of the change in loss caused by removing a filter. The first-order Taylor criterion scores filter k in layer l as:

$$\text{importance}(l, k) = |E[dL/da_{\{l,k\}} * a_{\{l,k\}}]|$$

where $a_{\{l,k\}}$ is the activation produced by filter k , and the expectation is taken over a calibration batch. This product is the element-wise contribution of filter k 's activation to the gradient of the loss, accumulated over spatial positions and batch samples. Filters whose removal produces small first-order loss change are removed first. This

criterion is more accurate than magnitude pruning for heavily regularized networks where small-magnitude weights are not necessarily low-importance, but it requires a calibration forward-backward pass before each pruning step.

Iterative magnitude pruning interleaves training and pruning in cycles: prune a small fraction of filters, fine-tune for a few epochs to allow the remaining filters to recover accuracy, then prune again. This is more accurate than one-shot pruning but requires significant training compute. For drone deployment where the target model is being adapted from a pretrained backbone, a practical schedule is three to five cycles of five percent filter removal followed by ten epochs of fine-tuning, stopping when accuracy on the validation set drops below an acceptable threshold.

Knowledge Distillation

Pruning and quantization compress a model by eliminating or approximating its existing parameters. Knowledge distillation takes a different approach: train a smaller student model to replicate the behavior of a larger teacher model, transferring information that the student's capacity would not recover from the training data alone.

The original distillation loss augments the student's task loss with a soft-label term. The teacher's output logits, divided by a temperature T greater than one, produce a softer probability distribution that encodes the teacher's relative confidence between classes. Matching this soft distribution forces the student to learn inter-class relationships that the hard one-hot labels discard:

$$L_{\text{distill}} = \alpha * L_{\text{task}}(\text{student}, \text{labels}) + (1 - \alpha) * T^2 * \text{KL}(\text{softmax}(z_t/T), \text{softmax}(z_s/T))$$

where z_t and z_s are the teacher and student logits, T is the temperature, and α balances the task and distillation terms. The T squared factor compensates for the smaller gradient magnitudes produced by soft targets at high temperature.

For detection models on drones, output-only distillation on the classification logits is less effective than feature-map distillation because the detection head's output space is structured differently between teacher and student architectures, and the logit distributions are dominated by background predictions that contribute little signal. Feature-map distillation instead minimizes the distance between intermediate representations:

$$L_{\text{feature}} = \text{sum over layers } l \text{ of } ||f_s^l(x) - \text{phi}(f_t^l(x))||^2_F$$

where f_s^l and f_t^l are the student and teacher feature maps at paired layers, phi is a lightweight adapter (typically a 1×1 convolution) that projects the teacher's feature dimension to match the student's, and the Frobenius norm measures their difference. The adapter is trained jointly with the student and discarded after distillation is complete.

Attention-transfer distillation is a specific variant relevant for transformer-based architectures at Class III and IV platforms. The teacher's multi-head attention maps are summed across heads to produce spatial attention maps, and the student is trained to produce similar spatial attention distributions:

$$L_{\text{attn}} = ||A_s^l / ||A_s^l||_F - A_t^l / ||A_t^l||_F||^2_F$$

where A^l is the spatially-summed attention map at layer l . This enforces that the student attends to similar regions as the teacher without requiring identical feature magnitudes, which are harder to match across architectures of different widths.

A practical distillation training loop for a detection student combines all three terms: task loss on labeled detections, feature-map distillation at the neck and head feature pyramid levels, and class-logit distillation at the detection head output. The alpha and temperature hyperparameters for the logit term and the per-layer weights for the feature-map term are tuned on a held-out validation split. For Class II platforms where the student is a YOLOv8-nano being distilled from a YOLOv8-medium teacher, feature-map distillation at the P3, P4, and P5 feature pyramid levels consistently recovers two to four percentage points of mAP lost to the student's reduced channel width.

Operator Fusion

Quantization, pruning, and distillation operate on the model's parameter and activation content. Operator fusion operates on its execution graph. The insight is that the overhead of writing activations to memory and reading them back between consecutive operations can exceed the arithmetic cost of the operations themselves on memory-bandwidth-limited hardware.

The most common fusion pattern is Conv-BN-ReLU folding. During inference, batch normalization applies a channel-wise affine transform to the convolution output:

$$y = \text{gamma} * (Wx + b - \mu) / \text{sqrt}(\text{sigma}^2 + \text{epsilon}) + \text{beta}$$

This can be absorbed into the convolution's weight and bias before deployment:

$$\begin{aligned} W_{\text{fused}} &= \text{gamma} * W / \text{sqrt}(\text{sigma}^2 + \text{epsilon}) \\ b_{\text{fused}} &= \text{gamma} * (b - \mu) / \text{sqrt}(\text{sigma}^2 + \text{epsilon}) + \text{beta} \end{aligned}$$

After folding, the batch normalization layer is eliminated entirely, and the convolution's output passes directly to the activation function. The fused convolution requires one write to memory rather than three, which on a Cortex-M7 with a 2 MB L2 cache reduces the effective memory traffic of that layer by a factor that depends on whether the activation tensor fits in cache. For large activation tensors that spill to DRAM, the traffic reduction is proportional and directly reduces inference latency.

Conv-BN-ReLU fusion is performed automatically by most export-path toolchains (TFLite converter, TensorRT, ONNX Runtime) and requires no manual intervention. The

operator fusions that do require attention are the ones that interact with quantization boundaries. A quantized convolution followed by a quantized activation must fuse the requantization step, which rescales the INT32 accumulator output of the convolution to the INT8 range of the activation, into the same kernel execution to avoid writing INT32 tensors to memory and reading them back for requantization. If the export toolchain does not recognize the quantization scheme used during QAT, it may insert explicit requantization nodes that break the expected fusion and add memory round-trips.

The depthwise-separable convolution block, used in MobileNet variants and throughout the efficient detection backbones discussed in the next chapter, has its own fusion considerations. The block consists of a depthwise convolution (a convolution where each input channel is filtered independently), a pointwise convolution (a 1x1 convolution that mixes channels), and typically a batch normalization and activation after each. The depthwise and pointwise convolutions are separate operations but their intermediate activation is small enough that fusing them into a single kernel, where the depthwise output is kept in register and consumed immediately by the pointwise operation, eliminates one full tensor write-read cycle. This fusion is not universally implemented in all runtimes but is critical for Class I and Class II platforms where the memory system bottleneck is severe.

A written form of the depthwise-separable block with explicit fusion annotation:

```
class FusedDWSeparable(nn.Module):
    def __init__(self, in_ch, out_ch, stride=1):
        super().__init__()
        self.dw = nn.Conv2d(in_ch, in_ch, 3, stride=stride,
                           padding=1, groups=in_ch, bias=False)
        self.dw_bn = nn.BatchNorm2d(in_ch)
        self.pw = nn.Conv2d(in_ch, out_ch, 1, bias=False)
        self.pw_bn = nn.BatchNorm2d(out_ch)
        self.act = nn.ReLU6(inplace=True)

    def forward(self, x):
        # In deployment, dw + dw_bn + act and pw + pw_bn + act
        # are each fused into single kernels by the runtime.
        # The intermediate tensor between dw and pw blocks
        # is never written to DRAM when both fusions are active.
        x = self.act(self.dw_bn(self.dw(x)))
        x = self.act(self.pw_bn(self.pw(x)))
        return x
```

The comment reflects a guarantee that only some runtimes on some hardware actually provide. Verifying that the fusion occurred requires inspecting the runtime's execution plan, not assuming it from the model definition. On TFLite with the XNNPack delegate, both fusions are applied. On CMSIS-NN for Cortex-M, the fusion applies to the Conv-BN-ReLU6 pattern but the cross-layer dw-to-pw fusion is not implemented; the intermediate activation is written to SRAM. For Class I systems this means the intermediate tensor must fit within the SRAM budget, which constrains the spatial resolution at which depthwise-separable blocks can be used.

Composing the Techniques

In practice these four techniques are applied in a specific sequence for a deployment pipeline. The starting point is a pretrained model in FP32. The sequence is:

1. Structured pruning with iterative fine-tuning to reach a target parameter count that fits the memory budget at INT8 precision (the INT8 model occupies roughly one-quarter the memory of FP32, so the pruning target is determined by multiplying the INT8 memory budget by four to find the maximum allowable FP32 parameter count before quantization).
2. Knowledge distillation from a teacher, either simultaneously with the pruning fine-tuning or as a separate phase after pruning stabilizes, to recover accuracy lost to the structural reduction.
3. QAT fine-tuning for a small number of epochs (typically five to fifteen percent of the original training schedule) to adapt the pruned, distilled weights to quantization error.
4. Export with operator fusion applied by the target runtime or manually for fusions the runtime does not recognize.

The ordering is not arbitrary. Pruning before QAT ensures that the quantization observer statistics are calibrated on the final pruned weight distribution rather than the original unpruned distribution. Distillation during or after pruning but before QAT ensures that the soft-label supervision is applied to the full-precision model where gradients are most informative. Operator fusion is always the last step because it transforms the graph into a form that may not be differentiable or easily modified.

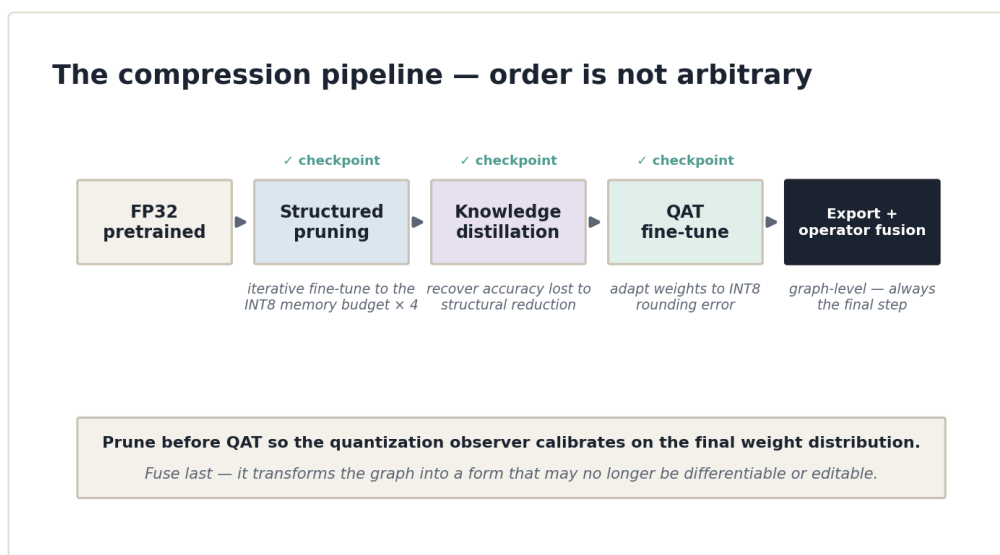


Figure 6.3 — The compression pipeline: prune, distill, quantize, fuse — applied in that specific order.

Each step introduces a validation checkpoint. The pruned model's accuracy drop before distillation recovery sets the ceiling for what distillation must recoup. The QAT model's

accuracy relative to the pruned FP32 model sets the quantization penalty. If the total accuracy budget, defined by the mission's minimum acceptable detection performance, is exceeded at any checkpoint, the choices at the preceding step must be revisited before proceeding.

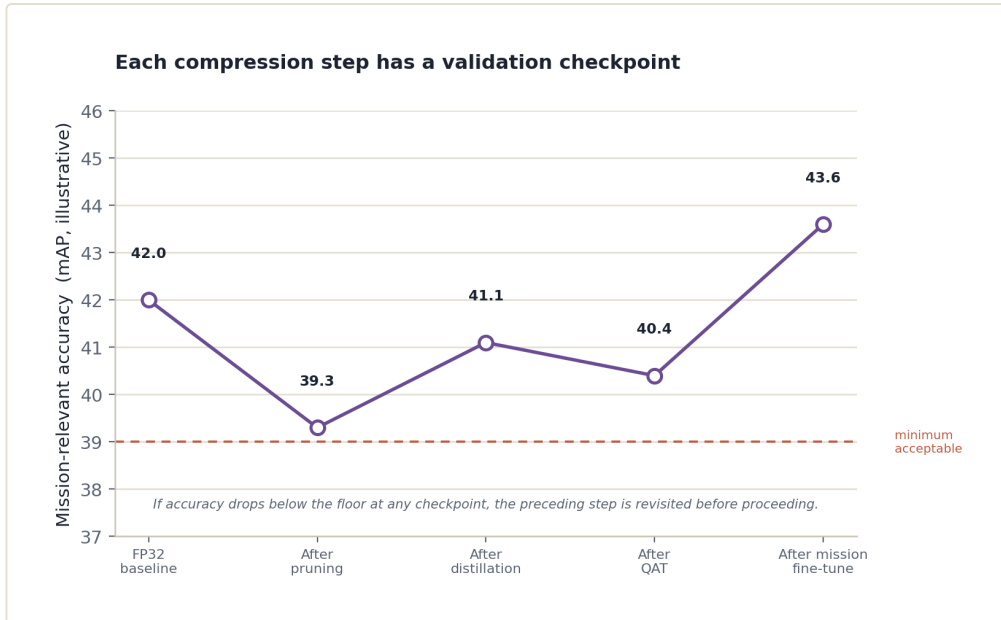


Figure 11.2 — Each compression step carries a validation checkpoint against the mission accuracy floor.

On drone platforms where there is no ground truth available at inference time, this offline validation step is the only opportunity to verify that the deployed model meets its specification.

The following chapters apply these techniques to specific architectures and specific drone classes. The decisions made at the architecture level, which backbone, which detection head, which fusion topology, determine what the compression techniques have to work with. A backbone with few inter-layer skip connections prunes more cleanly than one with dense residual connections, where removing a filter requires coordinating across many dependent paths. A model trained with explicit quantization awareness from the first epoch of fine-tuning reaches its INT8 accuracy target with less QAT compute than one quantized only at export time. The techniques are tools, and like all tools their effectiveness depends on whether the underlying structure is amenable to what the tool does.

Chapter 7: Vision-Based Inference Architectures by Drone Class

Every camera attached to a drone produces the same kind of raw data: a grid of pixel intensities arriving at a fixed frame rate. What differs radically across drone classes is what the inference stack is permitted to do with that data before a decision must be made. On a Class I nano-UAV, the camera's output must be processed by a network small enough to fit in 512 KB of SRAM and execute in under 50 ms on a Cortex-M7 with no floating-point acceleration beyond what the hardware's FPU provides. On a Class IV platform, the same conceptually simple task of detecting and localizing objects can be assigned to a vision transformer with hundreds of millions of parameters, compressed and compiled by TensorRT into a CUDA kernel sequence that saturates the Orin's tensor cores at 40 TOPS. The architectural decisions that follow from these constraints are not stylistic preferences. They are the output of a calculation: how many multiply-accumulate operations can be completed in the available time budget, given the available silicon, given the available watts.

This chapter maps camera-based inference architectures to the drone taxonomy established in Chapter 2, working up from Class I to Class IV. For each class the argument has the same structure: what the hardware can afford, which architectural choices respect that budget, and why the specific operations that define each architecture, depthwise separable convolutions, anchor-free heads, linear attention approximations, are the right tools for that operating point.

The Class I constraint is close to absolute. A nano-UAV or micro-UAV in the under-100g category carries a processor in the Cortex-M7 or RISC-V range, sometimes with a small neural processing extension but more often without. Memory is measured in hundreds of kilobytes. The camera, typically a low-resolution global-shutter module at 96x96 or 160x120 pixels, delivers frames at 30 fps. The inference budget per frame is therefore around 33 ms, of which perhaps 20 to 25 ms can be assigned to the network before the flight control loop is starved.

The architecture family that fits this envelope is shallow convolutional networks with aggressive channel reduction, often derived from MobileNetV3-Small. MobileNetV3 is built around the inverted residual block originally introduced in MobileNetV2, extended with hard-swish activations and squeeze-and-excitation modules tuned for mobile inference. The inverted residual structure expands channels in a low-dimensional space, applies a depthwise convolution to mix spatial information within each channel independently, then projects back down with a pointwise convolution. The key operation is the depthwise separable convolution, which replaces a standard k -by- k convolution over C input channels producing C -prime output channels with two operations in sequence: a depthwise convolution applying one k -by- k filter per input channel, followed by a pointwise 1×1 convolution that mixes channels.

The MAC count comparison makes the advantage concrete. A standard 3x3 convolution on a layer with 32 input channels producing 64 output channels over a 28x28 spatial map requires approximately 28 times 28 times 64 times 32 times 9 MACs, which is roughly 14.5 million operations. The depthwise separable factorization reduces this to the depthwise cost of 28 times 28 times 32 times 9 plus the pointwise cost of 28 times 28 times 32 times 64, totaling approximately 2.3 million operations, a factor of roughly 6 reduction. For a full network with many such layers, the cumulative reduction determines whether the model runs within the latency budget or not.

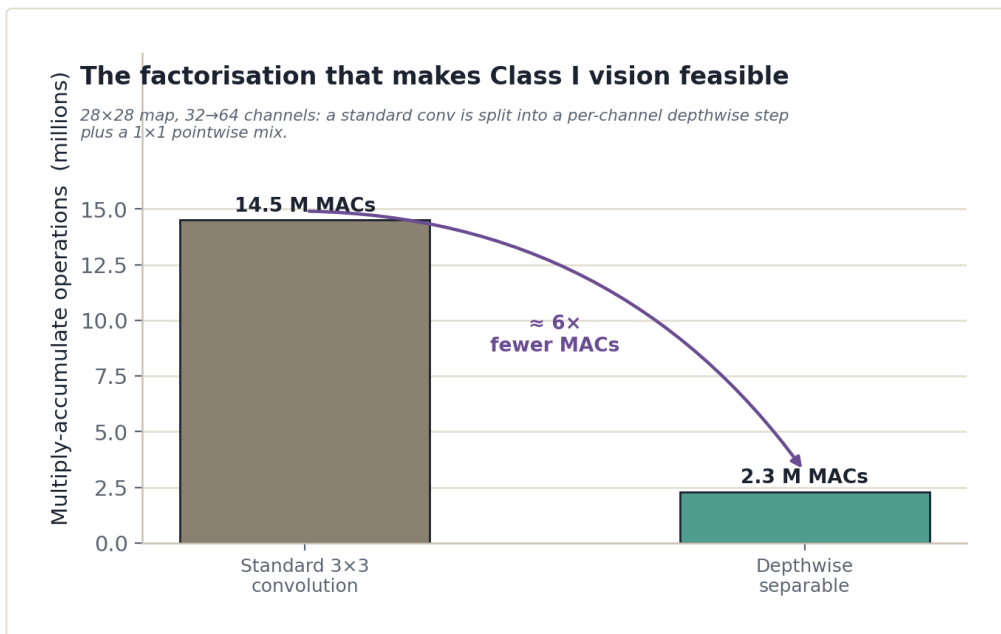


Figure 7.1 — The depthwise-separable factorization cuts multiply-accumulate count roughly sixfold.

A depthwise separable block in PyTorch looks like the following:

```

import torch
import torch.nn as nn

class DepthwiseSeparableConv(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.depthwise = nn.Conv2d(
            in_channels, in_channels,
            kernel_size=3, stride=stride,
            padding=1, groups=in_channels, bias=False
        )
        self.pointwise = nn.Conv2d(
            in_channels, out_channels,
            kernel_size=1, bias=False
        )
        self.bn_dw = nn.BatchNorm2d(in_channels)
        self.bn_pw = nn.BatchNorm2d(out_channels)
        self.act = nn.Hardswish()

    def forward(self, x):
        x = self.act(self.bn_dw(self.depthwise(x)))
        x = self.act(self.bn_pw(self.pointwise(x)))
        return x

```

The `groups=in_channels` argument is the mechanical realization of the depthwise assumption: each output channel in the depthwise step sees exactly one input channel. This is not a heuristic. It is the algebraic commitment that reduces the MAC count by the factor derived above, and it is what allows the block to run without exceeding the Class I cache budget.

For Class I detection, the network head is almost always a single-scale output with a small number of anchor priors or, increasingly, an anchor-free formulation where each spatial cell predicts an objectness score, a class distribution, and a bounding-box offset directly. Multi-scale detection heads, which require maintaining feature pyramids across several resolution levels, are not viable at Class I because they multiply the memory footprint by the number of scales. The practical resolution is to choose an input size and a receptive field that makes single-scale detection sufficient for the objects the mission requires detecting. For obstacle avoidance at 2 to 5 meter range on a nano-UAV, objects subtend large angles and a single-scale head at a coarse feature map is usually adequate.

EfficientDet-Lite0 represents the upper boundary of what some Class I platforms with hardware acceleration extensions can handle. It applies a compound scaling coefficient to jointly scale network depth, width, and resolution, and its BiFPN neck provides multi-scale feature fusion with weighted connections learned during training. On a Cortex-M55 with Helium vector extensions, a quantized EfficientDet-Lite0 at 96x96 input can achieve detection at around 100 ms per frame, which places it at the edge of the Class I timing budget. Without Helium or an equivalent SIMD extension, it is too slow.

Class II expands the operating envelope significantly. A 100g to 2kg multirotor carries a mobile SoC with a dedicated NPU, such as the Ambarella CV28 or an NXP i.MX 9 series part, or alternatively a Cortex-A55 cluster with NEON SIMD. Power budgets in the 2 to 5W range for the compute subsystem allow networks an order of magnitude larger than Class I. Camera resolution rises to 720p or 1080p, though the inference input is typically downsampled to 320x320 or 416x416. Frame rates of 30 fps are standard, giving a 33 ms per-frame budget.

The dominant architecture family at Class II is the single-stage object detector in the YOLO lineage, specifically YOLOv8-nano and similarly scaled variants. YOLOv8 introduces an anchor-free decoupled head, meaning the classification and regression branches are separated rather than sharing a convolutional stack. This matters at the edge because the decoupled head is cleaner to quantize: the two branches can have their quantization ranges calibrated independently, avoiding the range mismatch that occurs when a shared head must simultaneously represent classification logits, which tend to be small and near zero, and box offsets, which can span the full image dimension.

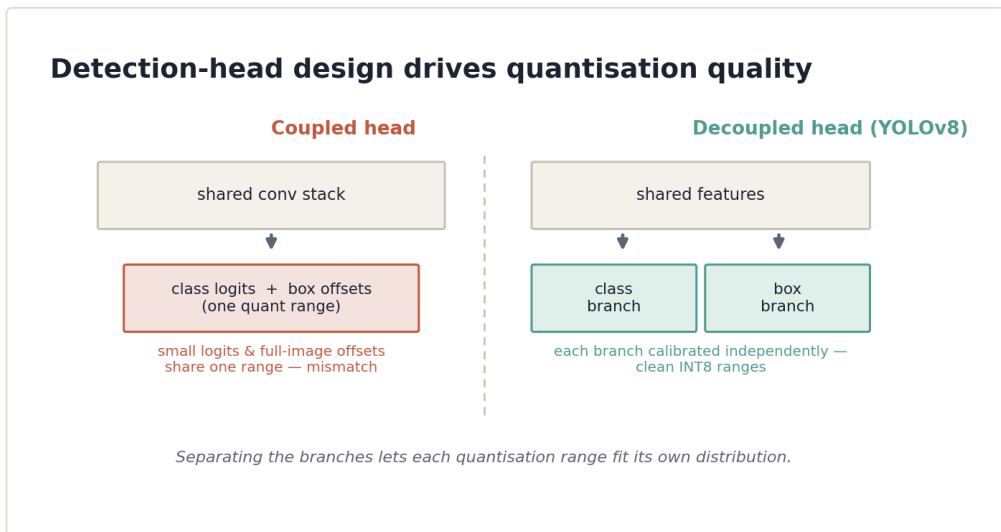


Figure 7.3 — Decoupling the detection head lets each branch's quantization range fit its own distribution.

The YOLOv8 backbone uses a C2f module in place of the CSP bottleneck of earlier versions. The C2f module splits the input tensor along the channel dimension, passes one split through a sequence of bottleneck residual blocks, concatenates all intermediate outputs with the other split, and projects to the output dimension. This design increases gradient flow during training, which matters for small models that are otherwise prone to vanishing gradients, while keeping the inference-time structure simple enough to map cleanly onto SIMD or NPU tiles.

At INT8 precision on a mobile NPU with 4 TOPS, YOLOv8-nano at 320x320 completes in 8 to 12 ms depending on the specific NPU architecture and memory bandwidth. This leaves headroom within the 33 ms budget for preprocessing and postprocessing,

including non-maximum suppression. NMS deserves brief attention here because it is disproportionately expensive relative to the network forward pass on embedded hardware. A naive $O(n^2)$ NMS implementation over a large number of candidate boxes can take several milliseconds on a CPU even when the network itself runs on an NPU. The practical fix is either to aggressively threshold objectness scores before NMS to reduce the candidate count, or to use a vectorized top-k selection followed by IoU filtering implemented with NEON intrinsics.

Exporting a YOLOv8-nano model to INT8 TFLite for deployment on a Class II platform with a TFLite-compatible NPU involves the following steps:

```
import tensorflow as tf
import numpy as np

def representative_dataset(num_samples=200):
    for _ in range(num_samples):
        sample = np.random.rand(1, 320, 320, 3).astype(np.float32)
        yield [sample]

converter = tf.lite.TFLiteConverter.from_saved_model("yolov8n_saved_model")
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.float32

tflite_model = converter.convert()
with open("yolov8n_int8.tflite", "wb") as f:
    f.write(tflite_model)
```

The representative dataset is not optional decoration. The TFLite INT8 quantization scheme is a post-training calibration process: the converter runs the representative samples through the model to collect activation range statistics, then uses those statistics to determine the scale and zero-point for each tensor. If the calibration data does not cover the tails of the activation distribution, the quantized model will clip values that appear at inference time and accuracy will degrade more than the typical 0.5 to 1.5 mAP penalty expected from clean INT8 quantization. For drone deployment specifically, the calibration set should include frames representative of the actual operational environment, not just generic object detection datasets.

Class III covers industrial multirotors and fixed-wing platforms in the 2 to 25kg range, carrying GPU-class SoCs such as the Jetson Orin Nano or the Qualcomm RB5. Power budgets for the compute subsystem reach 10 to 15W. Camera payloads move to 1080p or higher, stereo configurations become common, and the inference input may be 640x640 or even 1280x1280 for tasks requiring fine-grained detection at range. The latency budget remains the same 30 to 50 ms for video-rate inference, but the compute available to fill that budget is two to three orders of magnitude greater than Class I.

At Class III, the architectural choice shifts toward transformer-based or hybrid convolutional-transformer models. The motivation is not novelty. Transformers with

multi-head self-attention have a fundamentally different inductive bias than CNNs: they can model long-range spatial dependencies without the receptive field limitations of stacked local convolutions. For drone applications requiring detection of small objects at distance, where context from across the full image frame is informative, this matters. A pure CNN backbone at Class II cannot easily associate a small object in the bottom-left of the frame with contextual structure in the top-right; a transformer operating on patch embeddings sees the entire spatial context at every layer.

The practical candidates at Class III are MobileViT and EfficientFormer, both designed explicitly to avoid the quadratic attention cost that makes full vision transformers intractable at real-time rates on embedded hardware.

MobileViT replaces the local processing in select inverted residual blocks with a global self-attention operation by unfolding the feature map into non-overlapping patches, flattening spatial positions within each patch into the sequence dimension, and applying standard multi-head attention. This hybrid design preserves the efficiency of depthwise separable convolutions for early-stage feature extraction while adding global attention capacity at later, smaller spatial resolutions where the sequence length is manageable. On a Jetson Orin Nano running at 10W TDP, a MobileViT-Small backbone attached to a single-stage detection head runs at roughly 15 to 20 ms per frame at 640x640 input, leaving margin for sensor preprocessing in the 30 ms budget.

EfficientFormer takes a different approach by constructing a pure transformer architecture whose specific attention module, called MetaFormer in the design literature, avoids the full pairwise attention matrix when spatial dimensions are large. At early stages it uses pooling-based token mixing rather than self-attention, switching to full multi-head attention only at the final, spatially smallest stage where the sequence length is small enough that the quadratic cost is bounded. The result is a model whose compute profile resembles a hybrid but whose module structure is more uniform, which simplifies quantization and kernel fusion.

Both architectures benefit substantially from TensorRT compilation on Jetson hardware. TensorRT's graph optimizer identifies sequences of operations that can be fused into single CUDA kernels: for example, a layer normalization followed by a linear projection followed by a GELU activation becomes a single kernel with one read and one write of the activation tensor rather than three separate passes. For transformer models where attention blocks repeat many times, this fusion dramatically reduces memory bandwidth consumption, which is often the binding constraint rather than raw FLOP capacity.

For Class III detection tasks requiring high recall on small or fast-moving objects, the detection head architecture matters as much as the backbone. DETR-style transformer heads, which reformulate detection as a set prediction problem and eliminate NMS entirely, are attractive because they remove the NMS latency spike discussed in the Class II section. However, the original DETR convergence is slow and its encoder-

decoder cross-attention is expensive. Deformable DETR, which restricts each query's attention to a small number of learned reference points rather than the full feature map, reduces the cross-attention cost by roughly a factor of ten and is the practical variant for Class III deployment.

Class IV represents large autonomous platforms above 25kg, including cargo drones, long-endurance fixed-wing systems, and multi-rotor platforms carrying substantial payloads. Compute configurations at Class IV may include a Jetson AGX Orin, a multi-SoC arrangement combining an Orin with a separate radar or sensor processing FPGA, or custom accelerator configurations in advanced programs. The power envelope for the compute subsystem may reach 30 to 60W or higher. At this class, the vision inference problem becomes less about fitting within a budget and more about using the available hardware effectively.

Full vision transformers, specifically ViT-Base or ViT-Large with standard multi-head attention applied to all spatial positions, become viable at Class IV when compiled and quantized through TensorRT. A ViT-Base/16 model, which divides a 640x640 input into 1600 non-overlapping 16x16 patches and applies 12 transformer layers with 12 attention heads, requires approximately 35 GMACs at FP32. In TensorRT INT8 with attention kernel fusion, this executes in under 20 ms on an AGX Orin at its 60W mode, which is comfortably within the 30 ms frame budget for 30 fps inference.

For object detection at Class IV, DETR and its variants, including DINO and DAB-DETR, provide state-of-the-art performance in the small-object-at-distance regime that matters for autonomous navigation and surveillance tasks. The key TensorRT optimization for these models beyond standard INT8 quantization is the fusion of the multi-head attention operation into a single FlashAttention-compatible kernel that tiles the attention computation to stay within the on-chip SRAM budget, avoiding round-trips to DRAM for the attention score matrix. On the Orin AGX, this reduces the memory bandwidth consumption of each attention layer by a factor of three to four compared to a naive implementation.

Deploying a TensorRT-optimized transformer model requires building the engine from an ONNX export:

```

import tensorrt as trt
import numpy as np

logger = trt.Logger(trt.Logger.WARNING)
builder = trt.Builder(logger)
network = builder.create_network(
    1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
)
parser = trt.OnnxParser(network, logger)

with open("vit_detector.onnx", "rb") as f:
    parser.parse(f.read())

config = builder.create_builder_config()
config.set_memory_pool_limit(trt.MemoryPoolType.WORKSPACE, 4 * (1 << 30))

config.set_flag(trt.BuilderFlag.INT8)
config.set_flag(trt.BuilderFlag.FP16)

calibrator = EntropyCalibrator(calibration_data, cache_file="vit_calib.cache")
config.int8_calibrator = calibrator

engine = builder.build_serialized_network(network, config)
with open("vit_detector.trt", "wb") as f:
    f.write(engine)

```

The EntropyCalibrator is a custom class that feeds representative input batches through the network in FP32 and collects activation histograms. TensorRT uses those histograms to determine INT8 scale factors by minimizing the KL divergence between the FP32 and INT8 distributions, which is more robust than simple min-max calibration when activation distributions have long tails, as they often do in attention score tensors. The INT8 and FP16 flags together allow TensorRT to make per-layer precision decisions, keeping sensitive layers, typically the attention softmax and final detection head outputs, in FP16 while quantizing the bulk of the computation to INT8.

Looking across all four classes, the architectural evolution is coherent rather than arbitrary. Class I uses depthwise separable convolutions because they are the minimum-viable spatial mixing operation that fits within kilobyte-scale memory. Class II adds the anchor-free decoupled head because the cleaner quantization boundary it provides makes INT8 deployment reliable rather than fragile. Class III introduces global attention through hybrid or approximated transformer designs because local receptive fields are insufficient for the object scales and scene complexity the mission requires. Class IV lifts the approximation constraint and operates full transformers because the silicon budget can sustain them.

What remains constant across all four classes is the centrality of the detection head's design relative to the backbone. A practitioner who optimizes the backbone aggressively but neglects NMS latency, output tensor quantization range, or the coordinate space in which box offsets are predicted will find that the final deployed latency does not match profiling estimates. The backbone's MAC count is visible and easy to compute. The head's implementation details, the way objectness scores are

thresholded, whether box decoding is fused into the graph or executed on the CPU, how many candidate boxes survive to NMS, determine whether the model meets its real-time contract in practice.

The chapters that follow extend this analysis to lidar and radar modalities, where the data structures entering the inference pipeline are fundamentally different from pixel grids, and the architectural choices reflect those differences rather than inheriting from the vision literature.

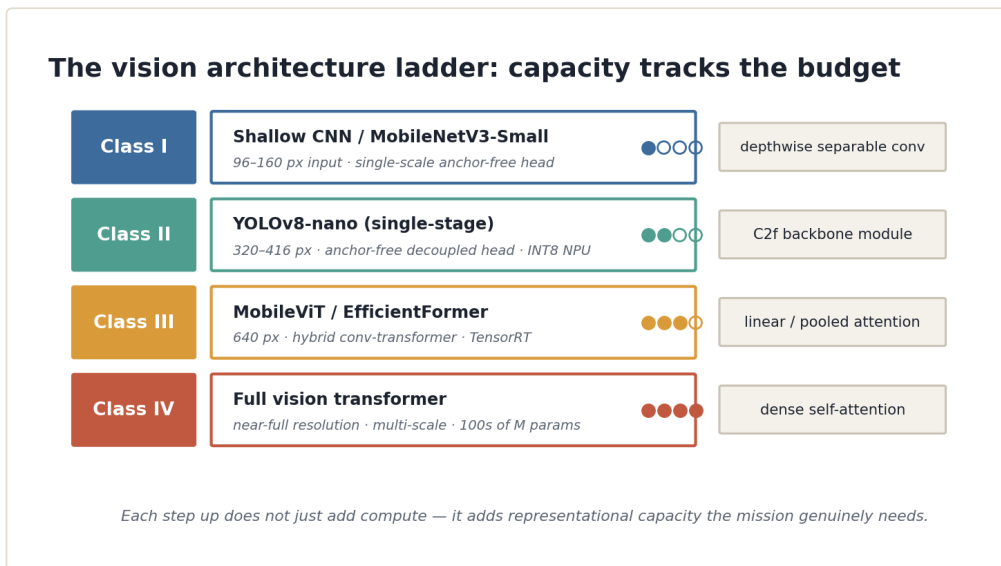


Figure 7.2 — The vision architecture ladder: representational capacity tracks the compute budget of each class.

Chapter 8: Lidar-Based Inference Architectures by Drone Class

Lidar data does not arrive as a pixel grid. It arrives as a set of points, each with a three-dimensional position and, depending on the sensor, a reflectance value and a return timestamp. That structural difference from camera data is not a surface-level formatting concern. It drives every architectural decision downstream, from how the data is stored in memory, to which compute units can process it efficiently, to which representations are feasible on which hardware. A convolutional layer that handles a 640x480 image tensor with predictable memory access patterns becomes impractical when the input is 30,000 unordered points whose spatial distribution changes with every scan. Understanding lidar inference architectures on drones means first understanding what makes point cloud data hard, then understanding the representations that make it tractable.

The core problem is irregularity. A point cloud from a solid-state lidar on a Class II drone might contain anywhere from 5,000 to 50,000 points per frame depending on scene geometry, range, and reflectance. Those points are not arranged on a regular grid. Processing them directly with standard convolution requires either sorting them into a fixed spatial structure, which costs memory and compute, or using operators that work natively on unordered sets, which are expensive and poorly supported by the NPU hardware available in Class II and below. The three dominant representations used to handle this irregularity are raw point clouds with set-based operators, range images that project the point cloud onto a cylindrical surface, and voxel grids or pillar columns that discretize the scene into a regular 3D or pseudo-2D structure. Each representation makes a different tradeoff between information fidelity, memory layout, and compatibility with standard neural network operators, and the right choice is dictated by where on the drone class spectrum the platform sits.

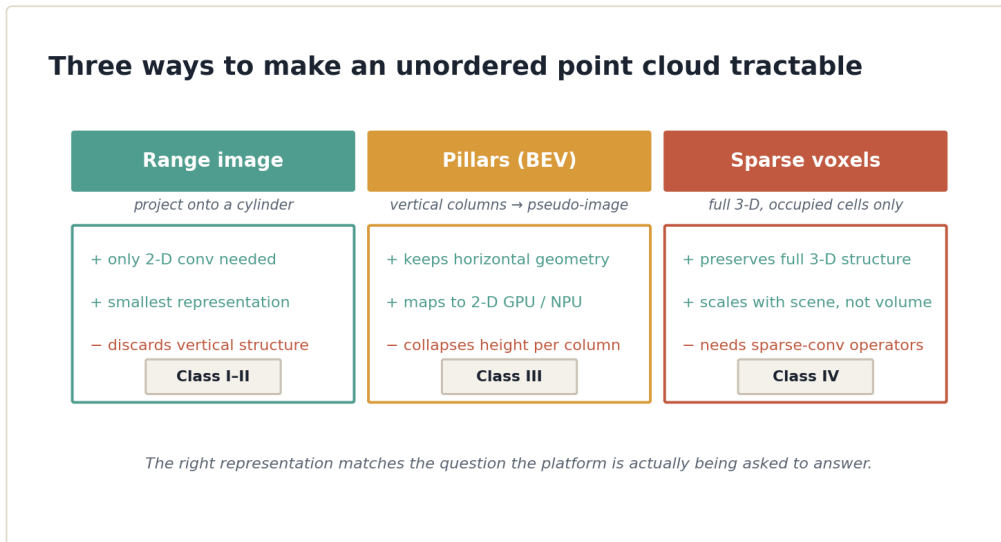


Figure 8.1 — Three ways to make an unordered point cloud tractable, each suited to a different class.

Class I: Avoiding the Point Cloud Entirely

For Class I nano-UAVs running on Cortex-M7 or RISC-V cores with tens of kilobytes of available RAM, the honest answer is that point cloud processing as practiced in the autonomous vehicle literature is not feasible. A full PointNet pass over 10,000 points requires shared MLP layers applied per-point followed by global max-pooling, and the intermediate per-point feature tensors alone would exhaust the memory budget before the first pooling operation. The lidar payloads on Class I platforms are correspondingly simpler: single-beam time-of-flight sensors, short-range solid-state lidars with limited angular coverage, or small-scale MEMS lidars producing on the order of a few hundred range measurements per scan rather than tens of thousands of points. The architectural approach on these platforms is to sidestep the point cloud representation entirely and operate on derived scalar or vector quantities instead.

The simplest viable approach converts the sparse lidar returns into a one-dimensional range profile: a fixed-length array indexed by angular bin, where each element contains the nearest measured range in that bin or a sentinel value indicating no return. This collapses the 3D point cloud into a structure that looks like a 1D signal, and it can be processed with a shallow 1D CNN or even a fixed DSP pipeline. For obstacle detection within a horizontal plane, which covers the most critical use case for low-altitude navigation, a 1D CNN with three or four convolutional layers, depthwise separable convolutions to reduce parameter count, and a simple regression or classification head can detect obstacles, free corridors, and approximate distances within a latency budget of a few milliseconds on a Cortex-M7 running at 200 MHz.

The information loss is real. Projecting to a range profile discards vertical structure, treats all returns at a given azimuth as equivalent, and loses any sense of the density or shape of the obstacle. For Class I drones operating at low altitude in cluttered environments, this is an acceptable tradeoff because the primary requirement is

binary: is there an obstacle within stopping distance in the direction of travel? The answer does not require a precise 3D bounding box. It requires a reliable proximity estimate with low latency and bounded memory, and the range profile representation delivers that at a compute cost that fits the hardware.

For Class I platforms with slightly more capable processors and small solid-state lidars that produce a genuine 2D scan pattern in a fixed plane, a lightweight PointNet variant becomes marginally feasible if the point count is constrained. A PointNet-Tiny configuration with a two-layer shared MLP of width 32, applied per-point to produce a 32-dimensional feature, followed by global max-pooling to yield a 32-dimensional scene descriptor, and a two-layer classification head, can run on a few hundred points with a memory footprint under 64 KB if weights are quantized to INT8. This is not the PointNet that processes full 3D scenes from autonomous vehicle lidars. It is a stripped variant that trades descriptive power for survival within the memory hierarchy. The shared MLP weights are reused across all points, which means the per-layer weight memory is fixed regardless of point count, and the only memory that scales with input size is the per-point feature buffer, which can be computed sequentially if the hardware cannot hold all per-point features simultaneously.

Class II: Range-Image CNNs and Micro-Pillar Architectures

Class II platforms running mobile NPU SoCs have enough memory and compute to process richer lidar representations, but the NPU acceleration available on these chips is optimized for regular 2D convolution, not for the dynamic indexing and scatter-gather operations that native point cloud processing requires. The architectural consequence is a strong bias toward representations that convert the lidar scan into a tensor with fixed spatial dimensions before any neural processing begins.

The range image representation is the natural fit here. A mechanical or solid-state lidar with defined azimuth and elevation resolution maps cleanly onto a 2D grid where one axis encodes azimuth angle and the other encodes elevation or scan line. Each cell in the grid stores the measured range and optionally the reflectance intensity. The result is a tensor of shape $H \times W \times C$ where H is the number of elevation beams, W is the number of azimuth bins, and C is the number of channels per measurement, typically two or three. For a 16-beam lidar with 512 azimuth bins, this produces a $16 \times 512 \times 3$ tensor, which is small, regular, and directly processable by a 2D CNN.

The CNN architecture applied to range images on Class II hardware follows the same design principles established for vision in Chapter 7. Depthwise separable convolutions reduce MAC count per layer. Early layers operate at full spatial resolution to preserve the fine angular structure of the scan, then stride or pooling operations reduce spatial dimensions before the classification or regression head. A range-image CNN for obstacle detection or simple object classification on a Class II platform might use five or six depthwise-separable convolutional blocks, INT8 quantized, with a final head producing either a binary obstacle mask or a small set of object class predictions.

Inference time on a mid-tier mobile NPU for this architecture runs under 5 ms for a 16x512 input, well within the 10 Hz lidar frame rate.

The limitation of range images is that the projection introduces distortions at close range and near-horizontal surfaces, and occlusion patterns in the range image do not correspond straightforwardly to occlusion in 3D space. For navigation applications where the primary goal is detecting obstacles and free space, these distortions are tolerable. For applications requiring precise 3D bounding boxes, they become a problem.

A partial alternative for Class II platforms with sufficient memory is a miniaturized version of the pillar-based approach. The full PointPillars architecture is designed for Class III, but the core idea of discretizing the scene into vertical columns and encoding each column with a lightweight set operator can be scaled down. A micro-pillar approach uses a coarser horizontal grid, say 128x128 cells at 0.2 meter resolution covering a 25.6 by 25.6 meter area around the drone, limits each pillar to a maximum of four points, and encodes each pillar with a two-layer shared MLP of width 16 before max-pooling into a 16-channel pseudo-image. The resulting 128x128x16 tensor is processed by a lightweight 2D CNN. Memory usage for the point encoding phase is bounded and predictable, the pseudo-image processing phase uses only regular 2D convolutions that map cleanly to the NPU, and the total pipeline fits within the memory and latency constraints of a Class II platform.

Class III: PointPillars at Operational Scale

The PointPillars architecture, introduced for autonomous vehicle perception, finds its natural home in the Class III drone context. Class III industrial multirotor or fixed-wing platforms carry solid-state or low-beam-count mechanical lidars producing 30,000 to 100,000 points per frame, run on NVIDIA Jetson Orin Nano or similar GPU-class SoCs, and require 3D object detection or detailed obstacle mapping for missions like infrastructure inspection, precision agriculture, or beyond-visual-line-of-sight flight. The memory and compute headroom available on these platforms makes the full PointPillars pipeline viable, and the architecture's design aligns well with the GPU execution model.

The pillarization step is the architectural key. Rather than voxelizing the scene into a 3D grid, PointPillars discretizes only the horizontal plane, creating vertical columns, pillars, that extend the full height of the scene. Each lidar point is assigned to the pillar cell corresponding to its x and y coordinates. Within each pillar, up to N points are retained, with empty pillars discarded. Each retained point is augmented with its position relative to the pillar center and its distance from the pillar's mean point, producing a D-dimensional feature vector per point. A shared PointNet-style MLP is applied to all points across all pillars simultaneously, treating the batch of point-in-pillar samples as a regular tensor of shape $P \times N \times D$ where P is the number of non-empty pillars. Global max-pooling along the N dimension reduces each pillar to a single C-

dimensional feature vector. These pillar features are then scattered back to their grid locations using a stored index tensor, producing a pseudo-image of shape HxWxC that is processed by a conventional 2D backbone and detection head.

The following snippet shows the core pillarization and scatter step:

```
import torch
import torch.nn as nn

def pillarize(points, x_range, y_range, grid_size, max_points_per_pillar, max_pillars):
    # points: (N, 4) tensor of [x, y, z, reflectance]
    # Returns pillar_features (P, max_points, D), pillar_indices (P, 2)
    x_min, x_max = x_range
    y_min, y_max = y_range
    nx, ny = grid_size

    dx = (x_max - x_min) / nx
    dy = (y_max - y_min) / ny

    ix = ((points[:, 0] - x_min) / dx).long().clamp(0, nx - 1)
    iy = ((points[:, 1] - y_min) / dy).long().clamp(0, ny - 1)

    pillar_id = iy * nx + ix
    unique_ids, inverse = torch.unique(pillar_id, return_inverse=True)

    num_pillars = min(len(unique_ids), max_pillars)
    D = points.shape[1] + 3 # original features + dx_center, dy_center, dist

    pillar_features = torch.zeros(num_pillars, max_points_per_pillar, D)
    pillar_indices = torch.zeros(num_pillars, 2, dtype=torch.long)
    pillar_counts = torch.zeros(num_pillars, dtype=torch.long)

    for p_idx in range(num_pillars):
        mask = (inverse == p_idx)
        pts = points[mask][:max_points_per_pillar]
        count = pts.shape[0]

        center_x = x_min + (unique_ids[p_idx] % nx + 0.5) * dx
        center_y = y_min + (unique_ids[p_idx] // nx + 0.5) * dy
        dx_c = pts[:, 0] - center_x
        dy_c = pts[:, 1] - center_y
        dist = torch.sqrt(pts[:, 0]**2 + pts[:, 1]**2 + pts[:, 2]**2)

        augmented = torch.cat([pts, dx_c.unsqueeze(1),
                               dy_c.unsqueeze(1), dist.unsqueeze(1)], dim=1)
        pillar_features[p_idx, :count] = augmented

        uid = unique_ids[p_idx].item()
        pillar_indices[p_idx, 0] = uid // nx
        pillar_indices[p_idx, 1] = uid % nx
        pillar_counts[p_idx] = count

    return pillar_features, pillar_indices, num_pillars

def scatter_to bev(pillar_embeddings, pillar_indices, grid_size, C):
    # pillar_embeddings: (P, C), pillar_indices: (P, 2)
    # Returns pseudo-image: (1, C, ny, nx)
    ny, nx = grid_size
    bev = torch.zeros(1, C, ny, nx)
    bev[0, :, pillar_indices[:, 0], pillar_indices[:, 1]] = pillar_embeddings.T
    return bev
```

The pillar encoding MLP then processes the (P, N, D) tensor, applies max-pooling along the N dimension to get (P, C), and passes the result to scatter_to_bev. The pseudo-image output of scatter_to_bev is a standard image tensor that feeds into a 2D CNN backbone identical in structure to what the vision pipeline uses, which is why the PointPillars approach maps efficiently to the same GPU compute and NPU paths that handle camera inference on Class III hardware.

For deployment on Jetson Orin Nano, the pillar encoding MLP is quantized to INT8 using TensorRT. The scatter operation is not compute-intensive but requires indexed memory writes that execute on the GPU's CUDA cores rather than tensor cores. The 2D backbone following the pseudo-image is the primary compute consumer and benefits directly from TensorRT kernel fusion and INT8 calibration. End-to-end latency for a PointPillars model processing 40,000 points on Jetson Orin Nano, with 128x128 pillar grid and a lightweight two-stage 2D backbone, runs in the range of 25 to 40 ms, which is compatible with a 10 Hz lidar frame rate with headroom for the rest of the perception pipeline.

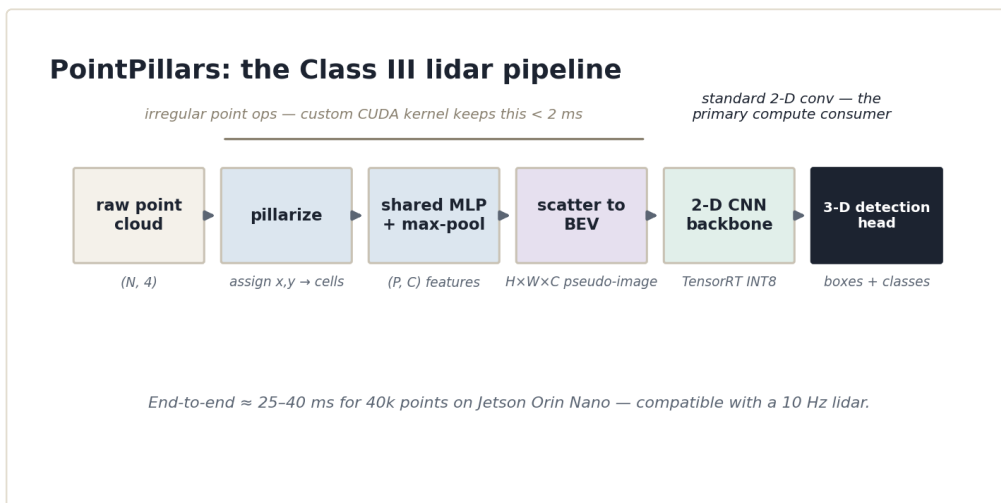


Figure 8.2 — The PointPillars pipeline: pillarization, a shared MLP, scatter-to-BEV, and a 2-D backbone.

One latency consideration that does not appear in profiling the neural network alone is the time spent in the pillarization step itself. The loop over points to assign pillar membership and compute per-point augmented features is sequential in naive Python and GPU-unfriendly due to irregular memory access patterns. In production, the pillarization step is implemented as a custom CUDA kernel that bins points in parallel and uses atomic operations to fill the per-pillar feature buffer. Alternatively, frameworks like OpenPCDet provide pre-built CUDA pillarization kernels that execute in under 2 ms for typical point counts. The network architect's job is not just to design the neural network layers but to ensure that the preprocessing step that feeds them does not consume the latency budget that the network was designed to fit within.

Class IV: Sparse 3D Convolution

Class IV platforms carry high-resolution spinning lidars or multi-sensor lidar arrays producing hundreds of thousands of points per frame. Their multi-SoC compute configurations can sustain workloads that would be impractical on Class III hardware. The appropriate architecture family for this regime is sparse 3D convolution, which processes the scene in full 3D without collapsing vertical structure into a pseudo-image.

The core insight behind sparse convolution is that a lidar scene, even a dense one, is mostly empty space. A 512x512x40 voxel grid at 0.1 meter resolution covering a 51.2x51.2x4 meter volume contains roughly 10 million voxels, but only a small fraction of them, typically less than 5%, contain any lidar returns. A standard dense 3D convolution would compute over all 10 million voxels regardless. Sparse convolution, as implemented in libraries such as `spconv` or `MinkowskiEngine`, computes only at occupied voxel locations, using hash-map-based spatial indexing to track which voxels are active and routing only those voxels through the convolution. The result is that MAC count scales with the number of occupied voxels rather than with the volume of the scene, which makes 3D convolution over large scenes feasible.

A VoxelNet or second-generation architecture like CenterPoint built on sparse convolution follows a fixed structure. The raw point cloud is voxelized into a regular 3D grid. Each occupied voxel retains up to a fixed number of points, which are encoded with a per-voxel PointNet to produce a feature vector. The voxel features are passed through a sequence of sparse 3D convolutional layers that progressively reduce spatial resolution while increasing feature channel depth. At a chosen resolution, the sparse feature map is converted to a dense BEV representation by collapsing the height dimension, and a 2D detection head similar to that used in PointPillars produces the final 3D bounding box predictions.

The memory profile of sparse 3D convolution is fundamentally different from dense convolution. The hash map that tracks active voxels introduces irregular memory access patterns that are bandwidth-bound rather than compute-bound on many SoCs. On NVIDIA AGX Orin or multi-Orin configurations available in Class IV, the tensor cores are underutilized during sparse convolution because the workload does not form large regular matrix multiplications. Batching strategies that accumulate multiple lidar frames before processing can improve hardware utilization, though they introduce latency that must be weighed against the mission requirement. For missions requiring continuous high-rate 3D perception, sparse convolution on AGX Orin achieves 10 to 20 Hz detection rates on full 64-beam lidar point clouds, which is the appropriate operating point for large autonomous platforms.

The other distinction for Class IV is that the lidar pipeline does not stand alone. Chapter 10 covers fusion in detail, but it is worth noting here that sparse 3D convolution architectures produce intermediate feature representations in 3D space that are directly compatible with camera feature volumes for early fusion. A camera

branch that lifts 2D image features into a frustum volume using estimated depth, and a lidar branch that produces a sparse 3D feature volume, can be fused by matching voxel coordinates and concatenating features before the dense detection head. This cross-modal 3D feature fusion is computationally expensive and requires precise extrinsic calibration between sensors, but on Class IV hardware it is feasible and provides measurably better performance on occluded or low-reflectance objects than either modality alone.

Choosing Between Representations

The choice between range images, pillars, and sparse voxels is ultimately a memory layout and operator compatibility decision shaped by the target hardware. Range images require only 2D convolution, map to every available NPU and GPU, and are the smallest representation, but they discard vertical structure and introduce projection distortions. Pillars preserve horizontal spatial relationships and produce a 2D pseudo-image that maps to GPU and NPU 2D convolution, but they also collapse height information into a single feature vector per column. Sparse voxels preserve full 3D structure, scale naturally with scene complexity rather than volume, and enable 3D detection at any orientation, but they require sparse convolution operators that are not supported on most embedded NPUs and are bandwidth-bound in ways that underutilize some hardware configurations.

For Class I the answer is range profile or fixed 1D projection because no other representation fits the hardware. For Class II the answer is range image or micro-pillar because the NPU accelerates 2D convolution and nothing else runs at the required speed. For Class III the answer is full PointPillars because the GPU can handle the pillarization kernel and the pseudo-image backbone at 10 Hz with acceptable latency. For Class IV the answer is sparse voxel or a hybrid pillar-voxel architecture because the application requires full 3D bounding boxes at high accuracy and the hardware can sustain the workload.

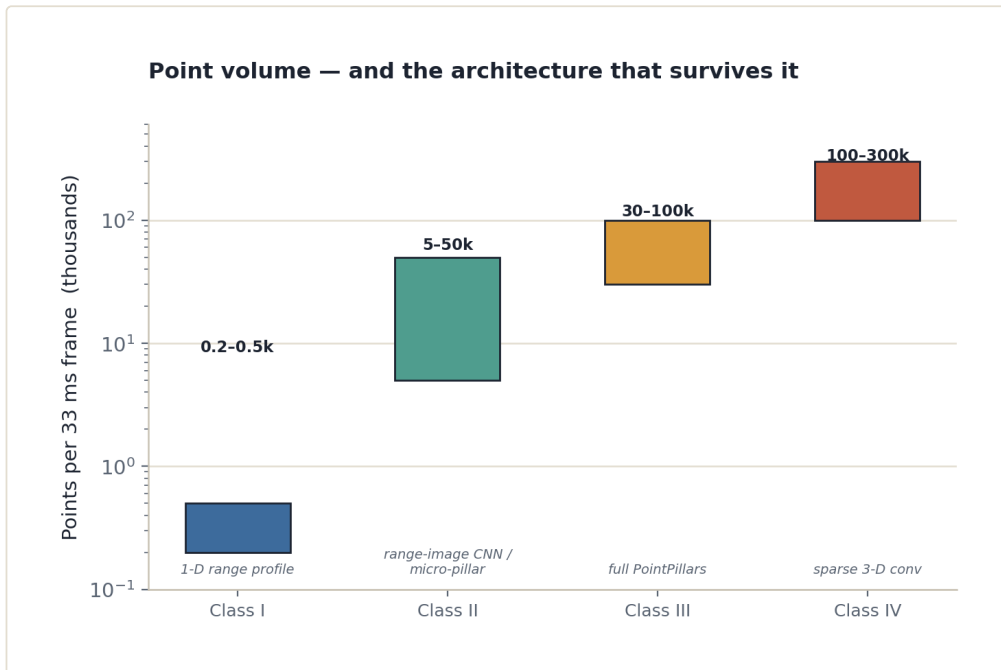


Figure 8.3 — Points per frame by class, and the LiDAR architecture that survives each volume.

What connects the chain from Class I to Class IV is the same logic that connected MobileNetV3 to full ViT in the vision chapter. Each step up the class hierarchy does not just add compute. It adds representational capacity that is genuinely necessary for the missions the platform is expected to carry out. A Class III industrial drone mapping a building facade needs to distinguish a window ledge from a structural beam at varying heights, which a range image CNN that discards vertical information cannot do reliably. A Class I nano-UAV navigating a corridor needs only to know whether the path ahead is clear, which a 1D range profile handles correctly and at a fraction of the cost. The architecture should match the representation to the question the platform is actually being asked to answer.

Chapter 9: Radar-Based Inference Architectures by Drone Class

Radar occupies an uncomfortable position in the drone inference literature. It is neither as photogenic as camera imagery nor as geometrically precise as lidar, and for years it was treated as a backup modality, something to fall back on when fog or darkness made everything else unreliable. That framing understates what radar actually provides. A 77 GHz FMCW radar on a 200g drone gives you radial velocity per detected point, direct from the Doppler shift, without any frame differencing or optical flow estimation. It works in rain, smoke, and complete darkness. Its measurements are inherently calibrated in metric units with no depth ambiguity. The inference architectures built on top of radar data are therefore not degraded versions of camera or lidar pipelines. They answer different questions, and in some cases they are the only sensor that can answer those questions at all.

The challenge is that radar data arrives in a form that is genuinely unfamiliar compared to images or point clouds. Before any inference model sees a tensor, the raw radar signal must pass through a signal processing front-end that transforms ADC samples into something geometrically interpretable. Understanding that front-end is a prerequisite for understanding why the downstream architectures look the way they do.

A frequency-modulated continuous-wave radar transmits a chirp: a sinusoidal signal whose frequency increases linearly over a short interval, typically 50 to 100 microseconds for automotive-derived chips used on drones. The transmitted chirp mixes with the reflected signal to produce a beat frequency proportional to the round-trip delay, which encodes range. A sequence of chirps forms a frame, and the phase progression of the reflected signal across chirps encodes radial velocity through Doppler shift. A 2D FFT over the slow-time axis of the beat signal produces the range-Doppler map: a 2D array where each bin represents a (range, radial velocity) hypothesis, and amplitude in that bin indicates the likelihood that a reflector with that combination exists in front of the sensor. A third FFT, this time across multiple receive antennas, produces angle estimates by exploiting phase differences in the wavefront across the antenna array. After that angular FFT, the output is a 3D cube with dimensions corresponding to range, Doppler velocity, and azimuth angle.

That 3D cube, or appropriate 2D slices of it, is what feeds into neural inference. The signal processing chain up to this point runs entirely in fixed-point arithmetic on whatever DSP or microcontroller hosts the radar front-end, often on the radar chip itself for integrated solutions like the Texas Instruments AWR family. The inference begins at the output of the FFT pipeline.

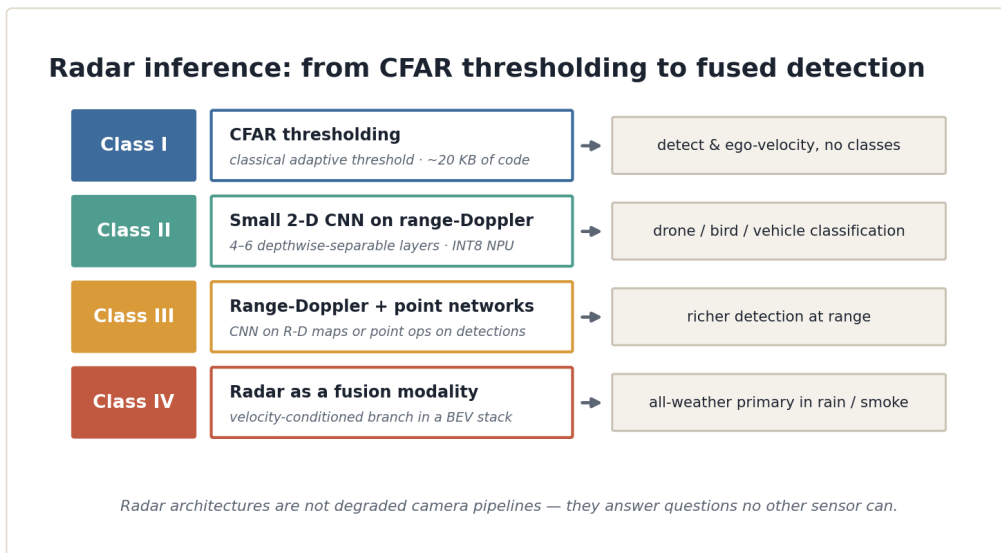


Figure 9.1 — The radar inference ladder, from CFAR thresholding to radar as a fusion modality.

For Class I nano and micro-UAVs, running even a small CNN on range-Doppler data is typically out of reach. The Cortex-M7 or RISC-V cores in this class do not have the multiply-accumulate throughput to process a 64 by 64 range-Doppler map with a neural model at 10 Hz while simultaneously running flight control. The practical architecture is a CFAR-based detection pipeline: constant false alarm rate thresholding applied directly to the range-Doppler map, producing a sparse list of detected peaks. CFAR is not a neural architecture. It is a classical adaptive threshold that compares each cell in the map to the average power in a surrounding window, declaring a detection when the ratio exceeds a threshold calibrated to a target false alarm probability. The implementation requires only integer arithmetic and a sliding window accumulator, both trivially cheap on a Cortex-M7.

The output of the CFAR stage is a short list of (range, velocity) pairs, each representing a candidate target. For drone applications, that list has two immediate uses. The first is obstacle detection: any return within a range threshold and with a relative velocity that implies an approaching object triggers an avoidance response. The second is ego-velocity estimation: a stationary ground return at a known depression angle produces a Doppler shift proportional to the drone's own speed along the radial direction, and averaging multiple stationary returns gives a reliable ground-truth velocity estimate that can correct IMU drift. Both uses require no neural model at all. They run on perhaps 20 kilobytes of code in a tight interrupt-driven loop, and they are remarkably reliable because the physics of FMCW radar is well understood and the signal chain has few moving parts.

The limitation of the CFAR-only approach is that it cannot classify targets. It detects reflectors but cannot distinguish a bird from a drone, a vehicle from a tree trunk, or a human from a static structure. For Class I platforms operating in uncluttered environments with simple mission profiles, that is acceptable. For anything more

complex, classification is necessary, which requires moving up the inference capability chain.

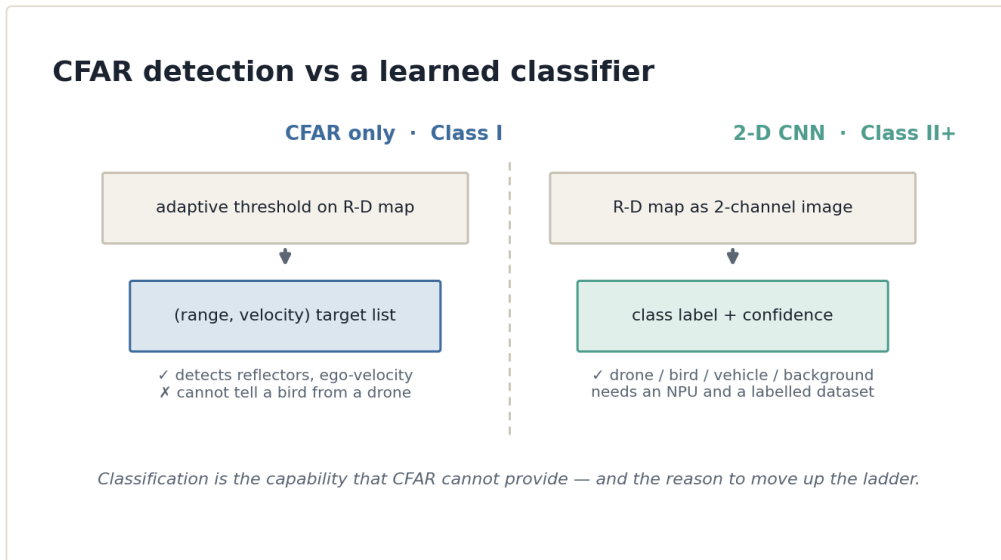


Figure 9.2 — CFAR detection versus a learned classifier: classification is the capability CFAR cannot provide.

Class II compact multirotors carry mobile SoCs with NPUs capable of running small 2D convolutional networks, and this is where range-Doppler map inference becomes genuinely useful. The range-Doppler map after CFAR-based clutter suppression is a 2D tensor, typically between 32 by 32 and 128 by 128 in resolution depending on the radar chip configuration, with real and imaginary components or magnitude and phase stored as separate channels. That tensor is structurally identical to a two-channel image. A small CNN can be applied directly, treating range as one spatial axis and Doppler velocity as the other.

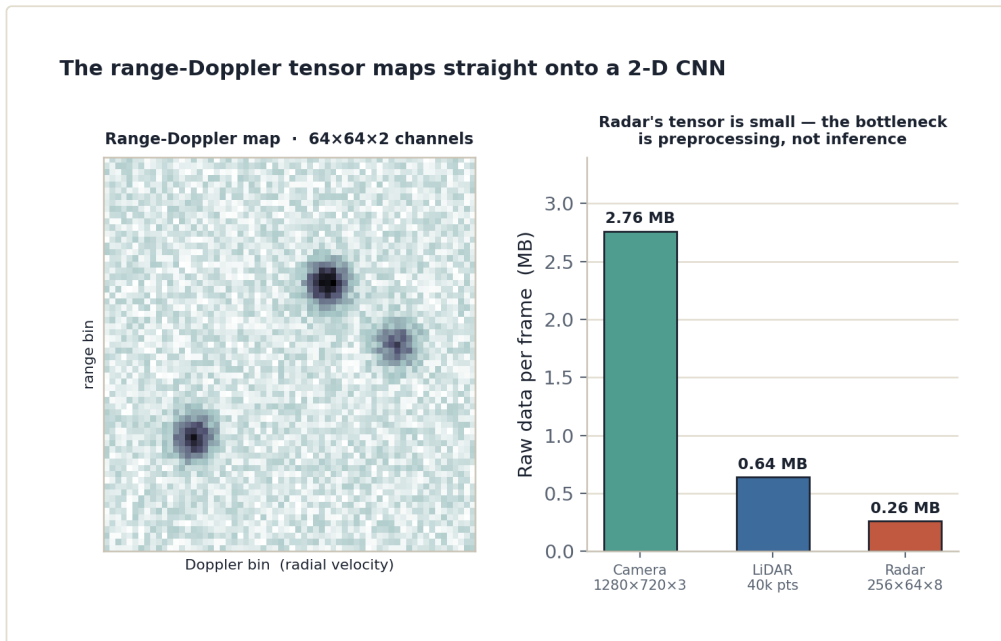


Figure 9.3 — The range-Doppler map is structurally an image; radar's small tensor makes preprocessing the bottleneck.

The network topology appropriate for Class II does not need to be large. A four to six layer depthwise-separable convolutional backbone followed by a global average pool and a classification or detection head runs well within the INT8 quantization constraint of mobile NPUs and fits in under 500 KB of parameters. The detection head can emit a small set of target hypotheses with associated class labels and confidence scores. For a drone detection application, for example, the input is a 64 by 64 magnitude map, the backbone extracts features at two or three spatial scales, and the head produces a score for each of a small number of object classes: drone, bird, vehicle, background.

Training such a network requires radar-specific data. Radar magnitude maps do not look like optical images, and pretrained ImageNet features transfer poorly. The range-Doppler patterns of different targets are governed by their radar cross-section profile and their kinematic signature. A multi-rotor drone produces a distinctive micro-Doppler signature: the rotating blades generate a frequency spread around the main body return, visible as a characteristic blur in the Doppler axis. A bird produces a different pattern, with periodic wing-beat modulation. Capturing sufficient training examples of each class in operational environments is non-trivial, and synthetic data generation using radar simulation tools is often necessary to augment real-world collections.

Quantization for range-Doppler CNN inference on Class II hardware follows the same post-training quantization logic described in Chapter 6. The key difference from vision models is that the input tensor has a dynamic range that varies substantially between frames depending on scene reflectivity, which makes per-channel activation scaling important. Fixed global scale factors calibrated on a small representative dataset tend to clip high-energy returns or lose resolution on weak ones. Per-channel or per-layer

dynamic quantization is preferred when the NPU supports it, and the accuracy penalty versus FP32 baseline is typically below two percent for INT8 with proper calibration.

One detail that matters for radar inference and does not arise in camera pipelines is Doppler ambiguity. FMCW radar with a given pulse repetition interval has a maximum unambiguous velocity: targets moving faster than half the wavelength per pulse repetition interval fold back into the map, appearing at an incorrect velocity. On a drone platform at 77 GHz with a 50 microsecond chirp interval, the maximum unambiguous velocity is around plus or minus 10 meters per second, which covers most relevant drone and bird velocities but not fast aircraft. When the inference model is expected to classify fast targets, either the waveform must be configured for higher velocity range at the cost of range resolution, or the model must be made aware of ambiguity and trained on data that includes wrapped Doppler signatures. Ignoring ambiguity and feeding wrapped signatures to a model trained only on unambiguous data produces systematic misclassification that does not show up in laboratory evaluation but appears immediately in field deployment.

Moving to Class III industrial multirotors and fixed-wing platforms running GPU-class SoCs, the inference budget expands enough to support both richer radar processing and tighter integration with other modalities. Two architectural directions become viable here. The first is recurrent processing of radar point clouds across time. The second is attention-based fusion of radar features with camera or lidar features in a shared representation space.

The radar point cloud in this context is not the same as a lidar point cloud. It is the output of CFAR detection applied to the full 3D range-Doppler-angle cube, yielding a sparse list of points each carrying (range, azimuth, elevation, radial velocity, amplitude). The point density is far lower than lidar: a well-configured 77 GHz automotive radar produces between 50 and 500 detected points per frame, compared to tens of thousands for a mid-grade lidar. Each point carries explicit velocity information, which lidar does not provide. The sparsity makes standard sparse convolution approaches wasteful; the velocity attribute makes point cloud methods that treat all attributes symmetrically suboptimal because velocity is semantically distinct from position.

A practical architecture for Class III radar inference is a lightweight PointNet variant with velocity as an explicit input feature, followed by a recurrent layer that accumulates evidence across frames. The input to the PointNet-style encoder at each frame is an N by 4 tensor of (x, y, z, v_r) tuples, where v_r is the radial velocity of each point after ego-velocity compensation. The encoder applies shared MLP layers across points, producing per-point features that are then aggregated with a max-pool into a global descriptor. That descriptor is fed into a GRU or LSTM to maintain a state vector across frames. The recurrent state captures the trajectory of detected objects, enabling the model to associate current detections with prior observations and to propagate

object class and velocity estimates across frames where the target may be momentarily undetected.

The ego-velocity compensation step deserves specific attention. The drone's own velocity, measured from IMU integration or from the static-return radar technique described for Class I, must be subtracted from each detected point's radial velocity before the point cloud is fed to the inference model. Without compensation, a stationary wall in front of a drone moving at 5 m/s appears with a radial velocity of 5 m/s, indistinguishable from a moving target. Compensation reduces this to zero, and the residual is the target's own motion. This subtraction is a trivial arithmetic operation, but it depends on accurate ego-velocity estimation, making the quality of the IMU or radar-derived ego-velocity estimate a direct input to detection accuracy.

Below is a compact sketch of the recurrent point cloud encoder described above, shown in PyTorch to make the structure concrete:

```
import torch
import torch.nn as nn

class RadarPointEncoder(nn.Module):
    def __init__(self, hidden_dim=128):
        super().__init__()
        self.point_mlp = nn.Sequential(
            nn.Linear(4, 32),
            nn.ReLU(),
            nn.Linear(32, 64),
            nn.ReLU(),
            nn.Linear(64, hidden_dim),
        )
        self.gru = nn.GRUCell(hidden_dim, hidden_dim)
        self.classifier = nn.Linear(hidden_dim, 4)

    def forward(self, points, h_prev):
        # points: (N, 4) tensor of (x, y, z, v_r) per frame
        feat = self.point_mlp(points)          # (N, hidden_dim)
        global_feat = feat.max(dim=0).values  # (hidden_dim,)
        h = self.gru(global_feat.unsqueeze(0), h_prev)
        logits = self.classifier(h)
        return logits, h
```

The GRUCell receives a single frame's global feature and updates a hidden state that persists across calls. In deployment, `h_prev` is initialized to zeros at the start of a tracking session and carried across frames. The classifier head emits class logits for the dominant detected object hypothesis, though in practice the head is more often a regression head emitting bounding box parameters in the radar's spherical coordinate system. The full architecture for a Class III deployment adds a learnable input normalization layer before the point MLP, because raw radar point attributes span very different value ranges: range in tens of meters, velocity in single-digit meters per second, amplitude across several orders of magnitude.

The second direction for Class III, cross-modal fusion with camera or lidar features, is covered in depth in Chapter 10. The radar-specific observation here is that the velocity

dimension of radar points is the unique contribution that makes fusion worthwhile. A fusion architecture that drops the velocity attribute and treats radar as a sparse alternative to lidar has discarded the most informative signal the radar carries. The correct design preserves velocity as a separate input channel through the early fusion stages, either as an additional feature in the point encoder or as a dedicated Doppler feature map projected into the bird's-eye-view space alongside spatial occupancy features.

For Class IV large autonomous platforms with multi-chip compute budgets, radar inference can move to attention-based architectures that process radar point clouds as sequences and attend over both spatial and temporal relationships simultaneously. Transformer-based encoders applied to radar points treat each point as a token with a learned positional embedding derived from its (range, azimuth, elevation) coordinates and an attribute embedding from its (velocity, amplitude) values. Multi-head self-attention across the token sequence captures relationships between simultaneously detected points that the max-pool aggregation in PointNet-style architectures cannot represent. A detected cluster of points with correlated velocities, for example, is a strong indicator of a rigid body in motion, and attention across those points can produce a more confident object hypothesis than processing each point independently.

The computational cost of attention over radar point clouds is manageable at Class IV because the point cloud is small. Unlike image tokens in a vision transformer, which number in the hundreds or thousands for reasonable resolution, radar point clouds at 100 to 500 points per frame produce attention matrices small enough to run efficiently even without sparse attention approximations. A 256-point frame with 8-head attention across 4 transformer layers runs in well under 5 milliseconds on an Orin-class GPU, leaving room for downstream fusion and decision making within a 100 ms control loop budget.

The all-weather value of radar is worth making quantitative rather than leaving as a general claim. Camera performance degrades continuously with precipitation: at 25 mm per hour rainfall, visible-light camera range drops by 20 to 40 percent depending on the detection task, and at heavy fog conditions the degradation reaches 60 to 80 percent at 100 meters. Lidar is somewhat more robust than camera in fog but degrades sharply in heavy rain because water droplets backscatter the near-infrared laser. A 905 nm lidar loses detection range rapidly at rain rates above 10 mm per hour. A 77 GHz radar attenuates at roughly 0.01 dB per kilometer per mm per hour of rainfall, a figure three to four orders of magnitude better than optical sensors. In practical terms, a radar operating at 100 meters in driving rain loses less than 0.001 dB of signal, which is undetectable. For drone missions that must operate through weather, radar is not a redundant modality. It is the primary one.

Velocity measurement is the second unique value, and it is worth being precise here too. Cameras can estimate velocity via optical flow, but optical flow requires texture

and fails on homogeneous surfaces. Lidar can estimate velocity by differencing consecutive scans, but this requires accurate scan-to-scan registration and introduces a minimum velocity threshold below the estimation noise floor. Radar measures radial velocity per detected point directly from the Doppler phase shift, at the same temporal resolution as the detection itself, with no differencing or registration required. The precision is typically within 0.1 m/s at sensor frame rate, which is sufficient for collision prediction, intent estimation, and fusion with IMU-derived ego-velocity. An inference model trained on radar data can therefore make velocity-conditioned predictions, for example, distinguishing a person walking from a person standing, without accumulating the temporal history that camera or lidar would require for the same inference.

The tradeoffs that make radar a constrained modality are angular resolution and point density. At 77 GHz with a compact antenna array fitting on a drone payload, typical azimuth resolution is 1 to 2 degrees with a field of view of plus or minus 60 degrees. At 50 meters, 1.5 degrees of angular resolution corresponds to roughly 1.3 meters of lateral position uncertainty. Lidar at the same range achieves centimeter-level lateral precision. This means radar point clouds carry meaningful range and velocity information but imprecise lateral position, which shapes the downstream inference problem. Classification architectures that rely on spatial arrangement of points, such as shape-based descriptors, are less effective on radar than on lidar. Architectures that rely on velocity and temporal trajectory, such as the recurrent encoder described above, are more effective on radar than on lidar precisely because radar provides what lidar lacks.

The radar-first inference architectures described in this chapter share a common structure: a signal-processing front-end that produces either a 2D tensor or a sparse point cloud, followed by a neural architecture sized to the platform's compute budget, with velocity treated as a first-class input feature throughout. The specific architecture ranges from CFAR-only pipelines on Class I microcontrollers through range-Doppler CNNs on Class II NPUs to recurrent point encoders and transformer-based architectures on Class III and IV GPU-class SoCs. In each case, the architecture is shaped by the same forces that shaped the vision and lidar architectures in earlier chapters: what the sensor physically outputs, what tensor representation that output can be converted into, and what inference operators the target hardware can run efficiently.

What radar adds to the system that neither camera nor lidar can fully replicate is the combination of all-condition operation and direct velocity measurement. Those two properties are not incidental. They define the missions where radar must be part of the inference pipeline rather than an optional addition. Any drone expected to operate through weather, detect non-cooperative targets by motion, or estimate relative velocity without depending on optical flow or scan registration needs radar inference,

and the architecture that handles it needs to treat velocity as its most informative signal rather than an auxiliary attribute.

Chapter 10: Sensor Fusion: Late Fusion, Early Fusion, and Joint Architectures

Every chapter up to this point has treated camera, lidar, and radar as separate inference problems. That treatment was deliberate: the architectures for each modality are genuinely different, and understanding them in isolation prevents the confusion that arises when fusion is introduced before the individual pipelines are clear. But a drone that runs three independent inference stacks and never combines their outputs is not a fused system -- it is three systems sharing a battery. Fusion is where the individual constraints compound and where the architectural decisions become the most consequential.

The central question in sensor fusion for edge inference is not which modalities to combine. The answer to that question is determined by the drone class and mission profile, as established in earlier chapters. The central question is when to combine them: before the neural network sees the data, after each modality has produced its own detection output, or somewhere in between. Each answer implies a different computational structure, a different failure mode, and a different set of constraints on the inference hardware.

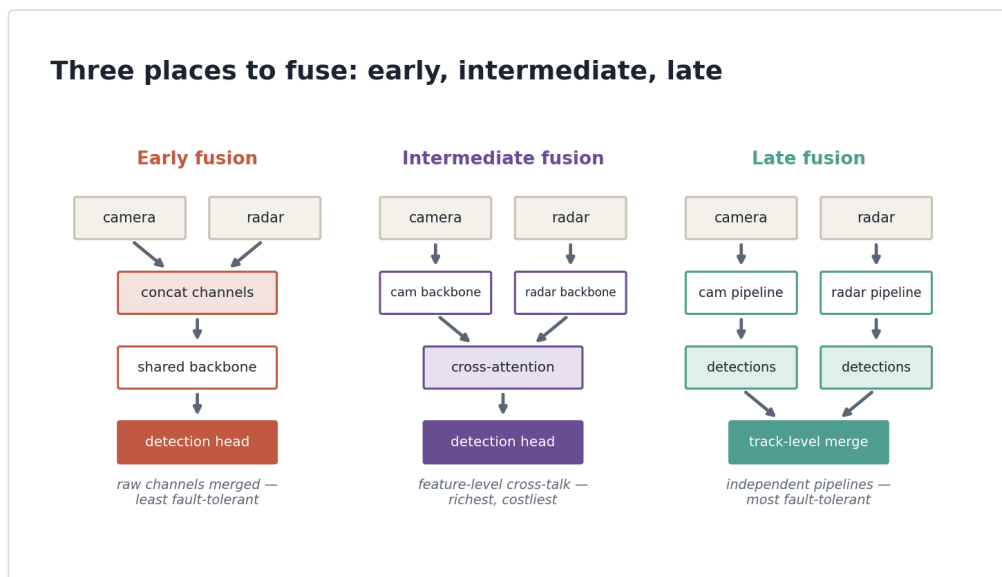


Figure 10.1 — Three places to fuse modalities: early, intermediate, and late fusion topologies.

Late fusion combines modalities after each has produced an output. Early fusion combines raw or lightly-processed sensor data before the neural backbone. Joint architectures -- sometimes called intermediate fusion -- combine feature representations partway through the network. These are not equally practical across drone classes, and they are not interchangeable. The rest of this chapter makes those distinctions precise.

Late Fusion: Combining Outputs After Independent Inference

Late fusion is the most common fusion strategy on constrained platforms, and its dominance is not arbitrary. When each modality runs its own inference pipeline independently, the failure of one pipeline does not corrupt the others. A camera that loses lock in a tunnel still allows the radar pipeline to produce valid velocity estimates. A lidar return obscured by rain still allows the camera to detect objects. Late fusion preserves this independence at the cost of not allowing modalities to inform each other's feature extraction.

The structural requirement for late fusion is a representation that can be compared across modalities. The most common representation is a detection output: a bounding box or position estimate with an associated confidence score. The fusion layer takes a set of such detections from each active modality and produces a single fused detection set.

The simplest late-fusion strategy is score-weighted averaging. If a camera branch and a radar branch both report a detection in approximately the same spatial region, the fused detection takes a weighted combination of their confidence scores, with the weight determined by the expected reliability of each modality under current conditions. This works when the spatial alignment between modalities is known and stable, which on a well-calibrated drone with fixed sensor geometry it approximately is.

A principled version of the same idea is Kalman-filter-based track fusion. Each modality maintains its own set of tracks -- objects being followed across frames -- with state vectors representing position, velocity, and uncertainty. The fusion layer runs a second Kalman filter over the track outputs rather than the raw sensor measurements. This is called track-to-track fusion. For a drone running three modalities, the implementation looks approximately like the following.

Each modality outputs a track list at its own rate: camera at 30 Hz, lidar at 10 Hz, radar at 20 Hz. The fusion filter runs at a fixed rate, say 50 Hz, and ingests whichever track updates have arrived since the last cycle. When a camera track and a radar track share overlapping uncertainty ellipses in the horizontal plane, they are associated via a nearest-neighbor gate or a more principled joint probabilistic data association step. The associated tracks are fused by computing the information-weighted average of their state estimates, which for Gaussian distributions reduces to a closed-form expression involving the inverse of each covariance matrix.

The measurement update for a single track being fused from two modalities with state vectors x_{cam} and x_{rad} and covariances P_{cam} and P_{rad} is:

$$P_{fused} = (P_{cam_inv} + P_{rad_inv})^{-1} \quad x_{fused} = P_{fused} * (P_{cam_inv} * x_{cam} + P_{rad_inv} * x_{rad})$$

This is the standard information filter combination and has constant compute cost regardless of detection count, which makes it attractive for Class I and II platforms where compute is genuinely scarce. The matrices involved are small -- typically 4x4 or 6x6 for a position-velocity state -- so the matrix inversions are not bottlenecks even on a Cortex-M7.

The weakness of track-to-track fusion is that it requires each modality to have already solved the association problem independently. If the camera tracker and the radar tracker have made different association decisions -- assigning the same physical object to different track IDs -- the fusion layer sees two tracks where there is one object. Recovering from this requires a cross-modality re-identification step, which adds latency and complexity. In practice, a generous gating threshold and a track lifetime limit that kills stale tracks before they confuse the fuser is often sufficient on drone-scale environments where the number of simultaneous objects is small.

Learned score aggregation is a third late-fusion approach that trains a small neural network to combine the per-modality detection scores rather than using a fixed rule. The inputs are the detection confidences and, optionally, context features like estimated range, weather flag from an onboard sensor, or time of day. The output is a single fused confidence per detection. The network is small -- typically a two-layer MLP with 32 to 64 hidden units -- and can be quantized to INT8 without meaningful accuracy loss. Its value is that it can learn modality reliability curves from training data rather than requiring a manually tuned weight schedule. The cost is that it requires labeled multi-modal data where the ground truth is available for all modalities simultaneously, which is harder to collect than single-modality data.

Latency in late fusion accumulates from two sources: the slowest single-modality pipeline and the association step. On a Class II platform running camera at 30 fps and lidar at 10 Hz, the fused output rate is limited by the lidar cadence unless the camera is allowed to produce single-modality outputs between lidar updates. A two-rate architecture -- camera-only detection at 30 Hz for fast response, camera-lidar fusion at 10 Hz for high-confidence outputs -- is commonly used in practice and is straightforward to implement because the pipelines are independent.

Early Fusion: Combining Sensor Data Before the Backbone

Early fusion moves the combination point upstream, merging sensor data before the neural backbone processes it. The most common form on drone platforms is treating projected lidar depth as an additional input channel to a camera-based network. Instead of a three-channel RGB tensor, the backbone receives a four-channel RGBD tensor where the D channel contains per-pixel depth values projected from the lidar point cloud onto the camera image plane.

The projection step is a standard perspective transform given the lidar-to-camera extrinsic calibration and the camera intrinsic matrix. For each lidar point with coordinates (x, y, z) in lidar frame, the projected pixel coordinates are:

$$u = f_x * (x_{cam} / z_{cam}) + c_x \quad v = f_y * (y_{cam} / z_{cam}) + c_y$$

where $(x_{cam}, y_{cam}, z_{cam})$ is the point in camera frame after applying the extrinsic rotation and translation, f_x and f_y are focal lengths, and (c_x, c_y) is the principal point. The resulting depth image is sparse: lidar point clouds from solid-state sensors on Class II/III platforms produce densities of a few percent of camera pixels, so most of the D channel is zero or marked as invalid.

Sparse depth requires care at the first convolutional layer. A standard convolution that averages the D channel across a kernel will be dominated by the zero values wherever the lidar projection is empty. Two practical fixes are used: depth completion, which interpolates the sparse lidar depth into a dense depth map before ingestion, and sparse convolution at the first layer, which masks zero-valued depth entries from the convolution accumulation. Depth completion adds a preprocessing step with nontrivial compute cost -- a small hourglass CNN or a guided filter using the RGB image -- but produces a cleaner input and is justified on Class III platforms where the preprocessing can run on a dedicated ISP or on a separate NPU core.

The benefit of RGBD early fusion is that the backbone sees aligned spatial and depth information simultaneously. When the network's first layer learns to detect an object, it can use both texture and depth cues jointly, which is not possible in late fusion where the depth signal appears only as a separate bounding box. For nearby obstacle avoidance, where depth is the primary cue and color is secondary, this alignment produces measurably better detection at small distances than late fusion from separately trained branches.

The cost is that early fusion couples the modalities at the data level. If the lidar fails or produces degraded data -- partial occlusion, return saturation, dew on the window -- the D channel becomes corrupted and the backbone, which has been trained to expect valid depth values, may degrade in unpredictable ways. Dropout augmentation on the D channel during training is the standard mitigation: at training time, the D channel is randomly zeroed with a probability matching the expected failure rate, which forces the backbone to learn features that are useful even when depth is absent.

Radar early fusion is less straightforward than lidar early fusion because radar and camera do not share a natural common projection space. Radar operates in spherical coordinates with coarse angular resolution; the camera operates in perspective projection with fine angular resolution. Projecting radar measurements onto the camera plane produces a very sparse set of pixels with associated Doppler velocity values, which is geometrically valid but informationally different from a depth channel.

Radar fusion is therefore usually deferred to intermediate or late stages rather than handled as an early channel concatenation.

Joint Architectures and Intermediate Fusion

Intermediate fusion combines modality-specific features partway through the network, allowing each modality to build its own initial representation before features are merged. This is the most flexible fusion paradigm and the most computationally demanding, which is why it appears primarily on Class III and IV platforms.

The Bird's Eye View representation is the most widely used intermediate fusion space for multi-modal drone systems that need to combine camera, lidar, and radar. BEV projects all sensor data onto a common horizontal plane -- the ground plane or a plane at fixed altitude relative to the drone -- and represents the scene as a 2D spatial map with channels encoding detection confidence, height, velocity, and other per-location attributes. Each modality contributes to this map through its own projection pathway, and the merged map is then processed by a shared detection head.

The BEV projection layer for lidar is a pillarization step: lidar points within a spatial voxel column are encoded into a feature vector and placed at the corresponding BEV grid cell. For camera, the projection is more involved because a 2D image does not directly encode depth. Lift-Splat-style architectures predict a depth distribution per pixel and use it to project image features into 3D, then collapse the 3D volume onto the BEV plane. For radar, the sparse point cloud with Doppler attributes is directly placed into the BEV grid with velocity as an additional channel. The following pseudocode illustrates the BEV projection for a combined lidar-radar input.

```

def project_to_bev(lidar_points, radar_points, grid_cfg):
    # lidar_points: (N, 4) -- x, y, z, intensity
    # radar_points: (M, 5) -- x, y, z, v_r, rcs
    H, W = grid_cfg['H'], grid_cfg['W']
    x_min, x_max = grid_cfg['x_range']
    y_min, y_max = grid_cfg['y_range']
    bev = torch.zeros(7, H, W) # channels: lidar_height, lidar_intensity,
                                # lidar_density, radar_vr, radar_rcs,
                                # radar_density, occupancy

    dx = (x_max - x_min) / W
    dy = (y_max - y_min) / H

    # Project lidar
    mask = (lidar_points[:, 0] > x_min) & (lidar_points[:, 0] < x_max) \
           & (lidar_points[:, 1] > y_min) & (lidar_points[:, 1] < y_max)
    pts = lidar_points[mask]
    col = ((pts[:, 0] - x_min) / dx).long().clamp(0, W - 1)
    row = ((pts[:, 1] - y_min) / dy).long().clamp(0, H - 1)
    for i in range(len(pts)):
        r, c = row[i], col[i]
        bev[0, r, c] = max(bev[0, r, c], pts[i, 2]) # max height
        bev[1, r, c] += pts[i, 3] # accumulated intensity
        bev[2, r, c] += 1 # point density

    # Project radar
    mask_r = (radar_points[:, 0] > x_min) & (radar_points[:, 0] < x_max) \
            & (radar_points[:, 1] > y_min) & (radar_points[:, 1] < y_max)
    rpts = radar_points[mask_r]
    col_r = ((rpts[:, 0] - x_min) / dx).long().clamp(0, W - 1)
    row_r = ((rpts[:, 1] - y_min) / dy).long().clamp(0, H - 1)
    for i in range(len(rpts)):
        r, c = row_r[i], col_r[i]
        bev[3, r, c] += rpts[i, 3] # radial velocity
        bev[4, r, c] += rpts[i, 4] # RCS
        bev[5, r, c] += 1 # radar point density

    bev[6] = (bev[2] > 0).float() # occupancy from lidar
    # Normalize density channels
    bev[2] = bev[2].clamp(max=10) / 10.0
    bev[5] = bev[5].clamp(max=5) / 5.0
    return bev

```

This BEV tensor is then processed by a 2D CNN backbone -- typically a lightweight ResNet or EfficientDet neck -- to produce class and box predictions directly in BEV space. The conversion back to world coordinates is a simple inverse of the grid parameters. On a Class III platform with an Orin NX, this full pipeline runs in roughly 30 ms for a 128x128 BEV grid with 16 cm resolution, leaving headroom in a 50 ms detection budget.

Cross-modal attention is the mechanism that allows one modality's features to guide another's processing within a shared backbone. The formulation follows standard scaled dot-product attention but applies it across modality feature maps rather than across token sequences. A camera feature map F_{cam} at a given backbone stage has spatial dimensions $H \times W \times C$. A radar BEV feature map F_{rad} has the same spatial dimensions after upsampling. The cross-attention operation treats flattened F_{cam} as the query source and flattened F_{rad} as the key and value source.

```

class CrossModalAttention(nn.Module):
    def __init__(self, dim, num_heads=4):
        super().__init__()
        self.num_heads = num_heads
        self.scale = (dim // num_heads) ** -0.5
        self.q_proj = nn.Linear(dim, dim)
        self.k_proj = nn.Linear(dim, dim)
        self.v_proj = nn.Linear(dim, dim)
        self.out_proj = nn.Linear(dim, dim)

    def forward(self, query_feat, kv_feat):
        # query_feat: (B, N, C) -- flattened camera features
        # kv_feat: (B, M, C) -- flattened radar/lidar BEV features
        B, N, C = query_feat.shape
        H = self.num_heads
        d = C // H

        Q = self.q_proj(query_feat).reshape(B, N, H, d).transpose(1, 2)
        K = self.k_proj(kv_feat).reshape(B, -1, H, d).transpose(1, 2)
        V = self.v_proj(kv_feat).reshape(B, -1, H, d).transpose(1, 2)

        attn = torch.softmax((Q @ K.transpose(-2, -1)) * self.scale, dim=-1)
        out = (attn @ V).transpose(1, 2).reshape(B, N, C)
        return self.out_proj(out)

```

The cross-attention block is applied once per fusion stage, not at every layer, because its complexity scales as $O(N * M)$ where N and M are the spatial token counts of the two feature maps. On a 32x32 feature map at a deep backbone stage, $N = M = 1024$, and the attention matrix is 1 million entries per head. With four heads and INT8 quantization, this is feasible on an Orin NX in under 5 ms per forward pass. On a Class II NPU with 512 KB of working memory, it is not feasible at all -- the attention matrix alone would not fit. Class II fusion must remain at the late or very-early level.

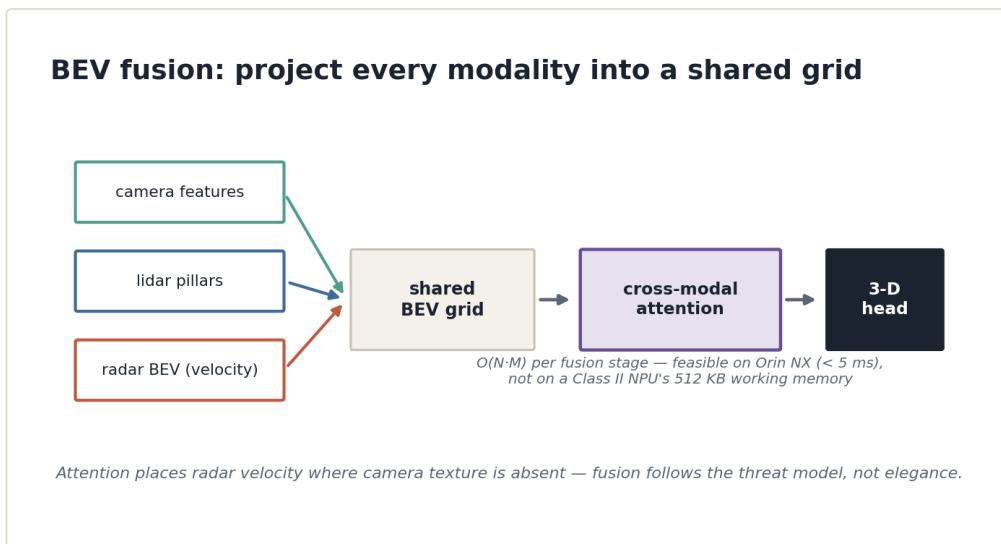


Figure 10.2 — BEV fusion projects every modality into a shared grid before cross-modal attention.

Fault Tolerance and Latency Tradeoffs Across Fusion Types

The fusion paradigm chosen has direct implications for what happens when a modality fails or degrades. Late fusion is the most fault-tolerant: each pipeline is independent, and the fusion layer can simply ignore inputs from a modality that has been flagged as unreliable. A confidence gate -- discard any track whose covariance exceeds a threshold -- is sufficient to prevent a degraded radar from poisoning camera-based detections. The cost is that the modalities cannot help each other at the feature level, so objects visible only under joint consideration -- a pedestrian whose texture is ambiguous to camera and whose velocity is ambiguous to radar but whose combination is unambiguous -- may be missed.

Early fusion is the least fault-tolerant in the naive implementation, because a corrupted input channel propagates directly into the backbone. With dropout augmentation during training, robustness improves significantly, but the network's behavior with a fully missing channel is trained rather than guaranteed. A missing D channel that was never zeroed during training will produce activations at the first layer that the rest of the network was not trained to handle.

Intermediate fusion sits between these extremes. Each modality builds its own initial feature representation before fusion, so early-layer features are modality-specific and robust to the absence of the other modality. The cross-attention layer can be made robust by masking out the key-value tokens of an absent modality and substituting a learned null embedding, which tells the attention mechanism that no cross-modal information is available. This requires explicit training with simulated modality dropout at the attention layer.

Latency in late fusion is the maximum of the per-modality pipeline latencies plus a small fusion overhead. For a camera pipeline at 20 ms and a lidar pipeline at 35 ms running in parallel, late fusion completes at roughly 37 ms. Early fusion runs a single backbone on the combined input, which is typically faster than running two independent backbones but slower than running the faster single-modality pipeline alone. Intermediate fusion adds the cross-attention block to the slower of the two per-modality paths, so its latency falls between late fusion and single-modality.

The practical implication for Class II platforms is that late fusion with a two-rate structure -- fast camera-only outputs between slower camera-lidar fused outputs -- is almost always the right choice. The NPU's memory and compute budget cannot support intermediate fusion at acceptable latency, and the fault-tolerance argument is strong for platforms that may encounter sensor degradation routinely. For Class III and IV platforms with GPU-class SoCs, BEV-based intermediate fusion with cross-modal attention is achievable within a 50 ms budget and provides detection quality that justifies its complexity for the missions those platforms fly.

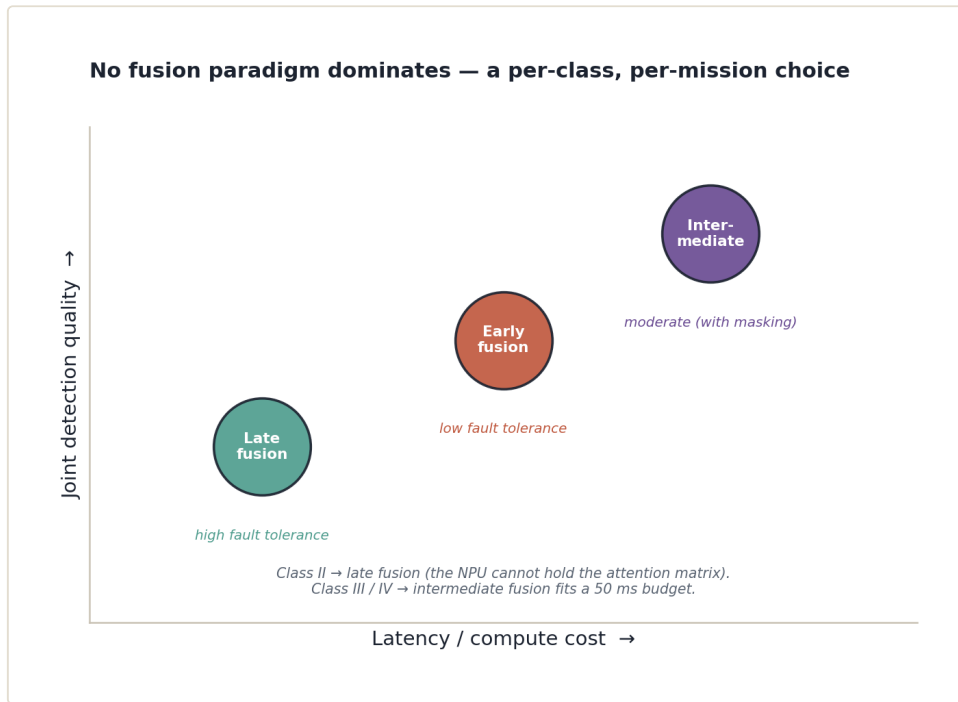


Figure 10.3 — Fault tolerance, latency, and detection quality across the three fusion paradigms.

The fusion architecture is not a single decision made once. It is a per-mission-type, per-drone-class choice that follows from the same constraints -- memory, compute, power, and reliability -- that have governed every architectural decision in this paper. A drone navigating an indoor warehouse at low speed under controlled lighting can afford to rely on camera-only late fusion and add lidar late-fusion for range confirmation. A drone conducting maritime search and rescue at night in rain has no option but to treat radar as a primary modality with early or intermediate fusion of whatever camera information is available. The fusion architecture follows from the threat model, not from a preference for architectural elegance.

What the chapters on individual modalities established is that each sensor has a regime where it is the best available source of information and a regime where it is nearly useless. The role of the fusion layer is to allocate inference weight toward whichever modality is currently most informative, whether that allocation is implemented as a fixed confidence weight in a Kalman filter, a learned gating function, a BEV projection that simply places radar velocity features where camera texture is absent, or a cross-attention block that learns which camera features to update based on what the radar sees. The mechanism differs by platform. The principle is the same across all of them.

Chapter 11: Architecture Selection Framework: Matching Models to Drone Class and Mission

Every preceding chapter has treated a single dimension of the problem in depth: what a given sensor produces, what a given hardware platform can sustain, what a given compression strategy costs. This chapter collapses those dimensions into a single question that a practitioner actually faces at the start of a project. Given a drone class, a sensor payload, a latency budget, and a mission type, which architecture family belongs in the system and what sequence of compression steps makes it deployable? The answer is not a single model name. It is a structured traversal of constraints that eliminates candidates until one or two families remain, at which point benchmarking on the target hardware decides between them.

The framework proceeds in four stages. First, the drone class fixes the outer bounds on memory, compute, and power that no architecture can exceed. Second, the sensor payload and mission type together determine the minimum capability the inference stack must provide. Third, those two sets of constraints are intersected against the architecture families catalogued in Chapters 7 through 10 to produce a candidate shortlist. Fourth, the candidates are compressed and benchmarked on-device until one meets all budget constraints simultaneously.

Stage one requires no measurement. The drone class is a physical fact established before any software decision is made. A Class I platform operating on a Cortex-M7 at 400 MHz with 512 KB SRAM and 2 MB flash will not run PointPillars. It will not run MobileViT. The L2 cache is too small for the intermediate activations of any architecture that was designed for a mobile phone, let alone a GPU. For Class I, the universe of viable architectures is already narrow before the mission type is considered: lightweight MobileNetV2 or MobileNetV3 variants pruned below 1M parameters for vision, range-image CNNs with fewer than four convolutional layers for lidar, and CFAR-based signal processing pipelines for radar. These are not approximations of the preferred solution. They are the solution space. Acknowledging that early prevents wasted engineering effort.

Class II platforms with a mobile NPU SoC in the 2W to 5W range expand the space modestly. YOLOv8-nano fits. DepthwiseSeparable single-stage detectors fit. A two-branch late-fusion structure with one camera head and one range-image head fits, provided the feature dimensions are held small and the branches are not run simultaneously on shared SRAM. Class III platforms with Jetson Orin Nano or equivalent open the architecture space to transformer-lite models, PointPillars with a compressed pillar encoder, and BEV-based intermediate fusion if the latency budget is 50 ms or above. Class IV is the only tier where full ViT variants, sparse 3D convolution networks, and cross-attention intermediate fusion are deployable without compromise.

Stage two is where mission type enters. Consider three representative mission profiles that span most operational use cases.

Obstacle avoidance at low altitude and moderate speed, say a warehouse inspection drone flying at 2 m/s through shelving corridors, requires dense spatial coverage of the near field, a latency ceiling around 50 ms to leave time for flight controller response, and robustness to close-range occlusion. It does not require long-range object classification. The architecture priority is spatial resolution in the detection output and low latency over classification accuracy. For this mission on a Class II platform, a depth-separable single-stage detector fused with a range-image lidar head via late fusion is appropriate. The lidar branch sets range, the camera branch classifies passable space, and the combination needs to run inside 40 ms to leave margin.

Object detection at extended range, for example a Class III fixed-wing conducting perimeter surveillance with a 60-frame camera and 905 nm solid-state lidar, requires high classification accuracy on small targets at distances where the lidar return is sparse. The camera is the primary source of texture and the lidar provides range confirmation rather than primary detection. The architecture priority here is classification accuracy and recall at small object sizes, which points toward transformer-lite encoders that preserve fine-grained spatial features better than purely convolutional backbones. BEV-based fusion is less useful here because the target geometry is not well described in a top-down projection. Late fusion where lidar depth confirms camera-detected bounding boxes is appropriate. A MobileViT-XS backbone with a detection head optimized for small object recall, quantized to INT8 with TensorRT, is the kind of concrete candidate this mission profile produces.

Mapping and simultaneous localization, the third representative mission, inverts the priority structure. Mapping missions on Class III or IV platforms do not require real-time object classification. They require dense, accurate geometric reconstruction of the environment, which means the lidar branch is primary and camera provides color or texture association. The inference latency ceiling is relaxed because mapping is not safety-critical in the same way obstacle avoidance is. This relaxed latency budget allows heavier lidar architectures: PointPillars with an uncompressed pillar encoder on Class III, or spconv-based sparse 3D convolution on Class IV. The camera branch may be downgraded to a lightweight feature extractor whose sole job is providing color for mesh reconstruction rather than running an independent detection head.

These three profiles illustrate the general pattern. The mission type determines which sensor carries the primary inference load, the required output format (bounding boxes versus depth maps versus occupancy grids), and the latency ceiling. The drone class determines how much compute is available. The intersection determines which architecture families are viable.

The following matrix encodes the key combinations. Each cell names the architecture family and fusion strategy, not a specific model, because model versions change faster than architectural families do.

1. Class I, obstacle avoidance: MobileNetV2 pruned below 500K parameters, range-image CNN with 3 layers max, CFAR for radar if present. Late fusion only. Latency target: 20 ms.
2. Class I, object detection: MobileNetV3-Small with INT8 post-training quantization, no lidar branch. Camera-only. Latency target: 30 ms.
3. Class I, mapping: Unsupported at inference level. Class I platforms offload mapping to ground station. Onboard compute handles only ego-motion estimation from sparse optical flow.
4. Class II, obstacle avoidance: YOLOv8-nano or DepthwiseSeparable single-stage detector INT8. Range-image lidar branch via late fusion with two-rate structure. Latency target: 40 ms.
5. Class II, object detection: YOLOv8-nano with INT8 quantization-aware training. Lidar late fusion for range confirmation. Latency target: 50 ms.
6. Class II, mapping: Lightweight monocular depth estimation network (MiDaS-Small variant) plus lidar range-image for depth correction. Late fusion. Mapping computation offloaded or duty-cycled.
7. Class III, obstacle avoidance: MobileViT-XS or EfficientFormer-L1, INT8 TensorRT. PointPillars compressed encoder for lidar. BEV late fusion or two-rate intermediate fusion. Latency target: 50 ms.
8. Class III, object detection: MobileViT-S with TensorRT INT8. BEV intermediate fusion with lidar PointPillars. Radar late fusion for velocity augmentation if radar present. Latency target: 50 ms.
9. Class III, mapping: PointPillars primary for lidar-based map accumulation. Camera feature extractor for color association. Late fusion. Latency target relaxed to 100 ms acceptable.
10. Class IV, obstacle avoidance: EfficientFormer or ViT-Small with TensorRT FP16. Sparse 3D convolution lidar branch. BEV intermediate fusion. Radar integrated via cross-attention on BEV grid. Latency target: 30 ms with multi-chip pipeline.
11. Class IV, object detection: DETR variant with deformable attention, TensorRT FP16. Full PointPillars or spconv lidar. Cross-modal attention fusion. Latency target: 50 ms.
12. Class IV, mapping: Full sparse 3D convolution stack as primary. Camera provides semantic labels. All-modality intermediate fusion in BEV space.

The matrix is a starting point, not a lookup table with guaranteed correct answers. Two cells in the same row may require different compression strategies depending on the specific SoC variant, the sensor resolution, and the inference runtime available. A Class III platform running YOLOv8-small that needs 50 ms may achieve it with INT8 quantization-aware training alone. The same platform with a higher-resolution input may need structured channel pruning to reach the activation size that fits the NPU's tile size before quantization helps at all. The matrix tells you the family to begin with. The compression sequence is determined by measurement.

Stage three is the compression sequence, and its order matters. The most common mistake is to apply all available compression techniques simultaneously and discover that the model no longer converges or that accuracy has collapsed from a combination of effects that are hard to attribute individually. The correct order is to prune first, then quantize, with distillation as an optional step inserted before quantization if accuracy after pruning is unacceptable.

Structured channel pruning with a magnitude criterion applied to convolutional layers is the right first step for vision-based models. It reduces the MAC count in a way that maps cleanly onto hardware execution because the resulting channel counts are smaller but the layer structure is preserved. Remove 30 percent of channels by lowest L1-norm magnitude per layer, then fine-tune on the downstream dataset for three to five epochs. Measure accuracy degradation. If the degradation is within 2 percentage points on the mission-relevant metric (mAP at the target IoU, obstacle detection recall at the relevant distance), proceed to quantization. If it is outside that bound, either reduce the pruning ratio or insert a distillation step before proceeding.

Quantization-aware training at INT8 follows pruning. The quantization-aware training loop inserts fake quantization nodes at weights and activations, trains for ten to twenty epochs at a reduced learning rate, and produces a model whose weights are nominally FP32 but whose activation statistics have been calibrated to the INT8 range. Post-training quantization without the QAT step is acceptable only for Class III and IV platforms where the base model is already large and accuracy degradation from static quantization is under 1 percent. For Class I and II platforms where the base model is already near the accuracy floor, QAT is mandatory. The pseudocode for a QAT training step was given in Chapter 6 and applies directly here; the only mission-specific modification is ensuring that the calibration dataset reflects the actual scene statistics of the target mission rather than a general dataset, because INT8 clipping thresholds set on ImageNet statistics will be wrong for a drone conducting nighttime maritime surveillance.

Knowledge distillation enters the sequence when the compressed student model's accuracy is still insufficient after QAT but further pruning would violate the latency budget. A teacher model that is two to three times larger than the student is trained on the full dataset with no compression, and the student is trained to match the teacher's

intermediate feature maps in addition to minimizing the task loss. Feature-map distillation on the neck or detection head features is more effective than logit distillation alone for detection tasks. The distillation loss weight must be tuned carefully; setting it too high causes the student to prioritize matching teacher features over minimizing detection loss, which produces a model with low distillation loss and poor mAP.

Stage four is on-device benchmarking, and it cannot be skipped or replaced by simulation. Inference latency measured on a desktop GPU with a TFLite interpreter is not predictive of latency on a Cortex-M7 running CMSIS-NN kernels. Throughput measured on a Jetson Orin using standard TensorRT benchmarks does not account for memory contention from the camera DMA pipeline running simultaneously. On-device benchmarking must be conducted with the full sensor pipeline active, not in isolation, because the inference SoC shares memory bandwidth with the image signal processor, the IMU DMA controller, and in some configurations a second inference core handling flight control.

The latency measurement procedure for a target platform follows a consistent pattern. The model is exported to the target runtime format: TFLite FlatBuffer for TFLite-Micro on Class I, TFLite or ONNX for NPU runtimes on Class II, TensorRT engine file for Jetson platforms on Class III/IV. The export is validated for correctness by comparing outputs against the FP32 reference model on a held-out set of inputs. Latency is measured as wall-clock time from input tensor copy to output tensor copy, not from the inference call alone, because tensor copy time is significant on platforms where the NPU and CPU do not share a unified memory pool. One hundred inference calls are made with the sensor pipeline active, and the 95th percentile latency is used as the benchmark figure rather than the mean, because outliers driven by memory contention or interrupt handling represent the worst-case the flight controller will observe.

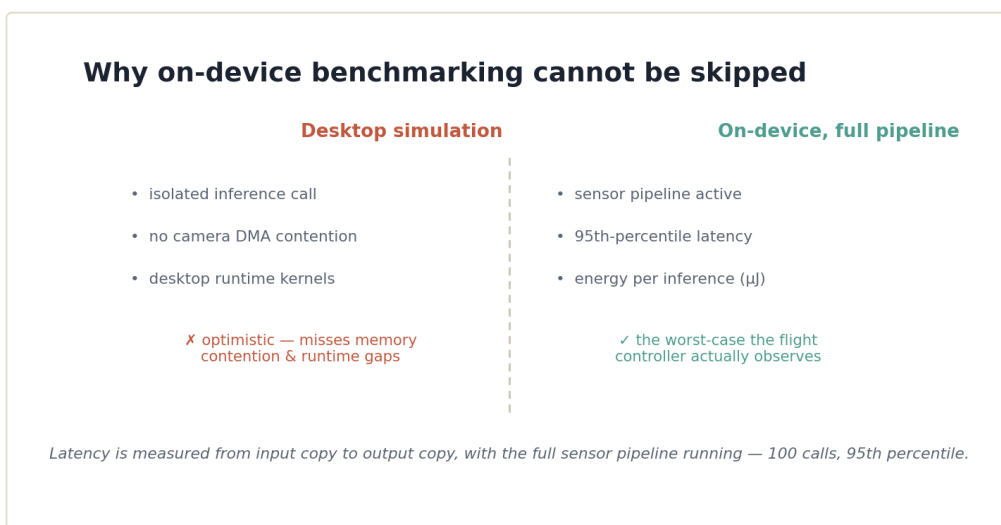


Figure 11.3 — On-device benchmarking with the full sensor pipeline cannot be replaced by simulation.

Energy measurement requires either a hardware power monitor on the supply rail or use of the SoC's built-in power estimation registers, where available. Jetson platforms expose power consumption via a tegrastats utility that reports per-module power at 100 ms intervals. For Class I and II platforms without built-in power monitoring, a shunt resistor on the compute supply rail with a current sense amplifier is the practical solution. Energy per inference is computed as average supply power during inference multiplied by the 95th percentile latency. This figure, expressed in microjoules per inference, is the metric that maps directly onto battery life impact and is more useful than TOPS or GFLOPS for mission planning.

A model that passes the latency and energy benchmarks still requires mission-specific fine-tuning before deployment. The publicly available datasets used to pretrain or train most architecture families, whether COCO for detection or nuScenes for lidar, do not match the view angle, altitude range, object scale distribution, or lighting conditions of drone-specific inference. A pedestrian in COCO occupies a large fraction of the image frame at human eye level. A pedestrian detected from a drone at 20 meters altitude appears as a small, foreshortened blob with a view angle between 30 and 70 degrees from nadir depending on gimbal position. Models trained only on ground-level data consistently underperform on aerial imagery at non-trivial altitude.

Dataset curation for drone-specific fine-tuning involves three sources. First, publicly available aerial datasets: VisDrone for detection at altitude, AU-AIR for multi-class aerial detection, and SeaDronesSee for maritime scenarios are the most commonly used. Second, synthetic data generated from simulation environments such as AirSim or Isaac Sim, which allows control over altitude, lighting, and object density to fill gaps in the real-world datasets. Third, real flight data collected during system integration testing, which is the most valuable but also the most expensive to annotate. Even a few hundred labeled frames from actual flight at the target altitude and lighting regime will shift model accuracy more than thousands of synthetic frames, because the synthetic domain gap for texture and lighting is significant even with modern rendering.

The fine-tuning regime for a compressed model should use a learning rate two orders of magnitude below the original training rate to avoid destroying the compression gains. If the model was quantized with QAT, the QAT training loop must remain active during fine-tuning; removing fake quantization nodes during fine-tuning and reapplying them afterward reintroduces calibration error. The fine-tuning dataset should be split with a held-out test set that reflects the exact operational envelope: same altitude band, same time of day range, same object categories present in the mission. Reporting accuracy on a general benchmark after mission-specific fine-tuning is misleading; the relevant metric is performance on the operational test set.

The decision framework as a whole can be expressed as a sequential checklist. Fix the drone class first; this eliminates entire architecture families immediately and prevents wasted effort on models that will never fit the hardware. Identify the primary sensing

modality from the mission type; this determines which branch carries the main inference load and which carries confirmation or augmentation. Select the architecture family from the intersection of class and mission. Apply compression in the correct order: structured pruning, then QAT, with distillation inserted only if pruning alone leaves accuracy too low. Benchmark on-device with the full sensor pipeline active, using 95th-percentile latency and energy per inference as the pass or fail criteria. Fine-tune on mission-specific data with a low learning rate and the compression pipeline active.

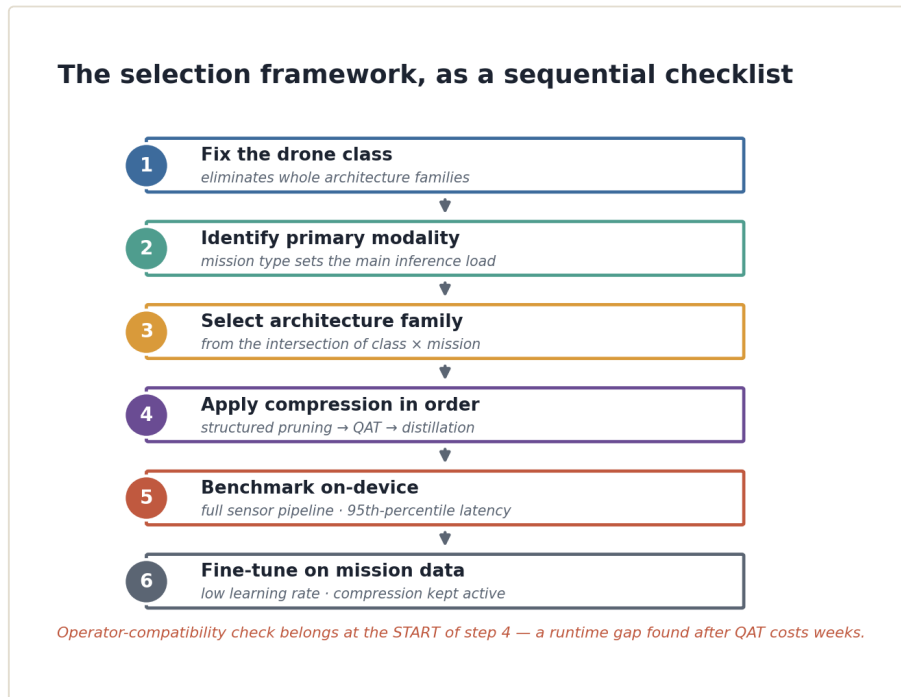


Figure 11.1 — The architecture-selection framework as a six-step sequential checklist.

One pattern appears repeatedly in practice and is worth naming explicitly. An architecture family that is technically feasible within the latency and memory budgets will sometimes still fail on-device because of runtime support gaps. A model that uses grouped convolutions with non-standard group counts may run correctly in PyTorch but produce unsupported operator errors in TFLite-Micro. A sparse convolution network designed for a Jetson Orin may rely on spconv kernels that have not been ported to the CUDA version shipping with the target JetPack release. Runtime support must be verified for every operator in the exported model against the specific runtime version on the target hardware before any compression or fine-tuning work begins. Discovering a runtime support gap after completing QAT is the kind of delay that adds weeks to a project. The operator compatibility check belongs at the start of stage three, not the end.

What the framework does not do is make the tradeoff between competing constraints disappear. It makes the tradeoff explicit and forces the decision to be made early, with

full awareness of its consequences. A Class II platform assigned a mission that genuinely requires BEV intermediate fusion for acceptable detection quality is a platform that is underpowered for that mission, and no compression strategy will fix the underlying hardware deficit. The correct response in that situation is either to accept lower detection quality by reverting to late fusion, or to escalate to a Class III platform. The framework clarifies that choice. Clarifying the choice early, when there is still time to change the hardware selection, is the most valuable thing a structured decision process can provide.

Chapter 12: Frontiers: Towards Ubiquitous Embedded Lidar and Radar on Drones

Every constraint described in the preceding eleven chapters is a snapshot, not a law. The power envelopes, memory ceilings, and sensor weight budgets that forced the architectural compromises documented throughout this book are products of 2024-era silicon and packaging. The trajectory of each underlying technology points toward a near-term future in which the constraints shift enough that today's taxonomy begins to strain at the edges. Class I platforms that currently run monocular-only inference pipelines will carry solid-state lidar. Class II platforms that treat radar as an optional luxury will treat it as a baseline. The inference architectures that survive that transition will be the ones designed to handle all three modalities from the start, even when some of those modalities are currently absent from the physical payload.

This chapter does not project a speculative horizon ten years out. It examines trends that are already in the prototype and early-production stage and traces their concrete implications for inference architecture and sensor fusion. The goal is to identify what must be done now in architecture design so that platforms built today remain extensible when the hardware arrives.

Solid-state lidar has been shrinking steadily since the first MEMS-mirror designs appeared in automotive development programs around 2019. The key figure is not beam count or angular resolution, which are already competitive with spinning units for many drone applications. The key figure is mass and idle power draw. Early solid-state lidar units suitable for outdoor ranging weighed 80 to 150 grams and drew 3 to 5 watts continuously. Units entering production in 2024 and announced for 2025 delivery are approaching 20 to 35 grams with sub-2-watt idle draw in reduced-field configurations. At 20 grams and 1.5 watts, a solid-state lidar clears the payload and power budget of a Class II platform without structural compromise. At 35 grams and 2 watts, it is within reach of high-end Class I designs that accept reduced flight time.

The inference implication is immediate. A Class II platform that previously required a range-image CNN operating on simulated or stereo-derived pseudo-depth can now ingest real sparse point cloud data at 15 to 20 Hz. This is not simply an upgrade in sensor quality. It changes the architecture feasibility profile in a specific way: the pillarization pathway described in Chapter 8 becomes viable on NXP i.MX 9-class SoCs, whereas previously the memory cost of the pillar feature extraction step exceeded what those platforms could sustain alongside the camera pipeline. The arithmetic of that tradeoff shifts when the lidar's native point density is low enough that the pillar grid stays sparse. A 20-gram solid-state lidar producing 10,000 to 30,000 points per frame at 15 Hz generates roughly half the pillar occupancy of the mechanical lidar datasets that PointPillars was originally benchmarked on. That sparsity is not a limitation to

work around. It is the exact property that makes pillarization cheap enough to run on a mid-range NPU.

What this means for architecture design today is that any Class II inference pipeline being designed for camera-plus-depth should be structured as a genuine two-branch architecture with late fusion rather than treating pseudo-depth as a channel concatenation hack. The pseudo-depth channel input is the expedient that works now. The pillar-based branch is the path that will be ready to accept real lidar data when the hardware arrives. Designing the camera branch and the lidar branch as separable modules with a well-defined fusion interface costs almost nothing in development effort and avoids a complete rewrite when the sensor payload changes. This is an argument for treating the fusion seam as a first-class design boundary, not an afterthought.

The radar trajectory is in some ways more advanced than lidar because the silicon path for millimeter-wave radar does not require novel optical assembly. Texas Instruments has shipped AWR family radar-on-chip devices since 2017. What has changed is the package size, the antenna configuration available in small packages, and the processing that happens on-chip before data is handed to the host compute element. The AWR6843AOP, a device in production now, integrates a 4-transmit 3-receive antenna array with on-chip DSP for FFT and CFAR processing into a package small enough to fit on a rigid-flex board appropriate for Class II payloads. The data that leaves the chip can be a radar point cloud rather than raw ADC samples, which shifts the host-side processing requirement from signal processing to inference.

This distinction matters architecturally. The radar pipeline described in Chapter 9 for Class II platforms centered on small CNNs operating on range-Doppler maps produced after the FFT stage. When the FFT and CFAR stages are completed on-chip, the host does not receive a range-Doppler map. It receives a sparse set of detected points annotated with range, azimuth, elevation, and radial velocity. The inference task on the host side becomes object classification and tracking from radar point clouds, which is structurally similar to lidar point cloud inference but with far fewer points, much larger position uncertainty, and the addition of velocity as a reliable per-point feature. This is a different problem from range-Doppler CNN inference, and it calls for a different architecture family.

Lightweight PointNet variants adapted for radar point clouds are the natural candidate, and several research groups have demonstrated that a three-layer PointNet with a 64-dimensional global feature is sufficient for object classification from AWR-class radar data with under 50,000 multiply-accumulate operations per frame. At that MAC count, the inference fits comfortably on the NPU of a Class II SoC within a 5-millisecond budget. The more interesting architectural question is how to fuse per-point radial velocity into the classification head. Treating velocity as a fourth input dimension alongside x , y , z is the naive approach and works reasonably well. The more principled approach is to train a velocity-conditioned attention weight that amplifies points with

velocity magnitudes inconsistent with platform ego-motion, since those are the points most likely to correspond to non-static objects of interest. This requires the platform to expose its own velocity estimate to the inference pipeline, which in turn requires the radar inference component to receive data from the flight controller's state estimator. The data dependency is straightforward but must be designed into the pipeline explicitly.

The 60 GHz band, distinct from the 77 GHz automotive radar band, is emerging as a candidate for short-range indoor and cluttered-environment operation. Silicon at 60 GHz allows smaller antenna geometries and benefits from stronger atmospheric absorption that limits interference range but also limits useful detection distance. For drone obstacle avoidance in confined indoor environments where 77 GHz lidar or structured light systems would saturate with returns, 60 GHz radar offers a useful complementary modality. The inference problem at 60 GHz is primarily occupancy and proximity, not classification, which means the CFAR-based detection pipelines discussed in Chapter 9 for Class I remain appropriate. The miniaturization progress on 60 GHz hardware is bringing it toward Class I payload budgets faster than the 77 GHz band, which is relevant for nano-UAV indoor navigation.

Neuromorphic inference hardware represents the most architecturally disruptive development in this space, though it remains further from production deployment than lidar or radar miniaturization. The core proposition of neuromorphic chips, exemplified by Intel's Loihi 2 and tentatively by Synaptics' Orca platform, is spike-based computation that is quiescent when the input is static and activates only when the input changes. For sensor fusion on drones, where the dominant compute cost often comes from running full inference on frames that contain no meaningful change from the prior frame, the potential energy reduction is substantial. A platform flying a straight-line survey route over featureless terrain is invoking the full forward pass of its obstacle detection model 30 times per second on frames that differ primarily in noise. A spike-based architecture that gates on detected change processes those frames for almost nothing and reserves its compute budget for frames with actual scene dynamics.

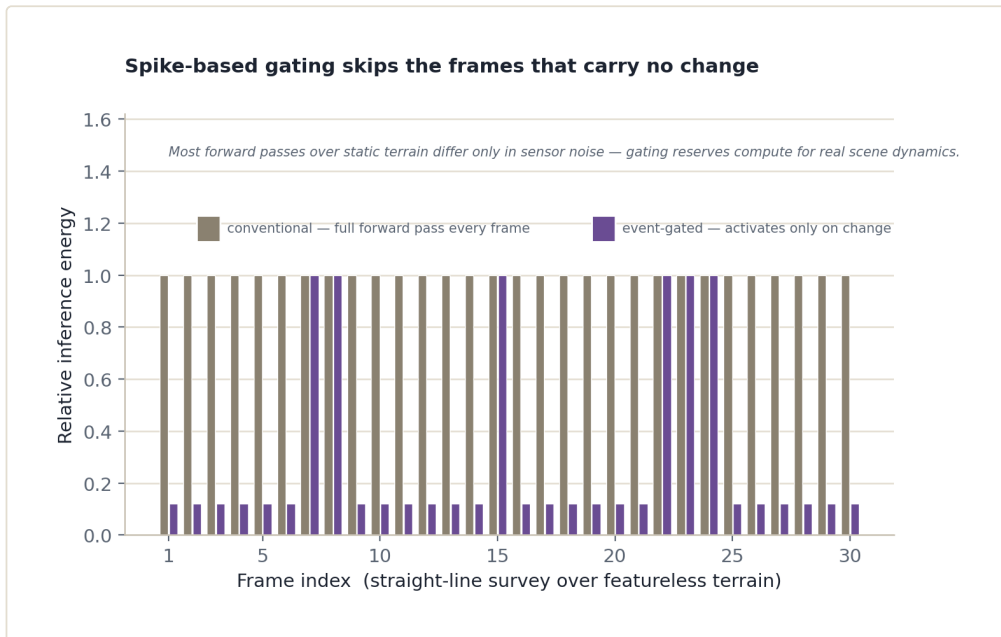


Figure 12.2 — Why spike-based gating saves energy: most frames over static terrain carry no real change.

The catch is software maturity. Programming a neuromorphic chip for drone inference today requires abandoning the standard model training ecosystem and working with spike-rate encoding schemes, temporal dynamics parameters, and learning rules that do not map cleanly to gradient descent on standard datasets. The gap between a PyTorch-trained detector and a deployable Loihi 2 implementation is not a quantization step. It is closer to a full redevelopment. Research groups working on event-camera integration with neuromorphic hardware are making progress because event cameras and neuromorphic chips share the spike-based data model natively, but extending that to fused camera-lidar-radar inference on neuromorphic hardware is an open research problem without a clear solution timeline.

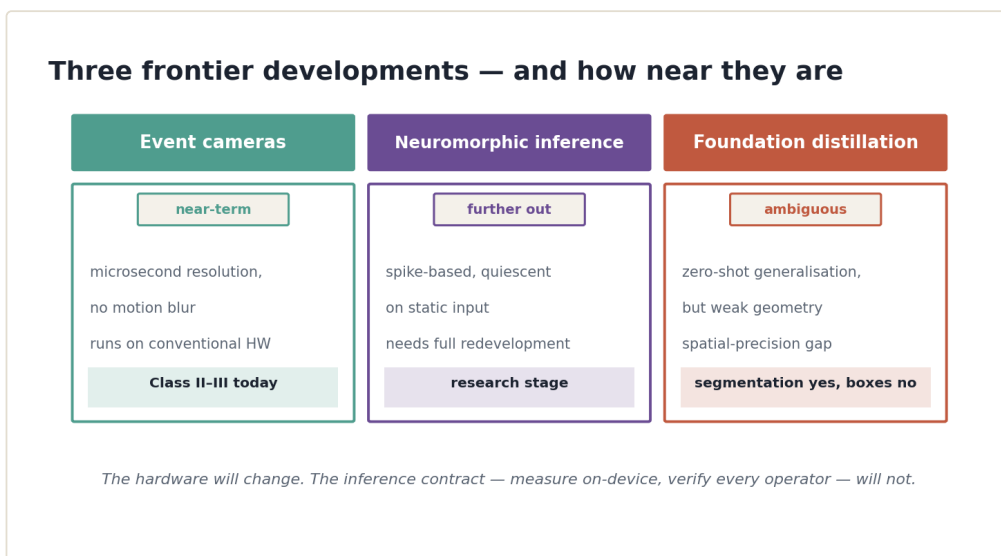


Figure 12.1 — Three frontier developments and their maturity for drone edge inference.

Event cameras, even deployed on conventional inference hardware, are worth treating as a near-term consideration for Class II and Class III platforms operating in challenging lighting conditions. An event camera produces output only at pixels where luminance changes exceed a threshold, with microsecond-level temporal resolution and no motion blur. For obstacle detection on fast-moving platforms where conventional frame cameras suffer from blur and HDR saturation simultaneously, event cameras offer an attractive alternative. The inference problem on event data is structurally different from frame inference: the input is a stream of timestamped pixel events that must be accumulated into a tensor representation before a conventional CNN can process it, or else processed directly with a recurrent or attention-based architecture that handles asynchronous input.

The most practical approach for near-term deployment is to accumulate events into a constant-count or constant-time histogram that approximates a frame, then run a standard lightweight CNN on that representation. This wastes much of the temporal resolution advantage but produces a pipeline that is compatible with existing TFLite or TensorRT runtimes and existing training infrastructure. The more sophisticated approach is to train a small spiking or recurrent network directly on the raw event stream. Several research groups have demonstrated SSD-style detectors operating on event histograms with latency competitive with frame-based cameras on the same hardware, while maintaining detection quality in low-light and high-motion conditions where frame cameras degrade substantially. The case for carrying an event camera as a supplementary modality alongside a conventional frame camera is already strong for Class III platforms in environments with high dynamic range or rapid ego-motion. As event camera unit costs continue to fall toward conventional global-shutter camera prices, the case extends to Class II.

Foundation model distillation is the development with the broadest potential impact and the most ambiguous near-term timeline. The premise is that large vision-language models trained on internet-scale data encode scene understanding that can be compressed into small edge-deployable models through distillation, without requiring the small model to be trained on a labeled drone-specific dataset at all. The appeal for drone inference is obvious: labeled datasets of drone-perspective obstacle and object imagery are expensive and sparse compared to ground-level datasets, and models fine-tuned on available datasets generalize poorly to novel environments. If a distilled edge model could inherit zero-shot generalization from a foundation model, the labeling bottleneck would shrink substantially.

The practical state of this approach in 2024 is that it works well for semantic segmentation and object classification in moderate-resolution imagery under benign conditions, and it works poorly for bounding box regression, depth estimation, and any task requiring precise geometric understanding of sparse or occluded scenes. The distillation process transfers the semantic knowledge in the foundation model's feature space reasonably faithfully but does not transfer the spatial precision that geometric

inference requires. For a drone obstacle avoidance system, that spatial imprecision is not a cosmetic limitation. A model that correctly classifies a tree but misestimates its distance by 30 percent in zero-shot deployment is not deployable. The gap between what foundation model distillation delivers today and what drone edge inference requires is real, and closing it is an active research problem involving better spatial grounding in the teacher model, better distillation objectives for geometric tasks, and drone-specific self-supervised data collection.

That said, the trajectory is toward closing this gap, not widening it. Foundation models with explicit 3D spatial grounding, trained on point cloud and depth map data in addition to RGB imagery, are beginning to appear. Distilling those models into compact architectures that run on Jetson Orin-class hardware is demonstrably achievable today for tasks that do not require millisecond latency. As inference efficiency of the distilled models improves through better structured pruning and hardware-aware neural architecture search, the zero-shot deployment scenario moves from laboratory demonstration to practical utility. The implication for this book's architecture recommendations is that the compression pipeline described in Chapter 6, specifically the sequence of structured pruning followed by quantization-aware training, remains the correct approach for models derived from foundation model distillation, because those models arrive over-parameterized in the same way that any large pre-trained model does.

Taken together, these developments point toward a specific convergence that is worth naming directly. Within a three-to-five year horizon, Class II platforms will routinely carry all three primary modalities: a conventional or event camera, a solid-state lidar producing a real sparse point cloud, and a radar-on-chip producing a velocity-annotated point cloud. The inference architecture that processes those three inputs simultaneously on a mid-range NPU SoC is not a scaled-down version of a Class IV multi-sensor fusion system. It is a genuinely new design problem. The BEV projection and cross-attention fusion techniques described in Chapter 10 provide the right conceptual starting point, but the memory budgets of Class II hardware will force simplifications that Class IV never required.

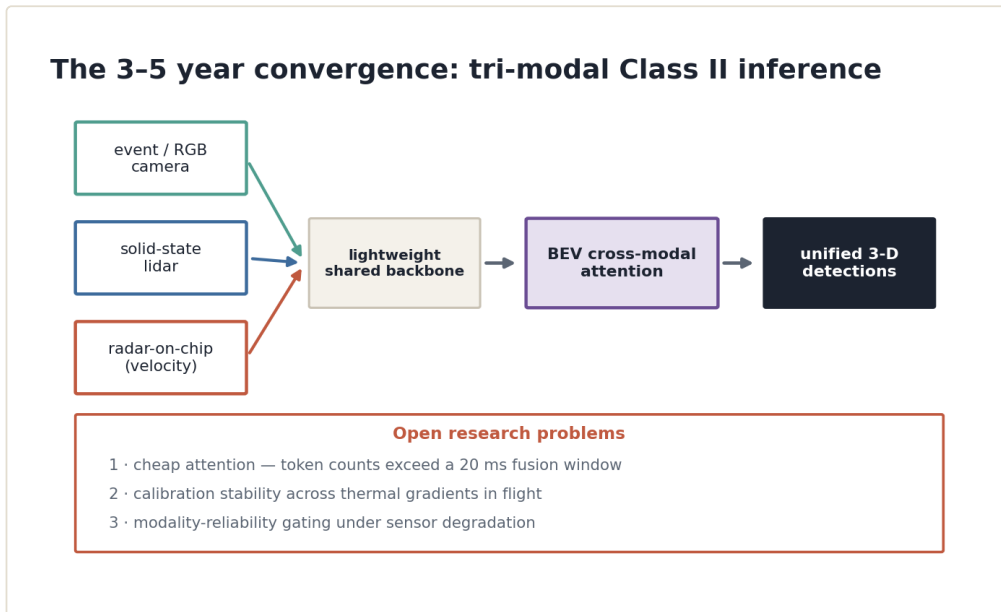


Figure 12.3 — The three-to-five year convergence toward tri-modal Class II inference, and its open problems.

The open research problem is not which architecture family to use. The family is approximately known: a lightweight shared backbone that ingests camera features, a small pillar or point-cloud branch that ingests lidar data, a velocity-conditioned point network that ingests radar data, and a cross-modal attention fusion head that operates in BEV space. The open problem is how to perform the attention computation cheaply enough. Standard multi-head self-attention scales quadratically with the number of tokens, and when BEV cells, lidar pillars, and radar points are all tokenized simultaneously, the token count at Class II resolution budgets exceeds what the attention mechanism can process within a 20-millisecond fusion window on current NPU hardware. Linear attention approximations, sparse attention with learned top-k gating, and factored cross-attention that operates on modality-pair interactions independently rather than jointly are all active areas of investigation, none of which has produced a clearly dominant solution for the drone-specific setting.

The second open research problem is calibration stability across flight conditions. Multi-modal fusion assumes that the extrinsic calibration between sensor frames is known and stable. For solid-state lidar and cameras sharing a rigid mount, this assumption holds well in benign thermal conditions. When the platform is climbing through a 30-degree Celsius temperature gradient between ground level and 300 meters, differential thermal expansion of the mount shifts the calibration by amounts that matter for tight fusion. Radar antenna phase calibration drifts with temperature on a different timescale and with different magnitude than lidar extrinsics. Online calibration estimation, where the fusion architecture continuously estimates residual calibration error from the data itself, is a research direction that intersects inference architecture and state estimation in a way that neither field has fully addressed for airborne multi-modal systems.

The third open problem is failure mode characterization under sensor degradation. Individual modality inference is relatively easy to evaluate: run the detector on degraded inputs and measure accuracy drop. Joint fusion architectures degrade in more complex ways. When one modality degrades and the fusion weights assigned to that modality do not update to reflect the degradation, the fused output can be worse than using the remaining reliable modalities alone. Teaching a fusion architecture to estimate its own modality reliability at inference time, without access to ground truth, and to gate fusion weights accordingly, is an unsolved problem for real-time edge deployment. Bayesian approaches to this problem exist in the offline literature but have not been reduced to forms that run within Class II or Class III inference budgets.

What unifies these open problems is that they are all problems about operating reliably under conditions that differ from training conditions. The edge inference contract on a drone, stated in Chapter 1, includes the requirement that inference be reliable without retry in an environment the model may not have seen. Foundation model distillation addresses part of this through better generalization. Online calibration estimation addresses part of this through adaptive geometry handling. Modality-reliability gating addresses part of this through graceful degradation. None of these is a substitute for the others.

The reason to treat lidar, radar, and vision as coequal modalities throughout this book, even for drone classes where one or two of them are currently absent from practical payloads, is precisely this convergence. An engineer designing a Class II inference pipeline today that is camera-only, because no Class II lidar or radar payload is in the current product budget, is designing either a dead end or an extensible system. The extensible version has a modular architecture with clean fusion seams, operator-compatible branches for the absent modalities, and a compression pipeline that will accommodate additional branches without complete redesign. That system costs almost nothing extra to design today and saves a significant amount of rework when the solid-state lidar arrives in eighteen months.

The engineering discipline this book has tried to describe, mapping constraints precisely before selecting architectures, verifying operator support before compressing, measuring on-device before declaring success, applies without modification to the frontier systems described in this chapter. A neuromorphic chip does not change the requirement to measure 95th-percentile latency on the target hardware with the full sensor pipeline active. A foundation-distilled model does not change the requirement to verify that every operator in the exported model is supported by the target runtime. The hardware will change. The inference contract will not.