

Article

An Open MCU-Embedded Platform for Real-Time Onboard Vision on Resource-Constrained UAV Systems

Bogdan Nedelcu * and Adina Magda Florea 

Faculty of Automatic Control and Computer Science, National University of Science and Technology, POLITEHNICA Bucharest, 60042 Bucharest, Romania; adina.florea@upb.ro

* Correspondence: radu_bogdan.nedelcu@stud.acs.upb.ro

Highlights

What are the main findings?

- A compact MCU–EdgeTPU platform can run a complete onboard perception pipeline—dual-camera sensing, hardware-assisted preprocessing, EdgeTPU inference, and post-processing—together with a scriptable MicroPython layer that exposes the detections to drone-control workflows for UAV experiments.
- Maximum embedded performance requires co-designing both ends of the system: the board/runtime must be optimized for stable real-time inference, while the detector and synthetic training dataset must match the camera resolution, viewpoint, object scale, lighting, weather, and flight conditions expected in deployment. The onboard throughput is bounded by the USB transfer of the input image, not by the accelerator—feeding the camera’s native YUV420 format instead of RGB halves that transfer and raising the pipeline from about 26 to about 40 detections per second at negligible accuracy cost.

What are the implications of the main findings?

- Beyond a passive camera or inference coprocessor, compact embedded AI hardware can host an onboard scriptable layer that exposes detections to the flight stack, providing an integration path toward person-aware navigation and ground-person avoidance—demonstrated as an interface rather than as fully validated closed-loop autonomy.
- Synthetic data can be used strategically to train EdgeTPU-suitable detectors for a known sensor geometry and mission scenario before large amounts of real flight data are available.



Academic Editors: Shunchao Wang, Meng Li, Shuo Cheng, Qi Cao, Rui Shang and Qi Liu

Received: 4 June 2026

Revised: 8 July 2026

Accepted: 8 July 2026

Published: 13 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Abstract

This paper presents a lightweight MCU–EdgeTPU platform—a microcontroller unit (MCU) paired with an Edge Tensor Processing Unit (EdgeTPU) accelerator—for onboard drone-perception experiments, extended from an open-source baseline originally limited to Quarter Video Graphics Array (QVGA) single-camera operation. Rather than treating hardware, runtime, model, and data as separate problems, they are developed as parts of the same continuous perception pipeline. The platform extends the hardware baseline toward dual 5 Mpx sensing, onboard inertial measurement unit (IMU) support, real-time embedded inference, and a high-level MicroPython control layer. In parallel, lightweight You Only Look Once (YOLO) detectors are trained and selected on a synthetic aerial-person dataset generated under the visual conditions expected by the drone camera, including target resolution, viewpoint, object scale, weather, lighting, and time-of-day variation. The resulting

workflow starts from both ends: the detector must be small and quantization-stable enough for the EdgeTPU path, while the dataset must match the images that the onboard sensor is expected to observe. To evaluate the system, the full path from camera capture and image conversion to TPU transfer, model execution, and post-inference processing is analyzed. In the tested setup, the optimized single-camera pipeline runs stably with no timeouts or inference failures at about 26 detections per second with standard RGB input; because each EdgeTPU invocation is bounded by the USB transfer of the input image, feeding the camera's native YUV420 format instead halves that transfer and raises throughput to about 40 detections per second at the same accuracy, while the selected 8-bit-integer (INT8) person detector preserves most of its 32-bit floating-point (FP32) accuracy. Detections are exposed to drone-control workflows (MAVLink/PX4 and Crazyflie) through the scriptable layer as an integration interface rather than a validated autonomy stack. The central contribution is therefore a co-designed embedded perception pipeline in which the board, runtime, detector, dataset, and even the camera pixel format are aligned around the same operating conditions.

Keywords: embedded AI; MCU; EdgeTPU; MicroPython; drone autonomy; MAVLink; PX4; Crazyflie; dual camera; FreeRTOS; embedded inference platform

1. Introduction

Small unmanned aerial vehicles (UAVs) are increasingly used as experimental platforms for onboard perception [1], but deploying real-time vision on them remains difficult when the full system is considered. The challenge is not limited to running a neural network. A useful onboard system must acquire images, prepare them for inference, execute the model, interpret the result, and communicate it to the flight logic within a time budget that remains meaningful for motion. This is especially difficult on compact platforms, where mass, power, memory bandwidth, sensor bandwidth, and software complexity are all constrained.

In many embedded-AI workflows, the perception problem is separated into independent parts. A board is first treated as an inference target, a model is then selected from an existing family, and a dataset is chosen or collected separately. This approach is convenient, but it often produces mismatches. A model may perform well in FP32 but fail after INT8 quantization [2]. A detector may be accurate on a generic benchmark but unsuitable for the object scale seen by the real camera. A dataset may contain relevant classes but not the viewpoint, resolution, lighting, weather, or altitude range expected in flight. On constrained UAV hardware, these mismatches are not minor details; they directly affect whether the complete camera-to-action loop can operate in real time.

This work addresses the problem from both ends. On the hardware and runtime side, this work develops a compact MCU–EdgeTPU platform intended for onboard drone experiments. The platform extends an open-source MCU-class embedded-AI baseline that was originally limited to a simpler single-camera workflow, adding dual 5 Mpx camera support, onboard IMU integration, hardware-assisted image preparation, EdgeTPU inference, and a high-level MicroPython control layer. The objective is not only to run a detector, but to expose onboard visual results through a scriptable interface so they can feed mission logic or be communicated to drone-control stacks—an integration path rather than a complete autonomy solution.

On the data and model side, a person detector is trained on a synthetic aerial dataset generated for the camera geometry and mission conditions expected by the platform. The

dataset is not treated as a generic image collection. It is built around the visual regime of the drone: aerial viewpoint, small human targets, 640×480 training resolution, daylight variation, weather variation, and scene types relevant to low-altitude UAV operation. This is important because the apparent size of a person in the image is determined by the altitude, field of view, and preprocessing resolution. For ground-person awareness and avoidance, the detector should learn the image distribution that the onboard camera is expected to observe, not only a broad distribution from unrelated datasets.

The model search follows the same deployment-oriented logic. Lightweight YOLO variants are compared not only by FP32 validation accuracy, but also by their behavior after INT8 conversion. This distinction matters because the target runtime uses an EdgeTPU-oriented deployment path. Some architectures that appear strong during training are not suitable after quantization, while other simpler variants preserve accuracy more reliably. The selected detector is therefore the model that best satisfies the combined constraints of accuracy, compactness, quantization stability, and compatibility with the embedded pipeline.

A central point of the paper is that real-time embedded perception depends on the whole chain. During development, camera capture, pixel conversion and scaling, tensor preparation, USB transfer toward the EdgeTPU, model execution, and post-processing were each measured and optimized. These measurements showed that the accelerator alone was not the limiting factor. Stability and throughput were strongly affected by memory placement, DMA behavior, SDRAM contention, USB transfer patterns, and RTOS scheduling. In several cases, a model could run in isolation, but the full pipeline became unstable when camera and preprocessing traffic were active at the same time.

For this reason, the platform was shaped through repeated profiling and correction. Hot buffers and selected runtime paths were moved away from contention-prone memory regions, camera buffers were isolated, and ownership rules were tightened between preparation and inference stages. The result is a more stable embedded vision platform rather than only a faster model benchmark. In the final configuration, the system runs the single-camera end-to-end pipeline stably at about 26 detections per second with RGB input and about 40 detections per second when the camera's native YUV420 format is fed directly to the accelerator, and supports dual-camera operation with controlled switching behavior.

The runtime is designed to connect perception to drone behavior. Low-level timing-sensitive work remains implemented in C/C++, while MicroPython exposes sensing, inference, communication, and mission logic at a higher level. This split allows experiments to be modified quickly without rebuilding the entire firmware. The same platform can therefore be used with small research drones such as Crazyflie and with more standard MAVLink/PX4-oriented workflows. In this setup, detections are not only visual outputs; they can become inputs to onboard decision logic, obstacle reporting, target-following behavior, or person-aware mission constraints.

The contribution of this paper is therefore not a detector alone and not a board alone. The primary contribution is the open MCU-class onboard perception platform together with the measurement-led co-design methodology behind it—a workflow in which the board, runtime, detector, and dataset are aligned around the same operating conditions. Its central scientific finding is that, on constrained UAV hardware, real-time perception is bounded by the complete camera-to-action chain—memory placement, DMA and SDRAM behavior, the USB/EdgeTPU transfer path, and RTOS scheduling—rather than by the accelerator in isolation. The board is optimized to run the detector in real time, while the detector is trained on images that match the resolution, object scale, viewpoint, and environmental variation anticipated from the onboard camera. This two-sided design is the main practical lesson of the work: maximum performance on constrained UAV hardware

is obtained when the embedded system and the data distribution are designed together. A direct consequence, demonstrated here, is that a detector adapted to the board's dominant bottleneck—the USB transfer of the input image—can match the accuracy of an unadapted model while running substantially faster, and that this best-fit configuration is found empirically, through on-board measurement, rather than from accuracy figures alone.

This primary contribution is realized through the following components, of which the platform and its whole-chain pipeline analysis are central and the remainder are enabling

- a. A compact MCU–EdgeTPU platform for UAV experiments, extended toward dual-camera operation, onboard IMU support, hardware-assisted preprocessing, and real-time embedded inference.
- b. A split runtime architecture in which C/C++ handles timing-critical camera, memory, USB, and TPU operations, while MicroPython exposes high-level mission logic and integration with drone-control workflows.
- c. A stage-level experimental analysis of the full onboard camera-to-detection pipeline, including capture, preprocessing, tensor transfer, inference, and post-processing under real runtime conditions.
- d. A synthetic aerial-person dataset generated around the expected drone-camera operating conditions, including viewpoint, target resolution, human pixel scale, weather, lighting, and scene variation.
- e. A deployment-oriented detector selection study comparing lightweight YOLO variants by both FP32 accuracy and INT8 quantization behavior, resulting in a compact person detector suitable for the EdgeTPU-oriented pipeline.
- f. An input-format co-design in which the detector consumes the camera's native YUV420 planes directly, halving the USB input-tensor transfer that bounds each EdgeTPU invocation and yielding a $1.5\text{--}1.6\times$ on-board throughput gain (about 26 to about 40 detections per second) at no accuracy cost.
- g. A scriptable integration interface that exposes onboard detections to Crazyflie- and MAVLink/PX4-based workflows, demonstrating the perception-to-action path as an interface rather than a validated closed-loop autonomy result.

The rest of the paper is organized as follows. Section 2 reviews related work in embedded UAV perception, lightweight detection, accelerator-based inference, and drone-control integration. Section 3 describes the materials and methods, including the hardware platform, camera subsystem, runtime structure, synthetic dataset generation, and experimental methodology. Section 4 reports the measured software, memory, runtime, detector, and integration results. Section 5 discusses the implications of co-designing the board, runtime, model, and dataset. Section 6 concludes the paper and outlines future work.

2. Related Work

Research on autonomous drone perception often sits between two quite different lines of work. One line focuses on small aerial platforms, where weight, power, and onboard compute are tightly constrained. The other focuses on modern vision models and tracking methods, which are usually developed and evaluated with less attention to the full embedded pipeline that has to support them in flight. This work is positioned between these two directions, with an interest not only in running detection onboard but in turning the result into a usable perception-to-action loop on compact MCU-class hardware.

A useful starting point is the work of Palossi et al., who presented an open-source and open-hardware deep-learning-based visual navigation engine for autonomous nano-UAVs [1]. That paper is relevant here because it shows, very clearly, that meaningful onboard vision is possible even on highly constrained aerial platforms when the full system is designed around that goal. It also motivates the kind of engineering trade-offs that matter

in this setting: sensor handling, low-power computation, and tight runtime control. The present platform follows a similar embedded spirit, but moves toward a different system point by combining MCU-class hardware with an EdgeTPU, dual high-resolution cameras, a MicroPython control layer, and direct integration with both small research drones and MAVLink/PX4-based systems.

At the model level, the YOLO family remains a natural reference point for real-time embedded detection. The original YOLO paper by Redmon et al. [3] established the one-stage formulation that made real-time object detection practical, while YOLOv3 [4] and YOLOv4 [5] improved the balance between speed and accuracy. These works are relevant because the detector path here follows the same general direction [6]. Recent surveys of onboard aerial detection [7] and benchmarks of object detectors on edge devices [8] confirm that lightweight architectures and accelerator deployment are now well studied; however, such studies typically evaluate the detector in isolation—on accuracy, latency, or energy—rather than inside the complete camera-to-action pipeline that has to sustain it in flight, which is the emphasis of this work. At the same time, existing models are not merely reused: a detector variant adapted to the embedded architecture is proposed and trained on a synthetic aerial dataset for ground-person detection. This makes the model part of the paper specific to the platform rather than generic. Even so, the focus remains the full onboard chain in which the detector is used: image capture, conversion, tensor staging, transfer, inference, and post-processing under RTOS constraints.

Tracking is also important when visual detections are meant to influence flight behavior rather than only produce per-frame outputs. SORT by Bewley et al. [9] is still a useful reference because it shows how much can be achieved with a relatively simple online tracking pipeline built around detections. ByteTrack by Zhang et al. [10] is relevant for a different reason: it shows that good tracking performance can come from keeping more of the detection stream instead of aggressively discarding lower-confidence boxes. Here, these references matter less as end goals and more as examples of the kind of lightweight online post-processing that can be integrated into an onboard perception loop. In other words, once detections are available at useful rates, the next question is how to stabilize them over time and turn them into something a drone controller can use.

Because the platform is more than a detector host, the software stack also matters in practice. FreeRTOS [11] is relevant because the results here depend heavily on how concurrent tasks are scheduled when capture, image preparation, transfer, and inference overlap. MicroPython [12] matters because one of the main ideas in this work is that high-level control code can stay onboard and still take part in real-time mission decisions. For MCU-side inference, TensorFlow Lite Micro [13] is the most relevant reference, since it was developed specifically for embedded machine learning on tiny systems. For the accelerator side, LiteRT [14] is still part of the technical background because it represents the on-device runtime path surrounding TensorFlow Lite deployment. These are not “related work” in the usual algorithmic sense, but they are part of the foundation that makes the present system design possible.

The hardware side also matters more here than in many model-centric works. The i.MX RT1170 reference manual from NXP [15] is relevant because the experiments are strongly shaped by MCU-side details such as CSI capture, PXP-based image preparation, DMA behavior, and memory placement across internal RAM and SDRAM. Similarly, the MIPI CSI-2 [16] overview is relevant because camera interface behavior is not a side issue in this system; it is part of the critical path that determines whether high-rate dual-camera acquisition remains stable, an important factor in end-to-end capability assessment [17]. The littlefs [18] project, and later the FileX/LevelX [19] stack, are also relevant at system level because practical embedded experimentation often requires robust local storage for

models, scripts, and configuration data—and storage whose directory operations scale as the number of artifacts grows—especially when the platform is used repeatedly in field or lab tests.

This work also connects directly to drone-control ecosystems rather than stopping at perception output. Previous work [20] classifies two platforms for piloting drones which are open and ready for integration with mission computers, PX4 and Crazyflie. PX4 [21] and MAVLink [22] are relevant because part of the contribution here is the ability to pass onboard visual results into standard drone communication and control flows. This is important for the main claim of the paper: the device is not only performing inference, but exposing those results to the control stack. Recent vision-based autonomous-navigation methods, such as deep reinforcement learning with privileged information [23], demonstrate full closed-loop UAV autonomy, but they typically assume a capable companion computer; by contrast, the present contribution keeps a scriptable perception-to-action interface on MCU-class hardware. Crazyflie [24] documentation is relevant for a complementary reason: Crazyflie provides a lightweight indoor research platform on which perception-driven behavior can be tested quickly and repeatedly, while PX4/MAVLink provides a path toward more transferable integration with larger autopilot-based systems.

A broader systems perspective also comes from hardware acceleration work outside drones. Jouppi et al. analyzed the performance of the Tensor Processing Unit in the datacenter setting [25]. That work is obviously much larger in scale than the present platform, but it remains relevant because it highlights an issue that still appears, in a different form, here: accelerator performance is never only about the compute unit itself. Data movement, memory access patterns, and the surrounding system architecture have a major effect on usable throughput. In this embedded setting, the same idea shows up at much smaller scale in the interaction between camera DMA, image conversion, memory placement, USB transfer toward the EdgeTPU, and post-processing latency.

Finally, this work also includes a person detector trained on synthetic aerial data generated for the expected operating conditions of the onboard camera. This part of the study is not treated as a separate dataset contribution, but as a practical component of the embedded perception workflow. The synthetic data were used to match the detector to the visual regime of the platform: aerial viewpoint, small human targets, fixed training resolution, daylight variation, weather variation, and scene types relevant to low-altitude UAV operation. Prior work on synthetic aerial-view human detection [26] shows, however, that a measurable sim-to-real domain gap remains and that hybrid real–synthetic training or domain adaptation is usually needed to close it; this limitation is revisited when interpreting the detector’s behavior under real-world conditions. In this sense, the detector is not only an application example, but a way to align the model with the images that the drone is expected to observe during person-aware navigation or ground-person avoidance.

Taken together, the prior work frames the problem from several directions: compact onboard vision for small UAVs, real-time object detection and tracking, embedded runtime design, accelerator-based inference, synthetic data generation, and drone communication through standard control interfaces. What remains less explored is the combination of these elements into a single MCU-class embedded AI workflow. Prior work typically optimizes one layer in isolation—a detector for a generic benchmark, an accelerator for raw throughput, or a dataset decoupled from the deployment sensor—and treats the board as a fixed inference target; the interactions between camera geometry, the quantized model, the memory and transfer path, and the training distribution are rarely co-designed. This is the gap addressed by this work: a compact, scriptable, dual-camera MCU–EdgeTPU platform in which the board, runtime, detector, and dataset are designed around the same mission

constraints, so that real-time onboard perception is obtained by aligning the model with its expected camera distribution and exposing the resulting detections to drone-control logic.

3. Materials and Methods

The embedded vision platform is evaluated by measuring the complete onboard processing path under controlled conditions, with every reported value kept traceable to raw per-iteration measurements acquired on the device itself. The methodology is built around repeatability, stage-level attribution, and explicit control of the firmware state before each benchmark. The materials—the hardware platform, camera subsystem, runtime, and synthetic dataset—are presented first, followed by the measurement methodology used to derive the timing and throughput results.

3.1. Materials

3.1.1. Embedded Hardware Platform

The experimental platform used in this work is a compact MCU–EdgeTPU embedded board (Figure 1), developed as an extension of the Coral Dev Board Micro design. The board preserves the general lightweight embedded-AI character of the baseline platform, but extends it with a richer sensing and runtime environment better suited to autonomous drone experiments. In addition to the main compute path formed by the MCU and the EdgeTPU accelerator, the board integrates two camera connectors, onboard inertial sensing, and removable storage support. In practical terms, this turns the platform from a single-camera embedded inference board into a more flexible dual-camera system intended for perception-driven drone tasks.

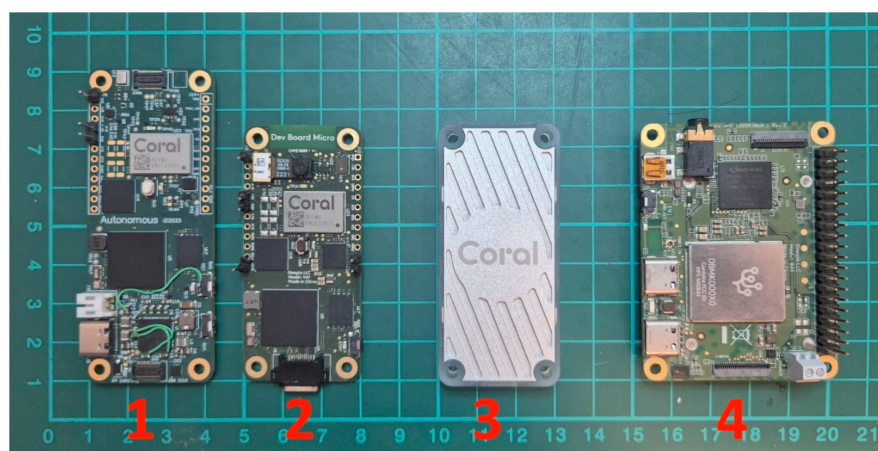


Figure 1. Visual comparison of the hardware platforms referenced in this study: (1) our board, (2) the Coral Micro baseline, (3) the Coral USB Accelerator, and (4) the Coral Dev Board (linux based).

From a form-factor perspective, the design remains closer to the Coral Dev Board Micro class than to larger Coral-based systems. This compactness was important for the present study because the platform was intended to be mounted on small aerial vehicles and used in repeated onboard experiments rather than only as a benchtop inference target. The board-level changes were therefore guided by sensing and integration needs (Figure 2) rather than by a complete redesign of the original hardware concept.

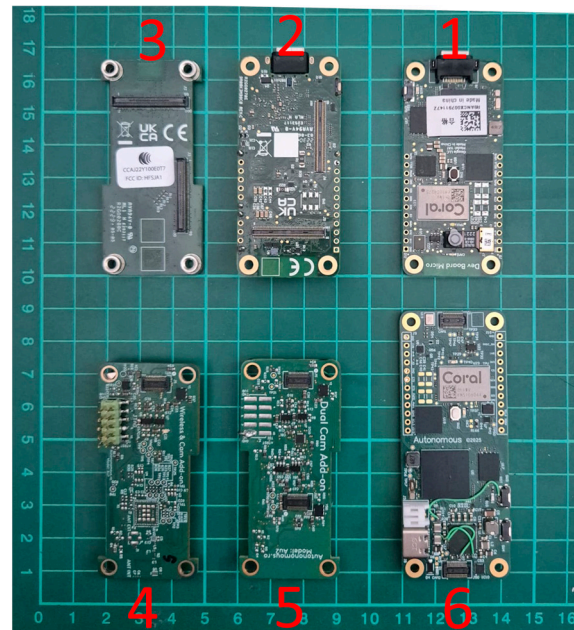


Figure 2. (1) and (2) front and back views of the Coral Dev Board Micro, which served as the starting point of the design effort. (3) The standard Coral add-on board, included here as the baseline expansion approach from which the later custom hardware departed. (4) is the first custom add-on board, designed to bring a single 5 Mpx camera onto an MIPI-connected path and to validate the higher-resolution sensing concept. (5) is the second iteration, which adds support for two MIPI cameras together with the switching logic required to alternate between them at runtime. (6) shows the final integrated board design, where the dual-camera concept is incorporated directly into a new compact board.

The fully integrated platform is light and compact enough to be carried by a small UAV. Figure 3 shows the board mounted on the airframe used for the onboard experiments reported in this work



Figure 3. The MCU-EdgeTPU platform mounted on the CrazyFlie Brushless UAV used for the onboard experiments.

3.1.2. Camera Subsystem

A central material difference between the baseline board and our board is the camera subsystem (Table 1). The baseline Coral Dev Board Micro uses the Himax HM01B0, a low-resolution sensor intended for lightweight embedded vision workloads. In the standard configuration, this camera provides a native resolution of 324×324 px, supports QVGA operation, and is connected through a low-bit-width synchronized sensor interface. By contrast, our board uses OV5640-1B camera module (Figure 4) with a configurable-resolution MIPI CSI-2 path; for the detector, frames are read as 640×480 (VGA) crops at the native pixel scale of the 1280×960 sensor geometry, matching the synthetic training tiles. The

OV5640-based configuration also brings a larger optical format and a narrower but more detailed field of view than the baseline camera.

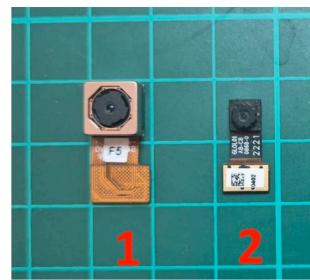


Figure 4. Our camera (1) and the Coral Micro original camera (2).

Table 1. Technical comparison between the baseline Coral Dev Board Micro camera and the OV5640-1B camera module used in our board.

Parameter	Baseline Coral Dev Board Micro Camera	Our Camera Module	Comparative Remark
Image sensor	Himax HM01B0 CMOS sensor	OV5640-1B	Higher-resolution image sensor class.
Native/nominal resolution	324 × 324 px native; QVGA (320 × 240) supported	1280 × 960 used in the described runtime configuration	Much larger source image for downstream processing.
Pixel size	3.6 μm × 3.6 μm	1.4 μm × 1.4 μm	The HM01B0 has larger pixels, whereas OV5640 targets denser imaging.
Optical format	1/11 inch	1/4 inch	Substantially larger optical format.
Field of view	110° diagonal	68.7° diagonal; 58.1° horizontal; 45.0° vertical	Coral Micro favors a wider low-resolution view, while our favors a narrower but higher-detail view.

This change is not only a matter of image resolution. It shifts the platform toward a different sensing regime. The baseline camera is well aligned with very lightweight embedded pipelines in which bandwidth and staging costs are modest. The higher-resolution OV5640 configuration used in our board provides substantially more spatial detail, but at the price of higher bandwidth demand and a greater dependence on memory placement, image preparation, and pipeline design. For the present work, this trade-off was intentional, since the target application was not minimal sensing, but onboard detection and mission adaptation under more realistic drone-view imagery.

The dual-camera arrangement (Figure 5) provides reconfigurable complementary viewpoints within a compact sensing geometry. Each camera is mounted on a flexible connector and can be oriented anywhere from nadir, observing the ground plane directly, to a forward or lateral view, so the viewpoint is chosen per mission rather than fixed by the hardware. A typical configuration keeps one camera at nadir for ground-referenced tasks—surface inspection, landing support, and ground-person detection, which is the regime the detector is trained for—while the second is tilted toward the forward scene for broader situational awareness and obstacle perception; at low altitude the second camera may approach a near-horizontal orientation. Because the mount is reconfigured manually rather than actuated at runtime, this flexibility avoids the mechanical and control complexity of a motorized sensing head; however, the in-flight orientation is not read by an encoder, so projecting detections onto the terrain is approximate unless the camera angle is supplied or calibrated.

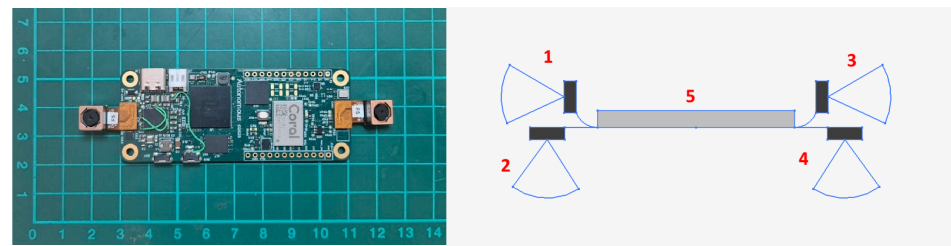


Figure 5. Left: our board positioned horizontally. Right: a profile schematic of the reconfigurable camera orientations. Each camera is mounted on a flexible connector and can be set anywhere between a downward (nadir) orientation and an outward (forward/horizontal) one: (1) and (3) indicate the outward-looking limit of the left and right cameras, (2) and (4) the downward-looking field of view of each, and (5) the central body hosting the main electronics. The figure illustrates the range of orientations the mount allows, not a single fixed geometry.

For low-altitude aerial perception, the higher-resolution camera path also has a practical consequence in terms of target visibility. Using the OV5640 field of view and the 1280×960 native pixel scale, a human target represented by an approximate 45 cm shoulder width and 22 cm head diameter remains only a few pixels wide at higher altitudes and becomes progressively more detectable as altitude decreases. This supports the motivation for using richer sensing and dedicated person-detection models when the goal is ground-person awareness or avoidance.

These pixel-scale figures and the footprint values use a simplified pinhole camera model; lens distortion is not modeled, as the modules were not intrinsically calibrated. Under this model the nadir ground footprint is $2 \cdot h \cdot \tan(\text{FOV}/2)$ per axis: at the 40 m operating altitude for which the detector is trained, one camera covers approximately 44.5×33.1 m at about 3.5 cm/px, which places a standing person at the ~ 10 –13 px scale of the training set. Tilting the second camera to a 45° oblique orientation extends coverage to a longer along-track ground strip, roughly 17–97 m ahead of the nadir point at 40 m altitude, for forward situational awareness, but at a variable and lower ground resolution and under a viewpoint outside the nadir distribution on which the detector was trained. At low altitude the second camera may instead be set near-horizontal (about 90° from nadir): in this configuration it looks forward at the vehicle's own flight level for obstacle perception—people, structures, or terrain ahead—rather than producing a ground footprint, and it is again outside the trained nadir distribution. Because the mount is reconfigurable rather than fixed, the combined coverage is a per-mission/drone monture trade-off rather than a single calibrated observation volume; we therefore characterize representative configurations rather than claim a fixed dual-camera sensing volume.

3.1.3. Camera Usage Model

The camera usage model in our board also differs from the baseline single-camera flow. In the original Coral Micro examples, camera handling is usually organized around a fixed acquisition path in which power-on, mode enable, frame acquisition, and downstream processing are performed sequentially. Our board extends this model toward explicit dual-camera initialization, runtime camera selection, hardware-assisted scaling and conversion, and a pipelined execution structure with overlapping preparation and inference stages. Camera switching is treated as a normal runtime operation rather than as a static configuration step. Per-camera configuration and geometry are also handled explicitly, which is important when the same onboard platform is used across different sensing layouts. For the detector, the onboard camera is read as 640×480 (VGA) crops at the native sensor pixel scale—the same 640×480 tiling used to build the synthetic training data—so a target projects to the same pixel size at deployment as during training (Table 2). Capturing the full

1280 × 960 frame is avoided at runtime for performance reasons. The runtime performance in Section 4.2 is measured directly at this native VGA input, in both the standard RGB format and the native YUV420 format that the inference detector can consume directly to reduce the per-frame USB transfer.

Table 2. Pixel size relative to camera FOV and altitude at the 1280 × 960 native pixel scale.

Altitude of Camera	cm/px Horizontal	cm/px Vertical	Shoulders ~45 cm in Pixels	Head ~22 cm in Pixels
100 m	8.68	11.51	5.2 px/3.9 px	2.5 px/2.5 px
80 m	6.94	9.21	6.5 px/4.9 px	3.2 px/3.2 px
60 m	5.21	6.91	8.6 px/6.5 px	4.2 px/4.2 px
40 m	3.47	4.60	13.0 px/9.8 px	6.3 px/6.3 px

This matters for the present work because the experimental platform is not used as a fixed single-view detector host. Instead, it is treated as a compact onboard sensing module that can adapt its viewpoint and inference path to the current task. The dual-camera design is therefore part of the material basis of the study, not only an implementation detail leading to integration of new image processing techniques [27–29].

3.1.4. Platform Access and Runtime Interaction

Another material aspect of the platform is the way it is accessed during experimentation. In the standard baseline workflow, interaction is organized mainly around host-side build, flashing, and reboot cycles. Our board extends this by exposing the running system through a composite USB interface that supports interactive console access, IP-based communication over the same cable, and optional storage export. In practice, this provides a more direct runtime-oriented workflow for scripting, diagnostics, and file exchange, which is useful in iterative embedded-AI experiments.

This access model is complemented by a split (Table 3) software structure (Figure 6). Low-level runtime services remain implemented in C/C++, where direct control over scheduling, memory handling, DMA, image preparation, and TPU invocation is required. On top of this, a MicroPython control layer is exposed for higher-level orchestration, mission logic, runtime configuration, and interactive experimentation. This separation is relevant as a material property of the platform because it determines how the same hardware can support both deterministic low-level behavior and rapid high-level adaptation during experiments.

Table 3. Comparison between the C++ runtime layer and the MicroPython control layer.

Aspect	C++ Runtime Layer	MicroPython Control Layer
Primary role	Low-level runtime implementation and hardware control	High-level orchestration and application logic
Typical responsibilities	Camera control, task scheduling, memory management, DMA/PXP handling, TPU invocation, USB/runtime services	Mission logic, runtime configuration, experimentation, scripting, and interactive control
Code structure	Often distributed across multiple source files, bridge layers, and subsystem APIs	Typically expressed as short scripts or direct REPL commands
Extension effort	Usually requires rebuilding or modifying lower-level code paths	New behaviors can be expressed with relatively small script changes

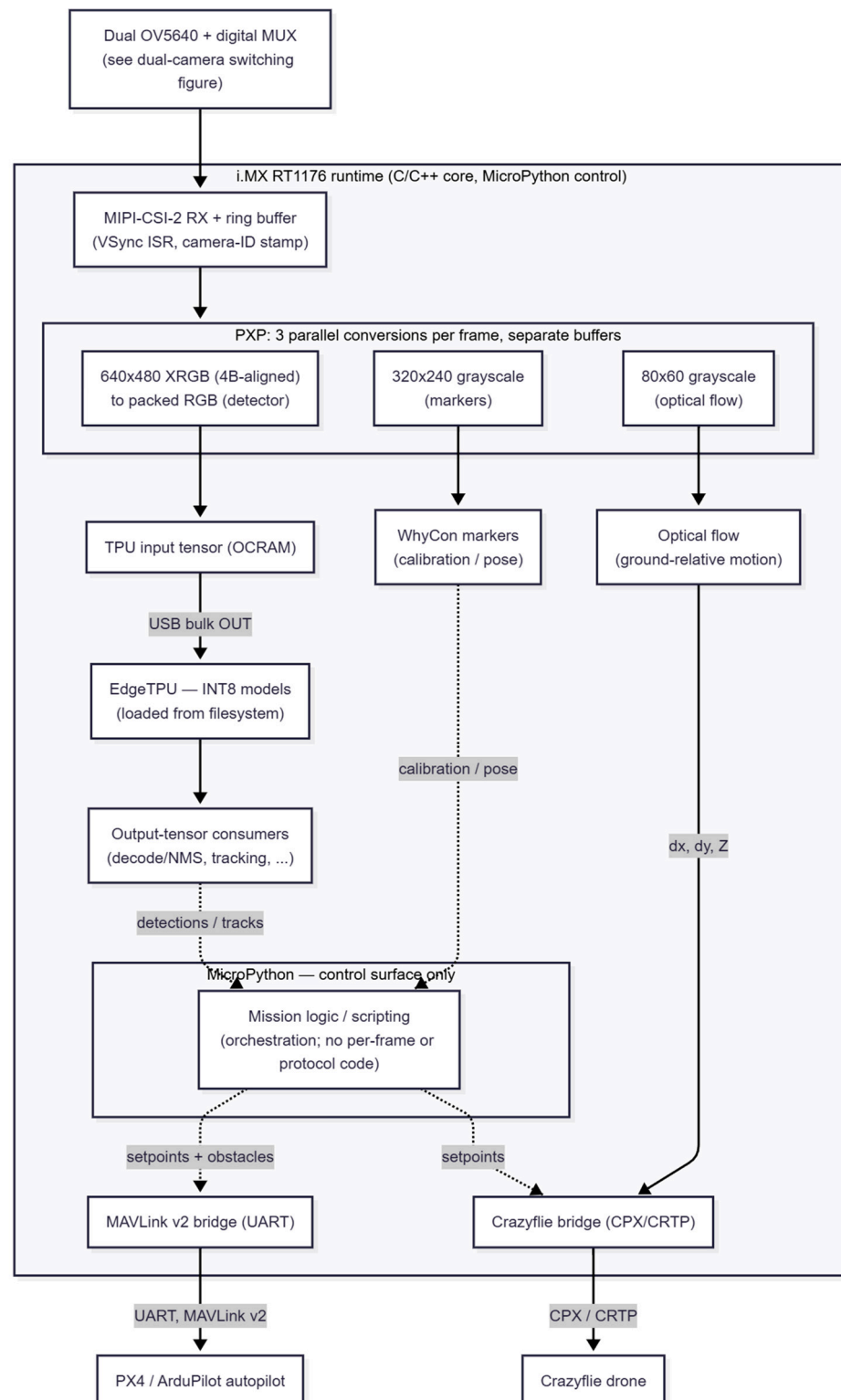


Figure 6. Onboard perception architecture. For each camera frame the PXP produces three parallel conversions into separate buffers—a 640×480 stream for the detector (4-byte-aligned XRGB/YUV capture converted to packed RGB/YUV), an 80×60 grayscale image for optical flow, and a 320×240 grayscale image for WhyCon markers—consumed independently by the output-tensor consumers (decode/NMS, tracking), the flow module, and the marker module. Every block except the MicroPython box runs in C/C++; the optical-flow module streams ground-relative motion (dx , dy) and height (Z) directly into the Crazyflie bridge for low latency, WhyCon markers are used internally for calibration/pose, and MicroPython (sentai.*) is a thin control surface that orchestrates mission logic—issuing setpoints to both autopilots and obstacle data to PX4—while holding no per-frame or protocol code.

3.1.5. Storage and Diagnostic Support

Beyond sensing and compute, we also introduce a storage and diagnostics layout intended to support repeated experimentation. In contrast to a monolithic or weakly separated flash layout, the platform uses partitioned storage with distinct system and user regions. The protected system partition holds core runtime content, while mutable artifacts such as models, MicroPython scripts, images, and logs are kept in a separate user partition. The user partition was migrated from LittleFS to FileX over LevelX [19] so that directory operations remain efficient as the number of stored session artifacts grows, while the smaller system partition remains on LittleFS [18]. This separation improves fault containment and allows experimental assets to evolve without interfering with the protected runtime environment.

The platform also includes persistent boot logging that remains available after startup through normal user-facing access paths. Compared with transient early-boot diagnostics observed only through live UART capture, this approach improves observability during repeated deployment-oriented experiments, especially when failures must be inspected after the fact rather than during active board bring-up. Although these features are not the main focus of the perception results, they are part of the material basis that makes the platform suitable for repeatable embedded drone experiments.

3.1.6. Synthetic Dataset Setup

The dataset was generated in a synthetic 3D environment built in Unreal Engine 5 [30] and controlled through the AirSim plugin [31]. Unreal Engine 5 was selected because it provides high visual quality, photorealistic rendering, and extensive customization options for creating diverse aerial scenarios. AirSim was used to simulate a virtual drone, reposition the camera in the scene, capture images, and generate segmentation outputs that supported automatic annotation of human instances.

The synthetic environment (Figure 7) included 11 scenario types: industrial area, snowy village, military unit, farm, beach area, abandoned city, university campus, military airport, modern city, mountain village, and downtown area. Human assets were selected to match the scene context and included beach people, workers, construction workers, shoppers, and urban pedestrians.

The Unreal Engine rendering and environment stack included Ultra Dynamic Sky, Ultra Dynamic Weather, Volumetric Fog, Lumen, Nanite, and Ray Tracing. These systems were used to generate visual variation in time of day, season, weather, illumination, atmosphere, and visibility. Weather conditions were randomly changed from image to image using probabilities adapted to the climate of each scenario.

The weather conditions used in the dataset were Clear Skies, Partly Cloudy, Cloudy, Overcast, Foggy, Rain Light, Rain, Rain Thunderstorm, Snow Light, Snow, Snow Blizzard, Sand Dust Calm, and Sand Dust Storm. The simulated time of day varied randomly between 06:30, corresponding to sunrise, and 17:50, corresponding to sunset.



Figure 7. High-fidelity synthetic worlds used in our dataset generation process together with human characters in different postures.

3.2. Methods

3.2.1. Experimental Design and Measurement Philosophy

The experiments were designed to characterize the system as an end-to-end embedded pipeline rather than as an isolated accelerator benchmark. For this reason, measurements were collected not only for model invocation, but also for the main stages that precede and follow inference: camera selection, frame capture, image conversion and resizing, tensor preparation, transfer to the EdgeTPU, and post-inference processing. This stage-level decomposition was necessary because early tests showed that the limiting factors were often outside the neural network itself and instead came from memory placement, peripheral interaction, or scheduling under concurrent load. The campaign was organized into two complementary tracks according to what each is the right tool to validate. Hardware-board experiments validated the properties that depend on the physical platform—EdgeTPU throughput, the camera-to-detection pipeline timing, dual-camera switching, and the memory, DMA, and USB behavior reported in Section Memory Mapping and Allocation Improvements and Section 4.2—because these cannot be reproduced faithfully off the target. Simulation experiments, using a Crazyflie (2.1) or PX4 (v1.17.0) software-in-the-loop model in Gazebo [32], validated the algorithms that consume the detections—marker-based calibration and closed-loop control, and optical-flow-driven position holding—where the quantity of interest is algorithmic correctness rather than hardware timing. The simulation was kept air-gapped from simulator ground truth: the onboard perception consumed only camera frames and the standard telemetry link and was never given injected world state, so a successful run reflects the algorithm rather than privileged information.

Three principles were followed throughout the experimental campaign. First, all measurements were acquired on-device, so that host-side transfer and collection did not affect the timing values. Second, each experiment was stored as a self-contained session with raw per-iteration data and a matching parameter description. Third, all timing loops were bounded by explicit repetition counts and timeout conditions in order to avoid

unbounded blocking and to make failed runs visible rather than silently absorbed into the measurement process.

Concretely, each experiment is a self-contained folder on the device (Figure 8), with one sub-folder per iteration holding that run's journal and a raw metric summary. Per-iteration counters are accumulated in RAM during the run and flushed to the FileX/LevelX user partition at the end, after which the session is downloaded to the host for analysis; the device, not the host, is treated as ground truth. A hardware and software watchdog recovers the board if a run crashes or hangs, and a host-side UART serial monitor follows progress without affecting the measured timing. All reported statistics are then computed off-device from the raw per-iteration data, so every value remains traceable to its raw source.

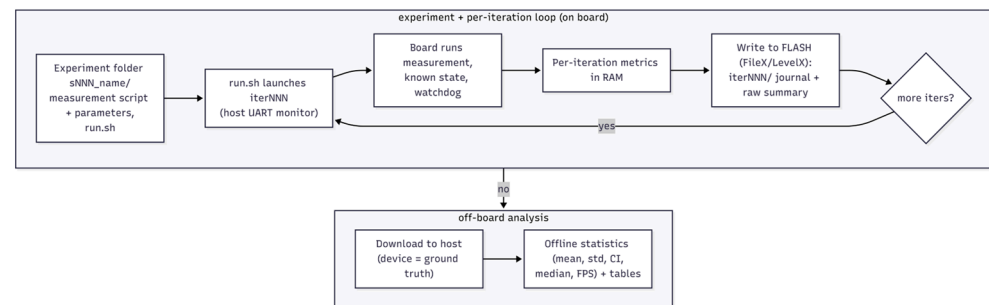


Figure 8. Per-experiment performance-measurement workflow. Each experiment is a self-contained folder (measurement script, parameters, per-iteration sub-folders). The host launches each iteration on the board and follows progress over a UART serial monitor; the board runs the measurement from a known firmware state, guarded by a watchdog that recovers it on a crash, accumulating per-iteration counters in RAM and flushing them to FLASH (FileX/LevelX) at the end of the iteration as a journal plus a raw summary. The session is then downloaded to the host (the device is authoritative), and statistics (mean, standard deviation, confidence interval, median, FPS) are computed off-device into the reported tables. Mission-success experiments add a separate verdict gate against pre-declared pass criteria, which is outside the scope of the timing results reported here.

3.2.2. Firmware State Control Before Measurement

Before each timing loop, the firmware was brought into a known and reproducible state. Verbose per-frame logging was disabled, automatic camera alternation was turned off unless it was the object of the experiment, any previously running pipeline was stopped, and garbage collection was forced before entering the hot loop. This preparation reduced hidden sources of jitter such as console backpressure, asynchronous ownership conflicts, or garbage-collection pauses during measurement.

This control step was especially important because the system under test was not a single function call, but a shared embedded platform in which the camera subsystem, preprocessing functions, USB transfer path, inference engine, and post-processing routines could interfere with one another. During diagnostic work, we also observed failure modes in which unstable low-level configurations could persist across tests. In those cases, fresh firmware reflashing was treated as part of the control procedure to prevent contamination between successive experiments.

3.2.3. Warm-Up, Repetition Counts, and Timing Acquisition

Each experiment began with a warm-up phase that exercised the full path at least once before timed sampling began. The first measured sample was then discarded. This was done to reduce the influence of one-time initialization effects such as CSI startup behavior, camera settling, initial buffer allocation, or first-use accelerator overhead.

After warm-up, each experiment was repeated a fixed number of times depending on the goal of the run. Sustained pipeline benchmarks were typically repeated twenty

times, alternating camera-switch studies around forty times, and visual capture studies as many times as allowed by the storage and memory budget of the platform. Each repetition corresponded to a full independent iteration and was recorded as a separate row in the output CSV.

Stage timings were measured around specific calls using the millisecond RTOS tick counter exposed to the runtime. The EdgeTPU invocation duration was returned by the firmware itself in order to avoid inflating that number through scripting overhead. For finer attribution within the invoke path, including transfer-side effects, a separate cycle-counter-based microbenchmark was also used during diagnostic optimization. These microbenchmarks were not the main reported result, but they were important for explaining why certain changes improved performance.

3.2.4. Scene Control and Visual Verification

Camera-dependent experiments were performed on a static scene so that timing measurements remained comparable across runs. For sessions involving image capture or camera switching, visual snapshots were recorded before and after the benchmark (Figure 9). These images were used to verify that the scene had not drifted, that the correct camera had been selected, and that switching artifacts or misalignment were not being misinterpreted as timing-related effects.

In experiments aimed at checking visual quality rather than throughput alone, such as seam validation during switching, image data were buffered in memory during the timed portion of the run and written to the filesystem only afterward. This avoided contaminating the timing loop with flash write latency.

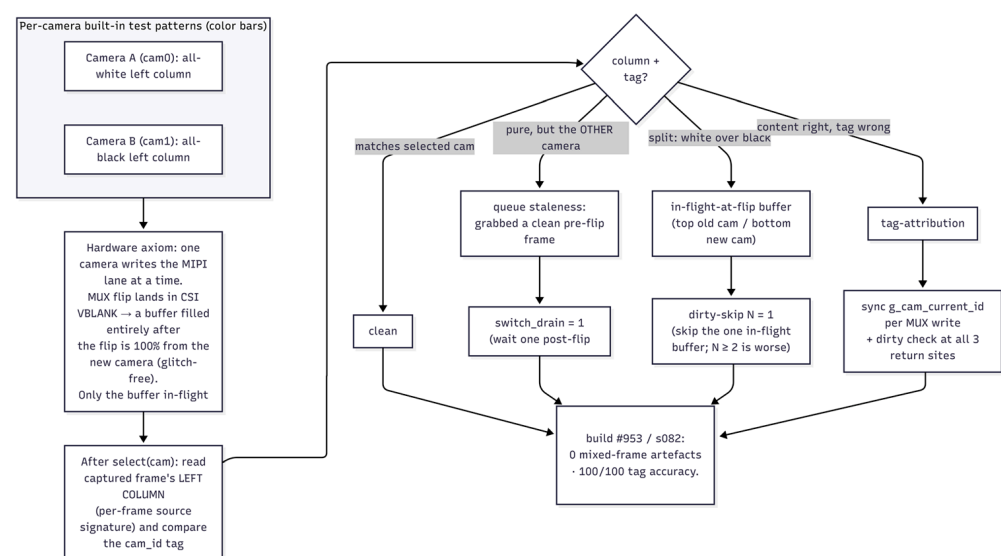


Figure 9. Visual verification of dual-camera switching. Each camera runs its own built-in color-bar test pattern whose leftmost pixel column is uniformly white (camera A) or black (camera B), giving every captured buffer a one-column signature of its source sensor that is cross-checked against the firmware camera-id tag. Because the analog MUX flip is taken inside CSI VBLANK, a buffer filled entirely after a flip is 100% from the new camera; the only physically mixed buffer is the single one in flight at the flip instant (a vertically split column). Reading the column plus the tag distinguishes four cases—clean, queue staleness (a clean but pre-flip frame), the in-flight-at-flip mix, and a mis-attributed tag—each closed by a targeted fix: one post-flip drain frame ($\text{switch_drain} = 1$), a single-buffer dirty-skip ($N = 1$), and synchronizing the camera-id with every MUX write. Across the captured frames zero mixed-frame artifacts were observed and tag accuracy reached 100%, confirming that a single shared MIPI lane delivers each camera's frame intact and correctly labeled.

3.2.5. Statistical Treatment of Results

All reported tables were derived offline from the raw per-iteration CSV files rather than from on-device summaries. For each experiment, we computed the mean, sample standard deviation, minimum and maximum values, and—for the headline rates—the median, a 95% confidence interval of the mean, and the invoke-time percentiles used to characterize jitter. Median values were also considered when the distribution showed alternating or asymmetric behavior, such as in repeated two-camera switching patterns.

Frame rate was derived from the mean total frame time as frames per second. For alternating camera runs, the data were also aggregated by camera and by switching direction to expose any systematic asymmetry between paths. This was useful in the dual-camera experiments, where the average performance alone could hide a directional penalty or residual switching imbalance.

3.2.6. Exploratory and Diagnostic Experiments

The experimental campaign was not limited to final benchmark runs. A substantial part of the work consisted of exploratory and diagnostic measurements whose purpose was to isolate bottlenecks and test architectural hypotheses. These experiments included single-variable parameter sweeps, staged isolation tests in which parts of the pipeline were selectively enabled or disabled, memory-layout comparisons across OCRAM and SDRAM, and control-flow variants implemented under different RTOS ownership models.

Keeping these exploratory runs as part of the methodology is important because the final system behavior was reached iteratively rather than by a single design choice. Several negative findings directly constrained the design space, and we report the most instructive ones explicitly. Placing the entire TensorFlow Lite tensor arena in OCRAM consistently produced a hard fault at tensor allocation, so only selected hot buffers—not the full arena—could be relocated to fast memory. Relocating the output tensor into OCRAM by pointer swap broke the EdgeTPU custom-operator invariants and collapsed the pipeline from about 41 to 0.2 frames per second, whereas the analogous input-tensor relocation was safe; only the input staging was therefore kept in fast memory.

Pushing the sensor beyond VGA (a 90 fps configuration) saturated the SDRAM write path and starved the USB transfer of EdgeTPU instructions, collapsing the pipeline to roughly 8 frames per second with frequent invoke errors, which fixed the operating point at VGA. Finally, a descriptor-caching path intended to skip re-uploading the model instructions did not remain reliable across repeated invocations and was abandoned in favor of re-uploading the instruction stream on each invoke. Retaining these negative results helped distinguish genuine architectural improvements from incidental changes and made the final optimization path more defensible.

3.2.7. Synthetic Dataset Generation and Curation Methodology

The dataset was produced through an automated capture and annotation pipeline. For each capture batch, a script initialized the virtual scene, selected the capture parameters, and stored the generated outputs in a structured local folder. Each run produced the RGB images, segmentation masks, debug images, YOLO bounding-box labels, polygon labels, and a log file containing per-frame metadata.

For every frame, the script first assigned segmentation identifiers to the relevant human meshes. The background was kept as a separate segmentation class, while each human instance was isolated through the simulator segmentation output. The resulting masks were then processed automatically to extract the visible human regions. From these regions, the pipeline generated two types of annotations: rectangular bounding boxes for object detection and polygon labels for segmentation-based experiments.

The initial exported dataset was then converted into a YOLO-ready structure. Since the raw captures were stored across multiple scenario-specific folders, the first curation step flattened the dataset into a common image-and-label layout. During this process, metadata such as the scenario, batch number, image index, weather condition, bounding-box count, and polygon count were encoded directly in the filename. This made the curated samples traceable to their original capture conditions without requiring access to the original folder hierarchy.

Because the target training resolution was 640×480 pixels, each original 1280×960 image was split into four non-overlapping quadrants. This approach preserved the original pixel scale of the human targets, unlike resizing, which would have made the already small objects even smaller. Bounding boxes and polygons were transferred to a quadrant only if they were fully contained inside it. Annotations crossing quadrant boundaries were discarded rather than clipped, in order to avoid introducing partial human instances into the training set.

After tiling, a minimum-size filter was applied. Bounding boxes with width below 5 pixels or height below 5 pixels were removed, as such objects provide weak localization signal and are close to the visual resolution limit of the model input. Patches that became empty after this filtering step were also discarded.

A final robust filtering step was used to remove annotation artifacts from the upper tail of the size distribution. Since the expected apparent size of humans at the chosen aerial viewpoint is limited, unusually large boxes were considered physically implausible. Width and height thresholds were computed using the Median Absolute Deviation criterion, and an additional 5:1 aspect-ratio limit was applied to eliminate elongated geometric artifacts.

The complete curation process transformed the raw dataset into a final training set of 34,040 images at 640×480 resolution, containing 534,106 validated bounding-box annotations. The final labels are therefore restricted to complete, visible, and physically plausible human instances at the target aerial scale.

Dataset split and leakage avoidance. The final 34,040-patch set was partitioned into training, validation, and test subsets at an approximately 80/10/10 ratio, stratified across all eleven scene categories (Table 4). To prevent data leakage between visually similar frames, the split was grouped at the level of the capture batch rather than the individual patch: each 150-frame capture run—and therefore all four quadrants of every source frame and all temporally correlated frames of the run—was assigned to exactly one subset. The grouping was verified to share zero source frames and zero capture batches between any two subsets (9593 distinct source frames), avoiding the optimistic bias that a random patch-level split would introduce, where near-duplicate quadrants or adjacent frames of the same run could otherwise appear in both training and evaluation. Weather was not used as an explicit grouping key, because the small number of capture runs per category (five to six) makes batch-level weather stratification infeasible; nonetheless, weather coverage is near-complete across the subsets, with only the two rarest classes absent from a single split.

Table 4. Train/validation/test split composition (leakage-free, batch-grouped).

Split	Patches	Boxes	Boxes/Patch	Scene Categories	Capture Batches
Train	22,562	353,875	15.7	11/11	43
Validation	5855	82,823	14.2	11/11	11
Test	5623	97,408	17.3	11/11	11
Total	34,040	534,106	—	11	65

The per-axis bounding-box size distributions are almost identical across the three subsets (median 11 px width and 12 px height in each, 95th percentile at 20 px, and no box

above 26×27 px after the outlier cap), confirming that the batch-grouped split does not introduce a scale shift between training and evaluation. The dataset is itself small-object-heavy: every box is below 32 px, 94% are below 16 px, and a quarter are below 8 px on the shorter side. The curation that produced this set is summarized in Table 5: a minimum-size floor removes sub-5-pixel boxes, and a Median-Absolute-Deviation outlier filter removes physically implausible boxes (width above 26 px or height above 28 px), leaving 534,106 of the 627,313 source boxes.

Table 5. Dataset curation stages and box retention. Two filters remove unusable annotations after the 640×480 quadrant split: a minimum-size floor and a Median-Absolute-Deviation (MAD) outlier cap; 85.1% of the source boxes survive into the final trained set.

Stage	Rule	Patches	Boxes
Source render (1280×960)	—	9608	627,313
640×480 quadrant split	—	38,432	606,926
Minimum-size floor	shorter side ≥ 5 px	34,170	540,029
MAD outlier filter	width ≤ 26 px and height ≤ 28 px	34,040	534,106

4. Results

4.1. Software Level Achievements

A substantial part of the work involved low-level software and architecture changes required to support dual-camera sensing, higher-resolution image paths, stable EdgeTPU execution, and repeatable embedded experiments under sustained load. These changes were driven mainly by practical bottlenecks observed during development, especially in memory placement, camera handling, USB transfer behavior, storage organization, and failure recovery. The main software design improvements introduced in our board are summarized below.

Taken together, these changes (Table 6) moved the platform away from a simpler single-camera embedded inference workflow and toward a more explicit and robust runtime architecture. In practice, they made it possible to sustain a richer camera pipeline, to switch safely between two sensors, to reduce contention in the camera-to-TPU path, and to improve observability and recovery during repeated experiments. The resulting software stack is therefore not only an extension of the original baseline, but a more suitable foundation for the perception-driven drone workflows studied in this paper.

Table 6. Main improvements in our board.

Area	Improvement Introduced in Our Work	Why It Matters
Memory architecture	Explicit separation between camera-related non-cacheable memory, hot internal memory, and general-purpose external memory	Reduced cache-coherency issues and removed major contention points between camera DMA, preprocessing, and TPU-related traffic
Linker-controlled placement	Fine-grained placement of critical buffers and selected runtime components in OCRAM instead of relying on default placement	Made memory decisions explicit and repeatable, and helped stabilize the camera-to-TPU pipeline under load
TPU input staging	Hot TPU input data moved away from SDRAM contention paths	Reduced interference between USB transfers and camera/PXP activity, which was one of the key bottlenecks found during optimization
Dual-camera support	Added software support for two MIPI cameras with a shared switching model	Enabled the platform to move beyond a fixed single-camera workflow toward multi-view sensing

Table 6. Cont.

Area	Improvement Introduced in Our Work	Why It Matters
Safe camera switching	Camera switching performed only at safe frame boundaries rather than during active acquisition	Prevented mixed or torn frames and made repeated camera alternation usable in practice
Frame-sequence handling	Corrected frame-sequence semantics so that software-visible frame cadence matched real sensor frames	Avoided timing and validation errors during switching, draining, and alternation experiments
Camera processing path	Extended the camera stack with hardware-assisted conversion and scaling	Reduced MCU-side preprocessing cost and made the higher-resolution sensing path more practical
EdgeTPU transfer path	Reworked parts of the USB/TPU software path to reduce overhead and improve timeout recovery	Improved sustained throughput and made TPU execution more robust during concurrent camera activity
Async USB handling	Improved in-flight transfer management and callback routing	Reduced failure cascades under concurrent submissions and made recovery from stalled transfers cleaner
Bounded failure behavior	Added bounded polling and stronger fail-fast handling in low-level runtime paths	Prevented indefinite hangs and improved debuggability during unstable or unexpected TPU states
Storage separation	Introduced clearer separation between protected system content and mutable user-managed content	Improved fault containment and made repeated experimentation with models, scripts, logs, and artifacts safer
Boot diagnostics	Added persistent boot logging and improved post-failure visibility	Made the platform easier to diagnose during repeated tests without depending only on transient early boot output
Anti-brick behavior	Added guarded boot-mode transitions and persistent state across warm resets	Reduced the risk of repeated startup failures leaving the platform inaccessible
Runtime robustness	Strengthened assert, stack-check, and crash-reporting behavior	Improved detection of runtime faults and made failures easier to trace during optimization work
Console and USB resilience	Improved console behavior and USB interaction during heavy testing	Made interactive control and diagnostics more reliable during long experiment sessions

Memory Mapping and Allocation Improvements

To make the linker and placement decisions easier to follow, Table 7 first summarizes the memory regions used by the runtime. The point of this table is not only to document the address map, but to show how the available on-chip and external memory resources were separated according to function. In the final design, tightly coupled internal memory, OCRAM, SDRAM, shared-memory space, and the dedicated non-cacheable camera region were no longer treated as generic storage pools. Instead, each region was assigned a clearer role in the runtime, reflecting the practical constraints identified during the camera-to-TPU optimization work.

Table 7. Memory allocation of the board.

Region	Start/End	Notes
m_interrupts	0x00000800, 0x00000400	Vector table and startup region in internal RAM.
m_text	0x00000c00, 0x0003F400	Main executable text/rodata region in tightly coupled on-chip memory.
m_ncache	0x20000000, 0x00008000	Non-cacheable data region for DMA-sensitive data.
m_data	0x20008000, 0x00038000	Main initialized data, BSS, and stack region.

Table 7. Cont.

Region	Start/End	Notes
m_ocram	0x20240000, 0x00080000	Large on-chip RAM used for selected libraries and runtime code.
rpmsg_sh_mem	0x202C0000	Shared-memory window for multicore/RPMsg use when enabled.
m_heap	0x80000000, 0x01000000	SDRAM heap reservation; default is 16 MB.
m_sdram	0x81000000, 0x01000000	SDRAM for large code/data objects after reserving the heap.
m_ncamera	0x82000000, 0x01000000	Dedicated non-cacheable camera buffer region.

Once the memory regions are defined, the next question is how the main software components are actually placed across them. Table 8 summarizes the main placement differences between the original repository and our runtime, together with the reason for each change. These decisions were driven mainly by contention and latency considerations: OCRAM was reserved for hot-path structures and code that directly affect the camera–USB–TPU pipeline, while larger or less time-critical components were moved to SDRAM. The table therefore captures the concrete linker-level changes that turned memory layout into an explicit performance and stability mechanism.

Table 8. Allocation of memory for compute intensive tasks.

Aspect	Original Repository	Our Runtime	Why It Changed
TensorFlow placement	.tensorflow placed in m_ocram	.tensorflow placed in m_sdram	Frees OCRAM for TPU buffers and USB host hot code.
TPU input tensor buffers	Not separated	Dedicated .tpu_input section in m_ocram	OCRAM is DMA-reachable by EHCI and avoids SEMC contention that affects SDRAM under camera/PXP load.
USB host EHCI hot path	Normal placement	Explicit .usb_host section in m_ocram	Keeps ISR/host-task code fetch off SDRAM to avoid IOC handling delays under SDRAM write saturation.
MicroPython runtime	Not present	Dedicated .micropython section in m_sdram	Interpreter is large; moving it out of OCRAM frees space for TPU buffers and USB host code.
libm math functions	Not separated	Dedicated .libm section in m_sdram	Same space trade-off: not on the per-frame hot path.
Audio + MP3 encoder	Not separated	.audio and .shine moved to m_sdram	Not on the TPU hot path; frees OCRAM for latency-sensitive subsystems.
Embedded ML/on-device learning	Not present	.aifes moved to m_sdram	Capacity trade-off; keeps OCRAM available for the pipeline hot path.
Telemetry/“slow” services	Not present	.sentai_slow moved to m_sdram	Groups non-hot modules away from OCRAM.
USB networking + camera service code	Not present	.cdc_ncm and .camera moved to m_sdram	Tolerates SDRAM latency to reclaim OCRAM for the pipeline hot path.
LWIP-related services	Standard LWIP in OCRAM	.lwip moved to m_sdram (includes HTTP server pieces)	Network stack is not part of the per-inference critical path.
Main .bss policy	Generic .bss stays in m_data	Large BSS for MicroPython/audio/httpd/BLE redirected into .sdram_bss	Preserves scarce on-chip m_data for true low-latency state.
Camera DMA buffers	No dedicated SDRAM camera region	.ncamera places NonCacheableCamera in m_ncamera	Avoids cache maintenance and makes camera buffers a first-class memory partition.
Boot-mode persistence	Not present	.noinit_boot_persist in m_data (NOLOAD)	Keeps boot mode/crash-loop guard across warm resets.
Heap strategy	Standard SDRAM heap reservation	Same SDRAM heap reservation	Heap is still 16 MiB by default; the change is making more state static and explicitly placed.

Beyond individual section moves, the runtime also reflects a broader architectural shift relative to the original software baseline. Table 9 summarizes this change at system level by comparing the assumptions behind the original repository with those of our runtime. The original software stack was well-suited to a compact single-purpose inference demonstration, whereas our target was a long-lived embedded runtime that had to support scripting, telemetry, camera switching, tracking, and auxiliary services without destabilizing the main vision pipeline. For that reason, the table should be read as a summary of the design philosophy behind the memory and placement changes described above.

Table 9. High-level decisions of memory usage.

Design Goal	Original Repository	Our Runtime
Baseline assumption	Single-purpose EdgeTPU demo application	Long-lived embedded runtime with scripting, telemetry, audio, camera switching, tracking, and auxiliary ML modules
OCRAM usage	Primarily “whatever fits” (512 KiB)	Treated as a scarce hot-path pool (~1 MiB minus RPMMSG): TPU input buffers and USB host hot code
SDRAM usage	Heap plus large allocations in a single cached region	Heap + most large code/modules + bulky BSS moved into SDRAM, keeping on-chip RAM small/clean
Camera buffering	No dedicated SDRAM camera partition	Dedicated non-cacheable camera SDRAM region (m_ncamera)
BSS philosophy	Mostly default placement	Explicit .sdram_bss policy to protect internal m_data
Overall effect	Suitable for a compact inference demo	Suitable for a multi-service runtime that must keep camera, PXP, USB, and TPU stable under load

A single finding drove this layout more than any other. As long as the region where the camera writes incoming frames and the EdgeTPU input-tensor arena from which the USB host reads shared the same cached SDRAM path, the camera DMA, the PXP color conversion, and the USB transfer to the accelerator all contended for the same SEMC memory controller, and the pipeline could not sustain a usable frame rate with camera acquisition, preprocessing, and inference running in parallel. The pipeline became stable only after these two roles were physically separated: the camera frames were placed in a dedicated non-cacheable region (m_ncamera, Table 7) and the EdgeTPU input tensor in a separate OCRM arena (the .tpu_input section, Table 8), with the PXP moving and converting each frame from the camera region into the input arena rather than the three tasks sharing one buffer. This separation—a camera ring buffer, PXP staging between buffers, and a DMA-reachable input arena that the USB host can read without crossing the camera’s memory path—is what allowed the acquisition, preprocessing, and inference stages to overlap in time instead of serializing, which is the concurrent behavior reported in Sections 4.2 and 5.

The final configuration is the result of an extended empirical optimization process rather than a purely theoretical design decision. After more than one hundred high-load experiments, including repeated board hangs, crashes, and unstable transfer conditions, the software and memory architecture converged toward a layout tailored to the actual behavior of the platform under stress. The resulting solution should therefore be seen as a runtime architecture shaped by measured bottlenecks and validated through repeated failure-driven iteration.

4.2. Runtime Performance Characterization

This section characterizes the runtime performance of the final embedded vision configuration on the physical board. The measurements cover the standalone EdgeTPU invocation path, the complete camera-to-detection pipeline, multi-patch inference, and dual-camera operation. The purpose is to report the sustained behavior available to the onboard runtime, rather than isolated accelerator-only estimates. The performance is characterized directly on the deployed detectors at their native VGA (640×480) input, measured on firmware build 1590; earlier board figures obtained through a 512×512 proxy resolution have been superseded by these native-resolution measurements. The per-architecture on-board numbers reported below are therefore the authoritative deployment values and are consistent with the detector study of Section 4.4.

All timing values in this section were measured on the embedded target using elapsed wall-clock time from the runtime microsecond timer. Throughput values were computed from complete operation intervals: a standalone invoke includes the USB input transfer, instruction transfer, output readback, and completion event; a pipeline output includes acquisition, preprocessing, EdgeTPU invocation, and publication of the detection result. Repeated pipeline measurements were taken after fresh boots to avoid residual state from previous runs. Visual camera-switch checks were used only to validate frame identity and were not included in timing loops when image serialization could perturb performance.

The runtime measurements were taken under the following fixed conditions. The camera ran at VGA (640×480) at 45 fps for all experiments, delivering frames in 4-byte-aligned XRGB that the PXP hardware converted to the detector's input format (packed RGB, or native YUV420 for the input-format co-design of Section 4.2.3). The sensor ran faster than the pipeline, so each invoke consumed the most recent available frame and stale frames were dropped. One patch per frame was used for the single- and dual-camera pipelines; the multi-patch experiments used two to four patches per frame as indicated. No recording to on-board media occurred inside a timed loop: timing counters were held in RAM and written to the filesystem only after the loop completed, and visual-verification snapshots were likewise buffered in RAM and flushed afterwards. The autopilot bridges (Section 4.3) were inactive during the performance measurements, so link traffic could not perturb the timing and the system under test was the perception pipeline in isolation. The active RTOS tasks during a timed run were the camera task, the preprocessing task, and the inference/detection task, together with the MicroPython control task that launched the run; per-frame verbose logging was disabled and automatic camera alternation was turned off unless it was itself the object of the experiment (Section 3.2.2).

4.2.1. Measurement Summary

Table 10 summarizes the sustained on-board performance of the deployed detector. Because the EdgeTPU invoke is dominated by the USB transfer of the input image (Section 4.2.2), the most effective throughput lever is the input format. With standard RGB input the accuracy champion (c2f_thick) sustains about 26 detections/s end to end; feeding the camera's native YUV420 directly (Section 4.2.3) halves the input transfer and raises this to about 40 detections/s at the same accuracy. Multi-patch configurations preserve the requested number of invokes per frame, and continuous dual-camera alternation at a 1:1 ratio reaches 22.5 combined detections/s (11.3 per camera).

Table 10. Measured runtime performance summary.

Metric	Measured Value	Measurement Method
Single-camera pipeline, RGB input (champion c2f_thick)	~26.2 detections/s	full camera-to-TPU pipeline, 3 × 6 s trials
Single-camera pipeline, YUV420 input (champion c2f_thick)	~40.5 detections/s	full camera-to-TPU pipeline, 3 × 6 s trials
Isolated EdgeTPU invoke, RGB/YUV420 (c2f_thick)	35.0 ms/21.9 ms	30 microsecond-timed invokes
Pipeline, 2 patches/frame (RGB/YUV420)	13.28/20.27 frame/s; 26.47/40.36 invokes/s	multi-patch wall-clock run
Pipeline, 4 patches/frame (RGB/YUV420)	6.66/10.41 frame/s; 26.80/41.65 invokes/s	multi-patch wall-clock run
Manual camera switch, select() -> to_tensor(), drain=1	~75 ms	post-switch tensor acquisition timing
Continuous 1:1 camera alternation, drain=1	22.5 detections/s combined, 11.3 per camera (RGB and YUV420 alike)	alternating pipeline with content validation
Metric	Measured value	Measurement method

These measurements show that the effective performance is determined by the complete onboard data path: camera capture, memory movement, preprocessing, USB transfer to the EdgeTPU, result readback, and task scheduling. The EdgeTPU invoke itself is bounded by the input-tensor transfer over USB 2.0, so the pipeline is invoke-bound rather than camera-bound, and reducing the transferred input volume is what raises throughput.

4.2.2. EdgeTPU Invoke and the Input-Transfer Bottleneck

The isolated invoke loop measures the USB-to-EdgeTPU path without camera acquisition or preprocessing: each invoke sends the input tensor, transfers the model instruction stream, reads the output tensor, and waits for the completion event. On the deployed 640×480 detector the invoke is dominated by the input-tensor transfer: for the champion c2f_thick the RGB input tensor ($640 \times 480 \times 3 = 921,600$ B) takes about 26 ms—roughly three-quarters of the ≈ 35 ms invoke—while the instruction stream, parameters, and output readback account for the remainder (Table 11). EdgeTPU compute is overlapped with the transfer (it begins as the instruction stream arrives and finishes by the output read), so it does not add to the invoke time. The invoke is therefore input-transfer-bound over USB 2.0, confirmed on real hardware, and the pipeline that wraps it is invoke-bound rather than camera-bound.

Table 11. On-board invoke breakdown for the champion detector (c2f_thick, 640×480 , RGB input).

Component	Wall Time (Approx.)
USB input tensor ($640 \times 480 \times 3$, 921,600 B)	~26 ms ($\approx 75\%$)
Instructions, parameters, and output readback	~9 ms
Invoke total (RGB)	~35 ms

Because the input transfer dominates, the single most effective throughput lever is the input format, not the model or the accelerator. This directly motivates the YUV420 input-format co-design of Section 4.2.3: transferring the camera's native YUV420 planes instead of host-decoded RGB halves the input tensor (460,800 B), cutting the input transfer to about 13 ms and the invoke to about 22 ms.

4.2.3. Single-Camera Pipeline—RGB and YUV420 Input

The single-camera pipeline measurement includes the complete runtime path from camera acquisition to detection publication. Two input formats are compared on the same detectors and the same board: standard packed RGB, and a native YUV420 dual-input co-designed for the transfer bottleneck identified in Section 4.2.2.

The camera’s native chroma is perceptually redundant for a single-class person detector, so instead of host-decoding to RGB the platform feeds the accelerator two planes directly—full-resolution luma Y ($640 \times 480 \times 1$) and half-resolution chroma UV ($320 \times 240 \times 2$)—a 460,800-byte input, half the RGB volume. A two-stem front (a space-to-depth stem on the luma plane and a stride-1 convolution on the chroma plane) fuses the planes early; the rest of the backbone is unchanged, the extra stem adds only about 6 KB, and the model still compiles to a single on-chip EdgeTPU subgraph with no CPU-offloaded operators.

On-board, this format change gives a consistent $1.53\text{--}1.58\times$ throughput gain on every architecture (Table 12): the champion c2f_thick rises from 26.2 to 40.5 detections/s (invoke $35.0 \rightarrow 21.9$ ms), and the whole nano family moves from about 26 to about 40 detections/s. Because the model graph barely changes the invoke, the gain is almost entirely the halved input transfer—the on-board confirmation that the pipeline is input-transfer-bound over USB 2.0. The rates are deterministic to within about ± 0.1 detections/s across repeated trials, with the isolated invoke stable to within ± 0.5 ms and no timeouts or inference failures.

Table 12. On-board pipeline throughput, RGB versus YUV420 input, per architecture (build 1590, 640×480 @ 45 fps). ΔmAP is the on-board float mAP@0.5 change from RGB to YUV420 with the same weights.

Architecture	$\Delta\text{mAP}@0.5$	RGB Pipeline (det/s)	YUV420 Pipeline (det/s)	Speedup
msblock	−0.123	27.2	42.8	$1.57\times$
c3	+0.017	27.0	42.6	$1.58\times$
c2f	+0.018	26.9	42.3	$1.57\times$
c2f_pan2	+0.000	26.4	41.0	$1.56\times$
gelan	+0.044	26.4	40.6	$1.54\times$
c2f_thick	+0.020	26.2	40.5	$1.55\times$
gelan_pan2	+0.051	25.4	38.8	$1.53\times$
c2f_deep	+0.031	unstable	38.9	—

Crucially, the speedup comes at no accuracy cost. Across an eight-architecture INT8 sweep on the synthetic split (Table 13) the YUV420 models are accuracy-neutral versus RGB (mean $\Delta = +0.013$ mAP50, six of eight at or above the RGB baseline), and on the on-board float evaluation accuracy is preserved or improved on seven of eight architectures (Table 10). The one exception is msblock, the smallest backbone (about 69 K parameters), which has too little capacity to learn the packed two-plane input and is not a YUV420 candidate. The accuracy champion c2f_thick actually improves slightly under YUV420 (INT8 $0.8296 \rightarrow 0.8437$ on the synthetic split; on-board float $0.803 \rightarrow 0.823$), so c2f_thick with YUV420 input is the recommended production detector: it is the most accurate model in the set and runs at 40.5 detections/s, a 55% throughput gain over the RGB champion. As a further deployment benefit, the c2f_deep variant—whose RGB pipeline is unstable under camera-plus-preprocessing bus contention—runs stably in its lighter YUV420 form, so the format change also removes a deployability failure.

Table 13. YUV420 versus RGB888 INT8 accuracy on the synthetic cover test (mAP@0.5, eight architectures).

Architecture	YUV420 INT8	RGB888 INT8	Δ
gelan_pan2	0.7428	0.6932	+0.050
gelan	0.7607	0.7157	+0.045
c2f_deep	0.7613	0.7296	+0.032
c3	0.7358	0.7149	+0.021
c2f	0.7541	0.7384	+0.016
c2f_thick	0.8437	0.8296	+0.014
c2f_pan2	0.7207	0.7423	−0.022
msblock	0.6266	0.6784	−0.052

This input-format co-design is a concrete instance of the paper’s central point: a detector adapted to the board’s actual bottleneck—here the USB input transfer—can match the accuracy of the classical model while running substantially faster, and the best-fit configuration is found empirically through on-board measurement rather than from accuracy figures alone.

4.2.4. Multi-Patch Throughput

The multi-patch mode evaluates configurations in which several regions from one camera frame are sent to the EdgeTPU. This mode increases the number of detections associated with each acquired frame, but it also consumes multiple invokes per frame. For this reason, Table 14 reports both frame throughput and invoke throughput.

Table 14. Multi-patch pipeline throughput (build 1590), for RGB and native YUV420 input.

Patches/Frame	Invokes/Frame	Input	Frame Throughput (Frames/s)	Invoke Throughput (Invokes/s)	Failures
2	1.99	RGB	13.28	26.47	0
3	2.99	RGB	8.84	26.43	0
4	4.02	RGB	6.66	26.80	0
2	1.99	YUV420	20.27	40.36	0
3	3.00	YUV420	13.74	41.23	0
4	4.00	YUV420	10.41	41.65	0

The measured invoke throughput stays close to the single-camera rate in every multi-patch configuration—about 26.5 invokes/s with RGB input and about 41 with YUV420—so the number of invokes per frame is preserved and the frame throughput is simply that invoke rate divided by the patches per frame. No timeouts or inference failures occurred. This behavior is useful when detection density per frame is more important than camera frame rate, because the runtime can trade frame rate for multiple inferences over the same acquired image while preserving stable execution; the YUV420 input-format gain (Section 4.2.3) carries directly into the multi-patch mode.

4.2.5. Dual-Camera Switching and Alternation

The dual-camera subsystem was evaluated using two complementary measurements. The first measures manual switching from one selected camera to the first usable tensor after the switch. The second measures continuous 1:1 alternation while the pipeline is running. These measurements answer different questions: manual switching captures the cost of changing the active sensor on demand, while continuous alternation reports the sustained rate when both cameras are sampled in a fixed schedule.

For manual switching, the application-visible cost is the complete `select()` -> `to_tensor()` interval, not only the MUX selection step with the one-frame drain setting the first usable

post-switch tensor is available after about 75 ms. In continuous 1:1 alternation, both the one-frame and two-frame drain settings are content-correct, but the one-frame setting sustains twice the rate (about 22.5 versus 10.6 combined detections/s), so it is the production configuration. The dual-camera rate is essentially identical for RGB and YUV420 input (22.52 versus 22.57 combined detections/s): unlike the single-camera pipeline, continuous alternation is bounded by the camera MUX switch and the VBLANK flip rather than by the USB input transfer, so halving the input has no effect here. The switch is therefore the dual-camera bottleneck, which is why the YUV420 co-design accelerates single-camera but not dual-camera operation.

Frame identity (Figure 10) was validated with camera-specific content: one camera delivered the live scene while the other was configured with a distinct OV5640 grayscale-bar test pattern, so the two sources are visually unambiguous. A correct frame's content matches its camera tag (Table 15); this check detects stale, mis-tagged, or mixed frames without relying on human inspection alone. The validation results are summarized in Table 16.

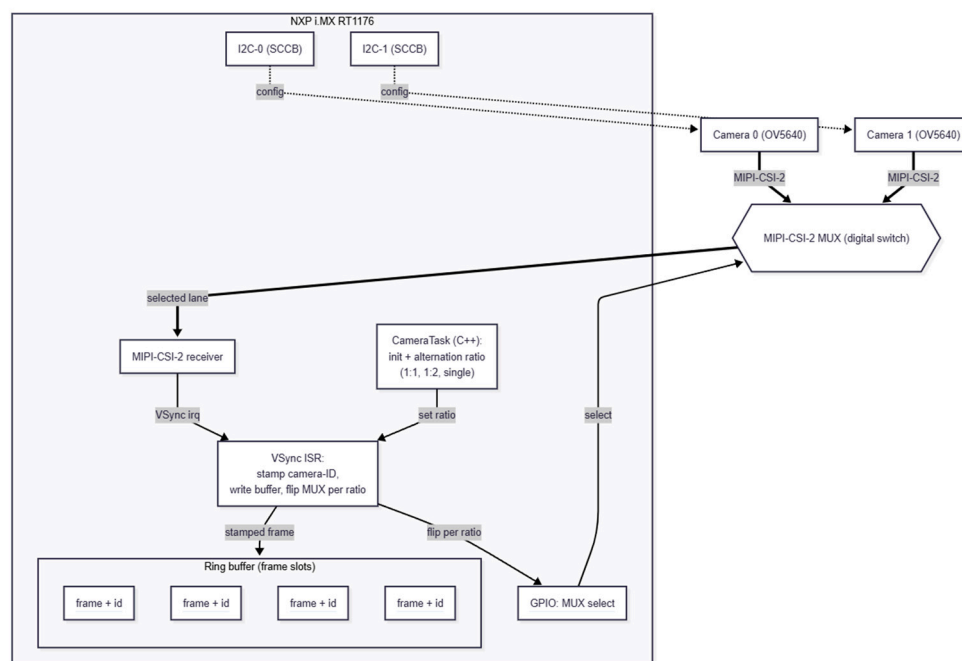


Figure 10. Dual-camera connection and switching. The two OV5640 modules are configured independently over separate I2C/SCCB buses and share one MIPI-CSI-2 lane through a digital MUX selected by a single RT1176 GPIO. The CSI-receiver interrupt service routine, triggered at each VSync, stamps every frame written to the ring buffer with its source camera ID and, when an alternation ratio is configured, flips the MUX; CameraTask only sets the cameras and the ratio (1:1, 1:2, ..., or single camera).

The sustained dual-camera configuration used for the final runtime is therefore continuous ratio(1, 1) alternation with drain = 1: every inspected frame was correctly tagged to its source camera (142 of 142, with no mismatched or mixed frames) for both RGB and YUV420 input, while sustaining 22.5 combined detections/s—twice the drain = 2 rate.

Table 15. Dual-camera timing and alternation measurements (build 1590), for RGB and native YUV420 input.

Scenario	Drain	Input	Measured Result	Content Verdict
Manual select() -> to_tensor()	1	RGB/YUV420	~75 ms	valid
Continuous ratio (1, 1)	1	RGB	22.52 detections/s combined, 11.26 per camera	valid, 0 resets
Continuous ratio (1, 1)	1	YUV420	22.57 detections/s combined, 11.29 per camera	valid, 0 resets
Continuous ratio (1, 1)	2	RGB/YUV420	~10.6 detections/s combined, ~5.3 per camera	valid, but half the rate

Table 16. Dual-camera content validation for continuous 1:1 alternation (build 1590). Every inspected frame was checked for correct per-camera tagging; the finer sub-frame seam breakdown was not capturable from the runtime.

Input	Drain	Frames Inspected	Correct	Mismatch	Mixed
RGB	1	142	142	0	0
RGB	2	71	71	0	0
YUV420	1	142	142	0	0

4.2.6. Detection Update Periods

The measurements above support update-period claims for the current runtime. A complete signal-to-detection latency distribution would require a same-frame timestamp trace through image exposure, CSI transfer, preprocessing, invoke submission, invoke completion, post-processing, and publication. The present measurements (Table 17) instead report the output periods and switching intervals directly observed by the runtime.

Table 17. Measured and derived update-period quantities.

Quantity	Value	Source
Single-camera output period, RGB/YUV420 (c2f_thick)	38.2 ms/24.7 ms	derived from 26.2/40.5 detections/s
EdgeTPU invoke time, RGB/YUV420 (c2f_thick)	35.0 ms/21.9 ms	measured isolated invoke
Continuous 1:1 combined output period	44.4 ms	derived from 22.52 detections/s
Continuous 1:1 per-camera update period	88.8 ms	derived from 11.26 detections/s per camera
Manual select() -> to_tensor(), drain=1	~75 ms	measured post-switch tensor acquisition timing

In single-camera operation at 45FPS, the runtime publishes detections every 38.2 ms with RGB input and every 24.7 ms with the native YUV420 input. In continuous dual-camera alternation, detections are produced every 44.4 ms overall, while each camera is refreshed every 88.8 ms on average (the same for RGB and YUV420, since alternation is switch-bound). These values are the appropriate performance quantities for control loops that consume the published detections

4.2.7. Summary

The final platform sustains real-time onboard vision at the native VGA (640×480) input while running the complete embedded camera and preprocessing pipeline. For the accuracy champion, the single-camera pipeline reaches about 26 detections/s with standard

RGB input and about 40 detections/s with the native YUV420 input, with no timeouts or inference failures. Because the EdgeTPU invoke is bounded by the USB input-tensor transfer, halving that transfer through the YUV420 co-design is what delivers the $1.5\text{--}1.6\times$ throughput gain. Multi-patch operation maintains approximately the same total invoke throughput while trading frame rate for multiple detections per frame. Continuous dual-camera alternation is stable and content-correct at 22.5 combined detections/s (11.3 per camera) with the validated `drain=1` setting, for both RGB and YUV420 input; because alternation is bounded by the camera switch rather than the transfer, the YUV420 gain does not extend to it.

These results show that the useful performance of an MCU-class EdgeTPU vision system is defined by the complete onboard chain, not by accelerator execution in isolation. The limiting factors include tensor transfer, instruction transfer, camera scheduling, memory placement, preprocessing, and synchronization. Reporting complete-operation wall-clock measurements therefore provides the most relevant characterization for embedded perception workloads.

4.3. Drone Pilot Connectors

A practical goal of the platform was not only to run onboard detection, but also to make those detections usable inside drone-control workflows. For this reason, the runtime exposes a MicroPython control layer that can read perception outputs from the embedded pipeline and translate them into telemetry, avoidance data, or direct flight commands. In the present implementation, this integration is available through two backends. The first is a MAVLink-oriented bridge, intended for PX4- or ArduPilot-style systems. The second is a Crazyflie-oriented bridge, intended for lightweight indoor experiments and rapid closed-loop testing. In both cases, the same basic logic is preserved: detections are produced by the onboard pipeline, interpreted in script space, and turned into actions visible to the flight stack.

Consistent with the two-track design of Section 3.2.1, the integration in this section is demonstrated through real, logged missions executed in software-in-the-loop simulation rather than through illustrative code. Each cited mission completed end to end and produced a recorded log; the simulation validates the algorithms that consume the detections, while the hardware-dependent perception path is characterized on the physical board in Section 4.2.

The simulation is air-gapped from simulator ground truth—the onboard perception consumes only camera frames and the telemetry link, never injected world state—so a successful run reflects the algorithm and not privileged information. On-hardware closed-loop flight is in progress and will be reported separately. In practical deployments, mission control additionally includes state estimation, filtering, safety logic, watchdog conditions, trajectory constraints, and mode management; the missions below validate the perception-to-action algorithms, not a complete production flight policy.

4.3.1. Integration Architecture

From an architectural point of view, the important point is that the high-level logic remains onboard. The embedded runtime does not stop at inference and hand results to an external host for interpretation. Instead, the device can execute a short control loop locally, using the MicroPython layer as a compact decision surface above the lower-level C/C++ runtime. This makes it possible to prototype behaviors such as target following, person-aware navigation, or obstacle reporting without rewriting the low-level camera and TPU path for every experiment.

In the current design (Figure 11), the control loop is built around the detection pipeline: a script loads the INT8 model, starts the background perception pipeline, and then polls either the current detections—bounding boxes, confidence scores, and class identifiers—or, when tracking is enabled, track-oriented events suitable for persistent obstacle reporting or track-based control. These two queries form the bridge between perception and drone-side logic and are the same interface exercised by the missions reported below.

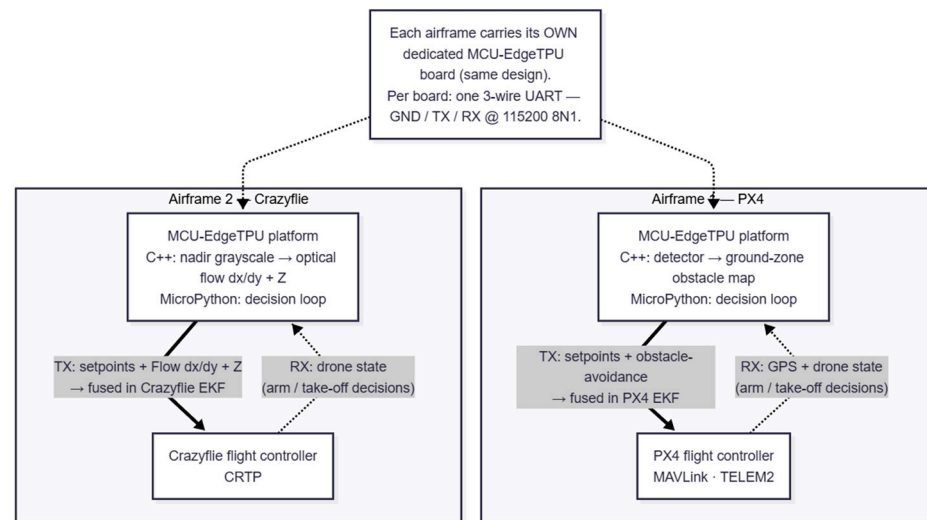


Figure 11. Autopilot interconnection. Each airframe carries its own dedicated MCU-EdgeTPU board (same design); the figure shows the PX4 and Crazyflie deployments. Each board integrates with its flight controller over a single three-wire UART (GND/TX/RX, 115200 8N1). Outbound, the platform sends setpoint commands plus perception products that are fused in the autopilot’s Enhanced Kalman Filter: ground-zone obstacle-avoidance commands from the onboard detector over MAVLink (PX4, TELEM2 port), or optical-flow (dx, dy) and height (Z) estimates computed onboard on the RT1176 in C++ from the nadir-camera grayscale image over CRTP (Crazyflie). Inbound, the flight controller returns GPS coordinates and drone state, which the platform’s MicroPython decision loop uses for higher-level actions such as arming and take-off.

4.3.2. MAVLink Protocol Integration for PX4 Drone Pilots

For PX4-style integration, the runtime exposes a MAVLink v2 bridge over UART. This path lets the onboard module communicate with a PX4 autopilot using standard message types rather than custom packets, and it supports two complementary uses publishing a tracker-derived obstacle map for collision-prevention logic on the flight side, and forwarding an onboard vision-based pose estimate as the vehicle’s position source.

This path was exercised in a real Gazebo [32] software-in-the-loop mission in which the drone’s own downward camera detected ArUco/WhyCon markers [33], recovered the vehicle’s absolute world pose by perspective-n-point [34], and streamed it to the PX4 estimator as a VISION_POSITION_ESTIMATE, enabling a GPS-denied hover driven entirely by onboard vision (mission s108). The same bridge can also publish track lifecycle events—appearance, update, and disappearance—so that the autopilot or a connected ground station receives structured, AI-side object understanding rather than only geometric avoidance cues.

4.3.3. Crazyflie-Based Closed-Loop Tests

For smaller indoor experiments, the runtime also drives a Crazyflie over a CRTP bridge, letting a compact perception-to-control loop run on a lightweight platform with minimal external infrastructure. Three executed software-in-the-loop missions, each producing a recorded log, validate this path end to end. In a marker-based calibration mission, WhyCon

markers [35] detected by the onboard camera were turned into a perspective-n-point pose and a Kabsch [36] frame estimate while the vehicle flew the full sequence—marker initialisation, take-off, logged hover, and landing—in simulation (missions s182 and s183). A yaw-sweep mission exercised closed attitude control while the perception pipeline streamed continuously (mission s185). In an optical-flow position-hold mission, onboard phase-correlation [37] flow estimates were streamed into the Crazyflie estimator at about 10 Hz, holding the vehicle over a detected target to within roughly 10–25 cm of the target position (mission s090). In each case the onboard pipeline did not merely return detections: it closed a local loop in which perception became a control input on the flight side. As emphasized above, these are algorithm validations rather than a complete flight-control law; in realistic deployment such loops sit inside a larger structure with explicit safety states, confidence filtering, temporal smoothing, and fallback conditions.

A more complete navigation demonstration closes the loop through a multi-axis trajectory. Here the onboard C/C++ controller, driven by marker detections and onboard optical flow, flew the vehicle through a cross-shaped path—center, then a left/right sweep along the X axis, then an up/down sweep along the Y axis, returning to center—while holding altitude. All seven trajectory segments reached their targets, and a post-mortem comparison of the autopilot's onboard position estimate against simulator ground truth (used only off-line and never provided to the controller) shows the estimate converging to within about 1.5 cm of ground truth and tracking it throughout the trajectory, both in the horizontal plane and in altitude (Figure 12). This gives a quantitative, if simulation-based, view of the perception-driven navigation and state estimation that the platform supports.

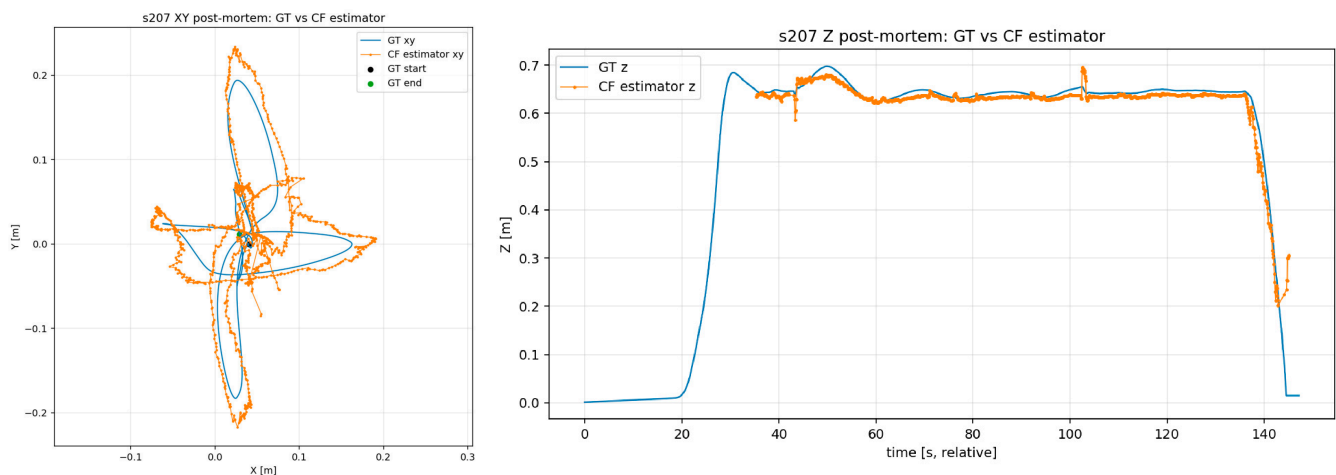


Figure 12. Cross-axis navigation—onboard estimate versus ground truth. A marker-and-flow-controlled Crazyflie flown through a cross-shaped path (an X sweep followed by a Y sweep) in software-in-the-loop simulation. Top: the horizontal (XY) trajectory; the onboard estimator (orange) tracks the simulator ground truth (blue). Bottom: the corresponding altitude (Z) over time. Ground truth is used only for this post-mortem comparison and is never fed to the controller; the onboard estimate converges to within about 1.5 cm of ground truth.

4.3.4. Implications for Onboard Autonomy

These examples show how AI outputs can be connected directly to drone-side actions without introducing a separate onboard computer or moving the decision logic off the device. In the MAVLink case, this means exposing detections and tracks to a standard autopilot interface. In the Crazyflie case, it means driving a lightweight drone from onboard visual cues through a short local control loop.

More broadly, this layer supports one of the main ideas behind the platform: the embedded device can do more than run a model and return boxes. By combining the

detection pipeline with a scriptable onboard control surface, it becomes possible to prototype person-aware flight constraints, obstacle-driven responses, target following, and other perception-based mission behaviors directly on the MCU–EdgeTPU system itself. The examples in this section should therefore be understood as integration patterns that demonstrate capability, while leaving room for more elaborate mission-control logic in future work.

4.4. Detector Model Trained on Synthetic Aerial Data

A separate set of experiments was carried out to select a person detector suitable for the onboard EdgeTPU pipeline. The objective was not only to maximize validation accuracy, but to identify a model that could remain small, quantizable, and useful under the same image geometry for which the drone platform was designed. The dataset was therefore generated and curated around the expected flight configuration: aerial images, small human targets, and camera viewpoints representative of low-altitude person-awareness and avoidance missions.

Concretely, candidates were ranked by the following criteria, in order of priority: (i) INT8 accuracy—the mAP50 of the quantized model actually deployed on the EdgeTPU, since the EdgeTPU path is INT8-only; (ii) EdgeTPU deployability and runtime stability—the variant must compile to a single on-chip subgraph and run inside the complete camera-to-TPU pipeline without breaking real-time behavior; and (iii) FP32 accuracy at the target object scale, used to shortlist candidates before quantization. Latency, memory, and power were treated as constraints rather than ranking variables: every shortlisted variant met the same real-time invoke budget—the invoke time is dominated by the fixed-size input-tensor transfer and is largely architecture-independent at this input size (Section 4.2)—and all INT8 model sizes (Table 18) fit the on-board runtime budget, while the EdgeTPU sets a fixed power envelope; a full power, weight, and thermal characterization of the platform is outside the scope of this paper. INT8 accuracy on the deployed model—together with the activation choice that most affects it (discussed below)—was therefore the decisive criterion.

All models were trained at the curated 640×480 resolution with a custom TensorFlow-native trainer that reimplements the YOLOv5-style architecture and training loop directly in TensorFlow, rather than training in PyTorch version 1.8+ and converting. This matters for deployment: PyTorch is channels-first (NCHW), whereas the EdgeTPU and the TensorFlow Lite runtime are channels-last (NHWC), so converting a PyTorch model injects transpose operations at every layout mismatch, while the native trainer emits the graph already in the target layout with zero such transposes (against four for a comparable converted pipeline, verified across all trained models), keeping each model a single clean on-chip subgraph and freeing instruction budget. Although the framework is YOLOv5-based, the compared variants are not all standard YOLOv5 architectures: the c2f bottleneck originates in YOLOv8 [38] and the GELAN block in YOLOv9 [39], adapted here into a common lightweight P3/P4 detection head. In this paper, “YOLOv5” therefore denotes the training and deployment framework rather than the architecture of every compared variant. Since the target persons occupy only a small number of pixels, geometric augmentations that change object scale were disabled. In particular, mosaic and random scaling were not used, because they would shrink the already small human targets and make the training distribution less consistent with the drone camera geometry.

The model search focused on lightweight YOLO variants that could plausibly be exported to an INT8 TensorFlow Lite format and used in the EdgeTPU deployment path. Optimization used stochastic gradient descent (momentum 0.937, weight decay 5×10^{-4}) with a cosine learning-rate schedule ($\text{lr}_0 = \text{lrf} = 0.01$, three warm-up epochs) and a weight exponential-moving-average (decay 0.9999), for 100 epochs at batch size 64 with early

stopping (patience 15) and a fixed seed; the only retained augmentations were HSV color jitter and a horizontal flip.

The complete training comparison is shown in Table 18, which reports each architecture in FP32 and after INT8 post-training quantization, for both the SiLU and the ReLU activation. On this synthetic split the accuracy champion is c2f_thick, which reaches 0.8430 mAP50 in FP32 and retains 0.8021 (SiLU) or 0.8296 (ReLU) after INT8 conversion; for this dataset, increasing feature-channel capacity was more effective than increasing network depth or adding detection heads. INT8 quantization is stable across all eight architectures—no architecture collapses under post-training quantization once the activation is chosen appropriately—so within this comparison model selection is governed by the accuracy-versus-size trade-off rather than by quantization survival.

Table 18. Detector training comparison—FP32 and INT8 mAP50 (480 × 640, both activations).

Model	FP32	INT8 (SiLU)	INT8 (ReLU)	INT8 Size	Model
c2f_thick	0.8430	0.8021	0.8296	874 KB	c2f_thick
c2f	0.7571	0.6469	0.7384	200 KB	c2f
c2f_pan2	0.7403	0.6868	0.7423	262 KB	c2f_pan2
c2f_deep	0.7469	0.7201	0.7296	378 KB	c2f_deep
c3	0.7420	0.7159	0.7149	193 KB	c3
gelan	0.7631	0.7361	0.7157	279 KB	gelan
gelan_pan2	0.7639	0.7271	0.6932	363 KB	gelan_pan2
msblock	0.6835	0.6308	0.6784	128 KB	msblock

Two deployment-oriented findings emerge. First, the activation function matters more for the quantized model than for the floating-point one. ReLU is marginally weaker in FP32, but its post-training-quantization accuracy drop is consistently far smaller than SiLU's—it never collapses under quantization—and, because ReLU fuses into the preceding convolution while SiLU compiles to a per-element lookup, the ReLU models use about 47% fewer EdgeTPU instructions on average [2]. Whether ReLU also improves the *absolute* INT8 accuracy is dataset-dependent: on this synthetic split it does (for example, c2f improves from 0.6469 to 0.7384), because SiLU happens to quantize poorly here, whereas on the real VisDrone distribution, where SiLU stays quantization-stable, SiLU retains a small INT8 lead (Section 4.5). The robust, dataset-independent advantages of ReLU are therefore quantization stability and lower latency rather than a universal accuracy gain, which is why the deployment path uses ReLU. Second, the deeper c2f_deep variant underperforms the wider c2f_thick, confirming that channel capacity is more useful than depth for this small-object aerial dataset; the value of the anti-downsampling blocks (the SPD-Conv downsample and the P2 head) is most visible on the longer-tailed real distribution and is reported in Section 4.5.

Two complementary outcomes follow. For maximum accuracy within the training distribution, the c2f_thick architecture with ReLU activation is the strongest deployable detector, retaining 0.8296 INT8 mAP50 on the synthetic split within the on-board flash and instruction budget; feeding the camera's native YUV420 format (Section 4.2.3) the same architecture is both slightly more accurate and much faster, running the on-board pipeline at about 40 detections/s, which makes c2f_thick with YUV420 input the recommended production detector. For the small-object regime that matters in flight, the validated deployment recipe adds the anti-downsampling blocks (the SPD-Conv downsample and the P2 head) to this configuration; this recipe gives the most robust real-data gains and is evaluated on VisDrone in Section 4.5. All of these keep the model as a single on-chip subgraph and detect people at the pixel scale expected from the drone camera.

For deployment, every trained model was converted to a full-integer (uint8-input, int8-output) TensorFlow Lite model by post-training quantization over 500 calibration images and compiled with the EdgeTPU compiler (version 16); each compiled to a single on-chip subgraph with no CPU-offloaded operators (for the champion, about 273 KB of per-invoke instructions and 764 KB of on-chip parameters, within an 874 KB INT8 model). The INT8 accuracies in Table 18 are the post-quantization validation of these compiled models, and the selected detector is additionally the one executed on the physical board in the pipeline measurements of Section 4.2. This single-subgraph constraint also bounds the architecture space: the detection head materializes a decode tensor whose on-chip transpose buffer scales with anchors $\times$ resolution $\times$ classes, so a single-class P3/P4 head fits comfortably, whereas a COCO-scale configuration (eighty classes, three anchors, with a stride-4 head) exceeds the on-chip memory and the compiler fragments the graph onto the CPU. Off-the-shelf YOLOv8- [38], v10- [40] and v11-class [41] detectors did not, in our attempts, compile to a single on-chip subgraph on this accelerator for the same kind of reason—their newer neck, head, or attention operators fall back to the CPU—which is why the platform builds EdgeTPU-deployable variants from the deployable blocks rather than deploying the stock models, and why a like-for-like on-device comparison against them is not possible; the external positioning is therefore made against published VisDrone results (Section 4.5). Post-training quantization is used throughout; quantization-aware training was evaluated as an alternative but produced no deployable improvement on this accelerator, because the full-integer export re-derives its quantization ranges from the calibration set and discards the QAT-learned ones (Appendix A).

All detector accuracies reported in this section are synthetic-domain results: the models were trained and evaluated on the synthetic aerial dataset, so these figures characterize each detector within its training distribution. A complementary evaluation on real aerial imagery (VisDrone [42]) is reported in Section 4.5; the values above are synthetic-domain.

4.5. Real-Data Evaluation

The synthetic results of Section 4.4 characterize each detector within its training distribution. To assess transfer to real aerial imagery, a compact detector of the same family and its anti-downsampling-block variants were also evaluated on the VisDrone benchmark, filtered to its person classes and sliced at native resolution into 640×480 tiles, so that object pixel sizes are preserved and the comparison with the synthetic 640×480 patches is like-for-like. Both datasets are dominated by small objects: on the synthetic set every box is below 32 px, 94% are below 16 px, and a quarter are below 8 px (median 11×12 px), and on a COCO-area split [43] 90.7% of the VisDrone test boxes are small. They differ instead in size variance and tail—the synthetic distribution is tightly curated, with a median-absolute-deviation cap that leaves the 95th percentile near 20 px and no box above about 26 px, whereas VisDrone is taller (an aerial pedestrian view, median 12×22 px), has a much longer tail (boxes up to a few hundred pixels), and a somewhat larger sub-8-px fraction (about 32% versus 25%). VisDrone therefore tests the anti-downsampling design on a real, higher-variance, genuinely tiny-object distribution rather than on the controlled synthetic one.

On real data, anti-downsampling design choices pay off. Two deployable, EdgeTPU-native blocks were evaluated on the compact baseline: SPD-Conv [44], a lossless $2 \times$ downsample that uses the supported space-to-depth operation rather than strided slicing (which would fall back to the CPU), and a stride-4 P2 detection head for the smallest objects. Both compile to a single on-chip subgraph with no CPU operations. Trained and evaluated on the tiled VisDrone person split, they improve the overall detection accuracy monotonically: the VisDrone mAP@0.5 rises from 0.402 for the baseline to 0.411 with

SPD-Conv, 0.436 with the P2 head, and 0.439 with the SPD-Conv + P2 combination in FP32, and from 0.224 to 0.255 on the deployed INT8 model (Figure 13). The same trend holds for the size-stratified COCO small-object metric AP_S, which rises from 0.054 for the baseline to 0.066 with the combination (Table A7), confirming on genuinely tiny real objects that the gain observed on the synthetic data is not an artifact of the synthetic distribution. The larger detection heads help most precisely in the small-object range where edge deployment cares most.

These block gains are statistically robust. Re-running the baseline and the SPD-Conv + P2 configuration over three random seeds, the FP32 improvement in the anti-downsampling blocks is about 0.05 mAP₅₀ with a 95% confidence interval that excludes zero, whereas the small FP32 difference between the SiLU and ReLU activations is not significant. After INT8 quantization the blocks still improve accuracy on the deployed model—on VisDrone the baseline rises from 0.224 to 0.255 with ReLU and from 0.232 to 0.268 with SiLU—but the per-seed INT8 deltas are individually inconclusive because post-training quantization occasionally collapses for either activation, which is handled in practice by quantizing several checkpoints and discarding the rare collapse. Two activation observations confirm that the ReLU-versus-SiLU result of Section 4.4 is a quantization effect rather than an architectural one: in FP32, SiLU is slightly ahead for every block pair, and on VisDrone—where SiLU does not collapse under quantization—SiLU keeps a small INT8 lead, while ReLU’s advantage remains its consistently smaller quantization drop and its lower instruction count. The full per-seed confidence intervals and the per-model INT8 results are given in Appendix A (Tables A8–A10).

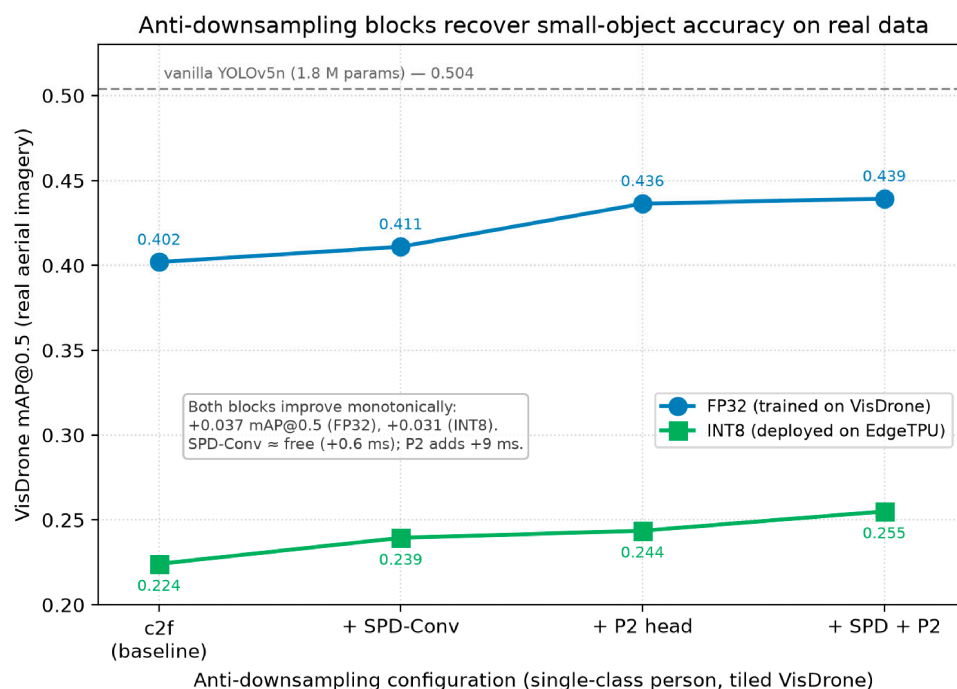


Figure 13. Anti-downsampling blocks on real VisDrone data. VisDrone mAP@0.5 for the compact detector as the anti-downsampling blocks are added (baseline → SPD-Conv → P2 head → both), in FP32 (trained on VisDrone) and after INT8 quantization (deployed on the EdgeTPU). Both curves rise monotonically (+0.037 mAP@0.5 in FP32, +0.031 in INT8), confirming that the small-object gain transfers to real aerial imagery. The dashed line marks the 1.8 M-parameter vanilla YOLOv5n reference (0.504); SPD-Conv adds accuracy at almost no latency cost (about +0.6 ms), whereas the P2 head adds further accuracy at about +9 ms.

For external context, a nano reference model reaches 0.50 mAP50 on the tiled VisDrone person split. This figure is not directly comparable to standard VisDrone-DET results, which score pedestrian and people as separate classes, use full images, and report average precision over the [0.5:0.95] range; under a single merged person class, native-resolution tiling, and mAP50-only reporting, published YOLOv5-class person/pedestrian AP50 spans roughly 0.33–0.55, and our result falls within this band. Its position is driven by the protocol—class merging and tiling—rather than by a larger backbone, since the reference model is a 1.8 M parameter nano network, an order of magnitude smaller than the tiled larger models at the top of that band; we therefore frame the result as falling within the published band from a much smaller model, not as exceeding the state of the art. A like-for-like comparison against off-the-shelf modern lightweight detectors (YOLOv8-, v10- and v11-class) is not included, because their newer neck, head, and attention operators do not compile to a single on-chip subgraph on this accelerator and fall back to the CPU (Section 4.4); they are therefore not deployable edge baselines on this platform and would at most provide a floating-point-only context point.

The on-board deployment trade-off—accuracy, speed, and model size together—is summarized in Figure 14.

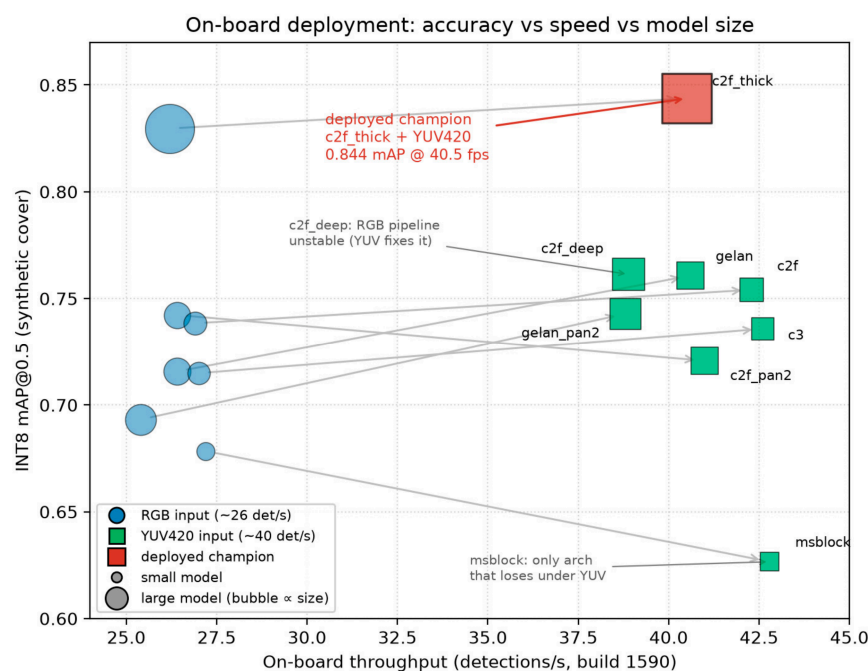


Figure 14. On-board deployment view of the detector family—accuracy vs. speed vs. model size. INT8 mAP@0.5 (synthetic cover, deployed activation) against measured on-board throughput (build 1590), with bubble area proportional to the INT8 model size. Each architecture is shown for both RGB input (circles, about 26 detections/s) and native YUV420 input (squares, about 40 detections/s); the arrows trace the RGB→YUV420 co-design, which shifts every architecture to a higher throughput at equal or better accuracy. The deployed champion—c2f_thick with YUV420 input—is the top-right point (0.844 INT8 mAP@0.5 at 40.5 detections/s); msblock is the only architecture that loses accuracy under YUV420, and c2f_deep, whose RGB pipeline is unstable, becomes deployable in its lighter YUV420 form. The figure captures the paper’s deployment message: the co-designed detector maximizes detection accuracy and on-board speed together.

5. Discussion

The results show that useful onboard perception on this class of hardware cannot be achieved by treating the detector, the dataset, and the embedded platform as separate problems. In the early configurations, the model could run in isolation on the EdgeTPU, but

the complete camera-to-detection path remained unstable because camera DMA, preprocessing, SDRAM traffic, USB transfer, and TPU invocation interfered with each other under load. The final improvement therefore came from system-level optimization: memory placement, buffer ownership, runtime scheduling, and TPU input staging had to be shaped together before the platform could sustain real-time operation.

However, the hardware side is only one end of the problem. The detector itself also has to be designed for the type of visual input that the drone will actually observe. For this reason, the person detector was not trained on a generic dataset, but on a synthetic aerial dataset generated around the expected operating conditions of the platform: low-altitude drone viewpoints, 640×480 model input, small human targets, varied scene types, different weather conditions, and daylight variation. This was important because the apparent size and shape of people in the camera image are not arbitrary; they are determined by the camera geometry, flight altitude, field of view, and preprocessing resolution.

This leads to one of the main lessons of the work: maximum embedded performance is obtained by starting from both ends at the same time. From one end, the model must be suitable for the board: small enough, quantization-stable, compatible with the INT8 deployment path, and simple enough to run inside the full camera-to-TPU pipeline without breaking real-time behavior. From the other end, the dataset must be suitable for the camera and mission: it should represent the object scale, image resolution, time of day, weather, and viewpoint that the onboard sensor is expected to encounter. Optimizing only one of these ends is not sufficient. A powerful model trained on the wrong visual distribution may not help the drone, and a good dataset may still fail if the selected architecture cannot be deployed efficiently on the target hardware.

The model comparison supports this interpretation. Across the candidate architectures INT8 quantization is stable, and the most robust accuracy result—significant across seeds—is that the anti-downsampling blocks improve floating-point accuracy by about 0.05 mAP50. The activation function has a larger effect on the quantized model than on the floating-point one, but its role is mainly one of deployment safety: ReLU's post-quantization drop is consistently far smaller than SiLU's and it uses fewer accelerator instructions, so the deployed path uses ReLU, while any absolute INT8 accuracy difference between the two is dataset-dependent. The c2f_thick variant gives the best in-distribution accuracy at an acceptable model size, but the validated small-object deployment recipe combines the anti-downsampling blocks with ReLU. The broader lesson is that embedded detector selection should be based not only on FP32 mAP but on the combination of quantization robustness, activation choice, model size, and runtime integration behavior.

A further lesson concerns the input to the accelerator, not only the model. Because the EdgeTPU invoke is bounded by the USB transfer of the input image rather than by compute, the pixel format is itself a design variable: feeding the camera's native YUV420 planes instead of host-decoded RGB halves the transferred input and gives a $1.5\text{--}1.6\times$ on-board throughput gain at no accuracy cost (Section 4.2.3). More generally, this illustrates that a detector adapted to the board's dominant bottleneck can match the accuracy of an unadapted model while running substantially faster, and that the best-fit configuration—architecture, activation, and input format—is found empirically, through on-board measurement, rather than predicted from accuracy figures alone.

The dataset curation choices also affected the architecture results. Since the synthetic dataset was curated for the expected aerial scale, extremely small boxes and physically implausible outliers were removed before training. This made the training distribution more consistent with the intended flight geometry. It also explains why adding an extra very-small-object detection head did not improve the results: the dataset had already been

filtered to remove annotations below the useful visual scale. For this application, wider feature capacity was more beneficial than additional depth or additional detection heads.

The dual-camera and runtime results should be interpreted in the same system-level context. Alternating between two cameras naturally reduces the effective per-camera rate, but the important result is that dual-view operation became stable and usable without mixed-frame artifacts. This matters because the platform is intended for drone tasks where different viewpoints may serve different roles, such as downward ground-person detection and forward situational awareness. The detector trained on synthetic aerial views therefore becomes more useful when combined with a runtime that can select, switch, and process camera streams reliably.

The software structure reinforces the same point. Keeping timing-sensitive work in C/C++ while exposing mission logic through MicroPython allows the platform to remain both performant and flexible. Concurrent load was also examined directly. The runtime drives several perception tasks from the same camera stream—the detector, an optical-flow estimator, and a marker detector—and the preprocessing and inference stages execute in parallel rather than serially, overlapping in time rather than adding. The optical-flow path, for example, costs only about 2.3 ms of compute per frame and is bounded by the camera rate rather than by computation, so running it alongside detection does not reduce the detection rate, and combined detection-plus-flow pipelines ran without frame-grab failures. Because the timing-critical path is moreover driven mainly by DMA and the USB/EdgeTPU transfer—during which the processor is mostly waiting on data movement—the cooperative MicroPython control layer, which executes no per-frame code, runs in the available slack and does not, in practice, contend with the perception pipeline. The board is not only a passive inference module returning bounding boxes. It can run high-level onboard logic that converts detections into perception-driven behaviors, such as person-aware constraints, obstacle reporting, target following, or communication through Crazyflie and MAVLink/PX4 integration. This makes the synthetic-data-trained detector part of a complete perception-to-action loop rather than a standalone model result.

The platform can also be positioned against existing embedded UAV-vision systems (Table 19). It occupies a distinct niche—an MCU-class, fully onboard module that performs INT8 object detection with dual-camera support—between ultra-low-power nano-UAV navigation engines and higher-power Linux companion computers. GAP8-based nano-UAV engines run a lightweight navigation CNN fully onboard at tens of frames per second within a sub-100 mW envelope [1], but on a steering/regression task rather than object detection. Jetson-class companions run lightweight YOLO detectors at roughly 40–65 frames per second, but draw about 8–14 W [8] and are far heavier than an MCU-class board. Our platform performs real INT8 person detection at about 26 detections per second with RGB input, and about 40 with the native YUV420 input, on a microcontroller-plus-EdgeTPU path; its power, weight, and thermal envelope are characterized separately and are outside the scope of this paper. The frequently cited figure of around 400 frames per second for an EdgeTPU running MobileNet [45] is a host-fed image-classification result on a Linux machine, not onboard MCU detection—which is precisely the regime this paper characterizes, since the USB transfer path, not the accelerator, sets the onboard ceiling; it is exactly this transfer bound that the YUV420 input-format co-design exploits.

Table 19. Positioning against representative embedded UAV-vision platforms. Task, input size, and precision differ per row and are stated; cross-platform throughput, latency, and accuracy are only comparable within that context. The latency column is the approximate per-frame inverse of throughput, and the detection-accuracy column is not a like-for-like comparison because the task, dataset, and precision differ per row (this work is a single-class synthetic-domain person detector; the Jetson rows run YOLOv8n on COCO; the Coral and GAP8 rows perform a different task entirely).

Platform	Compute	Class	Task	Input	Throughput	Latency	Detection Accuracy	Power
This work	i.MX RT1176 (M7) + Coral EdgeTPU (USB 2.0)	MCU-class, onboard	INT8 person detection	640×480	~26 det/s RGB, ~40 det/s YUV420	~35/~22 ms invoke	0.83 mAP50 (synthetic, in-distribution); ~0.50 in the VisDrone band	separate study (out of scope)
Palossi et al. [1]	GAP8 (PULP RISC-V)	nano-UAV, onboard	navigation/steering CNN	320×240	up to ~18 fps	~55 ms	n/a (regression task)	~64 mW
Coral USB/Dev Board	EdgeTPU + Linux host	SBC + accelerator	MobileNet classification	224×224	~400 fps (host-fed)	~2.5 ms	n/a (classification)	~2 W (TPU)
Jetson Orin Nano [8]	Ampere GPU SoC	Linux companion	YOLOv8n INT8 detection	640×640	~43 fps	~23 ms	YOLOv8n on COCO (different dataset)	~7–9 W
Jetson Orin NX/Xavier NX	Ampere/Volta GPU SoC	Linux companion	YOLOv8n INT8 detection	640×640	~65 fps	~15 ms	YOLOv8n on COCO (different dataset)	~10–14 W

Future work should therefore continue in two parallel directions. On the platform side, the end-to-end pipeline is bounded by the per-invoke USB transfer of the input tensor; the YUV420 input-format co-design already exploits this by halving the input transfer for a 1.5–1.6× throughput gain (Section 4.2.3), and further gains should come from reducing the transferred data still further—for example through lower-bit input encodings—or from changing the EdgeTPU invocation pattern, rather than from preprocessing alone. On the dataset/model side, synthetic generation should be expanded to cover additional altitudes, camera orientations, weather distributions, lighting conditions, and object classes. The most promising path is not to optimize the board first and the model later, or the dataset first and the board later, but to co-design them around the same mission constraints.

6. Conclusions

This work shows that a compact MCU–EdgeTPU platform can support real-time onboard drone perception when the complete system is optimized as a single camera-to-action chain. The final platform ran the single-camera pipeline stably—with no timeouts or inference failures—at about 26 detections per second with standard RGB input and about 40 with the camera’s native YUV420 format, which halves the bandwidth-bound input transfer; it also supported dual-camera operation and exposed detections to a scriptable onboard control interface. These results confirm that accelerator performance alone is not enough; stable embedded perception depends on camera handling, preprocessing, memory placement, transfer behavior, scheduling, inference, post-processing, and communication working together.

The work also shows that model and dataset design must be aligned with the embedded platform. The selected detector was not simply the largest or most accurate FP32 candidate. It was chosen because it preserved accuracy after INT8 quantization, remained compact, and matched the type of input expected by the onboard camera pipeline. The synthetic dataset was likewise not generated as a generic image collection, but as a controlled approximation of the drone’s expected operating conditions: aerial viewpoint, target resolution, human pixel size, time-of-day variation, weather variation, and scene diversity.

The main conclusion is therefore that high embedded performance is best obtained by starting from both ends of the problem. On one side, the board, runtime, memory layout, and EdgeTPU path must be optimized for reliable real-time inference. On the other side, the detector and its training dataset must be optimized for the exact visual distribution the camera is expected to see. When these two directions meet, the result is not only a faster benchmark, but a more practical onboard perception system. Concretely, this work was a search for the architecture and input format that jointly maximize detection accuracy and on-board speed under the board's dominant constraint—the USB transfer of the input tensor—and it is this search that led to training the detector on the camera's native YUV420 format, so that the transfer bandwidth no longer bounds the pipeline while the detection accuracy is preserved. In this sense, the contribution of the paper is not only a hardware platform, and not only a synthetic-data-trained detector. It is the integration of both into a co-designed perception pipeline for small UAVs.

The platform provides the real-time embedded path, while the synthetic dataset provides a way to train models for the specific mission conditions before large amounts of real flight data are available. This combination makes the system useful as a repeatable research base for developing person-aware navigation, ground-person avoidance, and other perception-driven drone behaviors. This paper is best understood as a deliberate intermediate milestone in that direction: an open, MCU-class onboard perception platform, validated where it can be validated today—the hardware-dependent perception path on the physical board (Section 4.2) and the perception-to-action algorithms in air-gapped software-in-the-loop simulation (Section 4.3)—and published as a reusable research base rather than as a finished autonomy system.

Several extensions are already planned and follow directly from the present results. On the flight side, the closed-loop control demonstrated in simulation will be repeated on the physical drone to obtain on-hardware navigation metrics. On the detector side, the synthetic-domain results are already complemented by the real-aerial evaluation of Section 4.5 (VisDrone, with small-object average precision at the small, medium, and large scales that target the regime of interest); the natural next step is to widen the synthetic generator to the additional altitudes, viewpoints, and classes noted above and to retrain. A like-for-like on-device comparison against off-the-shelf modern lightweight detectors is not available on this accelerator, since their newer operators do not compile to a single EdgeTPU subgraph, so such models can serve only as floating-point context.

On the deployment side, on-board energy and thermal characterization is outside the scope of this paper; quantization-aware training was evaluated as a remedy for the occasional post-training-quantization collapse but gave no deployable benefit here (Appendix A), so post-training quantization with checkpoint screening is retained. The long-term direction remains a workflow in which the expected camera geometry, the synthetic data generator, the detector architecture, and the embedded runtime are designed together from the outset rather than reconciled after deployment problems appear.

Author Contributions: Conceptualization: B.N. and A.M.F.; formal analysis, methodology, software, and validation: B.N.; investigation and resources: A.M.F. and A.M.F.; data curation and writing—original draft preparation: B.N.; writing—review and editing: B.N. and A.M.F.; supervision, project administration: A.M.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Acknowledgments: During the preparation of this manuscript/study, the author(s) used ClaudeCode 4.6 for the purposes of writing documentation code, review code. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CDC-ACM	Communications Device Class—Abstract Control Model
CDC-NCM	Communications Device Class—Network Control Model
CPX	Crazyflie Packet Exchange
C RTP	Crazy RealTime Protocol
CSI	Camera Serial Interface
DMA	Direct Memory Access
EHCI	Enhanced Host Controller Interface
FPS	Frames Per Second
GPS	Global Positioning System
HD	High Definition
HTTP	Hypertext Transfer Protocol
IMU	Inertial Measurement Unit
IOC	Interrupt On Completion
ISR	Interrupt Service Routine
LFS	Little File System
lwIP	Lightweight IP
MAVLink	Micro Air Vehicle Link
MCU	Microcontroller Unit
MIPI	Mobile Industry Processor Interface
OCRAM	On-Chip RAM
PXP	Pixel Pipeline
QVGA	Quarter Video Graphics Array
RAM	Random Access Memory
REPL	Read–Eval–Print Loop
RPMsg	Remote Processor Messaging
RTOS	Real-Time Operating System
SDRAM	Synchronous Dynamic Random-Access Memory
TPU	Tensor Processing Unit
UART	Universal Asynchronous Receiver–Transmitter
UAV	Unmanned Aerial Vehicle
USB	Universal Serial Bus
VGA	Video Graphics Array
YOLO	You Only Look Once

Appendix A

Appendix A.1. MicroPython-Facing Runtime Surface Exposed by the Platform

This appendix summarizes the MicroPython-facing runtime surface exposed by our platform through the `sentai.*` namespace. The purpose of this interface is to make the main embedded subsystems accessible from high-level scripts, while keeping timing-critical camera, TPU, memory, and communication logic implemented in the lower-level runtime. The namespace layout is therefore organized around the main roles required during experimentation: platform control, sensing, inference, communication with drone backends, diagnostics, and lightweight on-device analysis.

Namespace	Main Role	Typical Use in Experiments
sentai	Top-level runtime control	Version check, verbosity control, help, debug level, console routing, script execution
sentai.io	Basic platform I/O	GPIO and LED-style control
sentai.rtos	RTOS utilities	Sleep, timing, task inspection, heap and CPU status
sentai.uart	Serial communication	UART-based communication and peripheral interaction
sentai.usb	USB control	USB drive, serial, and runtime access control
sentai.fs	Filesystem access	Reading, writing, listing, and organizing on-device files
sentai.sys	System and recovery control	Reset, recovery-mode state, boot-attempt tracking
sentai.camera	Camera subsystem	Capture, resolution control, camera switching, tensor
sentai.imu	Inertial sensing	Accelerometer and motion-related measurements
sentai.mic	Audio input	Microphone capture and audio-triggered experiments
sentai.sleep	Low-power waiting	Idle wait, timeout-based wake, and sensor-triggered
sentai.tpu	EdgeTPU inference path	Model loading, accelerator control, and TPU inference
sentai.tfl	MCU-side inference path	CPU-only TensorFlow Lite Micro execution
sentai.pipeline	Detection pipeline	Background detection, tracking, projection, and geometry
sentai.flow	Optical-flow path	Block-matching flow experiments
sentai.mesh	Mesh radio bridge	Meshtastic-style communication experiments
sentai.link	MAVLink bridge	PX4/ArduPilot-oriented telemetry, vision, and obstacle
sentai.crazy	Crazyflie bridge	CPX/CRTP control, takeoff, landing, hover
sentai.diag	Diagnostics and health	Boot logs, crash state, counters, profiling, and health
sentai.aifes	Embedded ML helpers	Lightweight on-device learning experiments [46]
sentai.kmeans	Clustering	Small clustering experiments and diagnostics
sentai.pca	Principal component analysis	Dimensionality-reduction experiments
sentai.anomaly	Anomaly scoring	Runtime or sensor-data anomaly tests
sentai.dtw	Dynamic time warping	Sequence comparison and temporal-pattern experiments
sentai.hmm	Hidden Markov Models	Lightweight sequence-model experiments
sentai.rl	Reinforcement-learning helpers	Small reinforcement-learning prototypes
sentai slam	SLAM-related helpers	Early mapping and localization-oriented experiments

The exposed namespaces provide a convenient scripting interface for hardware access, sensing, inference, communication, diagnostics, and small experimental ML utilities, but the underlying work is still performed by optimized C/C++ runtime components and, where applicable, by the EdgeTPU. In this sense, MicroPython is used mainly as an orchestration layer: it makes experiments easier to express and modify, while the performance-critical parts remain in the lower-level implementation.

Appendix A.2. Supplementary Detector Results

This appendix reports the full per-architecture detector results summarized in Sections 4.4 and 4.5. All accuracies are mAP50 unless a column states otherwise; INT8 figures are full-integer post-training-quantized models compiled to a single EdgeTPU subgraph. The cover split is the synthetic curated test split of Section 3.2.7; VisDrone results use the tiled person split of Section 4.5.

Table A1. Name convention of blocks used in the paper.

Token	Meaning
c2f	C2f cross-stage feature block (from YOLOv8 [38])—the default block
c3	C3 feature block (the original YOLOv5 block)
gelan	GELAN block (from YOLOv9 [39])
msblock	a multi-scale convolution block variant
thick	wider channels (increased feature capacity)
deep	more stacked blocks (increased depth)
pan2	an additional PAN (path-aggregation) feature-fusion stage
spd	SPD-Conv lossless anti-downsampling stem (Section 4.5)
p2	an extra stride-4 (P2) detection head for small objects (Section 4.5)
vanilla5n/vanilla5s	stock YOLOv5 nano/small, used only as width references

For example, c2f_thick is the C2f block with widened channels, and c2f_spd_p2 is the C2f baseline with both the SPD-Conv stem and the P2 head.

Table A2. Backbone ablation on the synthetic cover split (eight architectures, 100 epochs, 480×640). INT8 is the SiLU full-integer model; ΔINT8 is the INT8 – FP32(test) drop. Sizes are from the compiled EdgeTPU model.

Model	FP32 Val	FP32 Test	INT8 Test	ΔINT8	Tflite (KB)	Instr (KB)	Param (KB)	Model
c2f_thick	0.8174	0.8430	0.8021	−0.041	874.5	273.4	764.0	c2f_thick
gelan	0.7293	0.7631	0.7361	−0.027	279.2	210.5	192.9	gelan
gelan_pan2	0.7215	0.7639	0.7271	−0.037	363.4	272.2	247.7	gelan_pan2
c2f_deep	0.7119	0.7469	0.7201	−0.027	377.6	277.0	265.4	c2f_deep
c3	0.7027	0.7420	0.7159	−0.026	192.5	155.0	130.5	c3
c2f_pan2	0.7068	0.7403	0.6868	−0.054	262.1	194.7	180.8	c2f_pan2
c2f (baseline)	0.7177	0.7571	0.6469	−0.110	199.8	151.5	139.7	c2f (baseline)
msblock	—	0.6835	0.6308	−0.053	127.7	139.2	90.9	msblock

Table A3. Activation instruction cost (identical weights re-exported with ReLU). ReLU fuses into the preceding convolution; SiLU compiles to a per-element lookup, and EdgeTPU latency scales with instruction count.

Model	SiLU Instr (K)	ReLU Instr (K)	Δ
c2f_thick	273.4	117.5	−57.0%
c3	155.0	78.0	−49.7%
gelan	210.5	113.8	−45.9%
c2f	151.5	82.1	−45.8%
8-architecture average	—	—	≈ −47%

Table A4. Anti-downsampling blocks on the synthetic cover split (INT8, ReLU). USB latency is the single-image Coral USB invoke time.

Model	INT8	Δ vs. Baseline	USB (ms)
c2f (baseline)	0.7384	—	28.2
c2f_spd	0.7617	+0.023	28.8
c2f_p2	0.7568	+0.018	37.3
c2f_spd_p2	0.7809	+0.043	37.9
Model	INT8	Δ vs. baseline	USB (ms)

Table A5. Block activation ablation on the synthetic cover split (INT8).

Model	INT8 SiLU	INT8 ReLU	ΔINT8	USB SiLU→ReLU (ms)
c2f_spd	0.7412	0.7617	+0.021	38 → 29
c2f_p2	0.7455	0.7568	+0.011	49 → 37
c2f_spd_p2	0.7658	0.7809	+0.015	50 → 38

Table A6. VisDrone (tiled person) FP32 validation mAP50. The vanilla models are width references, not part of the deployable block family.

Model	Params	ReLU	SiLU
c2f (baseline)	0.14 M	0.4020	0.4197
c2f_spd	0.20 M	0.4110	0.4302
c2f_p2	0.14 M	0.4364	0.4556
c2f_spd_p2	0.20 M	0.4393	0.4608
vanilla5n	1.77 M	—	0.5041
vanilla5s	7.0 M	—	0.5765

Table A7. Size-stratified COCO average precision on the VisDrone test split (pycocotools). AP_S/M/L are small/medium/large by COCO area. AP_S rises monotonically with the anti-downsampling blocks under both activations.

Model	Act	AP	AP50	AP_S	AP_M	AP_L
c2f (baseline)	ReLU	0.0651	0.2181	0.0543	0.178	0.045
c2f_spd	ReLU	0.0693	0.2306	0.0608	0.183	0.046
c2f_p2	ReLU	0.0722	0.2351	0.0624	0.174	0.030
c2f_spd_p2	ReLU	0.0764	0.2460	0.0655	0.188	0.041
c2f	SiLU	0.0698	0.2326	0.0595	0.187	0.061
c2f_spd	SiLU	0.0776	0.2457	0.0635	0.200	0.036
c2f_p2	SiLU	0.0792	0.2487	0.0672	0.199	0.057
c2f_spd_p2	SiLU	0.0824	0.2612	0.0708	0.203	0.084
vanilla5n	SiLU	0.1003	0.3039	0.0813	0.282	0.244

Table A8. VisDrone Phase B—deployed INT8 accuracy and quantization drop ($\Delta\text{INT8} = \text{INT8} - \text{FP32}$). ReLU's quantization drop is three to twelve times smaller than SiLU's, but SiLU keeps a small absolute INT8 lead because it does not collapse on this distribution.

Model	Act	INT8 Test	ΔINT8	Instr (KB)
c2f (baseline)	ReLU	0.2241	−0.0033	82.1
c2f	SiLU	0.2320	−0.0118	151.5
c2f_spd	ReLU	0.2394	−0.0009	85.9
c2f_spd	SiLU	0.2462	−0.0106	161.9
c2f_p2	ReLU	0.2436	−0.0018	98.1
c2f_p2	SiLU	0.2555	−0.0052	180.1
c2f_spd_p2	ReLU	0.2550	−0.0019	102.2
c2f_spd_p2	SiLU	0.2678	−0.0056	190.7

Table A9. Multi-seed robustness (three seeds: 42, 1, 2) on the synthetic cover test split. Mean with 95% confidence interval; CV is the INT8 coefficient of variation.

Config	Act	FP32 Mean [95% CI]	INT8 Mean [95% CI]	INT8 CV
c2f	ReLU	0.7338 [0.710, 0.758]	0.7268 [0.702, 0.752]	1.4%
c2f	SiLU	0.7434 [0.714, 0.773]	0.6920 [0.595, 0.789]	5.6%
c2f_spd_p2	ReLU	0.7862 [0.774, 0.799]	0.7506 [0.627, 0.874]	6.6%
c2f_spd_p2	SiLU	0.7926 [0.767, 0.818]	0.7559 [0.722, 0.789]	1.8%

Table A10. Paired differences (mean $\pm$ 95% CI, three seeds). The anti-downsampling block gain is significant in FP32; every INT8 difference is inconclusive at three seeds because post-training quantization collapses stochastically (two of twelve runs).

Comparison	FP32	INT8
c2f_spd_p2 − c2f (ReLU)	+0.0525 [+0.035, +0.070] (sig.)	+0.0238 [−0.058, +0.105] (n.s.)
c2f_spd_p2 − c2f (SiLU)	+0.0492 [+0.024, +0.075] (sig.)	+0.0639 [−0.002, +0.130] (n.s.)
SiLU − ReLU (c2f)	+0.0096 [−0.015, +0.034] (n.s.)	+0.0348 [−0.030, +0.099] (n.s.)
SiLU − ReLU (c2f_spd_p2)	+0.0063 [−0.012, +0.025] (n.s.)	−0.0052 [−0.088, +0.077] (n.s.)

Table A11. Published VisDrone YOLOv5-class person/pedestrian anchors used to position the result of Section 4.5. Protocols differ per row; mAP@0.5 and AP@[0.5:0.95] must not be mixed.

Reference	Model	Protocol	Person/Pedestrian Metric
EDNet [47]	YOLOv5n	full image, 10-class	0.315 mAP@0.5 (overall)
LAF [48]	YOLOv5n	full image	0.274 mAP@0.5 (overall)
LAF [48]	YOLOv10n	full image	0.335/0.248 mAP@0.5 (ped/ppl)
Keleş [49]	YOLOv5-L	640 tiles	0.547/0.348 AP50 (ped/ppl)
TPH-YOLOv5 [50]	YOLOv5x	1536, test-dev	0.290/0.168 AP@[0.5:0.95] (ped/ppl)

Table A12. EdgeTPU footprint and on-board latency of the width references (480×640 , INT8, single subgraph). The small-width model fits the accelerator but is four times slower than nano, which is why the deployable family uses nano width.

Model	Params	On-Chip	Off-Chip	Coral USB	Throughput
vanilla5n	1.77 M	1.82 MiB	0	77 ms	13.0 FPS
vanilla5s	7.0 M	6.80 MiB	42 KiB	310 ms	3.2 FPS

All deployed detectors use post-training INT8 quantization (PTQ). Quantization-aware training (QAT) was implemented and evaluated as the standard remedy for the occasional PTQ collapse, but it gave no deployable benefit on this accelerator: the composite blocks contain structural operations that the quantization-aware toolchain cannot fake-quantize, so the full-integer export required for the EdgeTPU re-derives its quantization ranges from a calibration set and discards the QAT-learned ones. On the collapse case the deployable QAT model reached 0.485 INT8 versus 0.693 for PTQ; the higher QAT score (0.806) corresponds to a non-deployable export in which only a small fraction of operations run on the accelerator. PTQ is therefore retained throughout, and the rare stochastic collapse (two of twelve seeds in Table A10) is screened by quantizing several checkpoints and keeping the best

References

- Palossi, D.; Conti, F.; Benini, L. An Open Source and Open Hardware Deep Learning-Powered Visual Navigation Engine for Autonomous Nano-UAVs. In *Proceedings of the 2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*; IEEE: New York, NY, USA, 2019; pp. 604–611.
- Gholami, A.; Kim, S.; Dong, Z.; Yao, Z.; Mahoney, M.W.; Keutzer, K. A Survey of Quantization Methods for Efficient Neural Network Inference. *arXiv* **2021**, arXiv:2103.13630. [\[CrossRef\]](#)
- Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. *arXiv* **2016**, arXiv:1506.02640. [\[CrossRef\]](#)
- Redmon, J.; Farhadi, A. YOLOv3: An Incremental Improvement. *arXiv* **2018**, arXiv:1804.02767. [\[CrossRef\]](#)
- Bochkovskiy, A.; Wang, C.-Y.; Liao, H.-Y.M. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv* **2020**, arXiv:2004.10934. [\[CrossRef\]](#)
- Zhang, Q.; Kanjo, E. A Multicore and Edge TPU-Accelerated Multimodal TinyML System for Livestock Behavior Recognition. *IEEE Internet Things J.* **2025**, *13*, 666–677. [\[CrossRef\]](#)
- Habash, N.; Abu Alqumsan, A.; Zhou, T. Recent Real-Time Aerial Object Detection Approaches, Performance, Optimization, and Efficient Design Trends for Onboard Performance: A Survey. *Sensors* **2025**, *25*, 7563. [\[CrossRef\]](#) [\[PubMed\]](#)
- Alqahtani, D.K.; Cheema, M.A.; Rodriguez, M.A.; Toosi, A.N. A Comprehensive Evaluation of Deep Learning Object Detection Models on Heterogeneous Edge Devices. *arXiv* **2024**, arXiv:2409.16808. [\[CrossRef\]](#)
- Bewley, A.; Ge, Z.; Ott, L.; Ramos, F.; Upcroft, B. Simple Online and Realtime Tracking. In *Proceedings of the 2016 IEEE International Conference on Image Processing (ICIP)*; IEEE: New York, NY, USA, 2016; pp. 3464–3468.
- Zhang, Y.; Sun, P.; Jiang, Y.; Yu, D.; Weng, F.; Yuan, Z.; Luo, P.; Liu, W.; Wang, X. ByteTrack: Multi-Object Tracking by Associating Every Detection Box. In *Computer Vision—ECCV 2022*; Springer: Cham, Switzerland, 2022; pp. 1–21.

11. Amazon Web Services. FreeRTOS Documentation. Available online: https://www.freertos.org/Documentation/RTOS_book.html (accessed on 4 June 2026).
12. MicroPython Project. MicroPython Documentation. Available online: <https://docs.micropython.org/en/latest/> (accessed on 4 June 2026).
13. TensorFlow Authors. TensorFlow Lite for Microcontrollers. Available online: <https://github.com/tensorflow/tflite-micro> (accessed on 4 June 2026).
14. Google. TensorFlow Lite and LiteRT Documentation. Available online: <https://ai.google.dev/edge/litert> (accessed on 4 June 2026).
15. NXP Semiconductors, I. MX RT1170 Processor Reference Manual. Available online: <https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus/i-mx-rt1170-crossover-mcu-with-arm-cortex-m7-and-cortex-m4-cores.i.MX-RT1170> (accessed on 4 June 2026).
16. MIPI Alliance. MIPI CSI-2 Camera Interface Overview. Available online: <https://www.mipi.org/specifications/csi-2> (accessed on 4 June 2026).
17. Cheng, M.; Ma, Z.; Asif, M.S.; Xu, Y.; Liu, H.; Bao, W.; Sun, J. A Dual Camera System for High Spatiotemporal Resolution Video Acquisition. *IEEE Trans. Pattern Anal. Mach. Intell.* **2021**, *43*, 3275–3291. [CrossRef] [PubMed]
18. Littlefs Project. Littlefs: A Little Fail-Safe Filesystem Designed for Microcontrollers. Available online: <https://github.com/littlefs-project/littlefs> (accessed on 4 June 2026).
19. Eclipse Foundation. Eclipse ThreadX—FileX and LevelX. Available online: <https://github.com/eclipse-threadx/filex> (accessed on 25 June 2026).
20. Foehn, P.; Kaufmann, E.; Romero, A.; Penicka, R.; Sun, S.; Bauersfeld, L.; Laengle, T.; Cioffi, G.; Song, Y.; Loquercio, A.; et al. Agilicious: Open-Source and Open-Hardware Agile Quadrotor for Vision-Based Flight. *Sci. Robot.* **2022**, *7*, eabl6259. [CrossRef] [PubMed]
21. PX4 Autopilot Community. PX4 Autopilot User Guide. Available online: <https://docs.px4.io/main/en/> (accessed on 4 June 2026).
22. MAVLink Development Team. MAVLink Developer Guide. Available online: <https://mavlink.io/en/> (accessed on 4 June 2026).
23. Wang, J.; Yu, Z.; Zhou, D.; Shi, J.; Deng, R. Vision-Based Deep Reinforcement Learning of UAV Autonomous Navigation Using Privileged Information. *Drones* **2024**, *8*, 782. [CrossRef]
24. Bitcraze. AB Crazyflie Documentation. Available online: https://www.bitcraze.io/documentation/system/doc_nav/ (accessed on 6 April 2026).
25. Jouppi, N.P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*; Association for Computing Machinery: New York, NY, USA, 2017. [CrossRef]
26. Lee, H.; Zhang, Y.; Shen, Y.-T.; Kwon, H.; Bhattacharyya, S.S. Exploring the Impact of Synthetic Data for Aerial-View Human Detection. *arXiv* **2024**, arXiv:2405.15203. [CrossRef]
27. Raju, S.R.K.P.; Singh, R.; Velmurugan, M.; Sanket, N.J. EdgeFlowNet: 100FPS@1W Dense Optical Flow for Tiny Mobile Robots. *IEEE Robot. Autom. Lett.* **2025**, *10*, 128–135. [CrossRef]
28. Boiteau, S.; Vanegas, F.; Gonzalez, F. Framework for Autonomous UAV Navigation and Target Detection in Global-Navigation-Satellite-System-Denied and Visually Degraded Environments. *Remote Sens.* **2024**, *16*, 471. [CrossRef]
29. Liu, X.; Yang, Y.; Ma, C.; Li, J.; Zhang, S. Real-Time Visual Tracking of Moving Targets Using a Low-Cost Unmanned Aerial Vehicle with a 3-Axis Stabilized Gimbal System. *Appl. Sci.* **2020**, *10*, 5064. [CrossRef]
30. Games Epic. Unreal Engine 5 Documentation. Available online: <https://docs.unrealengine.com/5.0/en-US/> (accessed on 4 June 2026).
31. Shah, S.; Dey, D.; Lovett, C.; Kapoor, A. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. In *Field and Service Robotics; Springer Proceedings in Advanced Robotics*; Springer: Cham, Switzerland, 2018; Volume 5, pp. 621–635. [CrossRef]
32. Koenig, N.; Howard, A. Design and Use Paradigms for Gazebo, an Open-Source Multi-Robot Simulator. In *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*; IEEE: New York, NY, USA, 2004; Volume 3, pp. 2149–2154. [CrossRef]
33. Garrido-Jurado, S.; Muñoz-Salinas, R.; Madrid-Cuevas, F.J.; Marín-Jiménez, M.J. Automatic Generation and Detection of Highly Reliable Fiducial Markers under Occlusion. *Pattern Recognit.* **2014**, *47*, 2280–2292. [CrossRef]
34. Lepetit, V.; Moreno-Noguer, F.; Fua, P. EPnP: An Accurate O(n) Solution to the PnP Problem. *Int. J. Comput. Vis.* **2009**, *81*, 155–166. [CrossRef]
35. Krajník, T.; Nitsche, M.; Faigl, J.; Vaněk, P.; Saska, M.; Přeučil, L.; Duckett, T.; Mejail, M. A Practical Multirobot Localization System. *J. Intell. Robot. Syst.* **2014**, *76*, 539–562. [CrossRef]
36. Kabsch, W. A Solution for the Best Rotation to Relate Two Sets of Vectors. *Acta Crystallogr. Sect. A* **1976**, *32*, 922–923. [CrossRef]

37. Kuglin, C.D.; Hines, D.C. The Phase Correlation Image Alignment Method. In Proceedings of the IEEE International Conference on Cybernetics and Society, San Francisco, CA, USA, 23–25 September 1975; pp. 163–165.
38. Jocher, G.; Chaurasia, A.; Qiu, J. Ultralytics YOLOv8 2023. Available online: <https://github.com/ultralytics/ultralytics> (accessed on 4 June 2026).
39. Wang, C.-Y.; Yeh, I.-H.; Liao, H.-Y.M. YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information. In *Computer Vision—ECCV 2024*; Springer: Cham, Switzerland, 2024. [CrossRef]
40. Wang, A.; Chen, H.; Liu, L.; Chen, K.; Lin, Z.; Han, J.; Ding, G. YOLOv10: Real-Time End-to-End Object Detection. *arXiv* **2024**, arXiv:2405.14458. [CrossRef]
41. Khanam, R.; Hussain, M. YOLOv11: An Overview of the Key Architectural Enhancements. *arXiv* **2024**, arXiv:2410.17725. [CrossRef]
42. Zhu, P.; Wen, L.; Du, D.; Bian, X.; Fan, H.; Hu, Q.; Ling, H. Detection and Tracking Meet Drones Challenge. *IEEE Trans. Pattern Anal. Mach. Intell.* **2022**, *44*, 7380–7399. [CrossRef] [PubMed]
43. Lin, T.-Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; Zitnick, C.L. Microsoft COCO: Common Objects in Context. In *Computer Vision—ECCV 2014; Lecture Notes in Computer Science*; Springer: Cham, Switzerland, 2014; Volume 8693, pp. 740–755. [CrossRef]
44. Sunkara, R.; Luo, T. No More Strided Convolutions or Pooling: A New CNN Building Block for Low-Resolution Images and Small Objects. In *Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2022); Lecture Notes in Computer Science*; Springer: Cham, Switzerland, 2023; Volume 13715, pp. 443–459. [CrossRef]
45. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv* **2017**, arXiv:1704.04861. [CrossRef]
46. Wulfert, L.; Kühnel, J.; Krupp, L.; Viga, J.; Wiede, C.; Gembaczka, P.; Grabmaier, A. AlFES: A Next-Generation Edge AI Framework. *IEEE Trans. Pattern Anal. Mach. Intell.* **2024**, *46*, 4519–4533. [CrossRef] [PubMed]
47. Song, Z.; Zhang, Y.; Abu Ebayyeh, A.A.R.M. EDNet: Edge-Optimized Small Target Detection in UAV Imagery—Faster Context Attention, Better Feature Fusion, and Hardware Acceleration. *arXiv* **2025**, arXiv:2501.05885. [CrossRef]
48. Farooqui, S.A.; Taha, Z.A.K.; Uddin, M.M.; Alam, S. LAF-YOLOv10 with Partial Convolution Backbone, Attention-Guided Feature Pyramid, Auxiliary P2 Head, and Wise-IoU Loss for Small Object Detection in Drone Aerial Imagery. *arXiv* **2026**, arXiv:2602.13378. [CrossRef]
49. Keleş, M.C.; Salmanoğlu, B.; Güzel, M.S.; Gürsoy, B.; Bostancı, G.E. Evaluation of YOLO Models with Sliced Inference for Small Object Detection. *arXiv* **2022**, arXiv:2203.04799. [CrossRef]
50. Zhu, X.; Lyu, S.; Wang, X.; Zhao, Q. TPH-YOLOv5: Improved YOLOv5 Based on Transformer Prediction Head for Object Detection on Drone-Captured Scenarios. In *Proceedings of the 2021 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*; IEEE: New York, NY, USA, 2021; pp. 2778–2788. [CrossRef]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.